overload|74
August 2006
ISSN: 1354-3172
www.accu.org

Comments Considered Good
William Fishburne

Introducing CODEF/CML
Fernando Cacciola

Fine Tuning for lexical_cast
Alexander Nasonov

C# Generics - Beyond Containers of T
Steve Love

The Kohonen Neural Network
Habdank-Wojewódzki and Rybarski

The Documentation Myth
Allan Kelly

OVERLOAD 74

August 2006
ISSN 1354-3172

Editor

Alan Griffiths
overload@accu.org

Contributing editor

Mark Radford
mark@twonline.co.uk

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddeaus Froggley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@gmail.com

Pippa Hennessy
pip@oldbat.co.uk

Tim Penhey
tim@penhey.net

Advertising enquiries

ads@accu.org

Cover art

Pete Goodliffe
pete@cthree.org

Design

Pete Goodliffe

Copy deadlines

All articles intended for publication in Overload 75 should be submitted to the editor by 1st September 2006 and those intended for Overload 76 by 1st November 2006.

The ACCU

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

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

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

Copyrights and Trade Marks

Take a Step Forward

"Nobody made a greater mistake than he who did nothing because he could do only a little." – Edmund Burke.

It is up to you to get involved

Once again I'm pleased with the response from authors – for the second issue running the advisors have been able to concentrate on providing assistance to the authors and to the editor and have not found it necessary to write articles to reach our minimum length. As you can see, we have comfortably achieved that again. Thanks to everyone who submitted an article! (I know some were redirected to C Vu, but the effort is appreciated.) I trust that those of you who have been thinking of writing will be inspired by this example and do so next time.

One of the benefits that ACCU can provide is the opportunity to learn new skills safely outside the work environment. I know that I have learnt a lot from my ACCU activities – as an author, as the chair, as a speaker and now as a journal editor. When people occupy a position for an extended period there are pros and cons: they provide us all the benefit of their experience, but they also block others from that same learning experience.

If you check your back issues you'll see that some of the advisors have been working on Overload longer than I've been editor – a long time. We've all benefited from their diligence and expertise over the years. Thanks team!

For a couple of reasons now is a good time for new people to get involved in producing this magazine. The first reason is to bring new skills to the task: over the last couple of issues I've had to ask for help from outside the team to review articles that require more knowledge of CSS, C# and neural networks than the current advisors have. (I'm always willing to do this, but CSS and C# are hardly obscure.) A second reason is that recently several of the current advisors have taken on new commitments in their lives. (Some of these commitments have to do with ACCU, some are personal, but all have the effect that the editorial team has reduced capacity and would welcome some new blood.)

If you have a clear idea of what makes a good article about software development and would like to help authors polish their writing then now is the time to get involved – while the old hands are here to "show the ropes" (and before some of them find their new commitments push contributing to Overload every issue out of their lives). It isn't hard – and the authors do appreciate the work done reviewing their articles.

The quest for usability

For many years I've puzzled to understand why, when people try to make a piece of software they are working on more useful (or "reusable") they take actions that have the opposite effect. I may have previously described how a simple "properties" class I wrote to hold configuration information was "improved" by another developer who needed to serialize it: by including the serialisation code he needed in the class he made it less useful in contexts which don't need the same approach to serialisation – I'm sure you can add many examples from your own experience.

What people repeatedly miss is that simple things that do one clearly defined thing are more usable than complex ones that do a host of things. There is even a clear tendency for people to add functionality "because we will need it later" – this is recognised in the Agile mantra "you aren't gonna need it" [YAGNI]. I've seen projects drowning in code that contributes nothing at the moment but was written and is being kept "because we will need it later". Not only are these projects typically over budget already, but they are also continuing to lose time because of the cost of compiling, changing and testing this *useless* code. My experience isn't unique – Tim Penhey wrote recently about a project that had recognised this condition and was investing a substantial effort to identify and remove dead code [Penhey].

The code that causes development problems may not always be completely useless – with a diverse or large user community there is often someone, somewhere, who uses it. If there are enough units sold then providing value to even a small proportion of users may justify the expense. The problem I've seen in this scenario is that there is no objective basis to make this decision: it is hard to quantify the cost of slowing down all changes to a system – not only is it difficult to quantify the lost productivity, but when new features are delivered slowly there can be significant opportunity costs (and these may be hidden – the benefits of possible features may not even be evaluated "because of the backlog").

Operating systems often meet the criteria of large, diverse user communities and it is no wonder that they have a history of developing lumps of near dead code that are needed for "backward compatibility" and have to be worked around for new development. I can remember IBM boasting proudly of how complex they had made MVS+S/390 – and, for many years, the press releases and briefings about forthcoming versions Windows have shown all the signs of optimism obscuring the difficulty of developing a codebase with so much baggage. Of course, "difficult" isn't "impossible" – we as a profession keep pushing back the boundaries of what is possible. S/390, WinNT, Win2K and WinXP were all delivered eventually – but without some of the functionality mooted in early "roadmaps".

It is, of course, easier in most projects to remove functionality that exists only in the plan than to remove functionality that has been developed in the code. In the latter case, more has been invested in creating that functionality – and there is also the risk that someone, somewhere is using it (or will want to use it someday). Hence the common experience that, in practice, as the codebase grows it accumulates increasing amounts of dead code that sucks vitality from the project.

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

Not just software

However, as I don't want to drift too far off topic, I'll get back on track with some comments about J2EE. Time was that J2EE was the new simple way to deliver a whole range of server-side applications. I remember those days – I worked on some systems in this category. As with many development tools, if what you wanted fitted the design parameters then it was great! (And if it didn't you either had to code around it or used different tools – I've worked on projects that took both approaches.) Over the years J2EE has accumulated new libraries, new technologies and grown in complexity until it gets comments like "In five years, Java EE will be the CORBA of the 21st Century. People will look at it and say, 'It had its time but nobody uses it any more because it was too complicated.'" Richard Monson-Haefel (Senior Analyst, Burton Group) – reported on The Register by Gavin Clarke [Clarke].

It isn't just software projects that that suffer from accumulating well motivated, but counter-productive, complexity. There is scarcely a day that goes by without something of this nature appearing in the news. Today's example was the UK's ID card scheme – a project filled with stuff that "might be useful" to the extent that it is far from clear that it addresses the anti-terrorism problem it purports to address.

I've not worked on a Java project for some time now, so I can't attest to the accuracy of these comments – but I heard very similar remarks about JDK5 at the ACCU conference a couple of years ago and noticed how rapidly the proportion of Java content had reduced at the last one. I'm sure Java will be around for a good while yet – but the complexity it has accumulated isn't helping.

I can imaging the C++ developers amongst you smiling at this and feeling smug that you avoided the Java "fad". Don't be! It is a lot like the decaying codebases I was describing earlier – much of the complexity in C++ is there because the standards committee doesn't dare to remove it. There are features that "might be useful" (while I've heard reports from people who are using custom allocators, these uses don't justify the cost to everyone else of maintaining them in the standard). There are features that someone started to work on, but didn't have time to finish (`valarray` does have users and there are some ongoing efforts to resolve the issues). There are even features that never seem quite right and were being reworked right up to the moment of shipping (like `auto_ptr`).

In another world

It doesn't have to be that way. While working on a codebase it is possible to delete the dead code and amputate the dying features. I know – I've done it – and the results are worth it. One of my more memorable days was when I deleted around 800 lines of code from a function and could then see that a condition had been coded incorrectly. (The deleted code had accumulated bit by bit to identify and fix up various specific scenarios that had been reported as being wrong. I deleted it bit by bit on the basis that "this shouldn't be doing anything".) Of course, it is prudent to have both version control and a good test suite in place before getting too eager with the "hatchet".

One of the keynotes at the last conference dealt with changes being made to the Python language. I know that many of us feel this was not a good topic for a plenary session, but it did give an insight into a very different strategy for maintaining a language. Changes are being made that break "backward compatibility"– something that Java and C++ go to tremendous lengths to avoid. Instead there were tools being considered that would help with the identification and migration of code that relied on the features obsoleted.

There are differences between the communities that Python and C++ support. But considerations such as "don't break backward compatibility" are not absolute requirements for Java and C++. C++ in particular supports a far wider range of environments.

Can you do anything?

You may feel that you can't do much about Overload, your codebase or the language standard. You are wrong: if you already find the time to read an article in Overload when it arrives then it doesn't take long to write some review comments; if you are developing some class it won't take any time to not add that bell or whistle; and. if you are looking at some code that isn't being used it doesn't take long to select and delete.

Progress is made one step at a time. ■

References

[YAGNI]"you aren't gonna need it" http://c2.com/cgi/wiki?YouArentGonnaNeedIt

[Penhey]"Dead Code", Tim Penhey, Overload 71

[Clarke]"JEE5: the beginning of the end?", Gavin Clarke, The Register http://www.regdeveloper.co.uk/2006/07/12/java_ee_burton_group/

Comments Considered Good

In our last issue, we offered the case against comments; in this issue, William Fishbourne responds in defence. He argues that it is not the practice of commenting that it is evil, it is the thoughtless commenter.

At the ACCU 2006 conference, the question was asked, "Is the total value of all the comments in all of the code out there less than or more than zero?" [ACCU2006] Mr. Easterbrook [Overload73] concluded that if the answer is less than zero, then it must be the practice of commenting that is at fault. Consider, for a moment, the same question with a different subject, "Is the total value of all the punditry in all the newspapers out there less than or more than zero?" If one could conclude that the pundits of the world offer little or detract from the world, following this line of reasoning, should one conclude that freedom of the press is inappropriate? Would it not be better to encourage more writing in hopes of finding better pundits? In a like manner, this author encourages more commenting for better commenters, not the elimination of comments.

Learning from the experts

Mr. Easterbrook took a look at the comments that experts put in books for beginners and found the comments woefully inadequate. This is an unfair test, however, as the books for beginners use comments to explain simple concepts to beginning programmers. A look at the complexity of the programs in beginning programming books would likewise be found wanting. Let us consider, however, how experts comment code designed for the advanced programmer. Stan Lippman in *Essential C++* [Lippman2000] has this example:

```
template <typename valType>
void BTnode<valType>::
lchild_leaf( BTnode *leaf, BTnode *subtree )
{
  while ( subtree->_lchild )
    subtree = subtree->_lchild;
  subtree->_lchild = leaf;
}
```

Mr. Easterbrook's argument is made! The code is, however, dense and while it is possible to figure out that `BT` probably means 'Binary Tree', especially with the help of `lchild` which is probably 'left child', it isn't intuitively obvious what this function is doing. After all, `lchild` could be 'least child' or `BT` could mean 'Branch of the Tree'. All in all, however, the code is easy enough for an good programmer to understand. Some might argue that longer or more descriptive names might make understanding easier, those same people might argue that such long names are hard to type in and lead to typos which are false errors. Mr. Lippman has chosen to comment upon invocation where explanation is useful:

```
// lchild_leaf() will travel the left subtree
// looking for a null left child to attach ...
```

William Fishburne is a graduate of the University of Maryland in Computer Science and works as a consultant to the United States Government. He is also a production coordinator for Project Gutenberg where numerous programming opportunities can be found for interested volunteers. You can contact William at bfishburne@gmail.com

These comments are useful and help elucidate the manner in which the function is being used.

Computers have often been compared to "universal tools" and procedures have also born this title. To the extent that a given procedure is generic or universal, the invocation of that procedure should be accompanied with comments that (as Mr. Easterbrook points out):

- Say why and not how
- Are meaningful
- Used where the code is non-obvious and/or non-portable

Tying the comment to the invocation, in this case, helps facilitate understanding of the code at the point where that understanding is needed.

Is code self-documenting?

Simply put, no. As an advocate of self-documenting code, this author is frustrated to admit it, but the simple fact is that code is not self-documenting no matter how hard a programmer may try. As the first example shows, even code that uses meaningful names in an intuitive manner can be very hard to understand.

Consider this snippet from Marshall Cline's *Essential C++* [4]:

```
class stack {
public:
  stack() throw();
  unsigned size() const throw();
  int top() const throw(Empty);
  void push (int elem) throw(Full);
  int pop() throw(Empty);
protected:
  int data_[10];
  unsigned size_;
};
```

Now consider the comments recommended by the author:

```
int pop() throw(Empty);
  // PURPOSE: Pops and returns the top element from
  // this
  // REQUIRE: size() must not be zero
  // PROMISE: size() will decrease by 1
  // PROMISE: Returns the same as if top() were called
  // at the beginning
```

The comments tell us what to expect from the member function and this type of information is critically important for interfaces, etc. While separate test functions might prove this functionality, it is worth taking the time to make these comments so that someone using the member function would be able to rely upon this functionality.

The comments offer us something that code alone cannot. There is no way for the declaration to state that size will decrease by 1 as a side-effect. One may argue that such side effects are components of bad code and be right

it is sad that so many programmers have abandoned the practice of actually designing code before implementing

in many cases, but that argument does not solve the problem – the code cannot document itself, particularly in side-effects.

Comments as design reference

Mr. Easterbrook refers to the use of pseudo-code as the programmer opting to code in another coding language to which the programmer has a preference. Many consider the use of pseudo-code a critical design step. In fact, it is sad that so many programmers have abandoned the practice of actually designing code before implementing (or, perhaps more correctly: hacking) it. Whatever the nature of the design process, it is worthwhile capturing the design as comments: first, as a reference for the programmer as the program is developed, and, second, as an insight into other follow-on programmers who are trying to fathom the whys and wherefores of a particular program.

The first code snippet given, might have been created in pseudo-code as:

```
// template static function lchild_leaf
// traverse the left subtree looking for a null leaf
// to attach
// accept a leaf and subtree as parameters
// while the left subtree exists
// step down the left subtree
// put the passed leaf into the left subtree
```

As comments, things like "while the left subtree exists" may seem redundant to the code that was written; however, it reflects the design process and the comments (with code removed) offer insight into the algorithm used. In this way, even "obvious comments" have a place, sort of like the paragraph headers which appear in this article. The paragraph headers aren't strictly necessary, the structure of the article should be self-evident, but the headers facilitate the reading of the article which is, after all, the point.

Comments that are used as part of the structure for writing a program should be kept in the program and not simply removed as the components are written. These comments help guide the reader through the code and help the programmer develop well-organized code which doesn't leave out crucial missing parts.

By the same token, comments should be maintained in the same manner as the code itself should be maintained. As the algorithm changes the comments, as well as the code, that was outmoded should be revised or removed. The examples given by Mr. Easterbrook are sad examples of code maintenance and simply another example of why peer code reviews are necessary. Think for a minute, would you let a peer slide on comments that are misleading or incomplete? It is in everyone's best interest to prevent that from happening.

Mr. Easterbrook has some great ideas

Compiler aware annotations are an interesting innovation, although there are still plenty of people who don't code in some graphic environment where annotations could pop up over the code itself. Mr. Easterbrook's concept of compiler aware annotations has hit on an improvement that only lacks an implementation. Perhaps some innovative programmers will get together and add this type of functionality to `gcc` and the programming environment of their choice. At the risk of inciting a riot, it seems that this would be easier to do with EMACS then with vi!

The call for meaningful comments is a good one, although banning headers is as bad as requiring them. Headers have a place (just like every book has a title and author on the cover page), but the place is to facilitate the reading and understanding of the code, not simply to meet a check box. When programming becomes a "step in a process" the art has been left behind and shoddy, cookie-cutter code is the result. Innovative thinking, like that of Mr. Easterbrook, is the hallmark of good programming, whereas meeting a series of check boxes is hardly short of automation.

As a modification of Mr. Easterbrook's call to action, leaders of code reviews should ask themselves why the comment would be useful. Perhaps most of the staff that maintains the code is composed of junior programmers and a comment that an experienced programmer would find unnecessary, should be there for the junior programmers. Perhaps there is a component of business logic that (though obvious) needs to be stated as such so that it is not later "mildly" modified without consideration for the importance of the business rule embodied in the code. Finally, the thinking process of the programmer can be documented in the comments, which is a singularly unique insight into why things are programmed in a specific manner as opposed to simply noticing that a particular path has been used.

Please remember, comments don't make code bad, programmers do. ■

References

[ACCU2006] By Russel Winder in Peter Sommerlad's session called "Only the Code tells the Truth".

[Cline1999] Cline, Marshall, et al, "C++ FAQs Second Edition, Addison-Wesley, 1999, ISBN: 0-201-30983-1

[Lippman2000] Lippman, Stanley C., "Essential C++", Addison-Wesley, 2000, ISBN: 0-201-48518-4

[Overload73] Easterbrook, Mark, "Comments Considered Evil", Overload, Issue 73

Introducing CODEF/CML

This article introduces a C# serialization facility that proposes a novel separation between object models, representing the serialized data, and class descriptors, representing the binding to the receiving design space.

I seldom find myself being completely comfortable with a development framework. I just can't help myself seeing a weakness here and there, and not even the .Net framework escapes from this criticism. On the other hand, I have used and sweated many different frameworks over the years, from Win32 in Windows 3.1, to DCOM and CORBA, and from that experience I have to admit that .Net gets its job done remarkably well. Being a long term C++ programmer, I do find many areas that I would have liked to have been different, but then I recall that there are ASP.Net, VB, Java and C# programmers out there that wouldn't agree with me on pretty much any of those points. Admittedly, the .Net frameworks allows all of us (programmers with different backgrounds, experiences and mindsets) to work together in an incredibly productive way; something I've never before had the opportunity to enjoy.

The *lingua franca* that .Net represents in a heterogeneous team got me to learn and even appreciate C#, which I've been using for the last two years.

I've also got to learn and appreciate many of the .Net subsystems, like Reflection, GDI+ (which is a royal pleasure for those with a Win32 GDI experience) and particularly .Net serialization (from framework version 1.1)

.Net 1.1 serialization just worked for us out of the box without trouble until we got a complaint from the boss because files were unacceptably large. We started to work around that problem from within the framework but in the process we discovered how .Net Reflection was *not* being used to the extent it could, so we ended up writing our own replacement framework instead, which has been used in production code for more than a year now.

.Net serialization 101:

The .Net framework provides two serialization facilities. They are significantly different and serve different purposes.

One is called XML Serialization and it's mainly targeted at representing XML Documents as .Net objects. In this form of serialization only public fields are serialized, and the type information included in the XML file, if any, is driven by the target specification, such as SOAP, instead of the actual types of the objects being serialized.

The other is called just Serialization and it's mainly targeted at *object persistence*, that is, saving and loading an object graph out of core memory.

XML Serialization is not suited for persistence of application objects unless the types of these objects are supported in the target schema. Therefore, in this article, I will always refer to the second form of serialization and I will use the term to refer to the complete round trip process; that is, including deserialization.

Fernando Cacciola Born and living in Argentina, Fernando has been producing software since 1984. In 2003 he became a freelancer and founded SciSoft, currently contracted by US and European companies. He is a developer of the Boost (www.boost.org) and CGAL (www.cgal.org) projects. Visit his home page at http://fcacciola.50webs.com

.Net serialization is controlled at the highest level by a `Formatter` object. Via a `Formatter` object you can `Serialize` and `Deserialize` an object graph into and from a `Stream`. The framework provides two of formatters `BinaryFormatter` and `SoapFormatter`. The first stores binary data in the `Stream` and the second stores SOAP-encoded XML Elements. The `SoapFormatter` is similar in effect to `XMLSerialization` but is not exactly the same.

Only those classes marked as `[Serializable]` are actually serialized when passed to a formatter.

If all you do is mark a class like that, all of its fields are automatically serialized without any further intervention. This is extremely programmer friendly, but like most magical things when it doesn't work it really doesn't. In our case, the problem was the size of the files, it was just way too big for us.

The logical solution was obvious: in theory, not all of the data members need to be saved, and in our case that was particularly true: our objects are geometric figures and they cache a lot of information like bounding boxes, polygonal approximations, lengths, areas, etc. none of them need to be saved since they can be recomputed on load.

Well, it turns out that you can mark fields with the `[NonSerialized]` attribute which prevents them from being saved and loaded. However, the deserialization process simply ignores them so this attribute alone is not enough if those fields are dependent, that is, their values must be computed after the other fields have been loaded. In that case, you must also implement the `IDeserializationCallback` interface which defines the method `OnDeserialization` called for each deserialized object after the complete object graph has been deserialized:

```
[Serializable] class Circle : IDeserializationCallback
{
  void OnDeserialization( object Sender )
  {
    m_area = Math.PI * m_radius * m_radius;
  }
  double m_center_x, m_center_y, m_radius;

  [NonSerialized] m_area;
}
```

If you make any change to a `[Serializable]` class and the Formatter finds a mismatch between the current class fields and the saved fields it will throw a `SerializationException`, even if the mismatch is just a field removed in the current class. Unfortunately, classes often change after they start being serialized. When that happens, you just need to ask .Net to hand you total control of the process.

When a `[Serializable]` class implements the `ISerializable` interface it makes itself completely responsible for the serialization/deserialization process. It is totally up to you to match the data saved and loaded with the object's state. This method allows you (and requires you) to fill in a dictionary called `SerializationInfo`, which is what the

Formatter actually stores in the Stream as a representation for your object. You still need to mark the class as `[Serializable]` though because interfaces don't define constructors and the `ISerializable` interface only defines a method used on save but doesn't provide the deserialization counterpart, a constructor that takes a `SerializationInfo` dictionary to restore the object state from the loaded data. See Listing 1: Implementing the `Iserializable` interface.

There are other low-level facilities in .Net serialization that won't be discussed in this article, like `SerializationBinder` objects that allow you to instruct the Formatter to map a saved type to its current counterpart, or `SerializationSurrogate` objects that you can use to serialize closed third-party types.

.Net serialization version 1.1 almost worked for us, but its weakness was that it offered two opposing extremes: total automation with no control at all with `[Serializable]` alone, or complete control with no automation at all with the interfaces and the helper objects. We felt like Reflection could be used to provide something in between that mixes automation and control.

After we invented our own framework, .Net 2.0 was released and .Net serialization was extended precisely to better use Reflection to give you some control without losing automation. The additions in .Net 2.0 are:

The data member attribute `[OptionalField]` which instructs the Formatter not to throw if this member is missing in a saved file.

And the method attributes `[OnDeserialized]`, `[OnDeserializing]`, `[OnSerialize]` and `[OnSerializing]` which let you hook on the 4 stages of the process and change your object's state if necessary.

However, we believe that the concepts and mechanisms developed in our framework are worth describing even with the .Net 2.0 Serialization extensions available.

CODEF/CML 101:

The main reason why we needed to implement `ISerializable` was to control which data members to serialize in order to reduce file size. Eventually we realized that the .Net serializer was using Reflection to detect `[Serializable]` classes and to automatically read the object fields, save them, then read and set them back to a newly created object. But reflection can be used even further to mark, in the code, which data members to serialize, so we created CODEF/CML as a replacement for .Net serialization.

All objects have a value (or state if you like), and two objects which are not equal are equivalent if they have the same value (or state).

Serialization can be viewed as the process of *transferring* the value of an object into another, where transferring here necessarily involves *getting* the value of an object, *storing* it into a *medium*, then *extracting* the value out of the medium, and *setting* the value into the receiving object (via initialization or assignment).

Under this view, copy-construction and assignment is not a form of serialization because the value is transfered directly and not indirectly through an external medium. On the other hand, saving/loading objects to a file, transmitting them across a boundary, and even cloning an object indirectly stepping through an external medium are all forms of serialization.

```
[Serializable] class Circle : ISerializable
{
  private Circle( SerializationInfo aInfo,
StreamingContext aContext)
  {
    m_center_x = aInfo.GetDouble( "m_center_x" );
    m_center_y = aInfo.GetDouble( "m_center_y" );
    m_radius   = aInfo.GetDouble( "m_radius" );

    m_area = Math.PI * m_radius * m_radius;

  }

  public void GetObjectData( SerializationInfo
aInfo, StreamingContext aContext )
  {
     aInfo.AddValue( "m_center_x", m_center_x );
     aInfo.AddValue( "m_center_y", m_center_y );
     aInfo.AddValue( "m_radius"  , m_radius   );
  }

  double m_center_x, m_center_y, m_radius, m_area;
}
  };
```

Listing 1

From that characterization, serialization can be considered as the composition of two layered processes. On the bottom layer there is the process of getting the value out of one object and setting the value into another object. On the top layer there is the process of storing the value of an object into an external medium and extracting that value back out of the medium. Such a decomposition is useful because it decouples the get/set step (bottom layer) from the store/extract step (top layer), allowing the top layer to be provided by different agents, like one storing/extracting values to and from a file and another transmitting/receiving values over a channel.

This decomposition implies that values are themselves objects, so the bottom layer can be seen as a *metadata codec* as it encodes and decodes the value of an object into metadata about it. The top layer can be seen itself as another codec as its encodes and decodes the metadata about the value of an object into some arbitrary specific code (a domain-specific XML for example).

You can see that these layers are implicitly present in many existing serialization frameworks. For example, in .Net the `SerializationInfo` object that is the metadata, which is used by different "top layers" like the `BinarySerializer` or the `XmlSerializer`.

CML is similar to the XML files produced by .Net's SoapFormatter but is more compact

When designing our framework I decided to formalize and even name these two layers:

CODEF

The bottom layer is called CODEF, which stands for Compact Object Description Framework.

CODEF uses two separate objects as metadata: descriptors and models. In conjunction, they codify the value of an object. Thus, CODEF encodes such a value as a pair (descriptor+model) and decodes a (descriptor+model) as a value set into a new object which CODEF instantiates.

A descriptor describes the type of the object in a generalized format. It is basically a list of data member fields along with some flags and some method fields . CODEF uses reflection to create descriptors automatically.

A model describes the value of an object in a generalized format. It is basically a list of named values (it is equivalent to `SerializationInfo`).

CODEF uses reflection to create models automatically.

The intersection between models and descriptors is the name of the data member field. That name matches each entry in a descriptor with the corresponding entry in a model.

CML

The top layer is called CML, which stands for Compact Markup Language.

CML encodes CODEF models as XML files, and decodes appropriate XML files back as CODEF models. We used XML as the final encoding not to interoperate with open standards, like SOAP, but to allow us to inspect saved documents in a text editor in case of versioning problems. This turned out to be very useful as I was able, many many times, to find out in a snap why some old file couldn't be loaded back with the current code. Our application compresses the CML text file using ZLib to produce small files (3 times smaller, on average, than what we had when we started) CML is similar to the XML files produced by .Net's `SoapFormatter` but is more compact because it doesn't follow all the SOAP protocol (that was not our goal).

Consider the following types:

```
class Point { int x,y }

class Bbox
{
  Point bottom_left;
  Point top_right;
}

class Figure
{
  Bbox bbox;
}
```

In CML this will look similar to this:

```
<Figure bbox.bottom_left="0,0" bbox.top_right="5,5"/>
```

Instead of this:

```
<Figure>
   <Bbox host="bbox">
     <Point host="botton_left">0,0</Point>
     <Point host="top_right">5,5</Point>
   </Bbox>
</Figure>
```

If you have ever seen serialization-based XML files you are likely to be familiar with the second verbose form but not with the first compact form.

The first thing to notice here is that CML can use XML attributes instead of XML elements, even for data members (the XML attributes are those `name="value"` tags right after the `Figure` markup).

If you look closely enough, you'll notice that the CML attributes, `bbox.bottom_left` and `bbox.top_right`, placed in the context of the encoding for the value of a `Figure` object, refer to a data member of a data member. That is, CML can encode the value of a data member nested any level deep directly from the root object as an XML attribute of the form:

```
"data_member.sub_data_member.sub_sub_data_member.····=
value"
```

Using CODEF/CML

Listing 2 shows some sample illustrative user code.

CODEF/CML is based on Reflection, but in order to keep it simple, it doesn't attempt to analyze each and every type in the system (though it could as Reflection permits that). Instead, you need to explicitly tell the framework which types it should cover. That is the purpose of the `[Described]` attribute prepended to the definition of the `Point` and `Circle` types.

As I've already mentioned, the main goal of CODEF/CML is to allow you to decide which data members must be saved. Hence, only those data members explicitly marked with the attribute `[DField]` are modeled (thus serialized by CML).

A CODEF descriptor object contains a list of `DField` objects, each one in turn encapsulating a .Net reflection's `FieldInfo` object which essentially contains all the needed information about a data member (including methods to set and get its value).

A CODEF model object contains a list of `MField` objects, each one it turn encapsulating a `string` with the field's name and an `object` with the field's value. There is no type information in a model's `MField`, but each `MField` is implicitly associated to the corresponding descriptor's `DField` by the field name. Together, a descriptor and a model completely codifies an object's value.

Each data member you mark as `[DField]` contributes a DField+MField pair in the encoding. Those data members which are not marked as `[DField]` are simply left out completely.

When CODEF needs to set the decoded value into a new object it uses the default constructor to instantiate the object and then it sets each `DField` automatically via reflection. Since the unsaved fields may depend on the saved fields (they usually do), CODEF calls the method marked with the attribute `[Fixup]`, if any, whose job is to recompute the unsaved dependent data members. This method can be private and can be named any way you like (because CODEF detects the method by its attribute, not by name).

The attribute `[InPlace]` is parsed by CODEF and merely becomes a flag in the corresponding `DField` and `MField`. CML then interprets that flags as indicating that the value must be encoded as an XML Attribute of the parent Element.

Versioning issues

Serialization, as a process, can be considered as the transfer of values from a sender object to a receiving object, with a dimension of time or space in between. Requiring the class definitions of the sender and receiver objects to match exactly is a desirable but largely unfeasible goal: I've never had the luxury of working on a system for which serialization facilities were added *after* the object model for the system was completely finished. In practice, you start serializing objects whose structure keeps changing after the first files are saved. Even if initially you simply do not support old files, sooner or later, earlier end users like testers begin to save files with objects still under development.

Versioning is the term used to refer to all the synchronization required to match the design subspace of the sender with the design subspace of the receiver. I speak of design subspace instead of class definitions because in some extreme scenarios the two subspaces might contain totally unrelated classes.

To my knowledge, the only systems that are capable of totally matching completely unconnected design subspaces are those which communicate via a high level generalized code. The best example that comes to mind is HTML and all its derivatives, from XML to SVG.

In classical versioning, problems appear when you start changing serializable classes but you need to read files saved with the old definitions. The simplistic solution is to populate the design space with the history of changes: that is, you never really change class A, instead, you create a new class B to replace it. Although this makes versioning a complete non-issue, it is, like most simplistic solutions, totally useless: imagine the design space after years of changes in a system with 200 (active) classes.

Class definitions continuing to change after 2, 5 or 10 years is not at all uncommon. In fact, it's called ***refactoring*** and is the best thing that can happen to old source code.

In versioning there are 3 archetypal scenarios of increasing complexity:

The first scenario is when you delete or add serializable data members to a class. This can be handled easily in most serialization frameworks: Deleted data members are extracted back but just left unset (because they are no longer in the class), and new data members are simply not extracted at all and you need to explicitly give them a sensible default.

Scenario 1 In the .Net framework, if you use the automatic approach (simply marking a class as `[Serializable]`) you'll get an exception whenever the *current* class definition contains data members that were not saved, but in the low level approach you can simply set the new data members to a default value in the serialization constructor (the one taking a `SerializationInfo` as a parameter).

Traditionally, a *version number* is saved along with every object so you can know, on load, which class definition was used when that data was saved. Unfortunately, version numbers are extremely error prone since it is totally up to you to relate a number to a particular historical class definition. Using version numbers successfully requires an uncommon discipline as you need to keep proper record of the definitions for each number, which in practice means a lot of side work whenever you change a class.

```
[Described] public struct Point
{
  public Point() {}

  public Point( float x_, float y_ ) { x  = x_; y =
y _; }

  [DField] float x,y;
}

[Described] public class Circle
{
  [Fixup] object OnLoad()
  {
    perimeter = 2 * Math.PI * radius;
    area      =     Math.PI * radius * radius;

    return this;
  }

  [DField] [InPlace] Pen   pen;
  [DField] [InPlace] Point center;
  [DField]           float radius;

  double perimeter;
  double area;

  Circle() {} // CODEF needs a default ctor,
              // but it can be private
}
```

Listing 2

Using the low-level .Net serialization approach you can add the version number to `SerializationInfo` even if that is not really part of the object.

Alternatively, using .Net serialization, you could also enumerate each entry in the `SerializationInfo` and match that, programmatically via reflection, with the actual data members in the class, setting only matching members.

Scenario 2 The second scenario is when the type of a data member changes, or the name of a type changes. This typically breaks most serialization frameworks, like .Net serialization, because the type of each value is saved so that the loader can read the value back (even if the static type of a value is generic, like "object", its dynamic type must be concrete and the loader needs to know which is it).

Scenario 3 The third scenario is when the design space changes radically (entire class hierarchies are replaced with new ones). The best and possibly only solution here is to keep the old classes around, read *them*, and make all the necessary conversions.

Scenario 4 There is a fourth scenario that is actually outside the domain of any serialization framework but which is related to serialization nevertheless: when serialized objects hold non-serializable objects. A non serializable object could be a Bitmap, or a Font, or some opaque third-party type whose state is hidden to the application. In these cases you cannot, or would not, save the actual object's state, so instead you save *something*, like a file name, or a string concatenating a Font Family name and Style, that, in your application, refers to the object. On load, typically as a global postprocessing stage after all the objects have been read back, you set the actual object within its parent locating it using the saved reference.

CODEF/CML Versioning facilities

I've described how CODEF uses both models and descriptors, and you might have asked why two separate objects with an implicit correspondence and not just one, like `SerializationInfo`?

Simple: to simplify some versioning issues. How? Because descriptors are always current. That is, when you load a class, both CML and CODEF uses the descriptor of the current definition of the class. Unlike any other

serialization framework I've ever seen, the loader is not tied to the potentially outdated description of a class that is stored in a saved file.

Removed and New data members:

Since descriptors are always current, CODEF knows when a saved `MField` (from the saved model) no longer matches a current `DField` (because a data member was removed) so it just ignores it. It also knows when there are unmatched `DFields`, that is, new data members. In this case though our current implementation simply assumes that the default constructor gives ALL fields a sensible default, so it also just ignores unsaved new data members.

CODEF always calls the default constructor to instantiate a new receiving object *before* the saved fields are set. This is suboptimal, yes, because saved fields are first initialized with a default value and then assigned their actual values. We just didn't consider this issue critical enough to complicate the design specially considering that managed objects, unlike unmanaged C++ objects, use a memory model in which *all* data members are initialized, either to zero or to the default value given in the member definition, before *any constructor is called* (thus you just cannot use a special constructor that does nothing as you could in unmanaged (pure) C++). [I do not know if the compiler optimizes away the default initialization of data members which are explicitly assigned in the constructor.. but I guess not]

The process of converting an arbitrary value to and from a string is far from trivial

Implicit typing:

Recall the figure example:

```
<Figure bbox.bottom_left="0,0" bbox.top_right="5,5"/>
```

If you look even closer than before you'll notice that there is only one type there: `Figure`.

When CML parses that line back it knows it has to produce a model for an object of type `Figure`, so it uses CODEF to get a descriptor of `Figure`. This descriptor is always up-to-date with the current definition of `Figure`. That `Figure` descriptor tells CML that, *currently,* a `Figure` has a field named `bbox` of type `BBox`. Similarly, CML gets a current descriptor for `BBox` so it knows that a `BBox` has two fields named `bottom_left` and `top_right` of type `Point`. As you can see, it doesn't at all matter which type `bbox` and its own members had when this was saved.

Normally, as in .Net serialization and every serialization framework I've ever seen, the saved data explicitly encodes the concrete type of the value being saved to allow the loader to regenerate the object that corresponds to that value. This introduces what I call an *early type binding*: by the time you get the value from the loader it is already of a concrete type that is defined by the saved data instead of the variable that is receiving it. However, since the type of the saved value must be, necessarily, constrained by the declared type of the variable that will receive the value on load, such early type binding can be worked around, in some cases, using descriptors as the loader knows the current declared type of the receiving variable and can use it to regenerate the object.

Suppose you have the following struct:

```
[Described] struct Point
{
  public Point() {}
  public Point( float x_, float y_)
              { x = x_; y = y _; }
  [DField] float x,y;
}
```

The data member fields `x` and `y` are not polymorphic so the objects reconstructed by the loader must be of type `float`, and CODEF knows that because the current descriptor says so. Consequently, there is no need at all to include the type in the serialized data. This not only saves space, which is significant by itself, but it also allows you the change the type of `x,y` provided that the encoded values of `x,y` (strings in the case of CML) can be decoded back into the new type. For example, you can change `float` to `double` and it just works, without any extra work on your part.

You might be thinking that you can also change `float` to `double` using the .Net serializer since you can simply convert the `float` read back to a `double` at the point where the value is assigned to the data member. That's correct and you can always use a conversion to handle type changes, but using the currently declared type of the receiving variable might skip the conversion altogether (as in the `float->double` case above).

Unfortunately, the declared type of the receiving variable cannot always be used to reconstruct the saved object so CODEF cannot always omit the type in the saved data. One case is when the declared type is explicitly or implicitly just `object` (implicitly is the case of a container like `ArrayList`). Another case is when the declared type is polymorphic: when the declared type of the variable is `Base`, but the concrete type of the object held by the variable is `Derived`.

Described and non-described types:

CODEF/CML fully understands only `[Described]` structs/classes (but this is by design and not an inherent impossibility since via Reflection you can create a descriptor/model for any type in the system). Non-described types are classified in 3 groups: containers, primitive types and everything else. CODEF/CML needs to detect containers because it has to encode the values of the contained objects differently than it does for data-members (there is no field "name" for instance).

Primitive types are detected as such because if the type is declared in a data member field (that is, the primitive value is not stored in a container or boxed in an object) CML does not need to encode this type (it is left implicit in the CML file). For everything else, for which CODEF has nothing to say, CML has no choice but to encode the concrete type, even if it is not polymorphic.

The Textualizer

Values of a non-described type are *atomic* from the CML point of view (CML cannot access its structure without a descriptor for it). In a CML file, atomic values are rendered as a single XML text.

The process of converting an arbitrary value to and from a string is far from trivial. In fact, the whole serialization framework can been seen as doing just that. I call that process *textualization:* A value can be textualized, that is, encoded as a string; and can be detextualized, that is, parsed back from a string. Textualization is not exactly the same as conversion to/from string. The difference is that textualization *requires* the conversion to be *round trip*: that is, `detextualize(textualize(val))==val` must hold for any value val. This requirement is often not fulfilled by string conversion functions.

The fundamental problem of textualization in CML is that it needs to textualize values of arbitrary types, including those it knows nothing about (though it could using reflection). For that reason, CML doesn't handle that at all. Instead, it uses a special singleton object called `Textualizer`, which can be seen as a side-product of the framework.

The textualizer knows how to textualize values of primitive type (it uses .Net XmlConvert for that). For other types you can either implement the interface `ITextualized` or register, non-intrusively, an `ItextualizedSurrogate` (Listing 3).

In CML, values of type Pen are rendered as a single string which is even more compact that using [Described] (that's why there is this option)

If the type is third-party you must implement the textualization agent as a separate class and register it with the Textualizer, Listing 4:

The TypeMap

Except in the case of implicit typing of unboxed primitive types, CML needs to encode a `Type` as a string and get a `Type` back from its string ID. This is similar to the textualization problem except that the object that needs to be recreated from a string is a `Type`.

Given an object `t` of type `Type`; `t.FullName` is a string encoding that is guaranteed to fulfill the round-trip requirement when used as an argument to its counterpart method: `Type.GetType(string)`.

This should be enough; but it isn't, because `Type.GetType()` returns null if the `Type` is in a different Assembly (DLL) than the one calling that `Type.GetType()`.

To get back a type from its `FullName` encoding you need to search for it in all the Assemblies of your application.

Again, CML itself doesn't handle this but instead it relies on a `TypeMap` to do that.

A `TypeMap` is anything implementing the following interface:

```
public interface ITypeMap
{
  Type GetType( string aID );
}
```

When you call `CML.Read()` to load a file you must pass some `ITypeMap` to it.

Typemaps are chained and the `GetType()` request is passed down the chain until someone returns a non-null `Type`. The current framework implementation comes with an `ExplicitTypeMap`, a `SystemTypeMap` which merely returns `Type.GetType(aID)`, and an `AssemblyTypeMap` which searches the type in the entire system in case none of the other maps find it.

The `ExplicitTypeMap`, normally the first in the chain, is there to help with a sort of versioning issue which is typically a huge problem when it shouldn't: type renaming. If the saved data speaks of type "animal" but the current class name is "Animal", you're in trouble even if that's the only thing that changed. But what if you could tell CML that "animal" is now called "Animal"?. Well, you can… using an `ExplicitTypeMap` registering with the `Type` that corresponds to a given string ID.

CML Encoding

The job of CODEF is to encode and decode types and values in a generalized form: descriptors and models. But that's just half the story. The job of CML is to encode and decode types and objects, using CODEF descriptors and models when it can, into an XML-like text file.

CML encodes objects based on the following rules which apply recursively to each distinguishable subobject.

If the object was already rendered as XML, then it has an Instance ID (administered by CML) and is encoded as an XML Element: `<HostField href="#instanceID />` or an XML Attribute: `HostField=#instanceID`. `HostField` is the name of the corresponding data member field on the parent object (if any, items in a container for instance have no host field).

The choice between an XML Element or Attribute is given by the `InPlace` flag of the field (controlled via the `[InPlace]` attribute).

If the object is modeled, which means that its dynamic type is described and CML can create a model for it via CODEF, it is encoded as an XML Element:

`<ConcreteTypeName id="#instanceID" host="HostName">`

along with XML Attributes or XML Elements corresponding to each `MField` (each data member).

If the object is unmodeled but is a container, it is encoded as an XML Element:

`<ConcreteTypeName id="#instanceID" host="HostName">`

along with XML Elements for each item in the container.

If the object is unmodeled but its *declared type* is primitive but not `object`, it is encoded as an XML Attribute: `"HostField=textualized-value"`

If the object is unmodeled but its declared type is object or non primitive, it is encoded as an XML Element:

`<ConcreteTypeName id="#instance" host="HostField">`
`textualized-value</ConcreteTypeName>`.

```
interface ITextualized { string Textualize(); }

public struct Pen : ITexutalized
{
  public Pen( uint color_, int width_ )
  {
    color = color_;
    width = width_;
  }

  public string Textualize()
  {
    return Color.ToString() + "," +
Width.ToString();
  }

  static public object Detextualize( string aTextual
)
  {
    string[] lTokens = aTextual.Split(",");
    uint Color = int.Parse(lTokens[0]);
    int  Width = int.Parse(lTokens[1]);
    return new MyPen(Color,Width);
  }

  uint Color;
  int  Width;
}
```

Listing 3

CML Decoding

Upon decoding, CML must regenerate objects, and for that it needs to get to its type first. If the CML encoding includes the type, as is always the case except for primitive unboxed fields, CML uses the `TypeMap` it receives to get the `Type` of the saved value. If the type is implicit, CML uses the `HostField` to lookup the corresponding `DField` in the `descriptor` of the parent object to get to the needed type.

If the object to be regenerated is modeled (that is, its type is described), CML decodes the XML Element creating a model object out of it and passes that to CODEF to complete the regeneration. If the object is not modeled, CML decodes the XML Element or Attribute using the singleton Textualizer to detextualize the string which is encoding the value into the resulting object.

Each regenerated object of reference-type (instead of value-type), which always has an instance ID, is kept in a dictionary with its ID as key. Thus, if the XML Element or Attribute is a reference to an instance ID, the object is just taken from the dictionary.

Mismatch handling and additional versioning features

CML uses the HostName to lookup a matching `DField` in the `descriptor` of a parent class. If there is no such `DField` the value is just ignored (as it normally corresponds to a data member deleted from the class), unless the struct/class contains the following special method:

```
[SetField] void SetField ( Type aType, string aName,
                           object aValue );
```

which is then called for any mismatching value.

If there is a `DField` for a particular `MField` (that is, the data member still exists) but the concrete type of the data member object (as encoded in the CML) is not a subtype of the declared type of the `DField`, CODEF throws a `TypeMistmatchException` unless the class has a `SetField` method, in which case it just calls it, passing the saved type as the `aType` parameter and letting that method take care of the conversion.

You can tell CODEF to call `SetField` directly without testing if the regenerated data member object is of the right type by marking the data member as `[ManualSet]`. By itself this isn't very useful, but it is when

```
public interface ITextualizedSurrogate
{
  string Textualize( object aO );
      object Detextualize ( string aTextual );
}
public class Color_TextualizedSurrogate :
 ITextualizedSurrogate
{
  public string Textualize( object aO )
  {
    return TextualizeColor((Color)aO);
  }
  public object Detextualize ( string aTextual )
  {
    return DetextualizeColor(aTextual);
  }
  static public string TextualizeColor(
     Color aColor )
  {
    uint lColorValue =  ( (uint)aColor.A << 24 )
                       +( (uint)aColor.B << 16 )
                       +( (uint)aColor.G << 8  )
                       +( (uint)aColor.R );
    return XmlConvert.ToString(lColorValue);
  }
  static public Color DetextualizeColor (
     string aTextual )
  {
    uint lColorValue =
       XmlConvert.ToUInt32(aTextual);
    byte lA =
      (byte)(( lColorValue & 0xFF000000 ) >> 24 );
    byte lB =
      (byte)(( lColorValue & 0x00FF0000 ) >> 16 );
    byte lG =
      (byte)(( lColorValue & 0x0000FF00 ) >> 8  );
    byte lR =
      (byte)(( lColorValue & 0x000000FF ) );
    return Color.FromArgb(lA,lR,lG,lB);
  }
}
Textualizer.RegisterSurrogate(typeof(Color),
   new Color_TextualizedSurrogate() );
```

Listing 4

you also mark the data member as `[ManualGet]`. `ManualGet` tells CODEF to simply bypass itself and do not encode the data member value in any way (as a model for instance). Instead, CODEF calls the following special method:

```
[GetField] object GetField ( string aName,
                              object aValue )
```

and lets *you* encode the object that CML will see and recreate.

The attributes `[ManualSet]` and `[ManualGet]` can be shortcut if used together as simply `[Manual]`. Manual fields are useful for data members that just can't be serialized via its data members, or for third-party objects which can't be serialized by textualization (CML will just textualize unmodeled objects).

A last but still interesting CODEF feature is the fact that the `Fixup` method returns an object. This is necessary because CODEF/CML automatically set data members unless they are marked as `ManualSet`. The object returned by the `Fixup` method allows you to keep a data member automatic even if its type changed critically.

Consider the follow scenario:

> At some point in time you have a `Collection` class, with some complex structure, and lots of files saved with that in. But then, later on, you refactor the design and the `Collection` class is replaced by a `Group` class which is *totally different*. You just have to keep the `Collection` class around (a stripped down version actually) so that CML can regenerate objects of that type when they are found in CML files. But that's not sufficient by itself: Data members that used to be of type `Collection` are now of type `Group`, so you need a way to convert a `Collection` read from an old file into a `Group` before assigning it to the data member. We already saw a case of type change that was handled by implicit typing, but implicit typing applies to declared primitive types only... here you need an explicit conversion.
>
> Using .Net serialization you would solve this by explicitly converting a `Collection` object extracted from a `SerializationInfo` to a `Group` object right before the assignment. In CODEF, all you need to do is to add a `Fixup` method to the deprecated `Collection` class that converts itself to a `Group`. That's it.
>
> The advantage of this is that the conversion is in the `Collection` class itself, and is always called by CODEF right before setting any field that used to be a `Collection` but now is a `Group`. This way, you just can't forget to add the conversion in a particular parent class, as you could using the .Net framework.

Future directions

CODEF/CML was developed to solve a specific problem during the lifecycle of a real application. It had concrete goals and was constrained by fixed resources (time). There are a number of improvements and open issues that become evident when you look back at the whole thing.

One of them is that fact that CODEF uses reflection but only on `Described` types. The idea was to avoid overloading the system with too much reflection, but I wonder now if given the fact that descriptors and models are generated on-demand from the set of types that are requested to be saved, if it really is a big overload to simply reflect on every type so that everything becomes described and all objects modeled. In our Application that is not really a problem because our design space is almost completely proprietary. We use at most 3 or 4 third-party simple structs, period; everything else comes from our own code.

But in most applications that's a very unlikely case.

The `[Manual]` attribute and the `GetField`/`SetField` method used with it is intended to give you some support for third-party types that can't be simply textualized (encoded as a string). Again, that totally worked for us because we just don't use third party objects except a few, and those are so simple that they can be textualized without trouble, but a better approach would be that you can register some form of CODEF *agent* that allows you to manually create CODEF models (and maybe even descriptors although these contain special Reflection types that can only be obtained via reflection).

CML needs textualization to get to and from the ultimate text representation, but it uses it for other things too. One of them is to handle types unknown to CODEF, yet, if CODEF is extended as proposed above this use of textualization won't be needed anymore.

Another usage of textualization in CML is to force compactness: If you go back to the textualizer example, you'll notice that the `Pen` class could have been `Described` instead of `Textualizable`. True, but if you want some type, possibly with 2 or 3 data members, to end up in CML as a single string, you just have to do that, but this is really an abuse of the current design. A much better approach is to let you register with CML your own codec for a given type. This would be similar in essence to the CODEF agents proposed above but it would be responsible for creating the XML elements that end up in the CML file (or part of them since some XML parts are mandatory). ■

Fine Tuning for lexical_cast

Alexander Nasonov takes a look at Boost's lexical_cast and addresses a common user request: "make it go faster".

A few weeks ago I created a patch that reduces a size of executable when arrays of different sizes are passed to `lexical_cast`. I sent the patch to Kevlin Henney and to a maintainer of `lexical_cast` Terje Slettebo. To my surprise, nobody is actively maintaining `lexical_cast` anymore. Terje kindly asked me to take a maintainership and I did.

To those of you who have never used `boost::lexical_cast` before, this is a function that transforms a source value of one type into another type by converting the source value to a string and reading the result from that string. For example, if the first argument of a program is an integer, it can be obtained with `int value = lexical_cast<int>(argv[1])` inside the `main` function. For more information, refer to [Boost].

There were several requests from users and I, as the maintainer, should consider them. The first request that I recalled was to improve the poor performance of the `lexical_cast`. Inspection of the `boost/lexical_cast.hpp` file showed that an argument of `lexical_cast` is passed to an internal `std::stringstream` object and a return value is constructed by reading from that `std::stringstream` object. It is well known among C++ programmers that a construction of `std::stringstream` is slow. This affects performance significantly because the object is created to format one value every time the `lexical_cast` function is called.

In the general case, not much can be improved but specialized conversions can be optimized. One example is where this can be done is `lexical_cast<char>(boolean_value)`, which could be expanded to `'0' + boolean_value` if this combination of types was supported.

Another example is a conversion from an integral type to `std::string`. It could be as fast as

```
std::string s = itoa(n); // n is int
```

I wish I could add a specialization that handled this case but `itoa` is not a standard function. The `sprintf` function might be an option but it formats differently compared to `std::ostringstream` and it also might be painful to mix C and C++ locales.

Sounds like I should implement a function similar to `itoa`. No problem. Though it's always good to study the subject before coding.

Some might argue that coding activity should be driven by tests. Well, that's a great approach but even having an excellent test suite doesn't guarantee that your code is bug-free. One goal of code analysis is to explore new test cases. You will easily recognize them during the course of the article.

Actually, the `lexical_cast` is shipped with a test (see `libs/conversion/lexical_cast_test.cpp` in Boost tree). It will help not to make a silly mistake but is not enough for checking hand coded conversion from an integral type to `std::string` because the test relies on correctness of `std::stringstream` code.

Analysis of several itoa implementation

There are several open-source UNIX distributions available ([FreeBSD], [NetBSD] and [OpenSolaris]) that include a code of `itoa`-like functions. I should make it clear that I'm not going to copy code between projects. Even if both projects are open-source, there are might be some incompatibilities between licences that don't let you redistribute mixed code under one licence. I only analyzed algorithms and pitfalls of implementations.

Some implementations of `itoa` look similar to this code:

```
char* itoa(int n)
{
    static char buf[12]; // assume 32-bit int
    buf[11] = '\0';
    char *p = buf + 11;
    int negative = n < 0;

    if(negative)
        n = -n;

    do {
        *--p = '0' + n % 10;
        n /= 10;
    } while (n);

    if(negative)
        *--p = '-';

    return p;
}
```

This code is plainly wrong. More precisely, `-n` has an implementation-defined behaviour. On 2s-complement systems, a negation of `INT_MIN` overflows. On systems where integer overflow doesn't interrupt a program execution, a result of `INT_MIN` negation is `INT_MIN`.

This fact has an interesting consequence for the algorithm. The first argument of the `n % 10` expression is negative, therefore, a result of this expression is implementation-defined as per section 5.6 bullet 4 of the standard [1]:

> If both operands are nonnegative then the remainder is nonnegative; if not, the sign of the remainder is implementation-defined

On my FreeBSD-powered Pentium-M notebook, this function returns: `"-./,),(-*,("`. This is definitely not a number!

Alexander Nasonov has been programming in C++ for over 8 years. Recently his interests expanded to scripting languages, web technologies and project management. A few months ago he became an occasional blogger at http://nasonov.blogspot.com. Alexander can be contacted at alexander.nasonov@gmail.com.

serialization can be considered as the composition of two layered processes

So, I threw this version away and went to the next implementation:

```
char* itoa(int n)
{
    static char buf[12]; // assume 32-bit int
    buf[11] = '\0';
    char *p = buf + 11;
    unsigned int un = n < 0 ? -n : n;

    do {
        *--p = '0' + un % 10;
        un /= 10;
    } while (un);

    if(n < 0)
        *--p = '-';

    return p;
}
```

It still applies unary minus to `int` type but the result is immediately transformed into `unsigned int`. This makes a big difference compared to the previous code. According to [1] 4.7/2, a conversion of some value `v` to `unsigned` type is the least congruent to `v` modulo `2^N`, where `N` is a number of bits used to represent values of the target type.

In other words, `v` belongs to one equivalence class with `v + 2^N, v + 2^N + 2^N, v + 2^N + 2^N + 2^N` and so on. By common convention, values within `[0, 2^N)` are used to represent classes of equivalent values.

Negative numbers are outside of this range and therefore, `2^N` should be added to convert them to `unsigned` type[1]. For example, `n=INT_MIN` in 2s-complement representation is `-2^(N-1)`. This number is equivalent to `-2^(N-1) + 2^N = 2^(N-1)`. Therefore, `un` is equal to an absolute value of `n`.

To get rid of implementation-defined behaviour of the `-n` expression, the line:

```
unsigned int un = n < 0 ? -n : n;
```

can be replaced with:

```
unsigned int un = n;
if(n < 0)
    un = -un;
```

It has already been discussed that a conversion in the first line has well-defined behaviour. Negative number `n` is converted to `unsigned` value `n + 2^N`, other values become `unsigned` by trivial conversion.

The third line operates on `unsigned` type and therefore, obeys modulo `2^N` arithmetic as per 3.9.1/4 of [1]. This means that a value of `un` after assignment is equal to `-(n + 2^N) = -(n + 2^N) + 2^N = -n`. Since `n` is negative, `-n` is positive and we can conclude that `un` is equal to an absolute value of `n`.

A reader might wonder why we don't just write `unsigned int un = abs(n)`? The answer is the same. When an absolute value of `n` cannot be represented by `int`, a result of `abs` function is implementation-defined.

Let's move on to the next problems. Illustrated functions define `static char buf[12]`.

First problem is that it is assumed that `int`'s representation is not bigger than 32 bits. A size of `buf` should be correctly calculated for all possible representations:

```
static char buf[3 +
  std::numeric_limits<int>::digits10];
```

The number 3 is a sum of 1 byte for minus sign, 1 byte for trailing `'\0'` and 1 byte for a digit not counted by `digits10` constant.

A more generic formula for arbitrary integral type `T` is:

```
2 + std::numeric_limits<T>::is_signed +
std::numeric_limits<T>::digits10
```

The second problem with `buf` is that it is shared between invocations of this function. You cannot call `itoa` from two threads simultaneously because both calls would try to write at the same location – to the `buf` array.

There are several solutions. One can pass a pointer to a buffer of sufficient length to `itoa`:

```
void itoa(int n, char* buf);
```

It introduces one inconvenience, though. Original `itoa` is very handy for code like `s += itoa(n)`. This modification would definitely break it because prior to making a call, a buffer variable should be defined. For the very same reason calling `lexical_cast` in-place is better then defining `std::ostringstream` variable and performing formatting.

The idea is to return an array from `itoa`. Builtin C/C++ array cannot be copied, so it should be wrapped into `struct`:

```
struct itoa_result
{
    char elems[12]; // assume 32-bit int
};
itoa_result itoa(int n);
```

The technique of wrapping an array into a struct can be generalized in C++. We don't have to do this though because `boost::array`[2] is (quoting

1. This statement is not correct if the source type has more bits then N. For such conversions, `2^N` should be added repeatedly until the result is within `[0, 2^N)` range.
2. The class is under discussion for inclusion into the next version of the standard; see N1836, Draft Technical Report on C++ Library Extensions.

Although it's tempting to turn itoa into a function template, there is a good reason to leave it and add overloads for other integral types instead

Boost documentation) *STL compliant container wrapper for arrays of constant size*:

```
boost::array<char, 3 +
  std::numeric_limits<int>::digits10> itoa(int n);
```

A typical call of this function would look like this:

```
s += itoa(n).elems;
```

or, if you'd like to save a result in a variable that has a wider scope than that of the temporary return value of `itoa`, take this route:

```
char buf[sizeof itoa(n).elems];
std::strcpy(buf, itoa(n).elems);
```

It's interesting to note how `buf` is defined in the last example. There is no magic formula to calculate a capacity to hold the `int` value, just the `sizeof` applied to an array returned by `itoa`. It is much easier to remember this trick then to recall the formula.

```
struct lexical_streambuf : std::streambuf
{
    using std::basic_streambuf<CharT>::setg;
};

// Adapted version of lexical_cast
template<class Target, class Source>
Target lexical_cast(Source const& arg)
{
  // Pretend that Source has a limited string
  // representation
  char buf[ lexical_cast_src_length<Source>::value ];
  char* start = buf;

  // Output phase ...
  // char* finish points to past-the-end of
  // formatted value

  // Input phase
  lexical_streambuf sb;
  std::istream in(&sb);
  sb.setg(start, start, finish);
  Target result;
  if(!( in >> result) || in.get() !=
        std::char_traits<char>::eof())
    throw bad_lexical_cast(typeid(Source),
                           typeid(Target));
  return result;
}
```

Listing 1

Unless you count every byte in `buf`, an access to `elems` in `sizeof` expression can be omitted:

```
char buf[sizeof itoa(n)];
```

This might increase the size of `buf` slightly but this is not dangerous, while it saves typing 6 characters.

Other improvements that can be applied to `itoa` are consistency with C++ streams and support for other integral types.

The first improvement is printing digits in groups delimited by a thousands separator if the current locale has grouping. It is worth noting that this increases the size of the buffer. Since the thousands separator is a `char`, this change doesn't add more then `std::numeric_limits<int>::digits10` characters to the buffer.

Integral types other then `int` are not different from an implementation point of view. Although it's tempting to turn `itoa` into a function template, there is a good reason to leave it and add overloads for other integral types instead. The fact is that some types cannot be used in templates. Such things as `bit` fields and values of unnamed or local enumeration types are incompatible with a template machinery.

That's it. Definitely too much for the patch but we created a good ground for another library.

Tuning for lexical_cast

As shown in a previous section, string representations of some source types have a limited length that can be calculated at compile-time. For every such type, we define a specialization of the `lexical_cast_src_length` metafunction that returns that limit. For example:

```
template<> struct lexical_cast_src_length<bool>
    : boost::mpl::size_t<1> {};

template<> struct lexical_cast_src_length<int>
    : boost::mpl::size_t<2 +
    std::numeric_limits<int>::digits10> {};
```

A source value can be placed in a local buffer `buf` of appropriate length without the overhead of `std::stringstream` class, e.g. a lexical conversion of `bool` value could be a simple `buf[0] = '0' + value` expression, likewise `itoa` could format an `int` value and so on.

Fine, the output phase is completed without calling to C++ iostream facilities. Next is the input phase. Unlike the output phase, where a set of source types is limited to those that have a specialization of `lexical_cast_src_length`, a target type can be any `InputStreamable`[1] type. This means that we have to use C++ streams. The implementation is straightforward. First of all, a `streambuf` object is constructed, then its `get` area (`setg` member function) is set to point to a location of formatted source value as shown in Listing 1.

1. A more correct list of requirements is `InputStreamable`, `CopyConstructible` and `DefaultConstructible`

write down a list of Source and Target types that have an optimized implementation... all combinations should be tested one by one

To solve a performance degradation of the generic solution, specialized versions can be added. For example, if `Target` is `std::string`, the input phase can be implemented in a single line of code:

```
return std::string(start, finish);
```

Testing

It's great that the library already has a test. We can safely assume that combinations of `Source` and `Target` types that are dispatched to old implementations are tested thoroughly. Some combinations of types that trigger an execution of the new code are tested as well but we shouldn't rely on it.

The plan is to write down a list of `Source` and `Target` types that have an optimized implementation. Then all combinations should be tested one by one. Implementation details should be taken into account so that there are no surprises like `INT_MIN` being formatted incorrectly or weird behavior for some locales.

Completeness of the test can be verified by running the test under control of a coverage tool. If some line is not hit, it's a good hint for a new testcase. There is one small problem with specializations of `lexical_cast_src_length` metafunction, though. They don't have executable code at all. How do you check that some particular specialization is in use? That's easy, just define an empty `check_coverage` function in each specialization and call it from a point of use of the metafunction.

Benchmark

A conversion from `int` digit to `char` has been measured on FreeBSD 6.1. The test has been compiled with gcc 3.4.4 with -O2 optimization flag turned on.

```
#include <boost/lexical_cast.hpp>
int main()
{
  int volatile result = 0;
  for(int count = 1000000; count > 0; --count)
    result += boost::lexical_cast<char>(count % 10);
  return result;
}
```

The next table shows execution times of the test compiled with Boost 1.33.1 and with the patched `lexical_cast`. The last column is the performance influence of not constructing `std::locale` and `std::numpunct<char>` objects and not calling `std::numpunct<char>::grouping` function. This change is not included into the patch because it is not consistent with the standard output operator for `int`.

Boost 1.33.1	Patch	Ratio	Patch that ignores locales
2.012 sec	0.322 sec	6.24	0.083 sec

More common is a conversion to `std::string`. To measure performance of this conversion, the `for` body in the test program has been replaced with

```
result +=
  boost::lexical_cast<std::string>(count).size();
```

The results are:

Boost 1.33.1	Patch	Ratio	Patch that ignores locales
2.516 sec	0.844 sec	2.98	0.626 sec

Conclusion

After careful tuning, the `lexical_cast` can run several times faster. I hope that future versions of Boost will have an improved version. It will likely happen in version 1.35. ■

References

[Abrahams] C++ Template Metaprogramming by David Abrahams and Aleksey Gurtovoy, ISBN 0-321-22725-5
[Boost] The Boost Library, http://www.boost.org
[FreeBSD] The FreeBSD Project, http://www.freebsd.org
[NetBSD] The NetBSD Project, http://www.netbsd.org
[OpenSolaris] The OpenSolaris Project, http://www.opensolaris.org
[Standard] ISO/IEC 14882, Programming Languages – C++, Second Edition 2003-10-15

C# Generics – Beyond Containers of T

Steve Love takes a look at generics in C# v2.0, how to use them to simplify code and even remove dependencies.

One of the bigger differences between the latest version of C# and its predecessors is the addition of Generics. This facility is in fact provided and supported by the runtime (actually the Common Language Infrastructure, or CLI, specification), and exposed in the language of C# from version 2.0. Programmers already familiar with C++ templates or Java generics will immediately spot that they share a common base motivation with C# – the provision of type-safe generic containers of *things*. Syntactically they are similar, too, but it would have been grotesque for C# to choose an entirely different syntax just to be unique. Where these languages really diverge is in the implementation details. A discussion of these differences is beyond the scope of this article; this isn't a comparison between languages, rather an exploration of what C# generics offer. If you think that means you can do:

```
public class Stack< T >
{
}
```

and

```
private T Swap< T > ( T left, T right )
{
}
```

well, you're right, but that's not the whole story. This article is for people who already *know* you can do those things and are starting to wonder if they can do anything else.

Groundwork

At its most basic level, using generics is about writing less code. At a slightly higher level, it's about saying what you mean *in code*. Without generics, the contents of a list have to be manually cast from `object` to obtain the real thing. With a generic list container, you write *less code* because the casts are no longer required. A list container parameterised on the type of its contents also *says what it is* right on the tin. Speak less, say more.

The next thing generics give you is support from your compiler. If you try to get an integer out of a container of strings, the compiler tells you you're dumb. Defects like this, if left until the program runs, are much harder to find and fix. Generics provide stronger *type-safety*. Wise programmers depend on their compilers being smarter than they are, however smug the error messages are.

Note that all the above is about *using* generic types, delegates, methods, etc.. If you're writing generic code, you need to invest a bit in allowing the programmers *using* your code to write less code, say what they mean and expect the compiler to spot their misuses. Given that often the "other programmers" will be you later on, it's worth every penny.

The example (nope – it's not a stack)

Double Dispatch [Wiki] is a common pattern used to figure out the concrete type of an object when all you have is an interface. The Visitor Pattern is often used for this, but the intent of Double-Dispatch is different.

```
interface Shape
{
  void Accept( Handler handler );
}
class Circle : Shape
{
  public void Accept( Handler handler )
  {
    handler.Handle( this );
  }
}
class Square : Shape
{
  public void Accept( Handler handler )
  {
    handler.Handle( this );
  }
}
interface Handler
{
    void Receive( Shape s );
    void Handle( Circle c );
    void Handle( Square s );
}
class ShapeHandler : Handler
{
    public void Receive( Shape shape )
    {
        shape.Accept( this );
    }
    public void Handle( Circle c )
    {
        Console.WriteLine( "Circle" );
    }
    public void Handle( Square s )
    {
        Console.WriteLine( "Square" );
    }
}
```

Listing 1

A common example is for `Shape` objects, and a basic implementation is in Listing 1. For the purposes of the example, the `ShapeHandler` class has a `Receive` method to stand-in for client code. The important point is that the code needs to access the concrete `Shape` instances when it has only a `Shape` interface available.

What's worth paying attention to in this example is what would be required if a new member of the `Shape` hierarchy gets added: the `Handler` interface

Steve Love Steve is a contract developer currently working in C# targeting the .Net Compact Framework version 2. He can be contacted at steve.love@essennell.co.uk

the whole point of using double-dispatch is to vary the behaviour on the concrete type of the object we have in hand

grows a new method, and the `ShapeHandler` implementation grows a new method in sympathy. This isn't too onerous, really, but it's a nasty dependency. All of the classes shown must be in the same assembly, because to do otherwise would introduce a circular reference, which is not allowed. In practice this means that the `Handler` interface is redundant. We could just as easily use the concrete `ShapeHandler` object in the `Accept` methods.

We can use C# generics to remove some of the code duplication in this example, and more importantly, we can break the dependency between the `Handler` interface, and the concrete implementations of the `Shape` interface.

Speak less

There are two areas in the example for Listing 1 where code is duplicated. Firstly, each of the concrete implementations of the `Shape` interface require an `Accept` method which contains identical code. Further implementations of `Shape` require *exactly the same code*. The second duplication is also a source of the dependency already mentioned: each of the implementations of `Shape` is mentioned in both the `Handler` interface and its implementation.

Say More

The ideal situation we'd like to achieve is to have the `Shape` and `Handler` interfaces living in their own assemblies (or perhaps a single common interfaces assembly), with concrete `Shape`s in one separate assembly, and the concrete `ShapeHandler` in another. We also want to divorce the `Accept` method from the `Shape` interface: it is not a `Shape`'s responsibility to do double dispatch, it has other things to think about. We can in fact go one step further, and remove the duplication in the `Square` and `Circle` classes by moving the `Accept` method to a base class.

```
interface Handler< HandledType >
{
  void Receive( Shape s );
  void Handle( HandledType c );
}
class ShapeHandler : Handler< Circle >,
   Handler< Square >
{
  public void Receive( Shape s )
  {
    s.Accept( this );
  }
public void Handle( Circle c )
    {
        Console.WriteLine( "Circle handled" );
    }
    public void Handle( Square s )
    {
        Console.WriteLine( "Square handled" );
    }
}
```

Listing 2

The Handler

The first pass at breaking this up is to remove the dependency between the `Handler` interface and the concrete `Shape` classes. This is where C#'s generics can help.

Generic code is representable regardless of the types it operates on (although we'll talk about type constraints later). "Representable" usually means some combination of:

- An algorithm works irrespective of the types on which it operates – e.g. Swap or sort.
- The storage of objects is the same whatever their type – e.g. Containers of "T".

Our requirements don't really fit either of these: the whole point of using double-dispatch is to vary the behaviour on the concrete type of the object we have in hand, and we're not really storing objects. What we can say is that the *interface* for the `Handler` is the same for each type of `Shape` object we consider. By using a generic *interface*, our `Handler` interface and `ShapeHandler` implementation become as shown in Listing 2.

This code demonstrates a simple use of generics, where the `Handler` has a generic parameter indicating the handled type. This handled type is then used by the `Handle` method declaration. When the interface is implemented, a `Handle` method for each of the generic *arguments* (in this case `Circle` and `Square`) must be implemented. `Handler` is a straightforward generative interface. In particular, it allows `ShapeHandler` to explicitly declare what types of `Shape` it is a handler for – it is like a label on a tin.

Note that the `Handler` interface no longer has the dependencies upon `Circle` and `Square`. This means we can safely put the `Handler` interface into a separate assembly, and have broken that part of the dependency circle. It presents other challenges, though.

Unrest in Shape County

Recall from Listing 1 that the `Shape` interface looked like this:

```
interface Shape
  {
    void Accept( Handler handler );
  }
```

Now that the `Handler` interface has a generic parameter, this is no longer valid. The problem is, what would we put as the argument to it?

```
interface Shape
  {
    void Accept( Handler< ? > handler );
  }
```

We could use the `ShapeHandler` class directly, but that would again make the `Handler` interface redundant, and would re-introduce the circular

dependency between **Shape** and **ShapeHandler**. We could try the same trick as with **Handler** and make **Shape** a generic, but that would cause a different kind of difficulty. If **Shape** were generic, we would have to name its type parameter at every use, precluding uses like

```
List< Shape< ? > > shapes;
```

The answer is to use a generic method. Instead of making the whole of the **Shape** interface generic, we make only its **Accept** method generic. A first attempt might look like this:

```
void Accept< ShapeType >(
   Handler< ShapeType > handler )
```

This would be valid, parameterising **Handler** on the generic parameter of **Accept**, but we have now moved the problem of what to put as a generic argument back to **ShapeHandler**, in its **Receive** method:

```
public void Receive( Shape s )
  {
    s.Accept( this );
  }
```

This does not compile because the call to **Shape.Accept** is ambiguous: "**this**" could be interpreted as either a **Handler< Circle >** or a **Handler< Square >** in this context, and we cannot explicitly specify which to use, because at this point, we know only that we have a **Shape.** The best we can manage is to make the entire type of the handler a generic parameter.

```
void Accept< HandlerType >(
   HandlerType handler );
```

```
interface Shape
{
  void Accept< HandlerType >( HandlerType handler );
}
class Circle : Shape
{
  void Shape.Accept< HandlerType >(
     HandlerType handler )
    {
      ( ( Handler< Circle > )handler ).Handle(
         this );
    }
}
class Square : Shape
{
  void Shape.Accept< HandlerType >(
     HandlerType handler )
    {
      ( ( Handler< Square > )handler ).Handle(
         this );
    }
}
```

Listing 3

We then need some way to tell the compiler the actual contents of the handler in the **Accept** method, and generics can't help here: a runtime cast is required. Listing 3 shows the new **Shape** hierarchy with this in place. **ShapeHandler** and its interface remain as in Listing 2.

Note that it's not possible to cast an *unconstrained* generic parameter to just anything. A cast to **object** is always valid, and a cast to an interface type (which is used here) is also OK. Constraining the parameter can make further casts valid.

This code works because generics in C# are a *runtime* artifact. The **Accept** method in **Shape** is implicitly virtual, because it's declared in an interface type. Derived classes (**Circle** and **Square**) inherit this method, and if code calls **Accept** using the **Shape** reference, the over-ridden method in the concrete (runtime) type will get called. If you're already familiar with C++ templates, this is probably the biggest difference between them and C# Generics. In C++, templates are a purely *compile-time* construction, and it's not possible to have a virtual function template.

Mission accomplished

We have reached the point where the circular dependency between concrete **Shapes** and the **Handler** interface is gone. We can create a new project structure reflecting rôles and responsibilities, with each different assembly having a specific rôle:

HandlerInterface
– interface Handler< HandledType >
ShapeInterface *depends upon HandlerInterface*
– interface Shape
Shapes *depends upon ShapeInterface and HandlerInterface*
– class Circle
– class Square
Application *depends upon Shapes, ShapeInterface and HandlerInterface*
– class ShapeHandler

There are no circular references, and apart from the main application, all dependencies are on interface-only assemblies. If nothing else, this makes testing easier. There remain some opportunities for overtime, however: we

```
public class Dispatchable< Dispatched >
   : Dispatchable
{
  void Dispatchable.Accept< HandlerType >(
     HandlerType handler )
  {
    ( ( Handler< Dispatched > )handler ).Handle(
       this );
  }
}
public class Circle : Dispatchable< Circle >, Shape
{
}
```

Listing 4

Taking a leaf from a pattern normally associated with C++ ... each derived class parameterises its base class with itself.

can still improve on what we have. It's time to make good on the promise that we can reduce the amount of duplicated code, and in the process make the whole a bit more tidy and self-describing.

The interfaces of shapes

It has already been noted that the `Shape` interface has too many responsibilities. Not only is it a shape, with all that implies, it is a shape that can take part in the double-dispatch mechanism we are describing here. That really isn't part of a `Shape` interface.

This indicates the need for a new interface, which we'll call `Dispatchable`. This interface exposes the double-dispatch mechanism in the same way as the old `Shape` interface did – a generic `Accept` method.

We can still do better. The implementations of the `Accept` method in the concrete `Circle` and `Square` classes are identical. Each class now implements `Dispatchable` as well as `Shape`, but the `Accept` method remains as it was in Listing 3. We have already identified that one of the uses of generic code is when an algorithm can operate regardless of the types using it: this is exactly that example.

We can therefore make a base class common to all concrete shapes which implements the `Accept` method. Listing 4 shows a first try at what we want to achieve. Taking a leaf from a pattern normally associated with C++ - the Curiously Recurring Template Pattern [Coplien95] – each derived class parameterises its base class with itself.

The observant among you will notice that we now have two types with the same name – `Dispatchable`. C# allows types to be "overloaded" based on the number of generic parameters, and thus `Dispatchable` and `Dispatchable< Dispatched >` are distinct types.

Unfortunately this doesn't compile because the call to `Handle` is passing `this` –which is an instance of `Dispatchable< Dispatched >`. Recall the `Handler` interface: it is a generic interface, where the `Handle` method declaration uses the generic parameter. `ShapeHandler` itself implements the `Handle` method with the actual concrete type of the shape being used, and so expects either a `Circle` or a `Square`. The `Dispatched` parameter is a generic handle for exactly that – depending on which concrete class implementation is in play – so we might try this:

```
( ( Handler< Dispatched > handler )
   .Handle( ( Dispatched )this );
```

Alas this doesn't compile either. The difficulty is that the compiler doesn't know the type of `Dispatched` because it is resolved *at runtime*. The types to which we can cast "`this`" are strictly controlled: we can cast to `object`, to any direct or indirect base class or interface of `this`, or to the same type as `this` – which is usually redundant, but permitted. It cannot be cast to a type that is, as far as the compiler is concerned, an unrelated type.

Constraints

A short diversion into a comparison between C++ templates and C# generics might be illustrative of what is going on. In C++, a template parameter represents *any* type, and being a compile-time construct, the compiler knows when some facility is used that the supplied argument to the template parameter doesn't support. In C# generic parameters are not resolved to types until *runtime,* so during compilation, the parameter still represents any type, but only as an `object`, the ultimate base class of all types. Therefore only those operations supported by `object` are permitted by the compiler, without extra information.

The extra information is provided either by explicitly casting the reference (as we've already seen), or by using *constraints* on the types allowed as arguments to that parameter. Listing 5 shows this in a simple way. Note there are several different types of constraint; a fairly detailed discussion can be found at [MSDN].

The `where` clause on the `PersonComparer`.`Compare` method tells the compiler that the parameters are `Person` objects (or are derived from `Person`), and thus have a `Name` property. Without the constraint, this code won't compile because `object` has no `Name` property. In addition, if `PersonComparer`.`Compare` is called with arguments which are *not* `Person` objects, the compiler also issues an error – the constraint applies to the client code as well.

Tying it up

So finally we should be able to finish the generic `Dispatchable` base class. From Listing 4, remember we need to be able to cast `this` to a type suitable for the argument to `Handler< Dispatched >.Handle` which accepts a reference to either `Circle` or `Square`. Depending on context, the `Dispatchable` class is either a `Dispatchable< Circle >` or `Dispatchable< Square >`, with the concrete type substituted for the generic parameter `Dispatched`.

In order to cast `this` to `Dispatched`, `Dispatched` must be constrained to ensure it's actually *the same type as* `this`, so the class declaration becomes:

```
struct Person
{
  public string Name
  {
    get { return name; }
  }
  private string name;
}
class PersonComparer
{
  public static bool Compare< Sorted >(
     Sorted left, Sorted right )
    where Sorted : Person
  {
    return string.Compare( left.Name,
                           right.Name ) < 0;
  }
}
```

Listing 5

restricting the types permitted by user code allows the generic code more freedom in its implementation

```
public class Dispatchable< Dispatched > : Dispatchable
   where Dispatched : Dispatchable< Dispatched >
```

and the cast is now legal, allowing:

```
void Dispatchable.Accept< HandlerType >(
   HandlerType handler )
  {
    ( ( Handler< Dispatched > )handler ).Handle(
       ( Dispatched )this );
  }
```

With this in place, we can now add the `Dispatchable` interface, and perhaps the `Dispatchable< Dispatched >` base class, since it depends only on the `Handler< Dispatched >` interface, to the `Handler` assembly, and the `Shape` interface is completely independent of the double-dispatch mechanism.

The effort now required to add a new object to the `Shape` family is to add the class, and inherit from `Dispatchable`, add a new derivation to `ShapeHandler`, and add the overloaded `Accept` method. No copy-and-paste, and many mistakes in usage will be caught by the compiler. Another benefit of using this generic double-dispatch framework is that `ShapeHandler` need not be the only handler: there could be multiple implementations of the `Handler` interface, each handling a different set of `Shape` objects, with no duplication in the interface or implementations.

One step too far

Using a `Constraint` for the `Dispatched` parameter in the `Dispatchable< Dispatched >` class gave the compiler extra information about `Dispatched` which allowed us to use it as a cast target. The question now arises – could we apply a constraint to `HandlerType` in the `Accept` method and so remove the runtime cast?

Unfortunately, the answer is "no".

An interface class specifies a contract which must be adhered to by implementing classes. A constraint on a generic parameter forms part of the interface, and therefore the contract, so implementing classes must match it exactly. The `HandlerType` parameter would need a constraint in the `Dispatchable` interface:

```
interface Dispatchable
{
  void Accept< HandlerType >(
   HandlerType handler ) where HandlerType :
   Handler< ? >;
}
```

and we have no way of specifying what to use as the argument to `Handler` at that point.

In conclusion

There is more to life – and generic code – than containers and simple functions. Generics in C# improve the type-safety of code, which in turn gives us, as programmers, much greater confidence that our code, once compiled, is correct.

The double-dispatch example shows how generics allow a generative interface to remove hard-wired dependencies, how classes can make use of a common generic base class to reduce code duplication, and demonstrated the run-time nature of C# generics, using virtual generic methods. In addition, it shows the trade-off of using type constraints on generic parameters, where restricting the types permitted by user code allows the generic code more freedom in its implementation.

There is much more to generics in C# than can be covered here, including using constraints to improve code efficiency, specifying *naked type constraints*, to match whole families of types (mimicking Java wild-cards), and creating arbitrary objects at runtime.

Generics in C# are not perfect – nothing is! – and there are limitations which can seem to be entirely gratuitous, but they provide a very powerful and expressive framework for improving code by allowing it to speak less, and say more. ■

Acknowledgements

The idea for using C# generics to implement double-dispatch is the result of inspiration from two sources:

Anthony Williams' article "Message Handling Without Dependencies" [Williams06] discusses managing the dependency problem of double-dispatch in C++ using templates. This got me wondering whether anything like it was possible in C# using generics.

Jon Jagger has an Occasional Software Blog [Jagger], where he uses the Visitor Pattern to demonstrate some properties of generics in C#. This showed me that double-dispatch probably *was* possible using C# generics.

Thanks to Phil Bass, Nigel Dickens, Pete Goodliffe, Alan Griffiths and Jon Jagger for their helpful comments and insights.

References

[Wiki] http://en.wikipedia.org/wiki/Double_dispatch

[Coplien95] James O Coplien, "Curiously Recurring Template Pattern", C++ Report February 1995

[MSDN] MSDN, "Constraints on Type Parameters (C# Programming Guide), http://msdn2.microsoft.com/en-us/library/d5x73970.aspx

[Williams06] Anthony Williams, "Message Handling Without Dependencies", Dr Dobb's Journal May 2006, issue 384, available on-line at http://www.ddj.com/dept/cpp/184429055

[Jagger] Jon Jagger, "Less Code More Software (C# 2.0 – Visitor return type)", http://jonjagger.blogspot.com

The Kohonen Neural Network Library

Seweryn Habdank-Wojewódzki and Janusz Rybarski present a C++ library for users of Kohonen Neural Networks.

Introduction

This article is about the Kohonen Neural Network Library written to support the implementation of Kohonen Neural Networks. The Kohonen Neural Network Library (KNNL) is written in C++ and makes extensive use of C++ language features supporting functional and generative programming styles. The library is available from http://knnl.sourceforge.net under Open Source BSD Licence [KNNL].

Artificial Neural Networks (Neural Networks for short) are designed to mimic the biological structure of the natural neurons in the human brain. The structure of a single artificial neuron is similar to the biological one, too. One important feature is that an artificial network can be trained in much the same way as a biological one. We can easily imagine, that some mental abilities of animals can be duplicated in the computer system e.g. classification, generalisation and prediction. Artificial Neural Networks have a wide variety of applications. They are used, for example, in speech and image recognition, signal processing (filtering, compression), data mining and classification (e.g. classification of projects, analysis of customer demand, tracking the habits of clients), knowledge discovery (recognition of different diseases from various diagnoses), forecasting (e.g. prediction of the sale rate or object trajectories), adaptive control, risk management, data validation and diagnostics of complex systems.

Kohonen Neural Networks are not the only type of Artificial Neural Networks – there are three basic kinds: Kohonen Network, Feedforward and Recurrent (of which the Hopfield Network is a particularly interesting example) [Wiki]. There are no precise rules to determine which kind of network to use for a given task; rather it is a matter of deep analysis to find the right one – sometimes a mixture of them is used. More important is what kind of training procedure we can use. The Kohonen Neural Network is tuned using an unsupervised training algorithm, so for that reason it is also called a Self-Organizing Map (SOM).

Other types of neural networks are tuned using supervised training algorithms. Supervised training sometimes is called training with teacher, because we compare output values from network with with a value which we expected that should be in the output (we are teachers). The comparison generally is created by calculation difference of the output value and given one. After that weights are modified with respect to calculated difference. Unsupervised training has no teacher, there are no comparisons. Training algorithms and network are prepared to modify parameters without external decisions what is proper – network is self-trained (self-organized).

The authors of this article both obtained Masters degrees from the AGH University of Science and Technology in 2003 and are now working towards a PhD.

Seweryn Habdank-Wojewódzki Seweryn's programing interests are computational software especially for data mining and knowledge disovery. Seweryn can be contacted at habdank@megapolis.pl

Janusz Rybarski Janusz's programming interests are data mining, knowledge discover and decision support systems.

Some example uses

Kohonen Networks are used mainly for classification, compression, pattern recognition and diagnostics. Here we want to offer some simple but quite different examples where the Kohonen Network can be used.

Example: recognising shopper categories

Let's consider a shop with electronic shopping carts. The neural network can recognize all carts and what each customer has in his or her cart. After recognition, we can observe representative carts – in reality, what a typical Mr. "X" bought. But the network is more flexible than the statistics, because it can divide clients into groups: women will probably have different purchases to men, and young people will have different purchases to older ones, the same between single and married people, and so on. Everyone knows how valuable this information is. If our aim is to help the customer shop quickly, the manager will place products from a particular group close to each other; or if our target is to sell everything from stock, we can place all elements from a recognised group in different parts of the shop and in the paths between those products we can place things that are expensive or hard to sell.

Example: compression

The next example is how it can be used for compression. Let's assume that we can lose information, so if there is a huge amount of data we can use it to train a network, and from the network we identify the most important information about data. In compressed storage we can put just the most important information. For this usage it is important that the number of recognised details is chosen to determined the length of the output independently of the length of the input. This fact can be sometimes very useful.

Example: spellchecker

Let say that we need to have an intelligent spell-checker. We can train a Kohonen network using all properly written words. After training we can put there a word which is written incorrectly, and the network can say which word is the closest one, and what are further opportunities.

Example: classification

Classification is similar to pattern recognition, but the important work of the network is done after training. We train network using some data – this data can define classes (groups). After training we put new data as the input of the network, and we have answer to which group belongs new item.

Example: diagnosis

Diagnosis is a combination of the pattern recognition and classification. But the core of the system is to define diagnostic rules (see sidebar) in the system using Kohonen Network. The sketch of the diagnostic system is that we train the network more or less manually. We define internal parameters in neural network to fit to different let's say diseases. Parameters can be the same as are typically when we control health in hospital. Then we put to the network data of other patient, and we can

observe what is a hypothesis of his disease. Another possibility is that we describe the state of the patient then train the network with some data. After training we probe the network, and notice where are placed typical situations. Then we put data from a new patient and we can have hypothesis of his own state.

As an example of advance medicine application of the Kohonen network is used for image segmentation of the scans from the magnetic resonance [Reyes-Aldasoro]. The target in this example is to separate on the image cranium, cerebrum and other parts of human head.

Example: Travelling Salesman Problem

A Kohonen network has been used to solve the Travelling Salesman Problem (TSP) [Brocki]. In this work the author assumes that every neuron represents single place that salesman visits. And then dedicated definitions, of the tasks and proper training of the networks, lead him to solutions which are quite good. Similar work is presented on the web page www.patol.com/java/TSP/ [TSP].

Generally Kohonen Neural Network is known as one of the algorithms for Data Mining and Knowledge Discovery in Data. Knowledge Discovery is very difficult task, but it is possible to use Kohonen Network for this purpose. The most important thing is to properly define the structure of the rules, which can be stored in the system, and then our data have to fit to this construction, so training will try to find in the system some rules which are satisfies.

Mathematical fundamentals of Kohonen Neural Networks

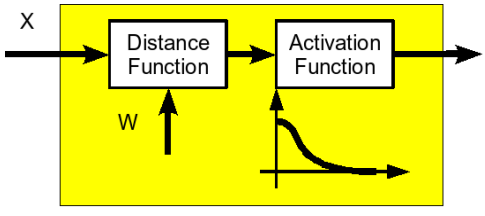
Kohonen Neural Networks are built from neurons which are placed in an organized structure. Thus:

KNN = container + neurons

The Kohonen neuron is defined as the composition of an activation function and a distance function, as shown in Figure 1.

The distance function (see sidebar "Distance") computes the distance between the input, *X*, and the neuron's internal weight, *W*.

In this paper, the input, *X,* and the weight, *W,* can be numbers, can be vectors on the plane, or on the 3D space, or in higher dimensional space. In that particular case the weights *W* belong to the same space as the input values *X*. We can calculate the distance between *X* and *W* using Euclidean formula (see sidebar "Euclidean formula").



Functional diagram of the neuron

Figure 1

In this article we will focus that particular distance function – it means on the Euclidean distance function. The activation function is the Gauss function (half of the full curve as shown in Figure 1.). The activation function controls the level of the output.

$$G_{(s,p)}(x) = \exp (- 0.5 (x/s)^p)$$

As with all neural networks, a Kohonen network is useless until it has been trained but, once trained, it is a very useful and intelligent mathematical object. It is *useful* because it can be applied in various branches of industry and science. It is *intelligent* because it can be trained without teacher – it can find some aspects in the data, that are not exposed.

Kohonen neural network must be trained. There are two main algorithms: one is called Winner Takes All (WTA), and the second is called Winner Takes Most (WTM). Details of these algorithms are described in section

Rules

A simple rule would be:

Patient has high level of the fat in the blood => Patient has coronary attack.

A more complex rule would take more objects, so we can add that patient is a smoker, and his live is nervous and so on. In addition to conditions we can add consequences – such as a cerebral haemorrhage.

So the new example is:

Patient has high level of the fat in the blood, patient is smoker => Patient has coronary attack, patient has cerebral haemorrhage.

The Kohonen Neural Network Library is fully equipped for examples like above – rules that can be described in numerical way as a vectors of numbers. It provides the implementation for some simple examples. For more complex examples the user may have to specialize templates for appropriate data structures, or add dedicated distance (maybe both). For details how to describe rules in the system we suggest to look in the books of Diagnostics Fuzzy Logic and Artificial Intelligence [Korbicz03][Yager94][Lin96].

Distance

In general case the are many kinds of sets where we can introduce distance. First very easy example is that we have a plane and we can measure distance between two points. Other possible example is, that we have a map of the surface of the earth and we have cities. Then we can assume that the distance is a minimal price which we have to pay to go from one city to another (we do not consider the mode of transport - it can vary, only the price). Other example of the distance can be considered in the dictionary. Let's consider such a situation. We have a set of words and we want to introduce a distance in this set. For example: the word "something" and its miskeyed versions: "somethinf", "sqmthing" and "smthng". Our intuition can say that "somethinf" is closer to correct version than word "smthng". So we can define distance in dictionary e.g. Levenshtein distance [Levenshtein]. The Kohonen network can work using such a distances if there will be a need.

Euclidean Formulas

Let's assume that:

$X = [x_1, x_2, ..., x_n]$

$W = [w_1, w_2, ..., w_n]$

Euclidean distance:

$d(X,W) = \text{sqrt} ((x_1 - w_1)^2 + ... + (x_n - w_n)^2)$

Sometimes is very useful to calculate weighted Euclidean distance, so let's assume that $P = [p_1, p_2, ..., p_n]$ are parameters:

$d_P(X,W) = \text{sqrt} (p_1 (x_1 - w_1)^2 + ... + p_n (xn - wn)^2)$

Training Process in this paper. Respectively to the training algorithm this structure can be either just a container (for WTA algorithm) for neurons or container equipped with distance function or topology (for WTM algorithm) – typically it is a two dimensional matrix. More generally, the network topology can be a graph or a multidimensional matrix. In medicine, especially, a 3D network topology is used for 3D image pattern recognition. In this paper the authors concentrate on the two dimensional case.

Code construction for the Kohonen Neural Network

In the library presented here we use `vector < vector < T > >` as a 2D matrix rather than the matrix template from the Boost uBLAS library [Boost]. This is because `boost::matrix` requires the value type to be `DefaultConstructible` and the neuron types in the library do not satisfy this requirement. We have suggested that the `boost::matrix` maintainers should add a suitable constructor to the matrix classes, so in the next version of the KNN library, the `boost::matrix` class will be used as a container to keep the network structure. This version of the library is restricted to 2D matrices. If need be, we will extend the structure of the network to 3D structures and graphs.

From a designer's point of view we can either construct a structure based on classes containing pointers to functions and functors (the latter being a class with declared and defined `operator()`) or build up template classes that store functors by value and provide an opportunity to parametrize the code by means of policy classes. We have decided to use templates, because many errors can be detected at compilation time, so the cost of testing and debugging the program is reduced to a minimum. Moreover template functors are quite small and such a construction is modular. Imagine how many different types of networks and training algorithms we are able to store in memory and use, and be sure that they will work, as they are fully constructed at compile time.

In the previous section the structure of the Kohonen neuron was described. Now we shall show how those neurons are represented in code. We use a class template, `Basic_neuron`, as shown in Listing 1. The template has two parameters: the type of the **distance function** and the type of the activation function. It is important that a neuron cannot be default constructed. This requirement is enforced by only providing a non-default constructor. So in the construction of a neuron there is no possibility of

```
template
<
  typename Activation_function_type,
  typename Binary_operation_type
>
class Basic_neuron
{
public:
  /** Weights type. */
  typedef typename Binary_operation_type
     ::value_type weights_type;
    /** Activation function type. */
  typedef Activation_function_type
     activation_function_type;
  /** Binary operation type. */
  typedef Binary_operation_type
     binary_operation_type;
  typedef typename Binary_operation_type
     ::value_type value_type;
  typedef typename Activation_function_type
     ::result_type result_type;
  /** Activation function functor. */
  Activation_function_type activation_function;
  /** Weak and generalized distance function. */
  Binary_operation_type binary_operation;
  /** Weights. */
  weights_type weights;
  Basic_neuron
  (
    const weights_type & weights_,
    const Activation_function_type &
       activation_function_,
    const Binary_operation_type & binary_operation_
  )
  : activation_function ( activation_function_ ),
  binary_operation ( binary_operation_ ),
  weights ( weights_ )
  {}
  const typename Activation_function_type
     ::result_type
  operator() ( const value_type & x ) const
  {
    // calculate output of the neuron as activation
    // function working on results from binary
    // operation on weights and value.
    // composition of activation and distance
    // function
    return activation_function ( binary_operation (
       weights, x ) );
  }

protected:
  Basic_neuron();
};
```

Listing 1

```
template
<
  typename Value_type,
  typename Scalar_type,
  typename Exponent_type
>
class Gauss_function
: public Basic_function < Value_type >,
public Basic_activation_function
<
  typename operators::Max_type < Scalar_type,
     Exponent_type >::type,
  Value_type,
  typename operators::Max_type
  <
  typename operators::Max_type
    <
      typename operators::Max_type < Scalar_type,
         Exponent_type >::type,
      Value_type
    >::type,
    double
  >::type
>
{
public:
  typedef Scalar_type scalar_type;
  typedef Exponent_type exponent_type;
  typedef Value_type value_type;

  typedef typename operators::Max_type
  <
    typename operators::Max_type
    <
      typename operators::Max_type < Scalar_type,
         Exponent_type >::type,
      Value_type
    >::type,
    double
  >::type result_type;
  /** Sigma coefficient in the function. */
  Scalar_type sigma;
  /** Exponential factor. */
  Exponent_type exponent;
  Gauss_function ( const Scalar_type & sigma_,
     const Exponent_type & exp_ )
  : sigma ( sigma_ ), exponent ( exp_ )
  {}
  const result_type operator() (
     const Value_type & value ) const
  {
    operators::power < result_type,
       Exponent_type > power_v;
```

Listing 2

making errors, because the neuron will work if the constructor parameter classes work.

We can easily see that the **`Basic_neuron`** class does not define anything more than a composition of functors similar to the **`compose_f_gx_hy_t`** in [Josuttis99]. The class stores functors by value, so as we mentioned earlier there are no pointers in that construction [Vandevoorde03].

The next thing to do is to design the code for the activation function. Activation function is a concept; we could easily use e.g. the Cauchy hat function or the Gauss hat function as an activation function. The Cauchy hat function is a generalisation of the Cauchy distribution function. The integral of the Cauchy distribution function over its whole domain is 1; this is not true, in general, for the Cauchy hat function.

$$C_{(s,p)}(x) = 1 / (1 + (x/s)^p)$$

The same is true for the Gauss hat function. An outline of the code for the Gauss hat functor is shown in Listing 2; the Cauchy hat functor is similar except for the formula that calculates the outputs.

We use a class template with three type parameters. **`Value_type`** is the type of the value that will be passed to the function. **`Scalar_type`** is the type of the scaling factor, and **`Exponent_type`** is the type of the exponent. All parameters are important, because generally **`Value_type`** and **`Scalar_type`** are some sort of floating point type like double or long double, and **`Exponent_type`** is used to choose the most appropriate power algorithm. This is important, because we won't always use the floating point power operation; sometimes we can use the faster and more accurate power function for integral exponents. So the library defines two specialisations of the power functor: one for integral exponents, and one based on the **`std::pow`** calculation for floating point exponents.

The library uses two coding tricks derived from modern C++ software design. The first one we call Grouping Classes. This technique is used to ensure that the class templates are only instantiated with the correct template parameters.

```
    // calculate result
    return
    (
      std::exp
      (
        - operators::inverse (
           static_cast < Scalar_type > ( 2 ) )
        * (power_v)
        (
          operators::inverse ( sigma ) * value,
             exponent
        )
      )
    );
  }
};
```

Listing 2 (cont'd)

In our library we have a lot of functors. They all define `operator()`. When we write code it would be easy to use the wrong functor as a parameter of a class template. We might try to solve this problem by putting related functors into separate namespaces, e.g. we might require that `foo::Foo` is only parametrized by functors from namespace `foo`. But this is just a hint to the programmer; it can not be checked by the compiler. There is also the problem of how to set up namespaces if there are some functors that could parametrize both class `bar::Bar` and `foo::Foo`. Multiple inheritance helps here. If we have a set of functors and some of them are only valid as parameters to `bar::Bar`, some are only valid as parameters to `foo::Foo`, and the rest will fit both templates, we could define two empty classes with name e.g. `Fit_to_Foo` and `Fit_to_Bar`, and all functors derive from them with respect to the represented traits. Sample code could be like Listing 3.

The `BOOST_STATIC_ASSERT` macro will generate a compiler error if `T` is not derived from `Fit_to_Foo`. So there is a compile-time check that the programmer has used a valid functor as a template parameter. Even though all the functor classes define `operator()`, they will not all fit our template. And the static assertion has no run-time overhead because it doesn't generate any code or data.

```
#include <boost/static_assert.hpp>
#include <boost/type_traits.hpp>

// helpful template
template<typename T, typename Group>
  struct is_in_group {
  enum {value = boost::is_base_of<Group,T>::value };
  };

// define groups
struct Fit_to_Foo {};
struct Fit_to_Bar {};

// set functors as a member of group
class Functor_A : public Fit_to_Foo { ... };
class Functor_B : public Fit_to_Bar { ... };
class Functor_C : public Fit_to_Bar,
   public Fit_to_Foo { ... };

// example template classes that check if parameter
// is in proper group
template < typename T >
class
Any_class_that_accepts_functors_only_from_Foo_group
{
  // and for example in constructor we can put:
  BOOST_STATIC_ASSERT(
     ( is_in_group < T, Fit_to_Foo >::value ) );
};
```

Listing 3

```
template < typename S, typename T >
struct Max_type_private
{};

template < typename T >
struct Max_type_private < T, T >
{
  typedef T type;
};

#define MAX_TYPE_3(T1,T2,T3) template <>\
struct Max_type_private < T1, T2 >\
{ typedef T3 type; };

MAX_TYPE_3(int,double,double)
MAX_TYPE_3(short,long,long)
MAX_TYPE_3(unsigned long,double,double)
// an example where we can explicitly use three
// different types  MAX_TYPE(std::complex < int >,
// double, std::complex < double >)
// macro below assumes that bigger type is the
// first one
// macro MAX_TYPE_2( argument_1_type,
// argument_2_type )
// simulate default result_type that result_type
// = argument_1_type

#define MAX_TYPE_2(T1,T2) template <>\
struct Max_type_private < T1, T2 >\
{ typedef T1 type; };

MAX_TYPE_2(double,int)
MAX_TYPE_2(long,short)
MAX_TYPE_2(double,unsigned char)
MAX_TYPE_2(double,unsigned long)

template
<
  typename T_1,
  typename T_2
>
class Max_type
{
private:
  typedef typename boost::
     remove_all_extents < T_1 >::type T_1_t;
  typedef typename boost::
     remove_all_extents < T_2 >::type T_2_t;

public:
  typedef typename Max_type_private < T_1_t,
     T_2_t >::type type;
};
```

Listing 4

```
template
<
  typename Value_type
>
class Euclidean_distance_function
: public Basic_weak_distance_function
<
  Value_type,
  Value_type
>
{
public:
  const typename Value_type::value_type operator()
  (
    const Value_type & x,
    const Value_type & y
    ) const
  {
  return
    (
      euclidean_distance_square
      (
        x.begin(),
        x.end(),
        y.begin(),
        static_cast < const typename Value_type
           ::value_type & > ( 0 )
      )
    );
  }
private:
  typedef typename Value_type
     ::value_type inner_type;
  const inner_type euclidean_distance_square
  (
    typename Value_type::const_iterator begin_1,
    typename Value_type::const_iterator end_1,
    typename Value_type::const_iterator begin_2,
    const inner_type & init
  ) const
  {
    return std::inner_product
    (
      begin_1, end_1, begin_2,
      static_cast < inner_type > ( init ),
      std::plus < inner_type >(),
      operators::compose_f_gxy_gxy
      (
        std::multiplies < inner_type >(),
        std::minus < inner_type >()
      )
    );
  }
};
```

Listing 5

In KNNL this technique is used for hat functions. The Gauss hat function will be used as the activation function and later as a helper function in the training algorithms. But this function is not valid for other purposes, so we derive **Gauss_function** from **Basic_function** and from **Basic_activation_function**. In template classes which will use them we could check which classes they are derived from using **boost::is_base_of**. If we use **BOOST_STATIC_ASSERT** or **BOOST_MPL_ASSERT_MSG** we can check if the hat function class is derived from the appropriate base, which guarantees that some minimal functionality is available and the class models the correct concept [Abrahams04]. If we don't do this and we use as a parameter any class that fits the type requirements, the template will compile, but the parameter (functor) may not meet the semantic requirements of the template. This

```
// type of the single data vector
// for example if we have tuples of the data like:
// {(1,2,3),(2,3,4),(3,4,5),(4,5,6),(5,6,7), ... }
// this data could describe position of pixel and
// it's colour
// V_d describes type of e.g. (1,2,3)
typedef vector < double > V_d;

// configure Gauss hat function
typedef neural_net::Gauss_function
  < double, double, int > G_a_f;
// Activation function for the neurons with some
// parameters
G_a_f g_a_f ( 2.0, 1 );

// prepare Euclidean distance function
typedef distance::Euclidean_distance_function
  < V_d > E_d_t;
E_d_t e_d;

// here Gauss activation function and Euclidean
// distance is chosen to build Kohonen_neuron
typedef neural_net::Basic_neuron
  < G_a_f, E_d_t > Kohonen_neuron;

// Kohonen_neuron is used for the constructing a
// network
typedef neural_net::Rectangular_container
  < Kohonen_neuron > Kohonen_network;
Kohonen_network kohonen_network;

// generate networks initialized by data
neural_net::generate_kohonen_network
//<
//  vector < V_d >,
//  Kohonen_network,
//  neural_net::Internal_randomize
//>
( R, C, g_a_f, e_d, data, kohonen_network, IR );
```

Listing 6

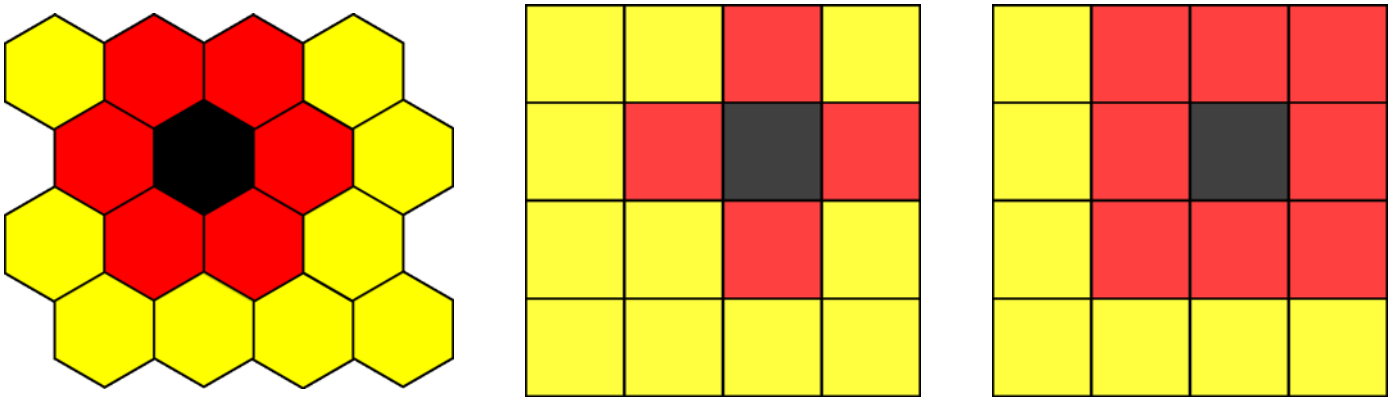


Figure 2

technique is an extension of the concept traits technique, because in concept traits we are sure only that the class contains the necessary methods; but if we ensure that it is derived from `Basic_activation_function` it is guaranteed to be a valid activation function. The authors call this technique Grouping Classes although there is no name in the literature for this coding style as there is for type traits or concept traits [Traits]. At this stage of the work the code does not include such checks; this will be added in the next version of the library after feedback from users. The rest of the `Gauss_function` class uses the `power` and `inverse` functors, which implement the common mathematical functions x^y and 1/x, to generate the proper results.

Another important feature of the library is the metafunction shown in Listing 4 (`Max_type < Scalar_type, Exponent_type >`).

The goal of this metafunction is to return a type that can represent any value on types `T_1` and `T_2`. This template is used to deduce the output type of functors from their input values types (presented here the number of the propositions of the usage of `MAX_TYPE_2` and `MAX_TYPE_3` is reduced – there are more examples in the library) .

The last part of the neuron is the distance function which is designed to work for any container. The Euclidean distance function, which is easy to understand, is presented in Listing 5 – the library also provides the weighted Euclidean distance function. To avoid problems (e.g. lost of performance) with the square root function, the functor actually returns the square of the Euclidean distance (we omit calculation of the square root function). If the user needs the real Euclidean distance from this functor he/she should either calculate the square root from this value, or set the proper exponent in activation function.

This function will be used as the `binary_operation` in neuron construction. The `Euclidean_distance_function` template is designed for any data stored in any container as defined in the C++ Standard Library. User-defined value types can be used with the Euclidean distance function provided the mathematical operations of addition, subtraction and multiplication are defined on them and the identity element under addition is formed by construction from the integer value 0.

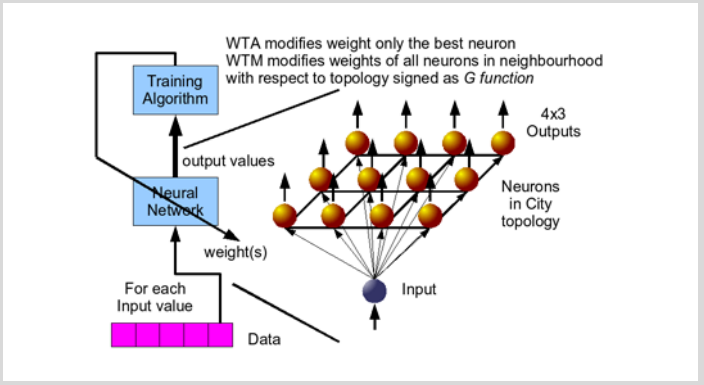


Figure 3

Listing 6 shows the complete code for a Kohonen network. Here, `R` is the number of rows in the network matrix, `C` is the number of columns, `g_a_f` is the chosen activation function, `e_d` is the chosen distance function, `data` is a data set of type `vector < V_d >`, `kohonen_network` is the network to be initialized and `IR` is a policy class for configuring the random number generator. (Random number generation is important for the implementation but it is too detailed for this short paper. Further explanation is included in the reference manual). Data are used just for setting up ranges in data space. In generation process ranges are used for randomly generated weights in network (uniformly in data space with respect to the ranges of the data). There are several algorithms for initialising the neuron weights. In this library the weights are assigned random values between the minimum and maximum values in the input data. So, for example, if the input data are:

```
{ (1,-2,-1), (-1,3,-2), (2,1,0), (0,0,1) }
```

the weights will be assigned random values in the range (-1,-2,-2) through (2,3,1). This completes the code to create and generate the weights of the Kohonen Neural Network. As we can see our desire for flexible code is fulfilled, even though we are not using pointers.

Training process

The training process is very important, because the untrained network has no idea why it exists. The more training passes are applied the greater the effect of the training. And after that the weights of the neurons can bring out many important properties of the data. For example, how the data can be generalized, how the data might be represented, what clusters (groups) exist in the data and so on.

There are two training algorithms for the Kohonen network. As we said in the beginning, the first is called Winner Takes All (WTA), and the second is Winner Takes Most (WTM).

WTA is very easy to understand and implement. When data is placed on the input to the neural network all the outputs are compared and the neuron with the highest response is chosen. This means that the weights of the winning neuron are quite close to the data. The winning neuron is the neuron with the highest response for the given input value. The weights for this neuron **only** are modified:

$$w(t+1) = w(t) + C(t)\,(x - w(t))$$

where: $w(t)$ is the weight(s) after training pass t, $w(t+1)$ is the modified weight(s), x is the input data, and C is a training coefficient which may vary as the training process proceeds.

Generally the behaviour of the WTA training algorithm is that it invests only in winners. Very radical method, isn't it? The WTM training process

```
template
<
  typename Network_type,
  typename Value_type,
  typename Data_iterator_type,
  typename Training_functional_type,
  typename Numeric_iterator_type
    = Linear_numeric_iterator <
      typename Training_functional_type
      ::iteration_type
    >
>
class Wta_training_algorithm
{
public:

  typedef typename
     Training_functional_type::iteration_type
     iteration_type;
  typedef Numeric_iterator_type
     numeric_iterator_type;
  typedef Network_type network_type;
  typedef Value_type value_type;
  typedef Data_iterator_type data_iterator_type;
  typedef Training_functional_type
     training_functional_type;
  /** Training functional. */
  Training_functional_type training_functional;
  Wta_training_algorithm
  (
    const Training_functional_type &
       training_functional_,
    Numeric_iterator_type numeric_iterator_ =
       linear_numeric_iterator()
  )
  : training_functional ( training_functional_ ),
  numeric_iterator ( numeric_iterator_ )
  {
    network = static_cast < Network_type* > ( 0 );
  }
  const int operator()
  (
    Data_iterator_type data_begin,
    Data_iterator_type data_end,
    Network_type * network_
  )
  {
    network = network_;
    // check if pointer is not null
    assert ( network != static_cast <
       Network_type * > ( 0 ) );
    // for each data train neural network
    std::for_each
    (
      data_begin, data_end,
      boost::bind
      (
        & Wta_training_algorithm
        <
          Network_type,
          Value_type,
          Data_iterator_type,
          Training_functional_type,
          Numeric_iterator_type
        >::train,
        this,
        _1
      )
    );
    return 0;
  }
```

Listing 7

```
protected:

  /** Pointer to the network. */
  Network_type * network;

  iteration_type iteration;

  Numeric_iterator_type numeric_iterator;

  void train ( const Value_type & value )
  {
    size_t index_1 = 0;
    size_t index_2 = 0;

    typename Network_type::value_type
       ::result_type tmp_result;
    // reset max_result
    typename Network_type::value_type
       ::result_type max_result
      = std::numeric_limits
      <
        typename Network_type
           ::value_type::result_type
      >::min();

    typename Network_type::row_iterator r_iter;
    typename Network_type::column_iterator c_iter;

    // set ranges for iteration procedure
    size_t r_counter = 0;
    size_t c_counter = 0;

    for ( r_iter = network->objects.begin();
      r_iter != network->objects.end();
      ++r_iter
    )
    {
      for ( c_iter = r_iter->begin();
        c_iter != r_iter->end();
        ++c_iter
      )
      {
        tmp_result = ( *c_iter ) ( value );
        if ( tmp_result > max_result )
        {
          index_1 = r_counter;
          index_2 = c_counter;
          max_result = tmp_result;
        }
        ++c_counter;
      }
      ++r_counter;
      c_counter = 0;
    }

   r_iter = network->objects.begin();
    std::advance ( r_iter, index_1 );
    c_iter = r_iter->begin();
    std::advance ( c_iter, index_2 );

    // train the winning neuron
    (training_functional)
    (
      c_iter->weights,
      value,
      this->iteration
    );
    // increase iteration
    ++numeric_iterator;
    iteration = numeric_iterator();
  }
};
```

Listing 7 (cont'd)

```
template
<
  typename Value_type,
  typename Parameters_type,
  typename Iteration_type
>
class Wta_proportional_training_functional
: public Basic_wta_training_functional
<
  Value_type,
  Parameters_type
>
{
public:
  typedef Iteration_type iteration_type;

  /** Shifting parameter for linear function */
  Parameters_type parameter_0;

  /** Scaling parameter for linear function */
  Parameters_type parameter_1;

  Wta_proportional_training_functional
  (
    const Parameters_type & parameter_0_,
    const Parameters_type & parameter_1_
  )
  : parameter_0 ( parameter_0_ ),
    parameter_1 ( parameter_1_ )
  {}

  Value_type & operator()
  (
    Value_type & weight,
    const Value_type & value,
    const iteration_type & s
  )
  {
    using namespace operators;
    return
    (
      weight
        = weight
        + ( parameter_0 + parameter_1 * s )
        * ( value - weight )
    );
  }
};
```

Listing 8

is more complicated. As in WTA we look for the best neuron, but we train all neurons in neighbourhood.

$$w_M(t+1) = w_M(t) + G(w(t), x, N, M, t)(x - w_M(t))$$

where: $w(t)$ is the existing weight(s) of the trained neuron (not only the winner), $w(t+1)$ is the modified weight(s) of this neuron, x is the input data, and G is a training function, N is neuron which is mostly activated (the winner), M denotes trained neuron.

Function G depends on the relation between the winning neuron N and the trained neuron M, also between the weight w_M and the data x. The library provides some ready-made WTM algorithms, too. The WTM algorithm trains groups of neurons, as much as they fit to the data.

The 2D topologies supported by the library are: Hexagonal topology, City topology and Maximum topology. In these diagrams (Figure 2.) we can see that the same 4x4 matrix can have different topologies which define different shapes of neighbourhood (on figures are shown neighbourhoods of the radius equals 1). In the WTM algorithm the winner will be improved a lot, but the neighbours will be improved too – the effect falling off with

```
// data container like:
// {(1,2,3),(2,3,4),(3,4,5),(4,5,6),(5,6,7), ... }
typedef vector < V_d > V_v_d;

// construct WTA training functional
typedef
neural_net::Wta_proportional_training_functional
  < V_d, double, int >
  Wta_train_func;

// define proper functional and its parameters
Wta_train_func wta_train_func ( 0.2 , 0 );

// construct WTA training algorithm
typedef neural_net::Wta_training_algorithm
  < Kohonen_network, V_d, V_v_d::iterator,
    Wta_train_func >
    Learning_algorithm;

// define training algorithm
Learning_algorithm training_alg ( wta_train_func );

// set number of the training epochs
// one epoch it is training through all data
// stored in container
const int no_epoch = 30;

// print weights before training process
neural_net::print_network_weights ( cout,
kohonen_network );

// train network
for ( int i = 0; i < no_epoch; ++i )
{
  training_alg ( data.begin(), data.end(),
     &kohonen_network );
  // better results are reached if data are sorted
  // in each cycle of the training
  // in target procedure this functionality
  // could be prepared using
  // proper policy like Shuffle_policy
  std::random_shuffle ( data.begin(), data.end() );
}

// print weights after training process
neural_net::print_network_weights
  ( cout, kohonen_network );
```

Listing 9

distance in the neural network and some other parameters defined by the ***G*** function.

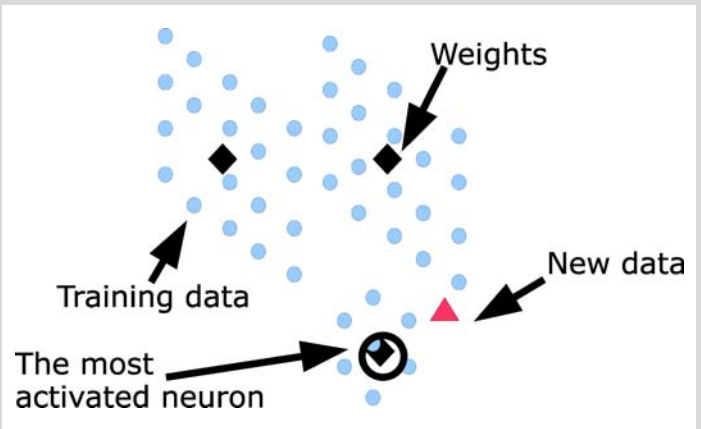


Figure 4

Implementation of the training algorithms

To be consistent with the design of the neuron, we decide to continue using class templates. A schematic of the training process is shown on Figure 3.

The code for the WTA algorithm is shown in Listing 7.

The function call operator calls the private `train` function for each item of data. The `train` function finds the winning neuron and applies the `training_functional` to it. Listing 8 shows the code for the `training_functional` class template:

So we could go deeper and deeper defining further class templates to build up the code for all the training procedures. Listing 9 presents part of the code that brings all the pieces together, creates the WTA training algorithm and, finally, trains the network generated earlier.

The code for the WTA method presented here is just an overview showing how the rest of the code is structured. `WTM` is much bigger and complicated, but also much more effective. **Both** algorithms **are implemented** in the library.

Usage of the network

Generally the structure e.g. topology, input and outputs of the trained network are shown in Figure 3. After training the network can be used in the same way as other neural networks. We can put the input value, and we can simply read outputs. The function below will return output value of the *(i,j)*-th neuron when we set the input value.

```
network ( i, j ) ( input_value );
```

or directly:

```
network.objects[i][j] ( input_value );
```

There are other useful functions for printing values and weights from network. The first one (from the `print_network.hpp` file) is:

```
print_network
```

The example of the usage is shown below. In the examples `kohonen_network` is a network and `*(data.begin())` is the first value from the set of training data, but there can be used other input values, too:

```
neural_net::print_network ( std::cout,
   kohonen_network, *(data.begin()) );
```

Other is for printing only weights:

```
print_network_weights
```

The example of the usage is shown below:

```
neural_net::print_network_weights ( std::cout,
   kohonen_network );
```

All details are available in the reference manual, both function are just loops through the network container. The first function calculates every output value. The second one take every weight of the neuron and print it.

Interpretation of the results

One interpretation of the network is that after training we are looking on values of the weights of all neurons. The weights can be treated as "centres" of the clusters.

If we have a network with 3 neurons, and we will train it with a set of data. We can noticed that weights of the neurons will concentrate in the centres of the data groups. We do not need define any group, Kohonen network will find as many group as is number of neurons what is shown on Figure 4. Thanks to the network and training process we have generalisation of the data. After training if we put in to the network input new data (similar to shown in Figure 4.) the network will activate neurons – it mean that we can read output values, and then we can look which neuron have the highest response – we call it "winner". We can say that new data belongs to the cluster (group) which is represented by this neuron and we can read the weights of this neuron.

This is important in all branches of the computation from business, through medicine, from signal processing e.g. image compression, to the weather prediction. Everywhere when we have a lot of data, and we do not really know what they are generally represent.

After training if we put input value to the network, we can observe responses from all neurons. Then we can say e.g. that the given input value belongs to the cluster e.g. "A" which is represented by the neuron with the highest response. Further we can say in "fuzzy" language that this particular value belongs the most to cluster e.g. "A", however it belongs to other clusters "B" and "D", but not to "C", where every cluster is represented by the single neuron.

Conclusions

In this paper we have presented the design and implementation of a Kohonen neural network library using class templates and a functional programming style. The KNNL library uses the Boost library. The training algorithms have been tested and found to produce quite good results. After implementation the library we observe that class templates are very useful for creating very flexible mathematical code. ■

Acknowledgements

Seweryn would like to thank Bronek Kozicki for help with the publication of this work, and Michal Rzechonek who always helps to improve my programming skills, and Phil Bass who gives many important remarks to the coding style and shape of this article. Moreover Seweryn would like to thanks Alan Griffiths for help in preparing final version of this paper more oriented to the programmers needs.

References

[Wiki] *Artificial Neural Networks*, http://en.wikipedia.org/wiki/Artificial_neural_network

[Boost] *Boost::uBlas*, http://www.boost.org/libs/numeric/ublas/doc/matrix.htm

[Josuttis99] *The C++ Standard Library: A Tutorial and Reference*, N. M. Josuttis, Addison Wesley Professional, 1999.

[Vandevoorde03] *C++ Templates: The Complete Guide*, D. Vandevoorde, N. M. Josuttis, Addison Wesley Professional, 2003.

[Abrahams04] *C++ Template Metaprogramming: Concepts, Tools, and Techniques from C++ Boost and Beyond*, D. Abrahams, A. Gurtovoy, Addison Wesley, 2004.

[Traits] *Concept Traits*, http://www.neoscientists.org/~tschwinger/boostdev/concept_traits/libs/concept_traits/doc/

[Reyes-Aldasoro] *Image Segmentation with Kohonen Neural Network Self-Organizing Map*, Contsantino Carlos Reyes-Aldasoro, http://www.cs.jhu.edu/~cis/cista/446/papers/SegmentationWithSOM.pdf

[Brocki] *Kohonen Self-Organizing Map for the Traveling Salesperson Problem*, Lukas Brocki, http://www.tokyolectures.pjwstk.edu.pl/files/lucas_tsp.pdf

[TSP] *Travelling Salesman Problem*, http://www.patol.com/java/TSP/

[KNNL] *Source code of the KNNL*, http://knnl.sourceforge.net

[Levenshtein] *Levenshtein Distance*, http://www.merriampark.com/ld.htm#WHATIS

[Korbicz03] *Fault Diagnosis: Models, Artificial Intelligence, Applications*, J. Korbicz, Springer-Verlag, 2003.

[Yager94] *Essential of Fuzzy Modelling and Control*, R. R. Yager, D. P. Filev, John Willey & Sons, 1994.

[Lin96] *Neural Fuzzy System – A Neuro-Fuzzy Synergism to Intelligent Systems*, C.-T. Lin, C. S. G. Lee, Prentice-Hall, 1996.

The Documentation Myth

Allan Kelly suggests that we don't invest more in documentation for a reason: that it isn't as valuable as we claim.

It appears that software developers, and their managers, have a love-hate relationship with documentation. On the one hand, we all seem to be able to agree that documentation is a good thing and we should write more of it. On the other hand, we are the first to acknowledge that we don't produce enough documentation. Frequently I talk to developers who say they don't have the time to do it, while managers often don't see the point "shouldn't you have written that before you started?"

Still, when a new recruit joins an existing project we're quite likely to sit them at a desk with a pile of documents and somehow expect that by reading them they will learn the system. Chances are the poor guy will just end up being bored and have difficulty keeping his eyes open by lunch.

Then, when you actually give him some code to work on he'll find it was never documented, or the document wasn't finished, or it's out of date, or it's just plain wrong. How often have we heard a new developer ask for documents, even though he knows they won't exist, One has to go through the ritual of asking - after all, this is a professional outfit isn't it?

Now at this point in the article, writers like me are expected to launch into exhortations on why documentation is important, why managers and developers should take it seriously, and maybe even recommend an amazing new tool which will help you do this. Surprisingly this tool is available from my company for just £199 (plus VAT).

However, I'm going to save my breath. To my mind it stands to reason that if documentation was as important as we say it is then we would take it seriously, we would do it, and it would be available. End of argument.

This is not to say documentation is useless. Particularly when we are planning and designing our system, the writing of papers, drawing of diagrams and discussions based around such documents are an essential way of analysing the problem and creating a shared understanding between team members.

Specification documents are somewhat more troublesome. They may be essential in order to start the project, indeed they may form part of a legal contract between customer and supplier, but we all know that specifications change. In fact, I'd argue that the process of writing the specification changes the specification. By documenting the problem we come to a better understand of the domain, this leads to new insights for all concerned and often leads us to view the original problem in a new way which also leads to new requirements.

OK, I'll accept that documentation has a role to play in helping new staff understand what is going on. However, diminishing returns are at work, the bigger the document the less information will be absorbed. The first few pages add more to understanding than the second hundred. Bigger documents delay the time when new staff actually do anything. Voluminous documentation can even discourage people from problem solving when they believe it is just a case of find the right page with a working solution (and how often have we each searched MSDN or Google for such a page?)

Even when there is documentation available we don't really trust it – and with good reason. We expect there to be a gap between what it says in the document and what is actually the case. The gap arises because things change over time, and because English and Java express different ideas.

In the worst case writing documentation becomes goal deferment – why complete the code when you can complete the document? I worked on one project that followed a rigorous ISO-9000 certified process. Despite very tight deadlines the documentation had to be kept up to date, as the documents grew the development work slowed, morale fell and the documents become more and more inaccurate. Even when they where proof-read by others there was no check to make sure the document actually described what was in the code. Increasingly the documents said what the managers wanted to hear and the programmers wrote the code they way wanted to. So what are we to do about the documentation myth? Which tool will solve my problem?

There is no technical fix for this problem. We need to rethink our view of documents. They are themselves a tool which allows us to discuss the problem domain, explore solutions and share information. However, they are a product of their environment. Inevitably they will address the issues at the time they are written, not the issues we find two years later. Documents will contain the information considered important by the people involved when they were writing. Other people, with different backgrounds and at different times, will consider different information important.

Documents only contain explicit information that we choose to express. Much of the important information on a project is actually tacit and held in people's heads without recognising it. Information is also held in the practices and processes adopted by a development team, and duplicating the processes won't necessarily replicate the information.

While our code base and deployed systems will always be the definitive source of information, we can supplement these sources if we value other forms of communication – particularly verbal and cultural. This means we need to look to our people, we need to encourage them to communicate and share what they know. New staff shouldn't ask "Where can I find the documentation?" but "Who can I ask?"

Postscript

The Documentation Myth was written a couple of years ago - January 2004 according to the file. For some reason I never got around to finishing it, I still stand by everything I've said: documentation can be useful, but it's probably not as useful as we often think it is. And the proof is: if it was that important we'd do more of it. The reason for dusting it off and publishing it now is down to Peter Sommerlad's presentation at ACCU conference: Only the Code Tells the Truth.

Peter suggested that at the end of the day the only definitive documentation of a running system is the code itself. Not the system documentation, not the specification, not the UML and not even the comments in the system.

Then he went further. He suggested that many of the practices we consider important in software development may be myths. For example, comments in the code, design before coding, test after coding and strongly typed languages, to name a few.

Some of these things were never true, we just thought they were a good idea at the time – well intentioned but not actually helpful. Others were useful once upon a time but things change and maybe they aren't so useful today – for example strongly typed languages. And others were just downright wrong, ISO-9000 for one.

I've written about this before in a way: it's the need to unlearn some things. Some ideas have passed their sell-by date and we need to let go. But this goes further, we need to constantly look at what we are doing and ask: does this add value? Does it contribute? These are hard questions to ask and the results can be even harder to accept but we have to do it if our profession is to move forward and not get caught in sub-optimal positions.

However, like Peter, I believe that the time has come for some Myth Busting. It's a new century, our young profession is entering middle age, the time has come to look at what passes for conventional wisdom and question it. So, do you know any Myths we should be exposing? ■

Allan Kelly Allan has many years' experience at the code-face and more recently in product and project management. He believes the future of software development looks very different to its past, and the key to high performing software teams is the ability to learn. He holds BSc and MBA degrees. Allan can be reached at allan@allankelly.net