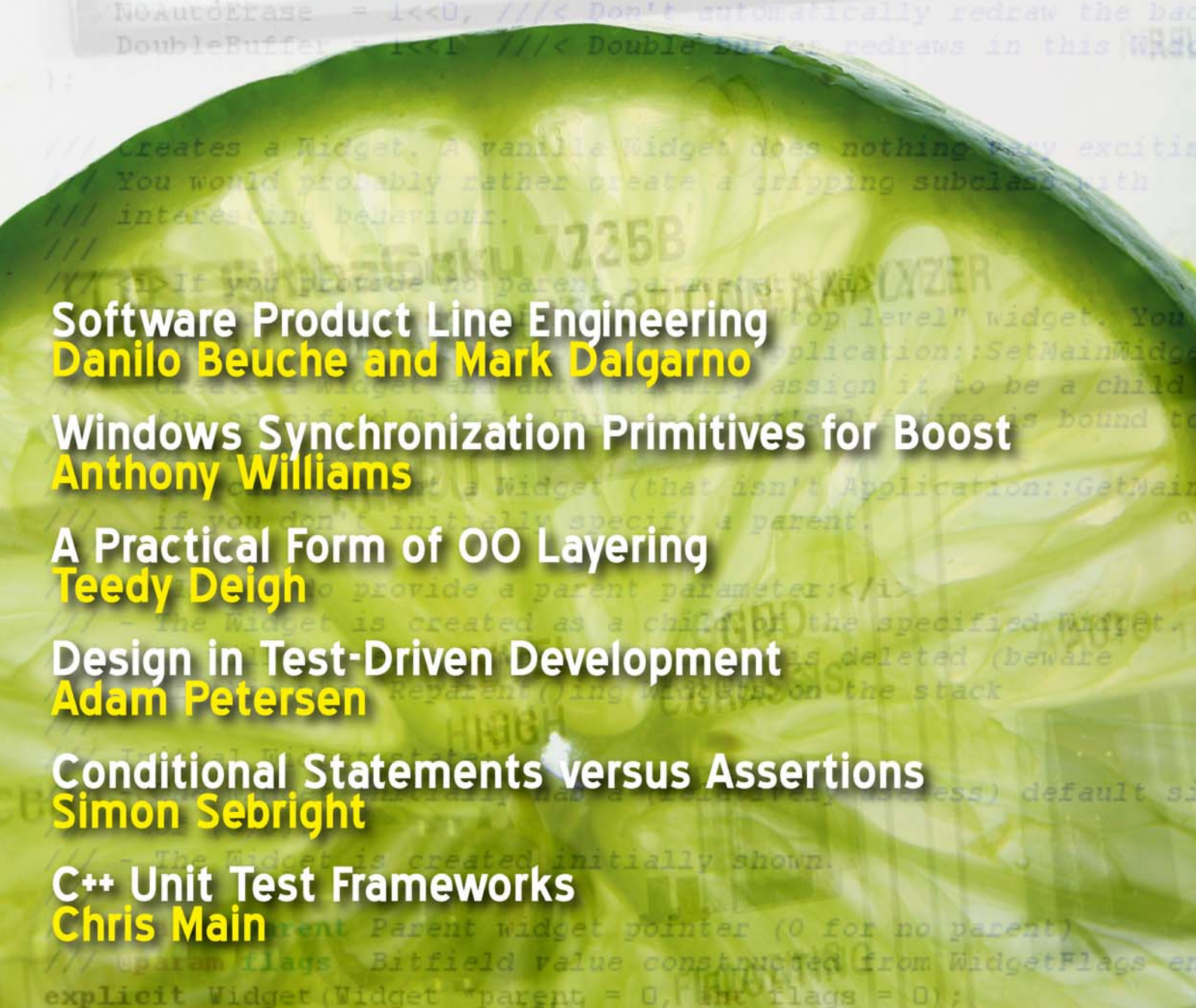
overload|78
April 2007
ISSN: 1354-3172
www.accu.org



OVERLOAD 78

April 2007

ISSN 1354-3172

Copy deadlines

All articles intended for publication in Overload 79 should be submitted to the editor by 1st May 2007 and for Overload 80 by 1st July 2007.

ACCU

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

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

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

Copyrights and Trade Marks

New Tricks for an Old Dog

Times change – as do development practices and tools. C++ was once the new kid on the block with all the latest gadgets. Agile methods have changed the focus away from language features and towards ways of working with the language. Can C++ continue to move with the times?

It is always hard to know what is happening in a field as broad as software development. Particularly as we so often are under pressure to deliver a new product, new version or just to fix the one we have delivered. This pressure takes its toll, and I know that I have, on occasion, 'surfaced' after months of intense focus to discover that there are new interesting things that everyone else knows all about.

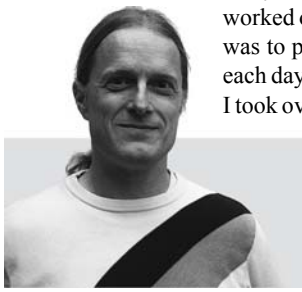
One way to ensure that these periods of several months do not turn into years is to ensure that I make it to the ACCU conference. If I've missed something, then someone will bring it to my attention there! (There are, naturally, other ways to achieve the same effect – the local meetings of the ACCU, the Extreme Tuesday Club and the SPA conference have all worked for me – and I'm sure that if I found time to attend XPDay that it would do the same. You will all have your own favourites – but I'm sure it isn't a coincidence that I meet the same people at many of these places.)

The point is that we all need to keep fresh and be on the lookout for the great new ideas and tools that will make it possible to deliver the next system we work on.

It has always been apparent that the tools are continually becoming capable of doing more for us. I remember re-reading *The Mythical Man Month* [Brooks 1974] some time ago and noticing that a task that it describes as requiring a dedicated person has all but disappeared. This was the process of incorporating updates to project documentation – which was done by merging replacement pages into a ring-binder. Managing documents in electronic form is definitely easier – especially when it comes to distributing and tracking changes.

I'm sure that everyone reading Overload will be aware of the popularity of 'agile methods'. What may not be so apparent is that many of the techniques advocated are reliant on today's technology. Honestly! Just think what 'continuous integration' relies on – the availability of a build server, automated access to a shared version control system and reliable and a way of publishing results. Some of us can still remember the days of 'sneaker net' – when the only way to get code from one machine to another was to copy it to a diskette (get one of the old timers to explain) and walk across the office with it.

Continuous integration



Shortly after the days of 'sneaker net', a team I worked on employed a contractor whose main task was to perform an integration build of the system each day. It took him all day every day. (Even after I took over and automated most of the steps so they could run overnight it still took around 8 hours. As the system grew this started taking longer – until it got to around 16 hours and steps were taken to speed it up).

Changes in technology make for dramatic changes in what is practical to achieve. We must not let the opportunity go by – because our value lies in the ability to deliver software. And delivering software is hard enough without denying ourselves the best tools available.

It is apparent that the agile software development movement has been effective in pushing effective tools. I've used CruiseControl to perform continuous integration on a number of projects now – it integrates nicely with several version control systems (I've used CVS, ClearCase and Subversion with it), it can build projects in different languages (I've used it for both Java and C++ projects), it runs on any platform that supports Java (I've used it on both Linux and Windows), it publishes notifications by email (I've used it in organisations using both Notes and ExchangeServer), and it presents its functionality via a nice web page. A great piece of software – it does one job and it does it well. There may be other products that do this job, but I'm not aware of one that is equally agnostic the environment it is used in.

Unit test frameworks

Unit testing underlies the agile practices of 'relentless refactoring' and 'test driven development'. Over the last decade or so a range of testing frameworks have become widely known. Although JUnit (for Java) is probably the best known it derives from the earlier SUnit (for SmallTalk).

C# has NUnit.

Python has PyUnit.

And C++ has boost.Test, CppUnit, CppUnitLite, Aeryn, CUTE, FRUCTOSE, CricketTest, CxxTest, and more.

While I know there are alternatives to JUnit in Java and PyUnit their use is unusual: I've not met a Java developer who is not familiar with this JUnit. The situation with C++ is somewhat different – although there are many frameworks, I've met many C++ developers who are not familiar with any of them! And many developers who prefer not to use the ones they are familiar with. This probably says something about both C++ and C++ developers. It definitely says something about test frameworks for C++: it isn't easy to produce one that is accepted by by the community.

Integration test frameworks

While I can assume that most of you know what unit tests and continuous integration are I feel I have to explain integration test frameworks. These

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

are tools that allow interactions that involve a significant portion (or all) of the system to be scripted in a way that allows the system to be tested. So for example, if the system being tested is a junk mail generator then it should be possible to supply a template letter, a mailing list and the expected list of letters. Running the test would supply the inputs to the system and validate the resulting addresses, salutations and letter contents.

I've dabbled with FitCpp (and the separate but related FitNesse) but, despite overcoming the technical barriers, cannot yet report a successful use in a real project. Others, working with Java have had mixed success applying Fit to this work.

Refactoring editors

One area where I've noticed significant progress is in the automation of editing tasks – after Fowler published his refactoring catalogue [Fowler1999], editing tools began competing against each other to provide automated support for these activities. Progress has been good in some modern languages[1] – in Java the Eclipse and IntelliJ IDEs initially ran out ahead of the field, and while I hear that NetBeans has now joined them all the Java developers I know are using Eclipse. C# also has refactoring tools – but progress has been slow in C++ tools.

What is that Edward? Oh yes, 'what is refactoring?' Yes, I should explain. It is a reorganisation of code that doesn't change the functionality, but makes it easier to work with. One example would be to change the signature of a function, and make the matching change to all the current invocations of the function. Another would be to extract a block of code from a function, place it in a new function and place an invocation of the new function where the code came from.

There are a number of reasons for the slow progress in C++: is is much more complex to parse, it has more dependencies of context, and the activities of the preprocessor causes significant difficulties for tools (as well as humans). Over the last couple of years I've been disappointed by the tools I've tried – but they are getting there at last. As always, as soon as someone proves that it can be done it will become commonplace. I hope that this will be the year that C++ gets its refactoring editors.

Functionality vs usefulness

The tools I've mentioned above – CruiseControl, JUnit, Fit, Eclipse – all focus on doing one clearly delimited job well. This is sadly very rare in the field of software – too many products make themselves less useful by tying in too many ancillary features. IBM once had (maybe still has) a great Java editor – VisualAgeJava, that was essentially useless because it included its own substitute for a file system and version control. These are useful features, but if you wanted to use text processing tools on your code, or your project required a common version control repository for Java and files that are not Java, then this approach got messy quickly.

I've seen code follow the same path at a lower level – classes (and functions) that try to do too much and fail to be useful. It is almost three years since I described the fate of a 'properties' class that I wrote as part of a client application. I think this was Overload 62 – but I don't have a copy to hand in order to check. Since I know that some of you won't have a copy either I'll repeat the story here.

This C++ class mirrored the Java class `java.util.Properties` – it allowed keys (of type `std::string`) to be used to store and retrieve values (also of type `std::string`). I suspect that classes with this functionality have been developed a number of times in different contexts. I tried to avoid this implementation being too specific to the context it was being used by limiting the functionality to that needed.

I left this class alone and went on with other things – until I was looking for something to do this same job and discovered that it was gone! It didn't take long to find what had happened: it had grown some additional functionality and changed its name. The new name and functionality related to the solution domain in which it was originally developed – it was now `calculation_options` which has the additional ability serialise and deserialise its contents into an application specific message format. Unfortunately, while the original 'properties' class would have solved my problem this new class had too much baggage to be useful to me (I didn't want to pull in the message format and associated protocols).

It is only a small example, but it illustrates a paradoxical point: the more functionality a piece of software acquires the less useful it becomes.

Is there life in the old dog?

At the conference this year I'm running a workshop [Griffiths2007] that I hope will help us determine how useful C++ is going to be in the future. As a language C++ is rich in functionality. But to exploit this effectively the developer need tools to match those available when working in other languages. My plan is to collate the experiences of those present and to identify the tools that we need for the third millenium.

See you there!

References

[Brooks 1974] Brooks, *The Mythical Man Month and Other Essays on Software Engineering*, Addison Wesley, ISBN 0201006502

[Fowler1999] Fowler, *Refactoring: Improving the Design of Existing Code*, Addison Wesley, ISBN 0201485672

[Griffiths2007] Griffiths, http://www.accu.org/index.php/conferences/accu_conference_2007/accu2007_sessions#Reviewing%20the%20C++%20toolbox

1. SmallTalk was probably the first language to support a refactoring browser – but is hardly mainstream

Letters to the Editor

Exceptional Design

Raoul Gough asks a question about Hubert Matthews' 'Exceptional Design' article, published in Overload 77 (February 2007):

> In the article 'Exceptional Design', Hubert Matthews shows a couple of ways of including error severity information in an exception. Try as I might, I cannot understand why he would even suggest this as a good idea.
>
> Surely the whole point of using exceptions is when non-local recovery is necessary, which means that the throw point cannot possibly know how severe the condition is to the system as a whole. Rather, it is only at the catch point where the severity is knowable. For example, whether a missing file is a warning, severe or fatal depends entirely on what the file is for – a missing config file might well be fatal, but a missing $HOME/.plan probably isn't! Or how about an out of memory error – surely that's a fatal error, right? Well, not necessarily – maybe you can discard some cached data and continue, so in fact it wasn't even really a warning.
>
> So I'm struggling to see what value there is in including severity information in an exception object (whether via its type or its members). I would even go so far as to suggest it leads to thinking about exceptions in the wrong way – rather than reporting the nature of an error, it is trying to pre-determine the 'correct' handling of that error. Maybe Hubert can explain his rationale for this a bit better?
>
> Regards,
>
> Raoul Gough.

Hubert replies:

> Exception handling acts in effect like a communications protocol between the point where the exception is thrown and the point where it is caught. Whenever there is a protocol there is coupling through shared knowledge. Think of something like an SMTP email server; both sides have to understand who says what and what the messages mean. In the case of exceptions, the two ends (throw and catch) have to agree on the message format (the exception object) and what information it contains. In some cases it is acceptable to throw an exception without any additional information. In others, a message alone suffices. In both these cases there is a one-way dependency from catch to throw; the throw knows nothing about the catch. In the case where severity information is included (usually where a multi-level recovery strategy is being used) then there is a two-way mutual dependency as the throw now needs to have knowledge about how to indicate what level of error this is. The alternative is to use some form of error type indicator and let the catch decide what to do, which then makes the dependency one way only again. Note that an error level is only advisory; it does not preclude the catch ignoring the level indicator and treating all exceptions as fatal for instance.
>
> This concept of exceptions as an error signalling protocol can be extended to dealing with retries, and alternative or compensating actions. Calls down towards the leaves of the call tree are attempts by the system to achieve some goal and exceptions are out-of-band signals in the reverse direction regarding problems.
>
> Hubert Matthewshubert@oxyware.com
> Software Consultanthttp://www.oxyware.com/

C++ Unit Testing Framework (CUTE)

Peter Sommerlad announces a new plugin for the CUTE framework:

> Hi all,
>
> In the past, you have shown interest in my C++ Unit Testing Framework, CUTE.
>
> I am sending you this mail to announce the pre-release of an Eclipse Plugin that makes C++ Unit Testing with CUTE almost as easy as Unit Testing for Java with JUnit in Eclipse. (Thanks to Emanuel Graf!)
>
> We created an eclipse plugin for Eclipse CDT (current release) that includes Cute and visualizes test results like the JUnit plugin for Java.
>
> You can install it from its update site (within Eclipse):
> http://ifs.hsr.ch/cute/updatesite/
>
> More information on my wiki:
>
> - http://wiki.hsr.ch/PeterSommerlad/wiki.cgi?CuTeDownload
> - http://wiki.hsr.ch/PeterSommerlad/wiki.cgi?CuTe
>
> NB: the current version of CUTE has some syntactic/cosmetic changes to my original Overload article. The wiki page reflects this already.
>
> More on it at ACCU conference.
>
> Pete Sommerlad

Software Product Line Engineering with Feature Models

Delivering on software product is hard, delivering a line of products is harder. Mark Dalgarno and Danilo Beuche share some ideas that help.

Although the term "Software Product Line Engineering" is becoming more widely known, there is still uncertainty among developers about how it would apply in their own development context. In this article we tackle this problem by describing the design and automated derivation of the product variants of a Software Product Line using an easy to understand, practical example.

One increasing trend in software development is the need to develop multiple, similar software products instead of just a single product. There are several reasons for this. Products that are being developed for the international market must be adapted for different legal or cultural environments, as well as for different languages, and so must provide adapted user interfaces. Because of cost and time constraints it is not possible for software developers to develop a new product from scratch for each new customer and so software re-use must be increased. These types of problems commonly occur in portal or embedded applications, e.g. vehicle control applications [Ste04] but are also seen in desktop applications. Software Product Line Engineering (SPLE) offers a solution to these increasingly challenging, problems [Cle01].

The basis of SPLE is the explicit modelling of what is common and what differs between product variants. Feature Models [Kan90] [Cza00] are frequently used for this. SPLE also includes the design and management of a variable software architecture and its constituent (software) components.

This article describes how this is done in practice, using the example of a Product Line of meteorological data systems. Using this example we will show how a Product Line is designed and how product variants can be derived automatically.

Software product lines

Before we introduce the example, we'll take a small detour into the basis of SPLE. The main difference from 'normal', one-of-a-kind software development, is a logical separation between the development of core, reusable software assets (the platform) and actual applications. During application development, platform software is selected and configured to meet the specific needs of the application.

The Product Line's commonalities and variabilities are described in the Problem Space. This reflects the desired range of applications ('product variants') in the Product Line (the 'domain') and their inter-dependencies. So, when producing a product variant, the application developer uses the problem space definition to describe the desired combination of problem variabilities to implement the product variant.

An associated Solution Space describes the constituent assets of the Product Line (the 'platform') and its relation to the problem space, i.e. rules for how elements of the platform are selected when certain values in the problem space are selected as part of a product variant. The four-part division resulting from the combination of the problem space and solution space with domain and application engineering is shown in Figure 1.

Several different options are available for modelling the information in these four quadrants. The problem space can be described e.g. with Feature Models, or with a Domain Specific Language (DSL). There are also a number of different options for modelling the solution space for example component libraries, DSL compilers, generative programs and also configuration files [Cza00].

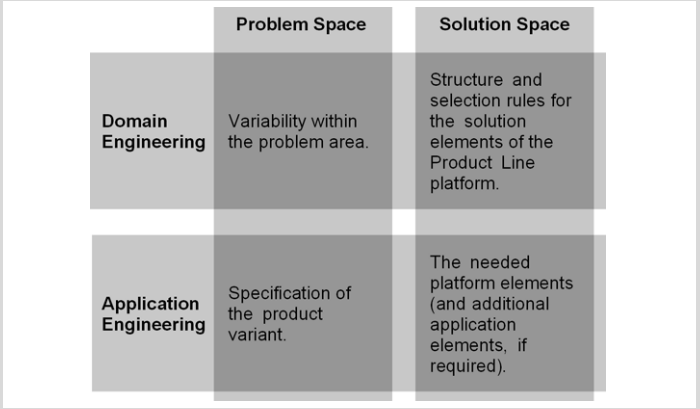


Figure 1

In the rest of this article we will consider each of these quadrants in turn, beginning with Domain Engineering activities. We'll first look at modelling the problem space – what is common to, and what differs between, the different product variants. Then we'll consider one possible approach for realising product variants in the solution space using C++ as an example. Finally we'll look at how Application Engineering is performed by using the problem and solution space models to create a product variant. In reality, this linear flow is rarely found in practice. Product Lines usually evolve continuously, even after the first product variants have been defined and delivered to customers.

Our example Product Line will contain different products for entry and display of meteorological data on a PC. An initial brainstorming session has led to a set of possible differences (variation points) between possible products: meteorological data can come from different sensors attached to the PC, fetched from appropriate Internet services or generated directly by the product for demonstration and test purposes. Data can be output directly from the application, distributed as HTML or XML through an

Mark Dalgarno's software industry experience spans over twenty years, primarily in product development and management. In 2004 he established Software Acumen as a specialist supplier of tools and services for organizations developing Software Product Lines. He blogs at 'The Variation Point' (http://blog.software-acumen.com/).

Danilo Beuche is the managing director of pure-systems GmbH, which specializes in services and tool development for the application of Product Line technologies in embedded software systems. He also works as a consultant in the area of Product Line development, mainly for clients from the automotive industry.

integrated Web server or regularly written to file on a fixed disk. The measurements to make can also vary: temperature, air pressure, wind velocity and humidity could all be of interest. Finally the units of measure could also vary (degrees Celsius vs. Fahrenheit, hPa vs. mmHg, m / s vs. Beaufort).

Modelling the problem space

We will now convert the informal, natural-language specification of variability noted above into a formal model in order to be able to process it. Specifically, we will use a Feature Model. Feature models are simple, hierarchical models that capture the commonality and variability of a Product Line. Each relevant characteristic of the problem space becomes a feature in the model. A definition of the term 'feature' is given in Definition 1.

Feature models have a tree structure, with features forming nodes of the tree. Feature variability is represented by the arcs and groupings of features. There are four different types of feature groups: 'mandatory', 'optional', 'alternative' and 'or'. When specifying which features are to be included in a variant the following rules apply:

If a parent feature is contained in a variant,

- all its mandatory child features must be also contained ('n from n'),
- any number of optional features can be included ('m from n, 0 < = m<=n'),
- exactly one feature must be selected from a group of alternative features ('1 from n'),
- at least one feature must be selected from a group of or features ('m from n, m>1').

Unfortunately, no single standard has yet been agreed for the graphical notation of feature models. However, in the literature, the graphical notation of the original Feature-Oriented Domain Analysis (FODA) method [Ste04] is common. However, this is representable with standard text tools and graph libraries only with difficulty. Therefore in this article a simplified notation has been used. Alternatives and groups of or features are represented with traverses between the matching features. In this representation both colour and box connector are used independently to indicate the type of group. Our notation is shown in Figure 2. Using this notation, our example feature model, with some modifications, is shown in Figure 3.

Each Feature Model has a root feature. Beneath this are three mandatory features – 'Measurements', 'Data Source' and 'Output Format'. Mandatory features will always be included in a product variant if their parent feature is included in the product variant. Mandatory features are not variable in the true sense, but serve to structure or document their parent feature in some way. Our example also has alternative features, e.g. 'External Sensors', 'Demo' and 'Internet' for data sources. All product variants must contain one and only one of these alternatives.

At this stage we can already see one advantage that feature modelling has over a natural-language representation – it removes ambiguities – e.g. whether an individual variant is able to process data from more than one source. When taking measurements any combination of measurements is meaningful and at least one measurement source is necessary for a sensible weather station, to model this we use a group of Or features. Usually simple optional features are used, such as the example of the Alarm. Further improvements can also be made by refining the model hierarchy. So the strict choice between Web Server output formats – HTML or XML – can be made explicit.

Definition 1: Features

Features are an abstract concept for describing commonalities and variabilities. What this means precisely needs to be decided for each Product Line.

A feature in this sense is a characteristic of a system relevant for some stakeholder. Depending on the interest of the stakeholders, a feature can be, for example, a requirement, a technical function or function group, or a non-functional (quality) characteristic.

Feature models also support transverse relationships, such as 'requires' and 'mutually exclusive', in order to model additional dependencies between features other than those already described. So, in the example model, a selection of the 'Freeze Point' alarm feature is only meaningful in connection with the temperature measurement capability. This can be modelled by a 'Freeze Point' requires 'Temperature' relationship (not shown in the figure). However, such relations should be used sparingly. The more transverse relations there are, the harder it is for a human user to visualize connections in the model.

When creating a feature model it can be difficult to decide exactly how problem space variabilities are to be represented in the model. In this case it is best to discuss this further with the customer. It is usually better to base these discussions around the feature model, since such models are easier for the customer to understand than textual documents and / or UML models. Formalising customer requirements in this way offers significant advantages later in Product Line development, since many architectural and implementation decisions can be made on the basis of the variabilities captured in the feature model. In the example, the use of the output format XML and HTML can be clarified. The model explicitly defines that the choice of output format is only relevant for Web Server, a format selection is not possible for File or Text output. However, in the context of a discussion of the feature model it could be decided that HTML is also desirable for the on-screen (Window) representation and could also be applicable for file storage.

This results in the modified feature model shown in Figure 4.

We have added 'Plaintext' to the existing features; this was implicitly assumed for output to the screen or to a file. We have modelled the mutual exclusion of XML and screen display ('Text') using a (transverse) relationship between these features (not shown).

The previous discussion describes the basic feature model approach commonly found in the literature, but a number of people have extended this basic approach. To complement the so-called hard relations between features ('requires' and 'conflicts') the weakened forms 'recommends' and 'discourages' have been added to many feature model dialects. A few tools also support the association of named attributes with features. This allows numeric values or enumerated values to be conveniently associated with features e.g. the wind force required to activate the storm alarm could be represented as a 'Threshold' attribute of the feature 'Storm Alert'.

An important and difficult issue in the creation of feature models is deciding which problem space features to represent. In the example model it is not possible to make a choice from the available hardware sensor types (e.g. use of a PR1003 or a PR2005 sensor for pressure). So, when specifying a variant, the user does not have direct influence on the selection of sensor types. These are determined when modelling the solution space. If the choice of different sensor types for measuring pressure is a major criterion for the customer / users, then appropriate options would have to be included in the feature model. This means that the features in the problem space are not a 1:1 illustration of the possibilities in the solution

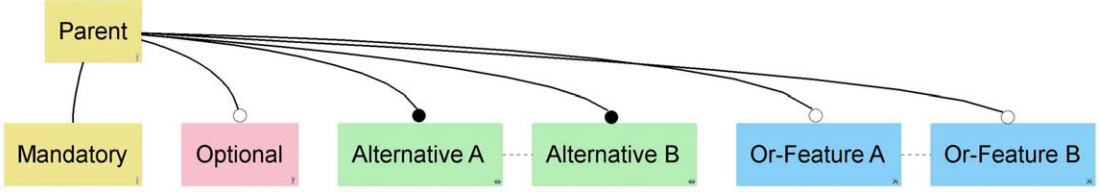


Figure 2

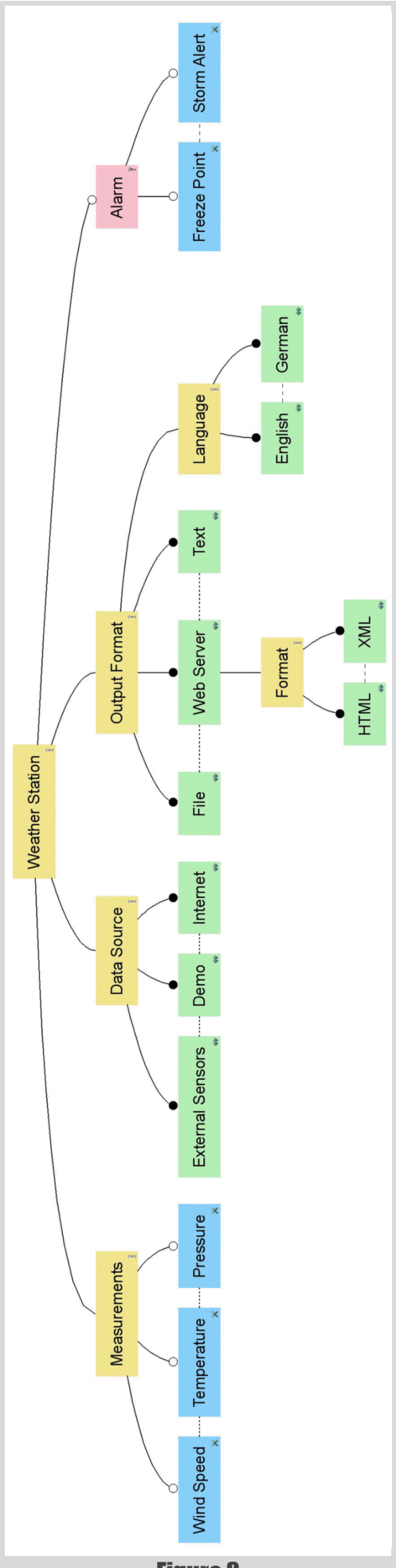


Figure 3

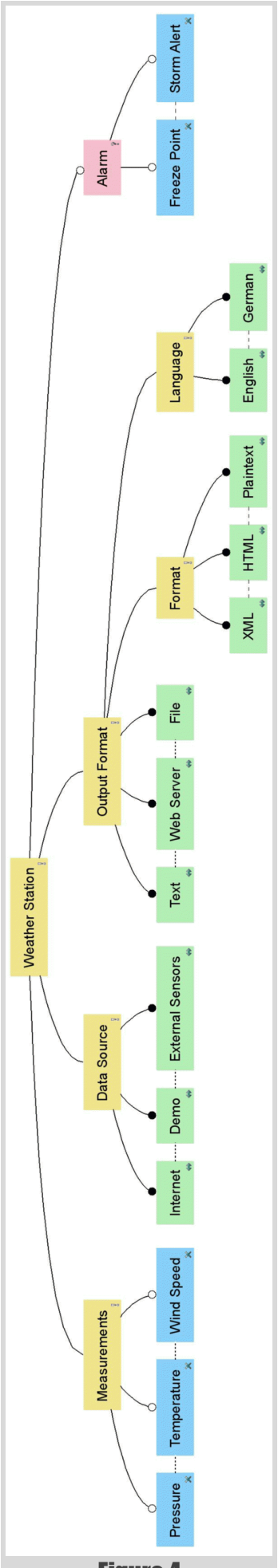


Figure 4

space, but only represent the (variable) characteristics relevant for the users of the Product Line. Feature models are a user-oriented (or marketing-oriented) representation of the problem space, not the solution space.

After creating the problem space model we can use it to perform some initial analysis. For example, we can now calculate the upper limit on the number of possible variants in our example Product Line. In this case we have 1,512 variants (the model in Figure 3 only has 612 variants). For such a small number of variants the listing of all possible variants can be meaningful. However, the number of variants is usually too high to make practical use of such an enumeration.

Modelling the solution space

In order to implement the solution space using a suitable variable architecture, we must take account of other factors beyond the variability model of the problem space. These include common characteristics of all variants of the problem space that are not modelled in the feature model, as well as other constraints that limit the solution space. These typically include the programming languages that can be used, the development environment and the application deployment environment(s).

Different factors affect the choice of mechanisms to be used for converting from variation points in the solution space. These include the available development tools, the required performance and the available (computing) resources, as well as time and money. For example, use of configuration files can reduce development time for a project, if users can administer their own configurations. In other cases, using preprocessor directives (`#ifdef`) for conditional compilation can be appropriate, e.g. if smaller program sizes are required.

There are many possibilities for implementation of the solution space. Very simple variant-specific model transformations can be made with model-driven software development (MDSD) tools by including information from feature models in the Model-Transformation process. [Voel05] gives an example using the openArchitectureware model transformer. Aspect-oriented programming (AOP) can also be used as a means for the efficient conversion of variabilities in the solution space. Product Lines can also be implemented naturally using 'classical' means such as procedural or object-oriented languages.

Designing a variable architecture

A Product Line architecture will only rarely result directly from the structure of the problem space model. The solution space which can be implemented should support the variability of the problem space, but won't necessarily be a 1:1 correspondence with the architecture. The mapping of variabilities can take place in various ways.

In the example Product Line we will use a simple object-oriented design concept implemented in C++ . A majority of the variability is then resolved at compile-time or link-time; runtime variability is only used if it is absolutely necessary. Such solutions are frequently used in practice, particularly in embedded systems.

The choice of which tools to use for automating the configuration and / or production of a variant plays a substantial role in the design and implementation of the solution space. The range of variability, the complexity of relations between problem space features and solution constituents, the number and frequency of variant production, the size and experience of the development team and many further factors play a role. In simple cases the variant can be

produced by hand, but automated tools in the form of Excel and / or small configuration scripts, and also model transformers, code generators or variant management systems will speed production.

One approach for modelling and mapping of the solution space variability is to use a separate Solution Model to model the solution space, to associate solution space elements with problem space features, and to support the automatic selection of solution space elements when constructing a product variant. This separation of concerns also has the advantage of allowing both models to evolve independently.

Solution models have a hierarchical structure, consisting of logical items of the solution architecture, e.g. components, classes and objects. These logical items can be augmented with information about 'real' solution elements such as source code files, in order to enable automatic production of a solution from a valid feature model configuration (more on this later). For each solution model element a rule is created to link it to the solution space. For example, the Web Server implementation component is only included if the Web Server feature has been selected from the problem space. To achieve this, a `hasFeature('Web Server')` rule is attached to the 'Web Server' component . Any item below 'Web Server' in the Solution model can only be included in the solution if the corresponding Web Server feature is selected.

In our example, an architectural variation point arises, among other possibilities, in the area of data output. Each output format can be implemented with an object of a format-specific output class. Thus in the case of HTML output, an object of type `HtmlOutput` is instantiated, and with XML output, an `XmlOutput` object. There would also be the possibility here of instantiating an appropriate object at runtime using a Strategy pattern. However, since the feature model designates only the use of alternative output formats, the variability can be resolved at compile-time and a suitable object can be instantiated using code generation for example.

In our example solution space a lookup in a text database is used to support multiple natural languages. The choice of which database to use is made at compile-time depending on the desired language. No difference in solution architectures can be detected between two variants that differ only in the target language. Here the variation point is embedded in the data level of the implementation.

In many cases managing variable solutions only at the architectural level is insufficient. As has already been mentioned above, we must also support variation points at the implementation level, i.e. in our case at the C++ source code level. This is necessary to support automated product derivation. The constituents of a solution on the implementation level, like source code files or configuration files which can be generated, can also be entered in the solution model and associated with selection rules.

So the existence of the Web Server component in a product variant is denoted using a `#define` preprocessor directive in a configuration Header file. In addition, an appropriate abstract variation point variable 'WEB SERVER' must first be created of the type `ps:variable` in the solution model. The value of this variable is determined by a Value attribute. In our case this value is always 1 if the variable is contained in the product variant. An item of type `ps:flagfile` can now be assigned to this abstract variable. This item also possesses attributes (file, flag), which are used during the transformation of the model into 'real' code. The meaning of the attributes is determined by the transformation selected in the generation step.

Separating the logical variation point from the solution makes it very simple to manage changes to the solution space. For example, if the same variation point requires an entry in a Makefile, this could be achieved with the definition of a further source element, of the type `ps:makefile`, below the variation point 'WEB SERVER'.

Deriving product variants

The solution model captures both the structure of the solution space with its variation points and the connection of solution and problem space. The direction of this connection is important since problem space models in most cases are much more stable than solution spaces; linking the solution space to the problem space is more meaningful than the selection of solution items by rules in the problem space. This also increases the potential for reuse, since problem space models can simply be combined with other (new, better, faster) solutions.

Now we have all the information needed to create an individual product variant. The first step is to determine a valid selection of characteristics from the feature model. In the case of some tools, the user is guided towards a valid and complete feature selection and for a large feature model this can reduce the time to create a complete and consistent selection by an order of magnitude. Once a valid selection is found, the specified feature list as well as the solution model serve as input for the production of a variant model. Then, as is described above, the rules of the individual model items are checked. Only items that have their rules satisfied are included in the finished solution.

Open issues in SPLE

So far we have highlighted some of the most common issues that will be encountered when working with Software Product Lines. In this section we highlight some additional issues.

Even with visualization support from specialist tools, the visual representation of very complex model structures is a not a completely solved problem. Larger feature models can have several hundred features and the solution space can have several thousands or more constituents. Thus it can be hard to understand the implications of modifications to these models just through use of model diagrams.

Issues around Product Line evolution are also very important. Evolution must be managed since changes that positively affect one or more variants could have a negative effect on other variants and these issues may only show up when variants are produced long after the changes have been implemented.

Finally, testing a Product Line also represents a significant challenge. Most Product Lines offer more potential variability than is in use at any one time, and testing all possible variants is usually impossible and in some cases a waste of time. Testing just those variants that are produced is already a difficult problem where there is a high number of variants. However, it is still necessary to co-ordinate testing with variant production in some way. One approach is to create test asset variants as one does for the (software) solution space variants – effectively creating a parallel test solution space that is driven from the Feature model. Reduction of test effort is still an open issue though for many Product Lines.

Closing remarks

We've shown above how the variability of the problem space of a Product Line can be described very simply using feature models. Automated production of solution variants is the logical next step, for which we have shown one example. We have also highlighted some of the approaches and issues that need to be considered when using Software Product Lines. These are covered in more depth in [Bos00], a standard work on Software Product Lines. The authors also welcome comments and questions on any aspect of Software Product Lines Engineering. ■

References and links

[Bos00] J. Bosch, *Design and Use of Software Architectures: Adopting and Evolving a Product Line Approach*, Addison-Wesley, 2000

[Cle01] P. Clements, L Northrop, *Software Product Lines: Practices & Patterns*, Addison-Wesley 2001 (see also www.sei.cmu.edu/productlines/framework.html

[Cza00] K. Czarnecki, U.W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000

[Kan90] K. Kang, et al., *Feature Oriented Domain Analysis (FODA) Feasibility Study*, Technical report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, 1990

[Ste04] M. Steger et al., 'Introducing PLA RK Bosch Gasoline of System: Experiences and Practices' in: *Proc. of the Software Product Line Conf*. 2004, S. 34-50

[Voel05] M. Voelter, *Variantenmanagement in the context of MDSD*, in: JavaSpektrum 5/05

A Perspective on Use of Conditional Statements versus Assertions

Simon Sebright offers us the benefit of his experience.

This article seeks to examine the use of two mechanisms to handle 'errors' in code. Two types of error are identified – runtime and design-time (see later for how these are defined) – both manifesting themselves when a program is run. Two mechanisms are identified for dealing with errors – program logic flow and assertions. I suggest that runtime errors should only be handled with logic flow (usually the `if` keyword) and that design-time errors should not. The latter should be handled by assertions or other design-time mechanisms.

I also put forward that if a function accepts a pointer to an object as an argument, then by default, the design decision should be that that pointer must not be `NULL`.

I know that some are going to have objections to what I say here, so I present this as my point of view, not industry best practice. Having said that, I have found in real projects that the technique put forward here helps to produce cleaner and more robust code, particularly not having silent failures, and not having to constantly check what parameters are valid, and which are not. I welcome any counter arguments, either as letters to the editor, or indeed another article.

First a proviso: my experience is based in the application development world, not safety-critical or embedded systems where continuity might be an issue (for example, an aircraft system would want to keep the plane flying in the right direction even if there was a bug in the toilet light module). The things I have to say may well apply there, with some modifications, but for the article, think of a desktop application, console application, or perhaps a web-based application. In these circumstances, I think it is better to fail fast – that is bail out altogether – than process things incorrectly.

This is a discussion relevant to C++ in particular. It may well be relevant to other languages. Indeed, C#, Java and VB.Net all have alias concepts which allow `NULL`, so the general principle will apply there.

Options for handling errors

We have at least two options for handling errors – program logic flow using a conditional statement (`if`, for example) and assertions. However, I usually favour design-time mechanisms to avoid error possibilities. This is briefly discussed later in the context of whether or not to allow `NULL` pointers into functions.

Conditional statements

These, in particular `if`, act as a fork for control flow. In the context of this discussion, I am talking about detecting error conditions, and doing something about it. That might be to silently ignore by returning early, throwing an exception, doing some extra work, missing out some work, diagnostic output, etc. The point is that it is the intention of the program to do something about it.

Assertions

Assertion refers to the general category of functions/macros which a programmer puts into code to catch specific 'errors' when a program runs, and alert the user (hopefully the developer acting as a user) in a drastic way. The results of this might be to terminate the program, or to pop up a nasty-looking error dialog offering the chance to debug or abort. The C++ standard only mentions assert in the context of the headers `<assert.h>` and `<cassert>`, which provides the `assert()` function on my system. Also defined in various implementations are `_ASSERT`, `ASSERT` and others. I'll deal with these collectively as 'assertions'.

They usually take an expression as an argument which can be converted to a Boolean value. In the case it evaluates to `false`, the drastic behaviour occurs.

Not orthogonal concepts

The crux of the matter is that a given condition should *either* be handled by a conditional statement *or* an assertion. The former should handle the error; the latter should bail out of the program (or do whatever drastic behaviour the particular assertion being used performs). Under no circumstances, should one see both in action for the same check.

But one does see that. Why? The answer usually comes in the form of 'belt and braces'. This is to me a pejorative term, referring to a style in which multiple attempts to 'handle errors' occur, just in case it is not handled properly by the first. The term comes from the real world scenario of the two ways to hold up your trousers. Of course in the real world, either your belt or your braces might wear out, break, or just come undone.

But in the software world, code does not rot once written (see sidebar – Bit Rot (or not?)). If your software belt works in a given environment, it will always work, ditto with the software braces. You have to decide which you really need, depending on the circumstances. That can be the hard part, and the reason why it is not done rigorously.

It boils down to distinguishing between two types of error, and using the appropriate mechanism for each.

Two types of error

Runtime/external errors

What I call runtime or external errors are things which are beyond the influence of your program. These are things it must take into account in order to behave sensibly in 'expected' circumstances. It may be the case that for your program to fulfil its purpose, certain external factors need to be in place, but as long as it cannot control them, it should be able to handle the absence of these factors in a purposeful way. That may be as simple as outputting a message to say that the program's installation is faulty, please reinstall, or it might prompt the user to go off and find something which it can't.

Simon Sebright Simon has been programming for 10 years, mainly in multi-tier C++ application development. Recently, he has been designing and developing web/database-based applications using C# and asp.net. He can be contacted at simonsebright@hotmail.com

Bit Rot (or not?)

Bit rot is a recognized concept, actually one that I hadn't come across until this article was reviewed! Wikipedia refers in the most part to decay of storage media, and that the suggestion that code itself can rot is facetious.

My position is that this is correct. Assuming that the bits of a program's code do not get corrupted, it will always execute the same way in response to the same input. Input, however, is not as simple as what the user types into the search box, or where a mouse click occurs in a UI. It must include memory state, available resources, and the like. Thus, there is the common phenomenon of seemingly-random errors in release builds of programs, where code is using memory which has (by incorrect coding) not been initialised. The input in this case changes if the memory address referred to happens to contain different values in different invocations of the program.

Particularly with regard to operating system changes, a program's environment can change. This may also lead to previously-unseen bugs appearing. In this case, we could include the environment in the term 'input'. For example, as mentioned elsewhere, if your code was passing a `NULL` pointer to `strlen()`, then from one version of Windows to the next, your code will suddenly start to crash. This is not bit rot, but a combination of incorrect coding and a change in input.

It should not crash, cause any inconsistencies in data, or do any other kind of harm to the system.

Let's look at some examples. It might be looking for a particular file, registry key, or database connection. These can fail to be there for a number of reasons. The installation may have failed, the network might be down, the user may have tinkered with something, there might not be enough memory available, or another program may have interfered. It doesn't matter what happened, just that you identify these possible scenarios. In programming, it doesn't matter how likely they are to occur, you have to program as if it will occur every time. It's all or nothing – you either check the condition or you don't. Whether the external error actually occurs when the program is run is obviously not knowable. You hope that it won't occur, but should not need to pray.

In addition, don't forget that just because you check something once, it still might go wrong later, a file could be deleted at any time, for example. This means that in principle, every use of that file should have a check first. However, especially considering concurrency, it becomes very difficult to guarantee absolutely that an external factor will be available, even if your code checks for it. Such a discussion is beyond the scope of the current article, though.

Design/internal errors

This is a totally different kettle of fish. These are things which are fundamentally about the program's design. When they go wrong, it means that you have made an assumption, an incorrect design decision, or just coded something incorrectly. It needs to be fixed. It will only be a matter of time, or particular data input, which will cause the thing to blow up.

Assertions often get omitted from release versions of code, for reasons of performance.

This kind of situation calls for an assertion in the code. This states that a given condition is expected to be true by design. It is a strong statement, one you have to make with confidence, because if it turns out not be true, things can go badly wrong with your program.

Assertions often get omitted from release versions of code, for reasons of performance. This is the source of much debate. It is arguable that leaving the assertions in the release build will give you better testing, as well as give good diagnostics should something go wrong in the field. Of course, a big hairy message box saying that the program has misbehaved is something you don't want your customer to see. But, more importantly, you don't want them to get misbehaving software in the first place. Certainly for some classes of application, it is better to bail out, than, for example to buy a lot of something you don't need, at the wrong price, or process only half the data, etc.

```
void MyFunc( Thing* p )
{
  ASSERT( p != NULL );
  if ( p == NULL )
    return;
  DoSomethingUseful( p );
}
```

or

```
void MyFunc( Thing* p )
{
  ASSERT( p != NULL );
  if ( p != NULL )
  {
    DoSomethingUseful( p );
  }
  DoABitMore();
}
```

Listing 1

Why not have them together?

So, we have a point in the code. We have been passed a pointer, and we want to access that object. Sometimes you will see one of the options in Listing 1.

Apart from causing the compiler to complain sometimes (because the `ASSERT` macro can expand to give a redundant check for `p`), I maintain that this is bad practice. At first sight, it might look very diligent. The careful programmer has covered all the options. The problem is that this lack of clarity blurs the issue of what we can expect from `p`, it blurs the issue of the design of the function, the whole class even.

Coming to this code from the outside, if you saw this in a function which you had cause to alter, what could you deduce about `p`? Suppose the person who checked it into source control is not available, or doesn't remember anymore.

You have to check in some code and you have to live with the consequences if it blows up. More than likely, you will opt to assume that `p` might be `NULL`. You might consider taking out the assertion in that case, but be careful, because we might really want to know if `p` is `NULL` here. You just don't know.

In one project, I spent time backtracking up the function-call trees to see where values were coming from. More often than not, it turned out that there was no way we could have got a `NULL` pointer, so I changed the code. It was quite common for certain members of the team to check collections of pointers for `NULL` before using them in a loop, a practice I tried to put a stop to. Simply don't put `NULL` pointers into the container in the first place!

In the ideal scenario, you would have 100% code coverage in your unit tests, so could simply run them and check. In the real world, you probably don't work in such an organised place.

Justification for putting them together

'Yes, but you just don't know what is going to happen in the future of that code. What if somebody does eventually pass a `NULL` pointer? Somebody might, and it might only happen in a release build where the assertions have been omitted.'

My answer here is that firstly, we put the assertion in as soon as the function is written, so we never have to shut the stable door after the horse has bolted. Secondly, this is just an argument propping up poor development practices. Are you saying that you can't write the code surrounding a function properly to honour its requirements?

Let's take an example. You have a UI showing objects, which the user can select. The user selects one and chooses to delete it. We end up at a call to `RemoveObject( Object* p )`. Code that ends up here with the possibility of passing a `NULL` pointer is going to have other problems, because it is not in control of things. The UI layers should be monitoring the selection, and only invoking actions which make sense. I have seen the result of the silent return early code. People don't realise when it happens, their broken code gets checked in, built upon, and creates a mess. Enter the onion.

The onion, a place for assertions

Let's take a step back, to the kitchen. If you cook much, you will probably be familiar with onions. They are comprised of a number of layers, each one shielding the one inside it. Often a bad onion is only bad in one or two layers, the rot stopping at the layer boundary.

Software can be like an onion, with the emphasis on the layering, rather than it making you cry (although that is often the case). Most software could be considered to be comprised of modules, there being many ways to define a module. However you choose to think of a module, think of it as an onion. The outer layer to that onion, or module, is the key one. It's where external factors call your code, where you can choose your policies for handling input. And that must include policies for handling input errors. See CodeCraft [Goodliffe] by Pete Goodliffe for a good discussion of different error handling strategies.

There are numerous policies you could follow. Doing nothing would be the most simplistic. Sometimes this is referred to as garbage in-garbage out, it being your own fault if you pass silly values.

Silently returning is also unfortunately common. Returning early with error codes is a viable option, requiring documentation of what the error codes mean.

In a language which supports it, throwing exceptions could be used for this purpose, as input errors ought to be exceptional.

Be aggressive with assertions inside the onion

Whatever your choice, once the control path gets inside the first layer, you can be justified in making assumptions about design choices. So, you can be very aggressive about checking and stating conditions with assertions.

That function above becomes cleaner:

```
void MyFunc( Thing* p )
{
  ASSERT( p != NULL );
  DoSomethingUseful( p );
}
```

That's better. You ought to do this from the outset of the project, though. Once code has been written and run with no assertions, you run the risk that come code elsewhere has gone through there and 'got away' with bad behaviour. Having a test suite somewhat mitigates this, as does taking a firm stance as soon as possible.

Documenting non-null pointers with the signature

Some say that references are the way to state non-nullability of an alias parameter. Personally, I disagree. I have only ever had bad experiences when null pointers were allowed. If you haven't got an object, don't call a function expecting one. You wouldn't try to call a member function on a `NULL` pointer, would you? Suffice it to say that assertions support the firm stance on parameters, if used once we are inside the outer layer of that 'module' we talked about.

As an aside, you could document your function's requirements on its pointer by making it take a smart pointer, whose policies dictate that nulls are not allowed [Alexandrescu]. However, personally, I don't think it makes sense to allow null pointers into a function. It is only there to alleviate the calling code from the responsibility. I'd favour a mechanism where the default is that pointer parameters are valid, and to document/police it explicitly if otherwise. Then everyone would know what to expect.

I remember a time when in the Windows API, the `strlen()` function was changed (implementation, that is). Previously, it used to accept a `NULL` parameter, and return `0`. Later versions didn't and crashed. Fair enough, I say. What's the point of measuring the length of a string object which doesn't exist? Of course, it might make some surrounding code have to do an extra check. I contend that by choosing a strategy of only using valid objects, the whole codebase can become cleaner.

The ambiguity of NULL pointers

Of course, certain things are going to be non-deterministic. For example, the user enters a search term, but none of your objects match that. So, your find function returns a `NULL` pointer? No, a better design would be to return an empty collection. The program design here is making the statement that the results of this find operation is a collection, and if you just specify a simple container like vector, then it can have nothing in it, no problem.

But what if you have a function which is only interested in one object? What might that be? One often finds interfaces of the kind: `FindFirstObject()`, `FindNextObject()`. This is quite common in the Win32 API, where the interface is a C interface. The failure to find any objects results in `FindFirstObject()` returning a `NULL` pointer. OK, but in C++ we have iterators and collections as standard, so I suggest that where possible, we construct interfaces to explicitly tell us when we have found nothing. An empty collection is one way, or return an iterator equal to `end()`. Note that in both these examples, we have an extra level of indirection introduced, which nicely solves the problem of duality of

Software can be like an onion, with the emphasis on layering, rather than making you cry

pointers (that duality being either valid object, or no object in one variable, a kind of union really). I note that pointers might be considered a type of iterator. However, idiomatic use of iterators is different to idiomatic use of raw pointers.

One of the most used functions which accepts this ambiguity is the `delete` operator. It is specified to be safe if given a `NULL` pointer, there being nothing done in that case. Whilst it means that code calling it does not have to make the `NULL` check first, I think it blurs the responsibility of the function, and is therefore a poor design. With a multitude of smart pointers to choose from, most code should not be calling `delete` directly anyway, and those places where it does should be limited to core functionality, so it would not hurt. In addition, consider that silently allowing `NULL` might be hiding errors where objects are not initialised properly.

Conclusion

There are two kinds of error in code – runtime and design-time. The former needs to be handled with logic flow in the code; the latter is a good candidate for assertions. Do not mix these two concepts; it should always be a clear choice.

Once inside the boundaries of a software module, making aggressive choices about the values you pass to functions can make code cleaner. It goes hand-in-hand with the practice of creating cleaner interfaces, which automatically remove ambiguity. ■

Acknowledgments

Thanks to Richard Blundell and Paul Thomas for comments and suggestions. Also, in my last article, I forgot to put the acknowledgements in. So, retrospective thanks to Phil Bass, Richard Blundell, Alan Griffiths and Anthony Williams.

Implementing Synchronization Primitives for Boost on Windows Platforms

Anthony Williams on the popular Boost library.

Introduction

The project to rewrite the Boost[1] thread library started in 2005, as part of the drive to have the whole of Boost covered by the Boost Software License[2]; Boost.Thread was under a different (though similar) license, and the original author could no longer be contacted. Though this issue has now been resolved, as the original author was contacted and agreed to the license change, efforts were underway to reimplement the library and they continue unabated. The reimplementation effort is driven in part due to the proposals for threading to be added to the next version of the C++ Standard, scheduled to be released in 2009.

This article describes the Windows implementations of mutexes for boost in CVS on the `thread_rewrite` branch[3] at the time of writing. As discussions continue, and alternative implementations are proposed, the final version used in Boost release 1.35 may differ from that described here.

What's a mutex

Mutex is short for 'mutual exclusion', and they are used to protect data – only one thread is permitted to 'own' the mutex at one time, so only one thread is permitted to access that data at once, the rest are excluded. The other important job performed by a mutex is to ensure that when a thread owns the mutex, the values it sees for the protected data are exactly the same as the values seen by the last thread to own the mutex, including any writes made by that thread. This is particularly important on systems with multiple CPUs, where the threads might be running on different CPUs, and the data will reside in the associated CPU caches for some indeterminate amount of time without specific instructions to ensure cache coherency, and visibility of the data between CPUs.

Native Windows synchronization objects

Windows provides an assortment of synchronization objects: `CRITICAL_SECTION`s, `Semaphore`s, `Event`s and `Mutex`es. Though all of them can be used to ensure that only one thread at a time has access to a given block of data, `CRITICAL_SECTION`s and `Mutex`es are the only ones designed for the purpose.

A `CRITICAL_SECTION` is a relatively light-weight object, and can only be used within a single process. Use of a `CRITICAL_SECTION` requires use of the `CRITICAL_SECTION`-specific API. In contrast, the other synchronization objects are kernel objects, and can be used to provide synchronization between processes, as well as within a single process. All these objects can be used with the generic `HANDLE`-based APIs, such as `WaitForSingleObject` and `WaitForMultipleObjectsEx`.

Anthony Williams is the Managing Director of Just Software Solutions Ltd. He has been programming professionally for over 10 years, having programmed as a hobby for a good many before that. He is a strong believer in the benefits of Test Driven Development, Refactoring, and being able to see the sea from his office.
He can be contacted at anthony@justsoftwaresolutions.co.uk

Memory visibility

Without explicit instructions to say otherwise, data loaded from main memory is generally kept in the CPU cache for as long as possible, since retrieval from main memory is a slow operation, and the CPU will stall if the data it needs is not in the cache. In addition, data may be stored in the CPU cache before being written to main memory. On a single-CPU, single-core system this is not a problem, but on systems with multiple CPUs, or CPUs with multiple cores, this can be a problem if data is to be shared, as stored data may not make it to main memory for other CPUs/cores to see, and they may not reload it from main memory anyway, if that memory location is stored in their cache.

With CPUs that can reorder instruction execution, reads and writes might not necessarily happen in the order anticipated, which therefore compounds the problem.

Consequently, in general, without special instructions, data written by one CPU may or may not be visible to other CPUs, and writes that appear sequential on one CPU may become visible on other CPUs in any order.

There are special CPU instructions which can force the CPU to synchronize the cache with main memory, and which restrict the reordering of instructions These are called memory barrier instructions, and come in several flavours, which determine the level of synchronization done.

Atomic operations

As well as the synchronization objects described above, the Windows API provides a set of atomic operations, commonly implemented as compiler intrinsics that compile down to a single processor instruction. The most basic of these is the `InterlockedExchange` API, which atomically swaps two 32-bit values, such that all CPUs and cores in the system will see either the original value, or the new value.

This property is common to all the atomic APIs (all of which share the `Interlocked` prefix), which include increment, decrement, add, and compare-and-exchange. The latter (which comes in both 32-bit and, on Windows Vista, Windows Server 2003, and 64-bit platforms, 64-bit variants) is the basic workhorse of thread synchronization – all other operations can be built on it.

The semantics of these basic atomic operations also include the full memory barrier semantics necessary for synchronization between threads running on different CPUs – all writes to memory that occur in the source code before the atomic operation will be completed (and made visible to other CPUs) before the atomic operation, and all reads that in the source code after the atomic operation will not proceed until the atomic operation has complete.

1 http://www.boost.org
2 http://www.boost.org/LICENSE_1_0.txt
3 http://boost.cvs.sourceforge.net/boost/boost/boost/thread/win32/?path-rev=thread_rewrite

The primary reason for using anything other than a Windows Mutex kernel object is the need for speed

Newer versions of Windows such as Windows Server 2003 and Windows Vista also provide additional variants of some of the atomic operations that only provide a write barrier (the **xxxRelease** variants), or a read barrier (the **xxxAcquire** variants), but these are not used for the implementation under discussion.

Choosing a mutex implementation

The simplest way to implement a mutex would be to write a wrapper class for one of the native Windows synchronization objects; after all, that's what they're there for. Unfortunately, they all have their problems. The **Mutex**, **Event** and **Semaphore** are kernel objects, so every synchronization call requires a context switch to the kernel. This can be rather expensive, especially so when there is no contention.

Boost mutexes are only designed for use within a single process, so the **CRITICAL_SECTION** looks appealing. Unfortunately, there are problems with this, too. The first of these is that it requires explicit initialization, which means it cannot reliably be used as part of an object with static storage duration – the standard static-initialization-order problem is compounded by the potential of race conditions, especially if the mutex is used as a local static. On most compilers, dynamic initialization of objects with static storage duration is not thread-safe, so two threads may race to run the initialization, potentially leading to the initialization being run twice, or one thread proceding without waiting for the initialization being run by the other thread to complete.

The second problem is that you can't do a timed wait on a **CRITICAL_SECTION**, which means we need another solution for a mutex that supports timed waits, anyway.

There is also another problem with using **CRITICAL_SECTION**s as a high-performance mutex, which is that a thread unlocking the **CRITICAL_SECTION** will hand-off ownership to a waiting thread. More on this below.

```
long flag;
void lock()
{
  while(!InterlockedExchange(&flag,1))
  {
    wait();
  }
}
void unlock()
{
  InterlockedExchange(&flag,0);
  wake_a_waiting_thread();
}
```

Listing 1

Race conditions

In multithreaded code, a race condition is anything where the outcome depends on the relative ordering of execution of instructions on two threads. They are particularly problematic where there are two separate operations that need to be done together: if the thread performing the operations is interrupted after doing the first part, and before doing the second, then another thread may see the overall operation as half-done. Not only that, but without synchronization, due to memory visibility issues, the thread might see the result of the second operation, and not the result of the first. If the second thread is also operating on the same data, this may then end in a garbled mess.

As a concrete example, consider a 64-bit counter on a 32-bit system. This is therefore implemented as a pair of 32-bit unsigned integers, one for the high part, and one for the low part. If we have two threads incrementing this counter without synchronization, we have a potential for problems:

Thread A reads low part=0xffffffff
Thread A read high part=0
Thread A increments counter
Thread B reads low part=0xffffffff
Thread A writes low part=0
Thread A writes high part=1
Thread B reads high part=1
  // oops thread B now sees low=0xffffffff, high=1
Thread B increments counter
Thread B writes low part=0
Thread B writes high part=2

Thread B has now just increased the counter by 2^32, rather than by 1. If this counter is a reference count, then the same issue in reverse when decrementing the count could cause the reference-counted object to be freed whilst other threads still had references.

Race conditions can be the causes of hard-to-find bugs, since they don't replicate well, and often disappear entirely in the debugger.

The need for speed

The primary reason for using anything other than a Windows **Mutex** kernel object is the need for speed – otherwise, the ease of implementation when using a **Mutex** object would make it an obvious choice. The fastest way to lock a mutex is using atomic operations – do an atomic exchange of the 'locked' flag to set it, and check the old value; if it was already locked, this thread has to wait, otherwise it has the lock – but the problem comes with the 'this thread has to wait' part. A waiting thread has to consume as near to zero CPU time as possible until the mutex becomes unlocked, in order to not slow down the running threads. (See Listing 1)

The simple answer, which is the one used by the Windows **CRITICAL_SECTION**, is to use a Windows auto-reset **Event** to handle contention. When a thread tries to lock the mutex, and finds it already locked, then it blocks on the event. When the thread that owns the lock

Plain lock() is essentially a try_lock() in a loop, waiting on an event

unlocks it, if there is a thread blocked on the **Event**, it signals the event to wake the waiting thread. This also handles contention and priority neatly, in that Windows will wake exactly one thread waiting on an auto-reset **Event**, and the kernel can choose the highest priority thread to wake.

The hard part here is the 'if there is a thread blocked on the **Event**' part. Windows won't tell us, so we have to maintain a count of waiting threads in the mutex. We could just set the **Event** every time, rather than keeping a count, but that would lead to a potentially-unnecessary kernel API call on unlock, and would also cause the first thread to enter the wait loop afterwards to wake immediately, as the **Event** would stay set. This spurious wake up would just consume CPU for no gain.

Keeping count

The simplest way to keep count is just to increment the flag rather than always set it to 1. If we just incremented it to 1, then we're the only thread, so we've got the lock. Otherwise, someone else has the lock, so we must wait.

On unlock, we can decrement the count. If it hits zero, it was just us, so we do nothing, otherwise we must wake another thread, and that thread now has the lock – see Listing 2.

Hand-off

This is essentially how **CRITICAL_SECTION**s work, and is a nice simple scheme. Unfortunately, it suffers from a problem called 'hand-off'. When the current thread unlocks the mutex, if there are any other threads waiting, it 'hands off' the mutex to one of these **even if that thread is lower priority than the unlocking thread**. If the mutex is being locked and unlocked in a loop, the high priority thread may come round to the lock again, before the lower priority thread (which now owns the mutex) wakes up. In tight loops, this is quite likely, since the high priority thread won't yield to the lower priority thread until it either blocks or hits the end of its timeslice. This means that the high priority thread now has to wait for the lower priority thread, even when it could have continued.

To solve this, we need a different scheme, where the mutex is truly unlocked by **unlock()**, however many waiting threads there are.

```
void lock()
{
  if(InterlockedIncrement(&flag)!=1)
  {
    wait_on_event();
  }
}

void unlock()
{
  if(InterlockedDecrement(&flag)!=0)
  {
    wake_a_waiting_thread();
  }
}
```

Listing 2

Avoiding hand-off with compare-and-swap

The solution I've implemented is to reserve one bit of the flag for the 'locked' state, whilst keeping the remaining bits for the count of 'interested' threads. Consequently, this can no longer be implemented as a single atomic instruction, since it is not a simple addition, or even a simple mask. This is where compare-and-exchange/compare-and-swap (CAS) comes into its own: by using CAS in a loop, you can do any operation you like to a single 32-bit field (or 64-bit field, on those platforms that support it). CAS only sets the field to the new value if it was equal to the specified 'old value', but it returns the old one, whatever it was. That way you know whether your **set** succeeded, and you can decide whether to try again, and what the new value should be based on the actual current value.

Consequently, **try_lock()** looks like Listing 3, which basically says: take the current value, increment the 'interested threads' count and set the lock flag. If we succeed in this, we've got the lock. If not, the state must have changed, so try again, unless another thread has the lock, in which case we're done.

Plain **lock()** is essentially a **try_lock()** in a loop, waiting on an event. The first time through, we increment the 'interested threads' count, whether or not we get the lock, to mark that we're waiting. If the lock flag

```
bool try_lock()
{
  long old_flag=0;
  do
  {
    long const new_flag=(
       old_flag+1)|lock_flag_value;
    long const current_flag=
       InterlockedCompareExchange(&flag,
       new_flag,old_flag);

    if(current_flag==old_flag)
    {
      return true;
    }
    old_flag=current_flag;
  }
  while(!(old_flag&lock_flag_value));
  return false;
}
```

Listing 3

```
bool lock()
{
  long old_flag=0;
  while(true)
  {
    long const new_flag=(
       old_flag+1)|lock_flag_value;
    long const current_flag=
       InterlockedCompareExchange(&flag,new_flag,
       old_flag);

    if(current_flag==old_flag)
    {
      break;
    }
    old_flag=current_flag;
  }
  while(old_flag&lock_flag_value)
  {
    wait_on_event();
    do
    {
      long const new_flag=
         old_flag|lock_flag_value;
      long const current_flag=
         InterlockedCompareExchange(&flag,
         new_flag,old_flag);

      if(current_flag==old_flag)
      {
        break;
      }
      old_flag=current_flag;
    }
    while(!(old_flag&lock_flag_value));
  }
}
```

Listing 4

```
void unlock()
{
  long const offset=lock_flag_value+1;
  long const old_flag_value=
     InterlockedExchangeAdd(&flag,-offset);
  if(old_flag_value!=offset)
  {
    wake_a_waiting_thread();
  }
}
```

Listing 5

was set already, then we wait on the event. When we wake, we just try and set the lock flag. If it was still set, we wait on the event again. See Listing 4.

In contrast, the `unlock()` code is quite straight-forward, since the modification to the flag value is simple: clear the lock flag, and take one off the count. Since we know the lock flag is always set, this is a simple subtraction, which we can do with the atomic exchange-and-add operation.

```
bool timed_lock(timeout_type timeout)
{
  long old_flag=0;
  while(true)
  {
    long const new_flag=(
       old_flag+1)|lock_flag_value;
    long const current_flag=
       InterlockedCompareExchange(&flag,
       new_flag,old_flag);

    if(current_flag==old_flag)
    {
      break;
    }
    old_flag=current_flag;
  }
  while(old_flag&lock_flag_value)
  {
    if(!wait_on_event(
       milliseconds_remaining(timeout)))
    {
      InterlockedDecrement(&flag);
      return false;
    }
    do
    {
      long const new_flag=
         old_flag|lock_flag_value;
      long const current_flag=
         InterlockedCompareExchange(&flag,
         new_flag,old_flag);

      if(current_flag==old_flag)
      {
        break;
      }
      old_flag=current_flag;
    }
    while(!(old_flag&lock_flag_value));
  }
  return true;
}
```

Listing 6

The only issue here is the fact that the timeout on the wait and the decrement of the count are not a single atomic operation.

If the old value returned by the exchange-and-add shows there was another thread waiting, signal the event to wake it. See Listing 5.

Timing out

Adding a timeout to the lock function is relatively straight-forward with this scheme. We need to add a timeout to the `wait_on_event()` call, and if it times out, then we decrease the count of 'interested threads' and return `false` to indicate that we haven't got the lock. If the wait doesn't time out, then we proceed just like `lock()`, returning `true` if we do get the lock.

The only issue here is the fact that the timeout on the wait and the decrement of the count are not a single atomic operation. Consequently, if this thread is the only waiting thread, and the current owner unlocks the mutex between the wait timing out and the flag being decremented, then the event will be signalled, even though there are no waiting threads. This will cause a spurious wake-up of the next thread to wait on the mutex, which is unfortunate, but not a disaster. The alternative, which is that rather than just decrementing the count, we try and acquire the lock, and only decrement the count if we don't get it, also has the potential for spurious wake-ups. If the timing-out thread is not the only waiting thread, but the mutex is unlocked between the timeout and the decrement, then another of the waiting threads will wake. If we acquire the lock instead of just decrementing the count, then the other waiting thread will have woken for no reason.

Events and initialization

Up to now I've glossed over the `wait_on_event()` and `wake_a_waiting_thread()` calls, but they're quite important to the whole scheme. They're also rather simple:

```
bool wait_on_event(unsigned milliseconds=INFINITE)
{
  return WaitForSingleObject(
     get_event(),milliseconds)==0;
}
void wake_a_waiting_thread()
{
  SetEvent(get_event());
}
```

Since we're using an auto-reset event, the event object is automatically reset when the first waiting thread wakes from the `WaitForSingleObject` call. The complication is `get_event()`. In order to permit static initialization, our mutex cannot have a constructor, so the members have to be initialized with fixed values – this was one of the problems with `CRITICAL_SECTION`s. Therefore, we need to create the event object in the `get_event()` call, in such a way that if multiple threads call `get_event()` concurrently, they all return the same event object. We do this with yet more atomic operations – first we read the current `Event` handle; if it's `NULL`, then we create a new `Event`, and try and swap it into place. If we succeeded, then we're using the `Event` we created, otherwise another thread beat us to it, so we use that, and destroy the one we created. See Listing 7.

```
HANDLE event;

HANDLE get_event()
{
  HANDLE* current_event=
     InterlockedCompareExchangePointer(
     &event,0,0);
  if(current_event)
  {
    return current_event;
  }
  HANDLE* new_event=
     CreateEvent(NULL,false,false,NULL);
  if(!new_event)
  {
    throw thread_resource_error();
  }
  current_event=
     InterlockedCompareExchangePointer(&event,
     new_event,0);
  if(!current_event)
  {
    return new_event;
  }
  else
  {
    CloseHandle(new_event);
    return current_event;
  }
}
```

Listing 7

Initialization is thus straight-forward: for objects with static storage duration, zero initialization will suffice, and for objects with non-static storage duration, explicit initialization to zero using an aggregate initializer will suffice.

```
void f()
{
  static mutex sm;
  // zero init works OK for static

  mutex m={0};
  // aggregate initialization required for auto

  static mutex sm2={0};
  // aggregate init works OK for static, too
}
```

This is important, as any use of a constructor would require dynamic initialization, which occurs at runtime, and therefore may be subject to race conditions. This is particularly a problem for objects with static storage duration at block scope, since the constructor is run the first time control passes through the definition of the object; if multiple threads are running concurrently, it is possible for two threads to believe they are 'the first', and thus both run the constructor. It is also possible that one thread will start running the constructor, and another thread will then proceed before the constructor has completed.

Cleaning up

Since such a mutex uses a Windows `Event` object, it needs to tidy up after itself, otherwise you have a resource leak, which is less than ideal.

For automatic and dynamic mutexes, clean-up is easy: the destructor should destroy the `Event`. For statics, clean-up is a bit more complicated.

Destructors for objects with static storage duration are called in the reverse order of their construction. Unfortunately, this is not necessarily the best order, especially for block-scope statics. Block-scope statics have the additional problem of a potential race condition on 'the first time through', which can lead to the destructor being run multiple times on popular compilers, which is undefined behaviour – compilers often use the equivalent of a call to `atexit` in order to ensure that destruction is done in the reverse order of construction, and the initialization race that may cause the constructor to be run twice may also cause the destructor to be registered twice. If any threads continue after `main`, this problem is compounded, as the unpredictable destruction order means that the mutex may be accessed after it has been destroyed, again leading to undefined behaviour.

We do know, however, that Windows Objects allocated by a program are freed when the program exits, so for objects of static storage duration, we can get by without a destructor, which neatly sidesteps the 'access after destruction' and 'multiple destruction' issues. It does make it hard to use the same class for static objects and non-static objects, though.

Copping out

The current boost spec requires that instances of the mutex type are usable without an explicit initializer. This means that it is not possible to satisfy the requirements for both static and non-static storage duration without resorting to compiler-specific techniques, or some form of 'named mutex' technique that doesn't require storing state within the mutex.

The current implementation of `boost::mutex` cops out, and has a default constructor and a destructor. This makes it dangerous to use as a block-scope static, but yields correct behaviour for objects with non-static storage duration, and objects of namespace scope, provided care is taken with initialization order.

Though this is the same as for previous boost releases, this situation is less than ideal, however, and the search continues for a way of ensuring correct initialization in all cases.

It has been suggested that boost adds a `static_mutex`, which would then be portable to other platforms, but this is not available at this time, and building a safe-for-all-uses mutex would be preferable.

The `basic_timed_mutex` used in the Windows implementation has no constructor or destructor, and could therefore safely be used with static storage duration as described above. Use at non-static storage duration requires calling the `init()` and `destroy()` member functions in place of the constructor and destructor. This class is a detail of the current `thread_rewrite` Windows implementation, however, and its use is not therefore recommended.

Generalizing to other platforms

Whilst the implementation described here is Windows-specific, the majority of the code is just using the `InterlockedXXX` functions. These functions are generally just compiler intrinsics for single CPU instructions, so could easily be implemented on non-Windows platforms with a bit of inline assembly, or by replacing them with the equivalent calls to OS functions.

The larger bit of non-portability comes from the use of `Event` objects. These are essentially just binary semaphores, so can easily be replaced by semaphores on those systems that support them (e.g. POSIX systems). POSIX semaphores are not quite ideal, though – they would have to be dynamically allocated using `new` or `malloc` in `get_event`, and they aren't limited to just `set`/`unset`, so there is the potential of spurious wake-ups. This wouldn't prevent the scheme described from working, since it allows for spurious wake-ups, but it would be less than ideal. A condition variable could be used instead, but that also has the potential for spurious wake-ups, and might have more overhead, since it requires use of an OS mutex in addition.

Future plans

Work is still under way to identify a solution to the initialization and clean up problems. Under the new C++ Standard, initialization may be easier, as there is a proposal under consideration for generalized constant expressions[4], which would enable simple default constructors to be used for static initialization, and thus solve the race conditions due to initialization. Unfortunately, this does not solve the corresponding clean up problems. There are also proposals under consideration to address thread-safe initialization of static objects.

it is not possible to satisfy the requirements for both static and non-static storage duration without resorting to compiler-specific techniques

In any case, the new C++ Standard won't be published before 2009, and it will then be a few years before compilers supporting it become common, so this is still an important issue.

Conclusion

Implementing synchronization primitives is hard. Ensuring they are correct is hard enough, but fairness issues and initialization problems just make the whole thing harder.

Thankfully, most of the time we can just rely on libraries such as Boost, and not have to think about the issues. It does mean that as the implementor of such a library it is even more important to get things right.

As a library user, understanding the issues involved can help us to see the reason behind particular restrictions the library places on us, and can help us write better code. ■

Acknowledgements

Thanks to Peter Dimov and Alexander Terekhov who pointed out the hand-off problem to me, and suggested using a swap-based method to avoid it. Thanks also to Roland Schwarz for reviewing the code, and proposing alternative initialization and implementation techniques.

4 http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2006/n2116.pdf

Design in Test-Driven Development

With its roots in Japanese just-in-time manufacturing, Test-Driven Development (TDD) puts the traditional development process on its head. Adam Petersen discusses when to use TDD, how to use it successfully, and its use in up-front design.

The roots of TDD

While best known to originate from Extreme Programming [Beck & Andres], TDD really has its roots in the Toyota Production System (TPS). Taiichi Ohno, the brilliant engineer behind TPS, was obsessed with eliminating waste in production. The main problem was how to supply the number of parts needed just-in-time. Ohno approached the problem by reversing the production flow: 'a later process goes to an earlier process to pick up only the right part in the quantity needed at the exact time needed' [Ohno]. And here's the core of TPS: now the earlier process only have to make the number of parts actually needed, thereby approaching zero inventory. It also puts focus on quality by detecting deficiencies early in the process. The communication between the steps in the production chain is solved by kanban (sign board).

How does TPS relate to software? Toyota is about cars, isn't it? Well, I consider eliminating waste very relevant for software development too. A very common type of waste is code that's written but not integrated until weeks and even months later. In this case, the main waste arises from the late feedback and the missed opportunity to improve the code from the knowledge gained. Further it indicates a waste because something was developed but there obviously wasn't any true need for it. At least not immediately and the time spent could have been invested in an activity adding immediate value.

TDD achieves just-in-time exactly the same way as Toyota does: by inverting the steps in the traditional process using a failing unit test as its kanban. That is, the failing testcase is the need that triggers production and ensures that no unnecessary code (i.e. waste) is developed. Of course it also provides immediate feedback.

TDD crash course

TDD is dead simple. At least in theory. Here are the two only rules [Beck]:

1. Write new code only if an automated test has failed
2. Eliminate duplication

Simple to remember but hard to apply. As Kent Beck puts it 'These are two simple rules, but they generate complex individual and group behavior with technical implications' [Beck].

The first rule forces us, at least if we're hardcore TDDers, to write a test before writing any production code. I always wanted to be a space pilot, so using Java and the unit test framework JUnit [JUnit] I'll try to explore the characteristics and responsibilities of a spaceship (Listing 1).

There are not many lines of test code and no production code at all, yet I have specified several design decisions in the test in Listing 1.

Adam Petersen is a software developer whose prime professional interests include C++, patterns, agile development, modeling and Lisp. Besides spending way too much time reading tech books, Adam also has somewhat healthier hobbies like chess, music, modern history and Russian literature.

```
// SpaceshipTest.java
public class SpaceshipTest extends TestCase {
  public SpaceshipTest(String testName) {
    super(testName);
  }
  public void testDrivingMode() {
    Spaceship spaceship = new Spaceship();
    assertEquals(spaceship.speed(), 0);
    final int speedOfLight = 299792458;

    DrivingMode hyperSpeed = new DrivingMode() {
      public int topSpeed() {
        return speedOfLight;
      }
    };
    spaceship.shiftDrivingModeTo(hyperSpeed);
    assertEquals(spaceship.speed(),
       speedOfLight);
  }
}
```

Listing 1

- Object creation: I have decided how a spaceship comes into existence and the initial state of a spaceship object (a speed of zero m/s). A typical spaceship will probably have many more characteristics, but I'll leave that for now. With TDD, I'll try to address one problem at a time (TDDers refer to this as organic design) and right now I want to drive at high speed; both with the development as with the spaceship.
- API design: The unit tests are the first usage of the code and provide immediate feedback, actually even before the code exists, on how easy it is to use the API.
- Decoupling: The `Spaceship` class is decoupled from the concrete driving modes and only knows about the `DrivingMode` interface. This is in line with the design principle of programming to an interface, not an implementation and is a typical TDD pattern. Besides being good design, it allows full control of the unit under test through the test stub (the anonymous class implementing the `DrivingMode` interface). I avoid depending upon concrete classes particularly because they add another factor of uncertainty and in TDD I never want more than one at a time.
- Side-effects specified: Shifting driving mode means that the new speed will equal the top speed in the current mode. It's stated explicitly in code what it means to shift driving mode.

The first test-case for a class is typically the one that involves most exploration. When I have it in place, I write the first version of the `Spaceship`:

Unless the implementation is obvious (as it is in Listing 2), I start with a stub implementation where I return a hardcoded value. The main reason is that it ensures that I'm testing the right thing, which gets harder as the

the single most common error developers make as they start to test-drive is taking too large steps

```
// DrivingMode.java
public interface DrivingMode {
  public int topSpeed();
}

// Spaceship.java
public class Spaceship {
  private DrivingMode drivingMode;
  public int speed() {
    int speed = 0;
    if(drivingMode != null) {
      speed = drivingMode.topSpeed();
    }
  return speed;
  }
  void shiftDrivingModeTo(DrivingMode newMode) {
    drivingMode = newMode;
  }
}
```

Listing 2

```
// Parked.java
public class Parked implements DrivingMode {
  public int topSpeed() {
    return 0;
  }
}
// Spaceship.java
public class Spaceship {
  private DrivingMode drivingMode;
  public Spaceship() {
    drivingMode = new Parked();
  }
  public int speed() {
    return drivingMode.topSpeed();
  }
  void shiftDrivingModeTo(DrivingMode newMode) {
    drivingMode = newMode;
  }
}
```

Listing 3

body of code grows. It also makes it easier to explore alternatives; because I haven't really put much effort into the code it is easier to just delete the code and start over if I am dissatisfied with it, something that's much harder mentally if I go for a full implementation directly. After I've run the test and got a green bar, I'll check for duplications and potential improvements. What I don't like above, is the `speed()` method; depending on state, the speed is set at two different places and the conditional is an unnecessary complexity. Let's factor it out by taking full advantage of `DrivingMode` (see Listing 3).

After my refactoring I run the test again and ensure that I still have a green bar in JUnit. In this last example, the testcases take on their second role; instead of driving the design they now function as regression tests.

Iterate again and again

These small iterations are the foundation of TDD and the single most common error developers make as they start to test-drive is taking too large steps. This is a hard balance; taking too small steps is inefficient, taking too large steps is a sure way to lose the feedback that TDD provides. With small steps, as a unit test fails it is immediately clear where the problem is (if it isn't you're not taking small enough steps). Every time I have to enter the debugger during development I know that I've rushed away with the coding and have to take smaller steps.

Small steps are also a good way to stay on track. In the average large software organization there are lots of disturbing factors such as phone calls, e-mails, and background noises. With a small testcase, there's less information necessary to regain as I pick up the coding after a distraction. Unit tests help relieve my mind by keeping knowledge in the world instead of in the head. It is also the way I prefer to leave a coding session at the end of the day; a small, failing unit test that functions as a memory aid, a written and executable note to my future self.

I believe that it is impossible to give a general guideline on the size of the steps; the optimal step probably varies depending on personality and experience of the programmer. For example, as I use a new language or start working in a new problem domain the steps I take are shorter than the one above.

Design at different levels

These days it seems popular to bash Extreme Programming (XP) where TDD is a vital component. Matt Stephens and Doug Rosenberg even devoted a whole book to dissecting and, partly, ridiculing XP [Stephens & Rosenberg]. While they credit unit testing as important and state that it can complement more traditional up-front design, they make sure to push their own silver-bullets (Use Cases and Sequence Diagrams, which is hardly surprising as Doug has written two books about it): 'The clean allocation of operations to classes you can achieve on a sequence diagram will eliminate the need for a whole bunch of Constant Refactoring After Programming'. This begs the question, how much design do I want up-front and how does it impact the role of my unit tests?

The first question is impossible to answer without a context. For example, I used to work on safety-critical software for the railway industry. One of the safety techniques was diversified programming, which basically means that the same program is written twice by two independent teams. The two programs are run in parallel and the results are compared between the programs at predefined points, everything in real-time. If the programs don't agree on the result, it means an emergency stop of all trains (hardly popular, particularly not for the passengers). It is a very expensive way to

Trying to test correctness into diversified software is a dead end ... the main design has to be defined up-front, if the two programs are ever going to agree

develop. Think about how hard it is to get one program working. Trying to test correctness into diversified software is a dead end and it is obvious that the main design has to be defined up-front, if the two programs are ever going to agree on their state of the world. Here well-defined use cases and complementary models are invaluable, particularly to make the transition from problem space, as defined by the requirements, to the solution space and the design. Still, at a certain point it makes more sense to switch to code as a design medium. The reason is requirements explosion.

Requirements explosion

The single most frequent question I've gotten with respect to TDD is: 'how do I know the tests to write?' It's an interesting question. The concept of TDD seems to trigger something in peoples mind; something that the design process perhaps isn't deterministic. I mean, I never hear the question 'how do I know what to program?' although it is exactly the same problem. As I answer something along the lines that design (as well as coding) always involves a certain amount of exploration and that TDD is just another tool for this exploration I get, probably with all rights, sceptical looks. The immediate follow-up question is: 'but what about the requirements?' Yes, what about them? It's clear that they guide the development but should the unit tests be traced to requirements?

My answer is a strong no, njet, nein. Requirements describe the 'what' of software in the problem domain. And as we during the design move deeper and deeper into the solution domain, something dramatic happens. Robert L. Glass identifies requirements explosion as a fundamental fact of software development: 'there is an explosion of "derived requirements" [..] caused by the complexity of the solution process' [Glass]. How dramatic is this explosion? Glass continues: 'The list of these design requirements is often 50 times longer than the list of original requirements' [Glass]. It is requirements explosion that makes it unsuitable to map unit tests to requirements; in fact, many of the unit tests arise due to the 'derived requirements' that do not even exist in the problem space!

Further, to capture all these derived requirements in a document or a UML model requires the level of detail of a programming language. Languages for that purpose do exist. The Object Constraint Language (OCL) [OCL] for example is a formal language for adding details to modelling artifacts. The problem is that extending the models with that kind of detailed information may actually limit their use. Unless you go for full code-generation a la MDA [MDA] where the model actually is the program (an approach which has problems of its own), you'll lose what I believe is the most valuable quality of models: a higher level view than the code. The models will in that case basically turn into a mixture of two different abstraction levels. Detailed design taken to such lengths will also result in a lot of overlap with the code; you'll get the feeling that you already coded the stuff ones before during the modelling. On projects where I worked with such detailed designs I found it terribly hard to keep them up-to-date. Every change results in a necessary update of the models, which isn't very productive. Sure, there are tools that may help by supporting reverse engineering, but basically they only help covering the symptoms of a real problem.

My advice is to care about designing the details but doing it in the medium most suitable to express that level of detail: unit tests, written in the same programming language as the production code.

The purpose of TDD

To me, TDD is primarily a design technique. Sure, the unit tests developed during TDD do serve a very valuable verification purpose. However, they verify code-correctness. Every software project has to complement them with testing on other levels, such as acceptance- and requirements-testing. The unit tests lay the foundation for the higher level tests and enable them to focus on their true purpose in a more efficient way; as I work with system tests I don't want to be stopped by coding errors and this is where TDD helps.

TDD is also a verification tool for the intent of the programmer. Like so many other techniques descending from Extreme Programming, TDD provides an interesting double-check mechanism. With TDD every programmer states his/her intent twice; once in the unit test and once in the production code. Only if they match do we get a green bar.

If you only want verification, you don't have to do TDD (although you need something else to carry out the low-level design, be it modelling, formal specifications, genius or plain luck). In this case I still recommend the unit tests to be written in close conjunction to the code. Writing unit tests only with respect to verification is more straightforward as there are no more design decisions to take. The major disadvantages are, of course, that you lose an excellent opportunity for design and run the risk of writing un-testable code.

Code coverage

Code coverage is a simple technique for providing feedback on the quality of the unit tests. A technique I've found valuable is to build code coverage analysis into the build system. In that way I can run the unit tests and get a report on the coverage with one single command. However, I typically don't bother with analysing the coverage until I've finished the first version of some module, but then it gets interesting. In theory, when using TDD we will always get 100 percent coverage (remember, we're only supposed to write code as an automated test fails). While I have written fairly large programs with full coverage I don't believe it is an end in itself nor particularly meaningful as a general recommendation; it's just a number. Instead the value I get is as feedback on my test writing skills. If I have missed a line or branch during my test-drive I try to analyse the cause; perhaps it is okay to leave it as it is, but more often there was some aspect of the solution that I initially overlooked.

What code coverage analysis actually implies is an implicit code review and that provides a great learning opportunity. But there's more to it. Another aspect where code coverage really helps is detecting broken windows.

The secret to successful modifications of existing programs is to keep one factor constant all the time

Broken windows

TDD works best when it's actually used. Let me elaborate by connecting to the heading. Broken Windows, at least in this context, has absolutely nothing to do with operating systems; it's a term from social psychology that comes from the following example: 'if a window in a building is broken and is left unrepaired, all the rest of the windows will soon be broken' [Wilson].

The analogy to software is apparent; a class without a unit test is a broken window and just makes such an excellent excuse to code yet another class without unit test. I think it's something very fundamental in human nature and I've been there myself. The original article on the subject puts it this way: 'one unrepaired broken window is a signal that no one cares, and so breaking more windows costs nothing' [Wilson]. From there things only get worse; trying to repair a broken window by covering the code with tests afterwards is tough, as the code probably isn't designed with respect to testing and now much effort has to be put into breaking dependencies in order to make the code testable (in fact it is such a tough problem that Michael Feathers has written a whole book about it [Feathers]).

On the other hand, as I extend or debug an existing program, if the program was developed with full TDD from the very start, I just continue to write tests as I go along. The value I get from unbroken windows is obvious and during maintenance I learn to appreciate the unit tests as a regression test suite. Writing code without covering tests would in such a case be breaking the first window and that's just too conspicuous.

TDD in a maintenance context

Software maintenance will always be hard but TDD may ease the pain. In fact, due to the small and rapid iterations in TDD, the software is put in maintenance mode almost instantly.

The secret to successful modifications of existing programs is to keep one factor constant all the time. In TDD terms this means either changing the unit test or the unit under test, but never both at the same time. After some initial analysis of the necessary changes I turn to the unit tests and write more of them. These may be either complementary tests to try out my understanding of the software to modify or a testcase for the required change. From now on the process is exactly the same as in TDD during greenfield-development.

TDD recommendations

TDD is a high-discipline methodology. That makes it easy to slip. Below are some recommendations on what I believe are the most important practices to adhere to during TDD.

1. *Keep the same quality on unit test code as on the code under test*. There's apparent danger in mentally and qualitatively differentiating between production code and test code. Remember, the unit tests are your primary interface to the code during development and maintenance and you do want that interface to evolve clean and nice over time in order to keep it alive.
2. *Write unit tests that are small and independent*. Particularly, avoid dependencies upon databases, network communication or files. It is in the vein of good design to keep software loosely coupled. Failure to follow this recommendation may have practical implications very soon, as such unit tests do not only require complicated set-up and clean-up code; they also take a long time to run. Unit tests that take a long time to run will probably not be run often enough (the same is true for the build process, if you're using a compiled language the unit tests have to be fast to build and run) and there's a risk that the unit tests get out of sync with the rest of the codebase and turn into heavy baggage that's finally abandoned.
3. *Use consistent naming*. My personal convention is to name the unit tests equally to the unit they're testing and appending 'Test' to the name. Returning to my initial example where I test-drove a `Spaceship.java` unit I named the corresponding unit test `SpaceshipTest.java`. The rationale is that most IDEs sort files and classes alphabetically making it easy to navigate between tests and production code.

Summary

Test-Driven Development is a design technique that pays off soon and, at the same time, an investment in the future that continues to add value in subsequent versions of the software. TDD is not the long-sought silver bullet of software. It doesn't really make any of the traditional phases in software development obsolete (possibly with the exception of desperate bug-hunting close to a release, but that rarely turns out to be a pre-defined and planned activity). Instead it inverses the order of coding and testing, thereby providing an excellent medium for detailed design with immediate feedback. TDD requires a lot of discipline and I hope that my recommendations will help you on the quest to great software. ■

References

[Beck] Kent Beck, *Test-Driven Development: By Example*, ISBN 10:0321146530

[Beck & Andres] Kent Beck and Cynthia Andres, *Extreme Programming Explained: Embrace Change (2nd Edition)*, ISBN 10:0321278658

[Feathers] Michael Feathers, *Working Effectively with Legacy Code*, ISBN 10:0131177052

[Glass] Robert L. Glass, *Facts and Fallacies of Software Engineering*, ISBN 10:0321117425

[JUnit] JUnit homepage: http://www.junit.org/index.htm

[MDA] OMG Model Driven Architecture, http://www.omg.org/mda/

[OCL] *The Object Constraint Language (2nd Edition)*, ISBN 10:0321179366

[Ohno] Taiichi Ohno, *Toyota Production System: Beyond Large-Scale Production*, ISBN 10:0915299143

[Stephens & Rosenberg] Matt Stephens and Doug Rosenberg, *Extreme Programming Refactored: The Case Against XP*, ISBN 10:1590590961

[Wilson] Wilson, Kelling, 'Broken Windows', *The Atlantic Monthly*, March 1982

C++ Unit Test Frameworks – a Comparison

There are many Unit Testing frameworks for C++, but which one to use? Chris Main shares his experience of some of them.

'Not another one' was the immediate reaction of a work colleague on seeing the article [FRUCTOSE] on the FRUCTOSE unit test framework [Overload07] open on my desk. It had also been my first reaction, but as I have always taken an interest in unit testing I made the effort to study the article. I was pleasantly surprised to find that FRUCTOSE has something to offer that other tools did not. The author, Andrew Marlow, also explained in some detail why he had not used any of a number of existing tools. I found this more satisfactory than a previous article [Overload06] on CUTE [CUTE] where the author, Peter Sommerlad, only explicitly considered CppUnit [CPPUnit] as an alternative, even though Aeryn [Aeryn] was referenced in the article. The FRUCTOSE article did not consider Aeryn, so I offer this comparative evaluation. The focus of my evaluation is a narrow one: I am considering frameworks specifically for unit testing C++, which can be run from a command line.

Boost

For the last three and a half years I have been using an in-house framework which is very similar to Boost [Boost] but with a richer set of assertion macros and the facility to select tests from the command line. We would have used Boost, except our compiler limited us to the header only libraries. This confirms Andrew Marlow's assertion that there are some environments where the Boost framework cannnot be used. Where the Boost framework can be used, though, it should not be dismissed too hastily. It gives me the impression that it has evolved from something more complex (an impression reinforced by the release notes). The documentation is comprehensive, but could be better organised. I fear that it obscures how very easy the framework is to use if you restrict your tests to standalone functions taking no arguments and returning void. Having worked within this restriction when using our in-house framework, I think this is no problem at all. Test files follow the format in Listing 1.

In addition, you need a suite creator which is trivial to implement, as shown below:

```
#define BOOST_AUTO_TEST_MAIN
#include <boost/test/auto_unit_test.hpp>
```

You then link together the suite creator, the boost test lib and as many test files as required (from a single test file to all the test files for a library or application). Hence Boost supports running the tests for a single file, but in a way that scales up easily to running a full suite of regression tests.

Chris Main has been a programmer since 1989, writing his first unit test framework in Ada soon after and porting it to C in 1992. C++ has been his main language since 1999, and he joined ACCU the following year.

```
#include "factorial.hpp"

#include <boost/test/auto_unit_test.hpp>

namespace {
  BOOST_AUTO_UNIT_TEST(test_factorial)
  {
    BOOST_CHECK_EQUAL(factorial(3), 6);
  }
}
```

Listing 1

Boost does not support selecting tests from the command line. This inconvenience can be minimized by ensuring that any long running test is isolated in its own test files.

Boost has a good selection of assertion macros, including allowing tolerances for floating point comparisons. It doesn't have **_LT**, **_LE**, **_GT**, **_GE** variants which I have come to prefer to boolean assertions where appropriate.

Boost supports two output formats (selectable from the command line) out of the box: human readable and XML. This is a reasonable design. The human readable format is good enough for most purposes, and the XML can be post processed into any conceivable format when it isn't.

In summary, Boost provides a simple to use, fully functional and scaleable unit test framework, so long as you can build it on your platform. It also provides many other features, which you can find in its documentation, but bear in mind that you may not need them.

Aeryn

Aeryn requires a modern C++ compiler, so if you can't build Boost you may not be able to build Aeryn. It has no dependencies upon other libraries. Unlike Boost, Aeryn feels like it has evolved from something simple. Its

```
#include "factorial.hpp"
#include <aeryn/test_case.hpp>
#include <aeryn/test_registry.hpp>

namespace {
  void test_factorial()
  {
    IS_EQUAL(factorial(3), 6);
  }
  Aeryn::TestCase numeric_tests[] = {
     Aeryn::TestCase( USE_NAME(test_factorial) ),
     Aeryn::TestCase() };
  REGISTER_TESTS_USE_NAME(numeric_tests)
}
```

Listing 2

design is very clean, and this is reflected in the User Guide which is a model of clarity (once you recover from the picture of the leather clad young woman on the title page!). You can use Aeryn in almost exactly the same way as Boost; the difference is that you have to wrap your test functions in `TestCase`s, as in Listing 2.

This leaves scope for a programmer to write a test function but forget to wrap and register it. It should be possible to write a macro to take a function name, wrap it in a test case and register it all in one to to prevent this happening. Building an executable is also very similar to Boost: link the Aeryn main and core libs with as many test files as required.

Aeryn does support selecting tests from the command line. It has a more limited set of assertion macros (no tolerance for floating point comparisons, for example, you have to implement that yourself).

Aeryn supports selection of output format from the command line. A number of human readable formats are provided out of the box, but not XML or HTML. Aeryn provides an elegant mechanism for building custom output formats into its framework: simply write a class which inherits from `Aeryn::IReport` and implements its well documented virtual member functions, then add the class to the report factory to make it available from the command line.

In summary, Aeryn provides a simple to use, fully functional and scaleable unit test framework, so long as you can build it on your platform.

Since, in my opinion, Boost and Aeryn are both excellent, mature unit test frameworks I don't understand what CUTE is trying to achieve, given that it requires the same kind of platform as them.

FRUCTOSE

FRUCTOSE does have a clear niche: it's a header file only framework that doesn't require a modern C++ compiler. I think it has the potential to be worthwhile even outside of that niche. It has the richest selection of assertion macros. I have had to insert tracing code to output loop indexes often enough to think that the loop assertion macros are a good idea. It supports selecting tests from the command line. It only supports a single, human readable format. Personally, I think that customisable output is a less important feature than a good set of assertion macros, so this doesn't put me off.

Although I usually work with a framework based upon standalone functions, I find that for a large proportion of my unit tests the standalone function is a wrapper around a test class. This is because there is usually some setup I want to put in a class constructor. I don't, therefore, see FRUCTOSE's requirement to derive test classes from a base class as a great burden. As for incorporating implementation code into the test class by public inheritance, I wouldn't normally consider this good design. However the sheer convenience it brings in this case, and the fact that test classes will not be further derived from nor have instances passed around as objects, leads me to take a pragmatic view of it (just as I pragmatically accept the use of Singleton in the implementation of automatic test registration, even though I normally avoid that pattern).

CxxTest

One weakness of FRUCTOSE when compared to Boost and Aeryn is that its main function has to be hand coded. Possibly this could be remedied using a similar automatic registration mechanism to them, but I think an approach more in keeping with the spirit of FRUCTOSE may be to adopt the design used by CxxTest [CxxTest]. This uses a Perl script to generate the main function from the test code. (CxxTest is quite similar to FRUCTOSE in that it has a good set of assertion macros, is distributed as header files, has no other dependencies and will work with older compilers. Its big drawback is that it does not allow selection of tests from the command line; you have to comment out code for a particular test if you wish to exclude it).

```
#define FRUCTOSE_STRUCT(_name_) struct _name_ :
   public fructose::test_base<_name_>
#define FRUCTOSE_CLASS(_name_) class _name_ :
   public fructose::test_base<_name_>
#define FRUCTOSE_TEST(_name_) void _name_(
   const std::string &test_name)
```

Listing 3

```
FRUCTOSE_CLASS(numeric_tests)
{
public:
  FRUCTOSE_TEST(test_factorial)
  {
    ...
  }
};
```

Listing 4

By defining a few macros, as shown in Listing 3, so that the test code looks like Listing 4, it becomes very easy to write a parser for the test code.

The parser only needs to do basic text processing; it does not need to understand the grammar of C++. The parser can then be used to generate the main function automatically, and it is straightforward to write the generator so that it can take any number of input test files. In this way FRUCTOSE, although initially designed for testing a single class, would be made as scaleable as the other frameworks.

The macros have the added advantage, in my opinion, of concealing the Curiously Recurring Template Pattern and of relieving the programmer of the need to remember the correct signature for test member functions. These are simplifications, but some may consider that achieving them by means of macros is a price not worth paying. ■

References

[FRUCTOSE] https://sourceforge.net/projects/fructose

[Overload07] Overload 77, February 2007

[Overload06] Overload 75, October 2006.

[CUTE] http://wiki.hsr.ch/PeterSommerlad/wiki.cgi?CuTeDownload

[Aeryn] http://www.aeryn.co.uk

[CPPUnit] https://sourceforge.net/projects/cppunit

[Boost] http://www.boost.org

[CxxTest] http://cxxtest.sourceforge.net

Checklist

Feature	Boost.test	Aeryn	FRUCTOSE	CxxTest
Requires modern C++ compiler	Yes	Yes	No	No
External dependencies	Boost	None	TCLAP (header files only)	Perl or Python
Automatically generated test suites	Yes	Yes	No	Yes
Assertion macros	Good	Adequate	Very good	Very good
Tests selectable from command line	No	Yes	Yes	No
Output format	Human readable and XML	Human readable and customisable	Human readable	Human readable, customisable and GUI

A Practical Form of OO Layering

Much has been written about the pattern identified by Kevlin Henney as **PARAMETERIZE FROM ABOVE**. Indeed, much has been written about it (just search the Web for 'Parameterize from Above' and 'Parameterise from Above'), but as a pattern it has never been written up. Much has also been written on accu-general about how Kevlin should get around to writing it up properly!

In lieu of a proper write-up, I think it is time to address some common misunderstandings concerning this pattern, including misunderstandings that Kevlin himself seems to have about it. Perhaps the most significant area of clarification concerns the relationship between **PARAMETERIZE FROM ABOVE** and **SINGLETON**, so we should start with that.

SINGLETON is a pattern that has come in for a lot of bad press in recent years – unjustifiably so, I would contend. **PARAMETERIZE FROM ABOVE** is often seen as an alternative design approach that clarifies the responsibilities of components and a design style that essentially removes the need for **SINGLETON**. This view of **PARAMETERIZE FROM ABOVE** is clearly mistaken, and any comparison between **SINGLETON** and global variables is nothing more than an attempt to introduce guilt by association. **SINGLETON** is a mature and well understood pattern. Of course, it is not without subtlety, but that's the point: patterns are not supposed to be easy; they require great skill to master. We know **SINGLETON** is well understood because of the amount of literature published – both online and on paper – that addresses its various issues. Indeed, the column acreage devoted to **SINGLETON**'s problems, and the ingenious workarounds proposed, are testament to the durability and quality of the pattern. Complex problems demand complex solutions, and **SINGLETON** can be seen as a natural hub for many complex solutions.

It is important to understand what is implied by the notion of above in **PARAMETERIZE FROM ABOVE**, because it then becomes clear how this pattern applies to and makes use of the role of **SINGLETON**. In the conventional architectural view of **LAYERS** each layer is stacked on top of the layer beneath it, with the lower layers holding mechanisms and representation concepts and the higher layers structured in terms of application and presentation concepts. The way **PARAMETERIZE FROM ABOVE** is often interpreted is that, given such a bottom-to-top, stacked view of layering, parameters that affect the behaviour of application objects and mechanisms should be passed in from the top of the stack, such as from manager objects and application controllers, rather than taken from objects in the lower layers, where the instinct is to fix and centralise the behaviour in **SINGLETON** objects. It is this topsy-turvy view of layering that is the root problem and, when righted, it becomes clear that **SINGLETON**s can play a dominant but benign role in any architecture.

When we consider layering more naturally from top to bottom, so that application and presentation concepts are seen as details built beneath and subordinate to more interesting mechanisms and representation-focused code, we can understand what above really refers to. This mechanism-oriented view is more natural to developers; architectures that work with such instincts rather than fighting them should be encouraged. In other words, the top layer is the one where we focus on the clever tricks of the trade and tackle the true complexities of software development, with patterns such as **SINGLETON**, and this layer sits above – and therefore dictates and parameterizes – the less significant and simpler ideas, such as the domain of the application. The value of simplicity is in keeping it in its proper place – which, in this case, is beneath **SINGLETON**.

Teedy Deigh has been a developer for a number of years and rates her programming skills as second to none, possibly closer. She is constantly on the look out for nifty techniques to impress other developers and to make her day job more interesting, whilst ensuring a certain level of job security.

The problem of gratuitous flexibility and speculative generality is that accidental and unwanted complexity is introduced into many systems. However, it appears that one of the selling points of **PARAMETERIZE FROM ABOVE** is that in loosening the coupling in a system, it naturally offers more opportunities for unanticipated flexibility and reduced cost of change than other approaches. This perspective is not only harmful, it is incorrect. Rather than encouraging flexible parameterization through pluggable patterns such as **STRATEGY** and **INTERCEPTOR**, a proper interpretation of **PARAMETERIZE FROM ABOVE** ensures that such flexibility is explicitly excluded, or at least made more difficult to achieve by accident, by using explicit conditional logic – whether using **if** or **#if** – and patterns such as the **NON-VIRTUAL INTERFACE** (NVI) idiom. Unanticipated flexibility should be discouraged not just by considering it a cautionary guideline, but by also writing code that makes it harder to introduce and take advantage of.

In this vein, another capability that is often seen as a virtue of the misinterpreted form of **PARAMETERIZE FROM ABOVE** is testability. For example, being able to use **MOCK OBJECT**s to isolate external dependencies and test interactions between core code and such dependencies. Rather than supporting or even arising from a test-oriented approach, the correct interpretation of **PARAMETERIZE FROM ABOVE** offers nothing that enhances code testability, nor should it. There is no virtue in being able to isolate units of code for fine-grained or integration-oriented testing for one simple reason: testing is not a developer's responsibility. Testing is, as the name suggests, strictly the job of testers. And it is the job of testers to test the software at the level of the system, not at the level of the code.

There is no benefit to testing at the code level as it is the responsibility of developers to ensure that the code is correct by design. When that fails, the approach they should fall back on is to drive a system through its user interface and use the debugger to focus on the code. Twisting an architecture to support some fashionable view of developer-based testing is wasteful. For one thing, we know that developers are unlikely to find all of the bugs they introduce, therefore it is not worth them doing any testing: they should leave that to others, which naturally leaves developers more time to debug code.

Testing is a tedious but necessary manual process that cannot be usefully automated, and attempts to do so are typically misguided. The idea of writing code to test code simply takes time away from writing more code, which is one of the main responsibilities a developer has. It has been suggested that coded tests act as a form of executable specification, and are therefore a requirements and design framing tool as well as a verification tool. Again, rather than taking time to write code that both defines requirements on behaviour and confirms satisfaction of these requirements, developers would be better off devoting time to debugging and letting others test their code in the context of the whole system. Because debugging is a difficult task to estimate, developers need all the time they can get. Rather than additionally support changeability and configurability in code – changing code typically introduces more bugs – the structure of the code should be explicit and fixed into place to prevent such change. There is no problem here that needs solving by **PARAMETERIZE FROM ABOVE** or, indeed, any other pattern.

I hope this article has helped to clarify the proper spatial metaphor to be used in interpreting the guideline to **PARAMETERIZE FROM ABOVE**, as well as dealing with other dysfunctional memes that have arisen from misinterpreting it. Beyond this clarification, I do not believe there is any benefit to having further write-ups as this has clearly reinforced the dominant position of **SINGLETON** in the developer's toolkit. It is, after all, the Highlander pattern: there can be only one! ■