contents

Writing Extendable Software by Allan Kelly 6

More Exceptional Java by Alan Griffiths 9

Programming With Interfaces In C++ by Chris Main 13

Building Java Applications by Vaclav Barta 16

An Overview Of C#.NET by Jon Jagger 18

Letter To The Editor 25

The Scoping Problem by Allan Kelly 26

credits & contacts

Editor: John Merrells
merrells@acm.org
241 Heartwood Lane,
Mountain View,
CA 94041-11836,
U.S.A

Website: http://www.accu.org/

Readers:
Ian Bruntlett
IanBruntlett@antiqs.uklinux.net

Phil Bass
phil@stoneymanor.demon.co.uk

Mark Radford
twonine@twonine.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Membership and subscription enquires:
David Hodge
membership@accu.org
31 Egerton Road
Bexhill-on-Sea, East Sussex
TN39 3HJ, UK

Advertising:
Peter Goodliffe
ads@accu.org
4 Malvern Road
Cherry Hinton
Cambridge CB1 9LD, UK
01223 518579

Editorial
Methodologies

In a recent issue of CVu, Pete Goodliffe wrote an excellent article surveying the history of popular methodologies. In this editorial I would like to chart my own personal journey through that history to discover where I now stand.

At the start of my computing experience, before any formal computing education, my methodology was to just hack everything into existence. I was programming, rather than engineering. I would assign myself a project, perhaps a simple line drawing program, and then set about coding. My methodology was to start with the part of the program that I knew least about. That way, if that piece defeated me, I would have wasted the least amount of time on the project. But, once the puzzle was solved, I would decide that the rest of the program was pretty obvious, and therefore there wasn't much point in completing it, so I might as well just move onto the next project.

Much later, at university, I learned about SSADM [1] and various lecturer-devised methodologies that seemed very suitable for building large payroll applications. They didn't seem to offer much for my day-to-day software engineering projects. My experience was with very small teams (a team of one), short term (midnight-6am), with a well-defined customer (me and my friends). My products were games. I spent four years porting and enhancing a variety of bulletin boards, chat systems, and multi-user dungeons. [2]

After graduation, I was straight off onto the games industry proper, at that time predominantly the vocation of small groups of underpaid youths in cramped quarters having a whale of a time making up crazy stuff all day. My chums and I were ensconced in a one room, fourth flour, lower Chelsea, studio that was reputed to have once been the office of a popular beat combo named the Rolling Stones. [3] Our methodology? Write a proposal, tart it about some publishers, code like crazy, beg for more money, code like crazy, beg for more money, etc.

I had to grow up a bit after that, and went to work on some terminal emulation software. It was all OO, so I dusted off my Booch and OMT books. OOP had always seemed pretty natural for me, so the methodologies made sense, but the process didn't do much for me. I just went through the motions. The diagramming notations were great though; a common language for communicating ideas efficiently and effectively.

Then I joined a small research and development company that was staffing a project. They were an independent start-up that used the RAD [4] approach to produce a demonstration product. A dozen engineers spent a year building a working demo to show off to investors and customers. This was a great success, Mr Gates said he thought it was 'very cool' and wanted to be a customer, so the company was sold to an American corporation.

They were a largish, established, telecommunications supplier, so the system was to deliver telecom level reliability. The project was blue sky; a paradigm shift in voice messaging. It was to be the next generation system to replace two aging product lines. There was plenty of money to invest, and plenty of time to do it right. This was an ideal opportunity to try a more formal engineering approach.

The project was already staffed with three teams of seven when I joined. We proceeded to write documents for six months. Yes, six months. We wrote functional specifications for the user facing components and detailed design documents for all the systems' components. Each document was based on a standard template to ensure that every engineer considered issues such as performance, testing, and internationalization.

As each document was completed it went to a five-person review team, each person having a distinct role in the review: author, two readers, scribe, and a moderator. The readers would raise defects against the document, which were recorded by the scribe. The moderator ensured that the correct process was followed, and that nobody got too upset. The review process would repeat until all defects had been dealt with to the satisfaction of the readers.

This was very formal, and more than some could take (me included). When no one was looking I would Alt-Tab out of Microsoft Word/Rational Rose and hack some code until I was happy my design was going to work. But this alternating between the abstract and the concrete was a valuable way to ground myself and prevent me from designing a castle in the air.

This is all very well, but did it work? Yes, and no. Yes, the software was developed as designed, and pretty much on time, without any serious unforeseen complications arising. And the software has provided a very solid foundation for the past eight years of functional enhancements. To my continued amazement my original code is still largely intact.

But the project did fail in a couple of respects. Firstly, for political expediency, management, against the advice of engineering, signed off on a poor quality design for one component. None of the engineers wanted to build the component, so a contractor was brought in. He did his best, but at the end of the project we had to junk his work and redesign and rebuild it from scratch. Secondly, the business has never been much of a success. The product still doesn't have many customers. But, it did help the company sell itself, again, to an even bigger company.

That was a natural point for me to move on, and, having had a taste of America, I decided to move to Silicon Valley to do the Internet thing. I thought I had missed the big boom of software development, and didn't see another one coming down the pike. I can recall thinking that $35 was far too much to register a domain name, and, in any case, that would be a pollution of the namespace. How would the inter-network be paid for now that the US government had backed out? By attaching an advert to each packet, or message, or something, we joked. Chuckle. Sigh.

I ended up at Netscape working on a server product. I had never experienced such contrasts between projects. That was where I realised the inverse relationship between code quality and profitability. I went from a four year project with a formally designed object-oriented architecture written in the most advanced C++, which made very little money, to a project with twelve month cycles with little design or expressible architecture written in the most awful C I'd ever witnessed which, of course, made more money than you could imagine. There was little regularity between anything, there was no data encapsulation, everything messed with everything else's privates; it was an orgy of code.

Their success came from moving quickly. If it worked it shipped, it didn't need to look pretty. We had internal customers who were very demanding and vocal. We had a very small and tightly nit team that intimately understood the problem domain. We had a program of continuous build and test cycles. We had no formal process of any kind, just gut feel and rule of thumb. If a concept was too hard to explain on a white board in a couple of minutes, then an email was needed, if that didn't suffice then a document was needed. All team communication, within and between teams, was via mailing lists, and internal web sites. When Kent Beck's XP book appeared the content seemed very familiar to us. [5]

Of course, as the project matured, and grew larger, it became exponentially harder to add new features to the code base. We spent increasing amounts of time rewriting swathes of code to componentise the system, and to try to cram the jinni back into the bottle.

I've now come full circle, and am back to the team of one, working at home, often in the evenings, on distributed semi-structured database systems. Not MUDs this time, but XML databases. Oddly enough, they have quite a lot in common.

Summary

So, what I've learned on this journey is to solve the hardest problems first, and that CASE tools draw nice diagrams. That writing documents works, if you have lots of time, and that having a good architecture in place provides a long-term development foundation. And, finally, that solving real problems is better than writing nice code, and that team communication is critical to project success.

New Reader

Richard Blundell has been dabbling with computers for a quarter century, but has been developing software professionally for half that time. His beginnings were in Basic and 6502/Z80 machine code on Commodore Pets and TRS-80s. He progressed through the likes of Pascal, Fortran, C and even things like PostScript, on PCs, Vaxen and Sparc systems. Today he is Principal Developer for a management consultancy company in Surrey, developing business visualisation software in C++ for the Windows platform. Some of his interests include: security/cryptography, DIY, extreme programming, gardening, algorithms, Linux.

John Merrells
merrells@acm.org

Notes

[1] Structured Systems Analysis and Design Method, a methodology favoured by the public sector in the UK.
[2] Essex Mud, Abermud, TinyMud, LPMud, Hunt, Sun of Bullet.
[3] To my knowledge the phrase 'popular beat combo' was coined by a barrister in response to a judge's query as to who 'The Beatles' were. 'I believe they are a popular beat combo m'lud'.
[4] Rapid Application Development.
[5] Extreme Programming Explained, Kent Beck, Addison Wesley.

Copy Deadline

All articles intended for publication in *Overload 50* should be submitted to the editor by July 1st, and for *Overload 51* by September 1st.

Writing Extendable Software

By Allan Kelly

"A hallmark - if not the hallmark - of good object oriented design is that you can modify and extend a system by adding code rather than hacking it.... In short, change is additive*, not* invasive*. Additive change is potentially easier, more localized, less error-prone, and ultimately more maintainable than invasive change."*

John Vlissides, The C++ Report, February 1998

This is one of my favourite quotes about software development – and I should apologise for mentioning it in more Overload columns than perhaps I should. By the time I originally read this quote I already had several systems under my belt, I had already read the seminal *Design Patterns* (Gamma 1995) and many, many other books so the essence of the quote shouldn't have come as a surprise to me but it did. Vlissides pinpoints and explicitly states something that is only implicit in a lot of writing.

You could view the entire contents of the *GoF* book as recipes for extensible code. Maybe, but it was never spelt out quite so explicitly.

Here, I'd like to spend a bit of time talking about what extendable code means to me, and look at some mechanisms for creating extendable systems.

What is extensible code?

One could argue that all code is extensible because all software is infinitely flexible. If we wished we could change our washing machine control software into a nuclear power station control system – we could do it, but it just isn't the best way to do it.

All software is infinitely modifiable, this is a big big problem because the point at which change is impractical is down to individuals' judgement. The decision is based on one's experience with the software, overall experience of software and your current business environment.

While it is possible to change and modify all software, only software which keeps its original shape and absorbs additions gracefully is truly extensible.

For example, I once worked on an evaluator for electricity futures contracts. To add a new type of contract meant: adding a big chunk of evaluation code, changing the user interface, changing the main control loop, adding a new case statement to half a dozen evaluation routines, and a myriad of minor changes throughout the code base. Many of the evaluation routines looked something like this:

```
double ContractPaymentMultiplier
                    (int contractType) {
  switch (contractType) {
    case 1 : return 1;
    case 2 : return 0.9;
    ....
    default : assert(false);
  }
  return 1;
}
```

True, this is C not C++ code, a properly object-oriented system wouldn't be like this but not all C++ is properly object-oriented[1].

Yes, OO supports extensibility better than procedural code but it doesn't force extensibility.

A system built like this can be changed, you can add to it but it involves intrusive changes in many places. Before you could add anything to this software you had to hack it. To borrow a metaphor from the days of 640K limited MS-DOS[2], the software could be *expanded*, but it could not be *extended*.

An extendable system would allow the new contract to be added without any changes to the existing source code. Realistically, we may have to accept "minimum changes" rather than "any changes" but the point is intrusive changes to existing code should be minimised, let's say three places at most – OK, I just pulled "three" out of the air. One or two would be better, but if we are attempting to separate the system elsewhere (e.g. GUI interface separated from calculation engine) then centralising all changes at just one point may break other abstractions.

If my contract evaluator was written as a truly extendable system I may only need to write a couple of new objects: one to represent the GUI aspects of the contract and one to represent the evaluation. Next I would recompile my system with the new objects – maybe I would need to add them to an existing list of contracts or maybe there is some magic in the make process that would do this for me.

To emphasise the point: the system has been changed without need to change the existing code – even though we may recompile the code the existing code is unchanged.

This may seem obvious when I write it here but stop for a minute and dwell on it. Can you do this with your current system? How would your life be improved if you could make changes like this? What would it mean if your system could be changed like this? How can you do this?

When is code extensible?

There are three points at which code may be extended:

- compile time : change our source code to pick up new functionality; this may mean adding new objects in new source code files and changing a factory function.
- link time : arrange for changes to be picked up by the linker only; this may involve some magic for new objects to be found, this can be self defeating as it inevitably adds some obscurity to the code and possibly the makefiles too.
- run time : dynamically loaded libraries were invented for this sort of thing. This can also lead to obscurity in the code and usually makes debugging more complex because you may have to wait for a library to be loaded before you can set break points.

Although run time extension is truest to the idea of extendable code (because you don't change any of the existing code) I don't think this buys much over a good compile-time extension system. Run-time extension has its uses, such as in very dynamic systems, or non-stop applications but it also complicates version tracking and configuration management.

Sometimes the simplest thing is to actually change some of the existing code. What is simplest and best depends on your circumstances. Actually adding a line and recompiling will be the simplest solution.

---

1 In fact, the system I'm actually talking about was actually written in Pascal.

2 For those who don't recall. MS-DOS was limited to 1Mb of accessible memory and 640K of user space. Initially to get beyond this Lotus, Intel and Microsoft introduced a system of page swapping which allowed memory to be "expanded", think of the memory map getting fatter as different pages were swapped in and out of the 1Mb memory map. Eventually this system gave way to "extended" memory where the CPU could access beyond 1Mb, instead of getting fatter the memory map got taller.

Mechanisms for extending code

Many of the classic design patterns are directly concerned with allowing code to be extended with minimal intrusion. It is easy to see how command, chain-of-responsibility and factory patterns can be useful but patterns are not the end of the story. (If this isn't obvious have another look at the GoF book and think about them for a few minutes.)

Interfaces and substitutability

The key to extensible code is common, well-known interfaces, which allow one object, module or library to be substituted for another – the *Liskov substitution principle* (see Martin 1996). The program framework handles all objects in a common fashion, no special cases are allowed, it is oblivious to the concrete type of the object. The same idea lies at the heart of the *dependency inversion principle* – see Griffiths.

In my extendable contract evaluator example, the framework would ask the contract to evaluate itself, it has no need to know anything about the contract class itself, only the interface for communicating with the contract class.

State of the program – data model

One problem we quickly run into when adding new objects to an existing system is that the objects must have access to the state of the program, that is, the data contained in the system at the current time.

Again, think of the extendable contract evaluator example. Before evaluating any contracts the system will load data models of the supply and demand for the period the contracts are being evaluated for. It is useful to centralise the data model for the system so that all contracts have equal access to the data.

Since the data model is used by all contracts we need to ensure it is accessible. The data model itself may be some easily accessible object, which contains the pre-loaded data and configuration information. All contracts have equal access to this data, there are no special allowances for Contract X to have special access.

Separating the state out also makes it clear what is data, and what is algorithms. This simplifies reasoning about the system.

Dynamically loaded .DLL/.so

Dynamic link libraries, shared libraries in Unix speak, are loaded by an application at run time, often we are only interested in them as libraries not their dynamic properties. However, most OSs allow you to explicitly specify the filename of a library you wish to load and, once loaded, use the functions contained within – this provides a powerful extension mechanism.

You can write several DLLs, each with a common set of functions, and decide which one to load and use at run time, thus you can extend the program at run time.

However, this comes at a cost. Firstly, you must take more care with your version management. Instead of having one large executable to manage you now have several discrete libraries.

Secondly, you must add configuration details to your system so it knows which DLLs to load.

Finally, there are portability problems. On Solaris the action of loading a DLL places all symbols in the run-time symbol table so extra care is required to ensure you don't call a function with the same name in another DLL, while Microsoft traditionally provide a stub library to link against.

If we wish to load a DLL and call a function by name the process is actually quite similar. On Windows we use `LoadLibrary` and `GetProcAddress`, while on Unix we use `dlload` and `dlsym` to the same effect.

COM & CORBA

Both COM and CORBA can be used to for extensible systems. However, the literature on both emphasises different aspects of each system. Essentially, both implement the loading of dynamic libraries.

If your system already uses, or you plan to use either COM or CORBA you can take full advantage to make your program extensible. However, if you are only interested in their extensibility properties I would advise against using either of them. There are simpler techniques (some outlined here) which provide the same benefit without the cost.

When I say cost I'm not talking about monetary cost – although simply buying the literature on either product is expensive, and purchasing a brand name ORB is not cheap – rather I am thinking of:

- both have steep learning curves : even if you know COM think of the maintenance requirements
- both have a reputation for poor performance
- both force you to design your system around them
- both have reference counting problems
- both force you to get into IDL writing which may be overkill
- COM locks you into Microsoft systems
- CORBA code can become specific to one vendor's ORB if care is not taken

Poor-man's COM

Even without using COM or CORBA you can pass objects out of dynamic libraries you have loaded. All that is required is three steps:

1. Simply define an abstract base class, e.g.

```
class Base {
public:
  virtual bool Action() = 0;
};
```

2. In each of your dynamic DLLs provide a concrete implementation of this base class, e.g.

```
class Concrete : public Base {
public:
  virtual bool Action() { return true; }
};
```

3. In each DLL provide a `Factory` function which returns a pointer to your `Base` class, e.g.

```
Base* Factory() { return new Concrete; }
```

You can now write as many objects as you like, each packaged inside a DLL, and choose which to load at run time.

Of course, should you decide to change the interface on the `Base` class you will need to recompile everything in your system. This is equally true if you change the IDL interface on a COM or CORBA class.

Exception handling

It may not be obvious at first that exception handling has a part to play in writing extensible code but it does, a very important part.

In the days before exception handling we typically had one file with a large number of error codes in it, such as `ErrorCodes.h`[3].

3 On a side note I urge everyone to avoid using the word "error" in filenames. Grepping a long compiler log for errors is much easier if there are no false positives.

Whenever a new error was added `ErrorCodes.h` needed updating and, since every file in the system depended on `ErrorCodes.h`, the entire system needed re-compiling.

By defining a hierarchy of exception classes derived from a simple base we can allocate error codes and messages as needed. We may still wish to provide each object with its sub-system code.

When a simple error code is passed up the call stack it is difficult for the top-level code to take any special action without knowing specifics about the circumstances. Contrast this with an exception class, which can itself provide specific methods for such a circumstance.

For example:

```
class ContractEvaluatorException
                : public std::exception {
public:
  virtual int SubSystem()=0;
  virtual int SpecificCode()=0;
  virtual const char* ExtendedDescription()=0;
  virtual bool EvaluationComplete()=0;
  virtual void StoreEvaluation()=0;
  virtual bool IsFatal()=0;
};
...
int EvaluateAll(std::list<Contract> contracts,
                DataModel& dataModel) {
  for (int i=0; i<=contracts.size(); i++) {
    try { contracts[i].Evaluate(dataModel); }
    catch (ContractEvaluatorException& exp) {
      cerr << exp.what()
        << " in subsystem " << exp.SubSystem()
        << " code = " << exp.SpecificCode()
        << ": " << exp.ExtendedDescription();
      LogError(exp);
      if (exp.IsFatal()) throw;
      if (exp.EvaluationComplete()) {
        exp.StoreEvaluation();
      }
    }
  }
}
```

The higher levels of the program are still ambivalent to what was actually happening – beyond the fact that some contract was being evaluated. Again, the exception system has allowed us to separate the cause from the effect (see Kelly, 2000) - this is dependency inversion at work.

State machines

State machines are particularly good at absorbing extra code. The simplest state machines (a big switch statement and a whole set of functions) can have extra states added with little pain but beware, beyond a certain point the big-switch statement becomes a pain to maintain.

More advanced state machines can be completely reconfigured at run time and may use look up tables rather than hard coded settings. Equally, I have read several articles on object based state machines over the years.

One of my favourite features of state machines is that they are very easy to debug and to explain to users. You can sit down with a piece of paper and trace the route against a diagnostic printout, or with a user who wants a change.

Putting it all together

This article draws heavily on my own experience. In these kind of extendable systems I frequently find a large number of "action objects." These are C++ classes which exist for one purpose only, indeed, as in the example above they may have just one significant method called *Action()*.

To keep these objects decoupled they are usually passed a means of accessing the program state when they are *action'ed.* These objects are ideal candidates for being placed in a queue and processed sequentially. Sometimes the processing order is important, sometimes the processing could be farmed out to worker threads to allow several objects to be *action'ed* at the same time.

Another characteristic is that the actioning of the objects may further populate the queue of objects to be action'ed. Sometimes this is direct, the action method will actually add a new item to the processing queue, other times it is indirect, the action method will trigger some other process which results in the queue being populated.

In fact, what I have just described is the *Command* pattern in a slight disguise.

Example code

By the time this article appears I should have some example code available on my web-site – www.allankelly.net. This demonstrates the use of dependency inversion to allow extensions to the code and poor-man's COM system. At the moment the code compiles on Windows 2000 using either Visual C++ or GCC. Overtime I would like to extend this code in several directions.

Summary

Extensibility is a worthy design goal. It is the goal of many design patterns and development techniques but it is seldom stated explicitly. Once we recognise extensibility as an explicit aim it is not rocket science. There are a variety of mechanisms for implementing it and with a little practice it becomes easy.

Of course even the most extensible systems suffer from sods-law - change requests can always occur for items your didn't expect to need changing – the classic *outside-context-problem*[4].

Allan Kelly
Allan.Kelly@bigfoot.com

References

Banks, Iain. 1996; *Excession*, Orbit, 1996

Bruntlett, Ian 2000; "User Defined Types: Qualities, Principles & Archetypes", *Overload 39*, September 2000.

Gamma, E., Helm, R., Johnson, R., Vlissides, J., 1995; *Design Patterns*, Addison Wesley 1995 - also called the *Gang of Four* book or *GoF* for short.

Griffiths, Alan; "Dependency Inversion", www.octopull.demon.co.uk/c++/dependency_inversion.html

Kelly, Allan 2000: "Error Handling and Logging", *Overload 35*, January 2000

Martin, R.C. 1996; "Liskov Substitution Principle", *C++ Report*, 1996, www.objectmentor.com/resources/articles/lsp.pdf

4 From Iain Banks' novel "Excession" (1996) – although as far as I know Ian Bruntlett (2000) was the first to use Outside Context Problem in connection with software.

More Exceptional Java

by Alan Griffiths

At the ACCU Spring conference last year I took my exception-safety pattern language [1] and redrafted the discussion of C++ idioms in Java. This wasn't as simple as I had hoped, many of the C++ techniques used have no equivalent in Java. The resulting presentation met the goal of relating the pattern language to Java and identified the necessary coding techniques. However, as Jon Jagger commented, "it lacks the same sense of resolution" that the C++ paper had.

The problem was that it raised a number of unanswered issues regarding the use of Java exceptions. During the course of the presentation these were discussed (with a lot of input from the audience) without reaching a solution. The only clear conclusion I could reach was that these were real problems that people were experiencing.

A year has passed and now I have tried to address the issue of how Java exceptions should be used in the companion article "Exceptional Java" [2] and it is time to returned to the original theme: that of writing exception safe code.

These two articles can be read separately, but each raises issues that are addressed by the other. One final point, although I arrived at the ideas presented in "More Exceptional Java" a year before writing "Exceptional Java" I would recommend reading it second.

A metaphorical landscape: the heights of exception safety

To give you some orientation in the exception-safety landscape I will first describe the principle landmarks: the heights of exception safety. These are the goals that we will be seeking during the discussion of implementation techniques, so if you are prone to asking, "why are we doing this?" it will be worth assuring yourself that you understand them first.

Let us begin with a program in which an exception is thrown and consider the call stack: a method `a` has called another method `b`, `b` has called `c`, and so on, until we reach `x`; `x` encounters a problem and throws an exception. This exception causes the stack to unwind, executing the code in `finally` (or `catch`) blocks, until the exception is caught and handled by `a`.

I'm sure that the author of `x` has a perfectly good reason for throwing an exception (running out of memory, disk storage, or whatever) and that the author of `a` knows just what to do about it (display: "Sorry, please upgrade your computer and try again!").

But exception safety is not about writing code that throws an exception (method `x`) nor about writing code that handles it (method `a`).

Exception safety is about writing the code that lies between - writing all the intervening methods in a way that ensures that something sensible happens when an exception propagates. It is about writing the typical method `m` in the middle of the call stack. If we want `m` to be "exception safe" then how should it behave?

Consider the options: If `m` completes its task by some other means (by using a different algorithm, or returning a "failed" status code) then it would be handling the exception. That isn't what we are concerned with - we are assuming the exception won't be handled until we reach `a`. Since `m` doesn't handle the exception we might reasonably expect that:

1. `m` doesn't complete its task.
2. If `m` has opened a file, acquired a database connection, or, more generally; if `m` has "allocated a resource" then the resource should not leak. The file must be closed, the database connection must be released, etc.
3. If `m` changes a data structure, then that structure should remain useable.

In summary: if `m` updates the system state, then the state must remain usable. Note that isn't quite the same as correct, for example, part of a name-and-address object may have been changed leaving mismatched name and address values.

These conditions are called the *basic exception safety guarantee*. Take a good look at it so that you'll recognise it later.

If you are new to this territory then the basic exception safety guarantee may seem daunting. But not only will we reach this in our travels, we will be reaching an even higher peak called the *strong exception safety guarantee* that places a more demanding constraint on `m`:

4. If `m` terminates by propagating an exception then it has made no change to the state of the program.

The basic and strong exception safety guarantees were first described by Dave Abrahams [3] to document an implementation of the C++ standard library.

Note that it is impossible to implement `m` to deliver either the basic or strong exception safety guarantees if the behaviour in the presence of exceptions of the methods it calls isn't known. This is particularly relevant when the client of `m` (that is `l`) supplies the methods to be called either as callbacks or as implementations of virtual member methods. In such cases the only recourse is to document the constraints on them.

If we assume a design with fully encapsulated data then each method need only be held directly responsible for aspects of the object of which it is a member. For the rest, the code in each method must rely on the methods it calls to behave as documented. (We have to rely on documentation in this case, since in Java, as in C++, there is no way to express these constraints in the code.)

As we proceed we'll find frequent outcrops of the rock that supports our mountains, which is named after the area in which it is found and is called the *no-throw exception safety guarantee*, as the name suggests this implies that the corresponding function will never propagate an exception. Clearly operations on the fundamental types provide this guarantee, as will any sequence of no-throw operations. If it were not for the firm footing that these outcrops of the no-throw exception safety guarantee provide we would have great difficulty ascending the heights. Although this was known and used in earlier work I think it was first made explicit and named by Herb Sutter [4].

Do forgotten objects live on?

In Java objects are in a very real sense immortal - they do not die - and are merely forgotten when there are no references to them. This has a number of obvious advantages - in particular, one cannot have a reference to an object that no longer exists (this is a major element of the Java security model).

However, the basic and strong exception guarantees both refer to "the system state" and it may not be obvious whether objects awaiting garbage collection should be considered part of this. These 'forgotten' objects cannot be accessed and I propose to ignore them, they are not part of the system state.

Java exceptions

Java has both checked and unchecked exceptions and the companion article "Exceptional Java" examines the consequences of this. For the purposes of this article I claim that both checked and unchecked exceptions must be considered when reviewing code for exception safety. A method that doesn't catch an exception doesn't care about the type of the exception, particularly whether it is checked or unchecked, so `throws` clauses are of little account when reasoning about exception safety.

Why must unchecked exceptions be considered? Because they can be thrown by `x` and caught by `a`. And the code that catches them requires guarantees about the state of the system. If, for example, it is going to restart the subsystem that encountered the problem, then it needs to know that the subsystem died in an orderly manner.

Throughout this article whenever I mention exceptions without qualification I mean it in the inclusive sense, "either checked or unchecked exception".

Not considering unchecked exceptions is a frequent cause of errors. Two factors contributing to this are: a tendency for developers to rely on the compiler to indicate any failure to consider exceptions, and that error handling is often omitted from demonstrative code – as it obscures the point. Consider the following example (widely quoted as a way to avoid flicker when using the AWT):

```
public void repaint() {
  Graphics g = getGraphics();
  paint(g);
  g.dispose();
}
```

It is possible for an unchecked exception to propagate from `paint(g)`, which would bypass the `g.dispose()` statement. It is important that this doesn't happen in a real application as it releases system resources. It is not sufficient to assume that the `finalize` method of the object formerly known as `g` will release the resources as there is no guarantee that the `finalize` method is called in a timely manner, or indeed ever.

In Java the only way to guarantee that a method on an object is executed is to call it before the object is forgotten. In code written for a production environment I'd expect to see:

```
public void repaint() {
  Graphics g = getGraphics();

  if (null != g) {
    try { paint(g); }
    finally { g.dispose(); }
  }
}
```

In this code fragment it should be obvious that the resources will be released whether or not `paint()` propagates an exception - look at the three significant steps: allocate, use, release and the logic that binds them together.

While on the subject of `finally`, please use it idiomatically: while it is legal to exit from a finally-block using `return`, `continue` or `break` doing so defies the 'principle of least surprise'. The next programmer to work on the code (it may be you) will expect that the exception continues to propagate. If the exception may be handled say so: by using `catch`!

Factoring out 'Allocate-Use-Release'

While repeatedly reproducing the same code in slightly different contexts may be good for productivity by metrics such as lines-of-code it is tedious and a possible source of errors. Wherever paired operations (such as allocation and release) need to surround a piece of code it may be useful to employ the Execute-Around-Method idiom [6]. This exists in a number of languages and in Java it is expressed by passing an anonymous local class to a method that allocates the resource, passes it to a method in the supplied code, and finally release the resource:

```
public class ExecuteAroundMethod {
  private ListSelectionModel sel;

  private interface Adjustment {
    public void use(
              ListSelectionModel sel);
  }

  private void executeAround(
              Adjustment adjustment) {

    sel.setValueIsAdjusting(true);
// 'allocate'

    try {
      adjustment.use(sel);
// 'use'
    }
    finally {
      sel.setValueIsAdjusting(false);
// 'release'
    }
  }

  public void removeOdd() {
    executeAround(new Adjustment() {
      public void use(
               ListSelectionModel s) {
        for (int i = entries; –i != 0;)
          if (0 != i % 2)
            s.removeIndexInterval(i, i);
      }
    });
  }
}
```

There is a trade-off here, we've ensured that there is only one method to examine to ensure that paired operations always occur correctly, but we've paid a price by introducing extra classes and method calls.

Strong exception safety guarantee – the frontal route

We are now going to look at strongly exception safe version of a simple method. The following example is a translation from C++ of an example introduced as a test case by Tom Cargill. The method we will examine, `copy()`, is one that aims to copy the state of `source` to `this`:

```
public class Whole {
  private static class PartOne
              implements Cloneable {
    /* omitted */
  }
  private static class PartTwo
              implements Cloneable {
    /* omitted */
  }

  private PartOne p1;
  private PartTwo p2;

  public void copy(Whole source) {
    /* What goes here? */
  }
}
```

A (very naïve) implementation might be:

```
public void copy(Whole source) {
  p1 = rhs.p1.clone();
  p2 = rhs.p2.clone();
}
```

Is this exception safe?

If we make the reasonable assumption that the `clone()` methods are themselves strongly exception safe then `copy()` supports the basic guarantee. (This is also the case if the `clone()` methods support the basic guarantee.) If the `clone()` methods are not exception safe there is little that `copy()` can do to achieve exception safety.

Only if the `PartTwo.clone()` method called in the second line won't throw an exception can this version of `copy()` support the strong guarantee (after modifying `p1` the system state has definitely changed). Of course, the nothrow guarantee is an unreasonable expectation of a `clone()` method, an 'out of memory' exception is a possibility for any reasonable implementation. On the other hand there is nothing to indicate that modifying `p1` alone will make the object unusable so we can claim to meet the basic guarantee.

Now consider a slightly updated version that addresses the problem of an exception being thrown by the second `clone()` call:

```
public void copy(Whole source) {
  PartOne temp = rhs.p1.clone();
  p2 = rhs.p2.clone();
  p1 = temp;
}
```

With the presumption that cloning does nothing that needs reversing then an exception propagating from either of the first two lines permits any changes to the system to be forgotten on exit from `copy()`. This is the strong guarantee.

In the above we've assumed that cloning doesn't do anything that needs to be undone. This isn't always true – for example `PartOne` and `PartTwo` may own a resource that needs releasing. Adding the complexity of ensuring `dispose()` methods are called and generalise slightly (I wouldn't normally expect to need the checks against `null` for `clone()` but other methods used in this context might):

```
public void copy(Whole source) {
  PartOne t1 = rhs.p1.clone();

  if (null != t1) {
    try {
      PartTwo t2 = rhs.p2.clone();

      if (null != t2) {
        try {
          // examples of methods
          // that might throw
          t1.setParent(this);
          t2.setParent(this);

// ******************************
// This is the pivotal point of the
// code - everything that could
// fail is before this point.
// Nothing that makes persistent
// changes to the state of the
// system is before this point.
// ******************************

          // The following commits the
          // change to the system state.
          // Importantly it won't throw.
          PartOne swap1 = t1;
          t1 = p1;
          p1 = swap1;
          PartTwo swap2 = t2;
          t2 = p2;
          p2 = swap2;
        }
        finally {
          // either frees the original
          // resources or of the
          // temporary - depending
          // whether we passed the
          // pivot uneventfully.
          t2.dispose();
        }
      }
    }
    finally {
      // either frees the original
      // resources or of the temporary
      // - depending whether we
      // passed the pivot uneventfully.
      t1.dispose();
    }
  )
}
```

The code is structured in such a way that for each object creation that succeed the `dispose()` method will be invoked. It will be invoked on either the original instance if no exception is propagated or the replacement instance in the case of an exception. The only assumptions needed to demonstrate the strong guarantee being attained are: no exceptions are propagated by the `dispose()` calls and that the `setParent()` calls are themselves exception safe.

This code structure once again shows the ALLOCATE-USE-RELEASE idiom we observed earlier with a subtle variation, if no error occurs then it's a different resource that is released. This has been described before in a C++ context[5] and goes by the name ALLOCATE-BEFORE-RELEASE. (Strictly, the original reference refers to COPY-BEFORE-RELEASE, a special case equivalent to the current example.)

A close examination of the above example should make it clear that when committing updates to the system state we need operations that don't throw exceptions. The assignments of references used to swap the new state into the object are obviously safe – they are guaranteed not to throw by the language, but if either call to `dispose()` were to throw an exception then the strong guarantee would be violated.

This is an area in which Java standard library documentation is deficient, as it only addresses checked exceptions. Consider the method `Graphics.dispose()` used in the first example: could this propagate an unchecked 'null pointer' exception? I hope not - but without documentation of this point we don't know.

A lower peak – the basic guarantee

On terrestrial mountains above a certain height there is a "death zone" where the supply of oxygen is insufficient to support life for long periods. Something similar happens with exception safety: there is a cost to implementing the strong exception safety guarantee. The technique illustrated above can involve the creation of extensive duplicate data structure: the additional objects created and the resources they allocate can be expensive in both space and time. If repeated at all the levels of our call stack the costs of doing this can suffocate an application.

The alternative to changing a copy of an object is to change the original and either accept that an exception could leave a series of changes incomplete. The result is that either the system will be in an unknown state or we must be prepared to back out changes. For either approach what we need to know is that nothing will go horribly wrong, the basic exception safety guarantee.

To provide an example I'm going to elaborate on the previous example by extending the class and adding a (potentially) large container to the derived class:

```
public class BiggerWhole extends Whole {
  private Properties parameters
                    = new Properties();

  public void copy(BiggerWhole source) {
    /* What goes here? */
  }
}
```

Using the techniques we examined earlier we would take a clone of `source.parameters` then call `super.copy()`, and finally update `parameters`. If we decide that the cost of creating a copy of `parameters` is unacceptable then we can update it with the understanding that if an exception occurs then we make no promise to the client code of the exact state of `BiggerWhole`, the client code must take appropriate action. Vis:

```
public void copy(BiggerWhole source) {

  super.copy(source);
  parameters.clear();

  Enumeration e =
       source.parameters.propertyNames();

  while (e.hasMoreElements())
  {
    String key = (String)e.nextElement();
    parameters.put(key,
     source.parameters.getProperty(key));
  }
}
```

At any point after `super.copy()` returns then the state of the system has changed. However an exception is still possible. However, if failing to update `parameters` completely leaves the object in a sensible state then this may be acceptable.

Conclusion

The exception safety landmarks are useful in Java, as they are in C++. The basic, strong and nothrow guarantees clearly make sense and can be applied when writing or reviewing code. There are techniques for writing code to these guarantees and these have been demonstrated. While I must agree with those C++ developers who consider these techniques less elegant than those available in C++ I see the principal issue to be the failure of a significant part of the Java community to believe that the problem they solve exists.

Ignoring a problem does not make it go away and as unchecked exceptions can encapsulate rare but plausible events (e.g. out of memory) and even exceptions explicitly thrown by the programmer it is unreasonable to ignore them. It is unfortunate that they also include that shouldn't happen in a correct program.

The path that led Java to the current handling of unchecked exceptions is paved with good intentions: rather than the JVM having undefined behaviour when "bad things" happen the behaviour is defined. But this has only shifted the problem, because the programmer is not working with the raw JVM, but with library code that doesn't fully document its behaviour.

In addition to unchecked exceptions being undocumented, there are no compile-time tools for verifying exception safety. This once again leaves the problem with the developers, who must document and check the requirements for themselves.

Alan Griffiths
alan@octopull.demon.co.uk

References

[1] Alan Griffiths, "Here be Dragons", *Overload 40*
[2] Alan Griffiths, "Exceptional Java", *Overload 48*
[3] Dave Abrahams, *Exception Safety in STLPort*, http://www.stlport.org/doc/exception_safety.html
[4] Herb Sutter, *Exceptional C++,* Addison-Wesley, ISBN 0-201-61562-2
[5] Kevlin Henney, "Self Assignment? No Problem!", *Overload 21*
[6] Kevlin Henney, *Java Patterns and Implementations*, http://homepages.tesco.net/~jophran/UKPatterns/plunk1/JavaPatterns.html

Programming with Interfaces in C++

By Chris Main

In a previous issue of Overload [1], Lois Goldthwaite gave an illuminating explanation of compile time polymorphism using templates.

This has the advantage over run time polymorphism that it does not require classes to be derived from a common base class to share an interface, and that it does not incur the cost of a virtual function table.

It is usually pointed out that it suffers the disadvantage that objects of different types with the same interface cannot be held in a type safe container, since such containers do require contained objects to share a common type.

However we can in fact get the best of both worlds since it is possible to convert objects that share compile time polymorphism into objects that share run time polymorphism. I infer from Kevlin Henney's article in Overload 48 [2] that the technique I describe uses the External Polymorphism pattern.

First, a recapitulation of Lois' classes:

```
// talkers.h (include guards not shown)
#include <iostream>

class Dog {
  public:
    void talk() const {
      std::cout << "woof woof"
                << std::endl;
    }
};

class CuckooClock {
  public:
    void talk() const {
      std::cout << "cuckoo cuckoo"
                << std::endl;
    }
};

class BigBenClock {
  public:
    void talk() const {
      std::cout
          << "take a long tea-break"
          << std::endl;
    }
    void playBongs() const {
      std::cout << "bing bong bing bong"
                << std::endl;
    }
};

class SilentClock {
};
```

We can provide compile time polymorphism by means of a function template, which can be specialised to adapt functionality (in the case of `BigBenClock`) and to supply missing functionality (in the case of `SilentClock`):

```
// talkative_generic.h
// (include guards not shown)

class BigBenClock;
class SilentClock;

template< class T >
void talkativeGenericTalk(const T& t) {
  t.talk();
}

template<>
void talkativeGenericTalk(const
             BigBenClock& bigBenClock);

template<>
void talkativeGenericTalk(const
             SilentClock& silentClock);

// talkative_generic.cpp
#include "talkative_generic.h"
#include "talkers.h"
#include <iostream>

template<>
void talkativeGenericTalk(const
             BigBenClock& bigBenClock) {
  bigBenClock.playBongs();
}

template<>
void talkativeGenericTalk(const
             SilentClock& silentClock) {
  std::cout << "tick tock"
            << std::endl;
}
```

The equivalent interface in run time polymorphism can be defined as:

```
// talkative_interface.h
// (include guards not shown)

class TalkativeInterface {
  public:
    virtual TalkativeInterface*
                              clone() const = 0;
    virtual void talk() const = 0;
    virtual ~TalkativeInterface(){}
};
```

We then need a mechanism for converting objects with the compile time interface to this type. We can do this by means of a factory class:

```
// talkative_interface_factory.h
// (include guards not shown)
#include "talkative_interface.h"
#include "talkative_generic.h"
```

[*continued over page*]

[*continued from previous page*]

```
class TalkativeInterfaceFactory {
  public:
  // Callers should take ownership of the
  // pointers returned by the public functions

    template< class T >
    static TalkativeInterface* convert(T* t) {
      return new TalkativeImplementation<T>(t);
    }

    template< class T >
    static TalkativeInterface*
                    copy(const T& t) {
      return new TalkativeImplementation<T>(t);
    }

  private:
    TalkativeInterfaceFactory();
    TalkativeInterfaceFactory(const
                 TalkativeInterfaceFactory&);
    TalkativeInterfaceFactory& operator=(
            const TalkativeInterfaceFactory&);
    ~TalkativeInterfaceFactory();

    template< class T >
    class TalkativeImplementation
              : public TalkativeInterface {
      public:
        TalkativeImplementation(T* t)
               : t_(t), owner_(false) {}

        TalkativeImplementation(const T& t)
               : t_(new T(t)), owner_(true) {}

        virtual TalkativeInterface*
                          clone() const {
          return new
              TalkativeImplementation<T>(*t_);
        }

        virtual void talk() const {
          talkativeGenericTalk(*t_);
        }

        virtual ~TalkativeImplementation() {
          if(owner_) delete t_;
        }

      private:
        TalkativeImplementation(const
                    TalkativeImplementation&);

        TalkativeImplementation& operator=(
              const TalkativeImplementation&);

        T* t_;
        bool owner_;
    };
};
```

The key to the factory class is the nested adapter class template `TalkativeImplementation` in the private part. This is derived from `TalkativeInterface` to give it the required type. It is a template so that it can be instantiated with any class `T` that supports the compile time polymorphism; the required member function `talk()` is implemented by using the function template `talkativeGenericTalk()`.

The factory provides two static public functions `convert()` and `copy()` which use the `TemplateImplementation` class template to perform the conversion (in the former by sharing an existing pointer, in the latter by creating a new, independent one).

For the sake of clarity I have used raw pointers throughout the code. In practice I would use a reference counting smart pointer everywhere there is a raw pointer. This would avoid the need for the `owner_` member variable and make the destructor of `TalkativeImplementation` trivial. For a full discussion of smart pointers see Alexandrescu [3]).

Here is some test code to show the use of the factory to build a vector of talkative objects using `copy()`:

```
// test_talkative.cpp

#include "talkers.h"
#include "talkative_interface_factory.h"
#include <vector>
#include <algorithm>

namespace
{
  typedef std::vector<TalkativeInterface*>
      Talkers;

  struct Talk {
    void operator()(const
             TalkativeInterface* talker) {
      talker->talk();
    }
  };

  template< class T >
  struct Destroy {
    void operator()(const T* t) {
      delete t;
    }
  };
}

int main(int, char**)
{
  Dog aDog;
  CuckooClock aCuckooClock;
  BigBenClock aBigBenClock;
  SilentClock aSilentClock;

  Talkers talkers;

  talkers.push_back(
       TalkativeInterfaceFactory::copy(aDog));
```

```
  talkers.push_back(
        TalkativeInterfaceFactory::copy(
                              aCuckooClock));

  talkers.push_back(
        TalkativeInterfaceFactory::copy(
                              aBigBenClock));

  talkers.push_back(
        TalkativeInterfaceFactory::copy(
                              aSilentClock));

  std::for_each(talkers.begin(),
                talkers.end(), Talk());

  std::for_each(talkers.begin(),
                talkers.end(),
                Destroy<TalkativeInterface>());
  return 0;
}
```

This produces the output:

```
woof woof
cuckoo cuckoo
bing bong bing bong
tick tock
```

One practical application of this factory class would be to implement the Observer pattern [4]. In fact, it was Pete Goodliffe's series of articles on that subject [5] that set me thinking about this in the first place. In this pattern, a Subject maintains a list of Observers. With this factory class we would be able to convert a variety of Observers, not necessarily sharing a common base class, so that they could be added to such a list. The `convert()` operation would be used in this case. The following test code demonstrates the effect achieved by using `convert()` rather than `copy()`:

```
// guard_dog.h (include guards not shown)

#include "talkers.h"
#include <iostream>

class GuardDog: public Dog {
  public:
    GuardDog() : barkLoudly(false) {}

    GuardDog& deterIntruder() {
      barkLoudly = true; return *this;
    }

    void talk() const {
      if(barkLoudly) {
        std::cout << "WOOF WOOF" << std::endl;
      }
      else {
        Dog::talk();
      }
    }
  private:
    bool barkLoudly;
};

// test_talkative.cpp
// other includes as before

#include "guard_dog.h"

// typedef Talkers, classes Talk and
// Destroy as before

int main(int, char**) {

  GuardDog* aGuardDog = new GuardDog();

  Talkers talkers;

  talkers.push_back(
        TalkativeInterfaceFactory::convert(
                                  aGuardDog));

  std::for_each(talkers.begin(),
                talkers.end(), Talk());

  aGuardDog->deterIntruder();

  std::for_each(talkers.begin(),
                talkers.end(), Talk());

  std::for_each(talkers.begin(),
                talkers.end(),
                Destroy<TalkativeInterface>());

  delete aGuardDog;
  return 0;
}
```

This produces the output:

```
woof woof
WOOF WOOF
```

Chris Main
chris@chrismain.uklinux.net

References:

[1] Lois Goldthwaite, "Programming With Interfaces In C++: A New Approach", *Overload 40*, December 2000
[2] Kevlin Henney, "Function Follows Form", *Overload 48*, April 2002
[3] Andrei Alexandrescu, *Modern C++ Design*, Addison Wesley C++ In Depth Series, 2001
[4] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995
[5] Pete Goodliffe, "Experiences of Implementing the Observer Design Pattern Parts 1-3", *Overload 37*, May 2000, *Overload 38*, August 2000, *Overload 41*, February 2001

Building Java Applications

By Vaclav Barta

This is an article about the boring process of running `javac` on Java source files to obtain Java class files. It explains that it's not so simple (when you come down to it, it's actually pretty complicated), what has been done to make it seem simple and what, in my opinion, should be done instead.

Dependency management

So, what's so complicated about running `javac`? Well, making class files may be the most obvious part of a Java build, but it's rarely the only one. In my experience (which is admittedly limited, but I did work on more than one Java project), Java projects are typically multi-language: more often than not, there's some legacy library which must be accessed through native code, a CORBA IDL interface to be implemented, an XML DTD to conform to and so on.There are many code generators producing Java, and therefore much code to be generated before compilation; and when the class files are compiled, they must be packaged into `jar`s (which may need to be signed), and then there's no limit to how complicated you want your automated *testing* to be. I subscribe to the adage that "the program that has not been tested does not work", and even if you don't, reflect on the dynamic nature of Java (runtime checks for null pointers, casts and so on) and ask yourself whether you really want your users to be the first running your code... And if you want to test what is delivered (as opposed to some other, quite similar code), you need version control, and that should also mesh with your build. Builds are hard, and Java builds are harder than average.

Of course, if a build does a lot of things, it shouldn't do them every time. Not only do I not want to run integration tests every time I change a line - I don't even want to package the `jar`s. I want to run the unit tests early and often and they must use class files corresponding to my source files. Therefore the build must figure out what needs to be updated (whether it's one class file, or every class using a changed constant) and build that and no more. In other words, a serious build for Java applications must understand Java dependencies. That is the problem; let's look at some existing solutions.

Current solutions

IDEs

The most common answer I get when asking about a project build is, I'm sorry to admit, "Why don't you just use JBuilder?" - so my answer is practiced by now: JBuilder doesn't scale. I believe (I didn't try) that it may be useful for a home-based individual developer, but an environment which does not handle multiple languages, multiple users and variant (at least unit test vs. delivery) builds is simply not good enough for production work. That should not be taken as criticism of JBuilder (in fact, I'm told that some of its more irritating limitations have been fixed in latest versions), but rather as a criticism of "visual" environments in general. Large-scale software development requires *programmable* tools, going beyond "what you see is all you get".

make

And indeed there are programmable build tools. The problem of dependency management has been recognized (and solved) long ago, and its solution is now a de-facto standard build tool: `make`. From a popular manual [1]: "The `make` utility automatically determines which pieces of a large program needs to be recompiled, and issues commands to recompile them."

It would be perfect, if only it were true... Standard `make` does not automatically handle dependencies (not beyond "`$NAME.o` always depends on `$NAME.c`"), and in consequence, projects using `make` either generate most of the content of their makefiles by various add-ons (either `make` extensions, or external programs), or just ignore the problem and build from scratch every once in a while to flush the bugs out. A good overview of `make` problems can be found (perhaps unsurprisingly) in the documentation of an alternative build tool [2].

And if the general problems of `make` weren't enough to look for alternatives, there are also Java-specific ones: `make`'s build model is based on C compilation (source files are compiled to object files, which are linked to make an executable), but Java doesn't work that way. Multiple class files can be generated from a single source, circular dependencies are quite common and the generation of `jar`s is strictly optional.

Considering that the purpose of dependency analysis is to minimize compilation time, it is worth noting that `javac` is *very* slow to start up. For a few simple classes, it may not pay to check dependencies at all - the startup time (or perhaps it is time `javac` itself spends checking dependencies - see below) dwarfs time actually spent compiling, and so it saves practically no time to compile just part of the project.

On the other hand, there clearly are projects big enough that `javac` should not compile all their sources, every time - there's no justification for build times going up linearly with the size of the project. In theory, `javac` could handle dependency management internally. In practice, that functionality (the `-depend` flag of `javac`) existed in early versions of Sun's JDK, but it has been dropped in Java 2, and `javac` now recompiles only direct out-of-date dependencies, when their sources happen to be found on `sourcepath` or `classpath`. `Sourcepath` at least is an explicit command-line argument, but the recompilation of source files on `classpath` cannot even be turned off... `Make` has a particular problem with this "feature", because `make` builds normally create derived files in the same directory where the source is, and since `javac` must be able to find the class files (to compile other sources depending on them), it also checks the sources even when it shouldn't.

A model for a Java build

So, if the current solutions are unsatisfactory, what should be done instead?

I believe that a Java-aware build tool should have language-specific support (not only for Java, but for multiple languages - at least C/C++) and that the support should be comprehensive.

The tool should know which class files are generated from which sources (this is important for further derivation, i.e. making `jar`s), and which class files are required to compile each source file.

The tool should handle circular dependencies, and even when there aren't any, it should compile more than one updated source file at once (by default probably all sources in one directory).

Overall, it's a lot of "should" - are these requirements realistic?

Enumerating derived files

In the simplest case, a Java source file defines one class; if that class is public, it must have the same name as the file. But not all classes are public, and non-public (top-level) classes can be defined in any source file declaring classes of a given package. Java also has inner classes, defined inside a top level class and not necessarily having any name at all. As the JVM spec [3] says, "Typically, a class or interface will be represented using a file in a hierarchical file system. The name of the class or interface will usually be encoded in the pathname of the file." - but there are no guarantees, and certainly no published algorithm to derive the class file name. Sun's "anywhere" (as in "run anywhere") just doesn't seem to include Java environments from other providers...

But at least the package part is clear: a package corresponds to a relative directory. The absolute, top-level directories prepended to the relative ones are listed (together with `jar`s - in this context, a `jar` is just a directory abstraction) in `classpath`, which is specified by a command-line argument (for command-line tools) or a user-settable option (for GUI applications), an environment variable or some hardcoded default (good only for the system classes). Pretty much everybody handles `classpath` the same way.

For the name part, let's get empirical: what class file names are actually created by Sun's `javac`? It appears that all inner classes have names constructed from multiple segments, separated by '`$`', where the first segment is the name of their enclosing class. Named inner classes (i.e. the only inner classes which can be referenced from files other than the one defining them) simply concatenate all names of their enclosing classes - for example, a class whose Java name is:

```
pkg.name.TopLevel.Inner.NestedInner.
                          DeeplyNestedInner
```

is compiled into a file

```
TopLevel$Inner$NestedInner$
                    DeeplyNestedInner.class
```

(in the `pkg/name` directory). Anonymous classes use two or three segments only. Unnamed classes have names constructed from two segments, while local classes (i.e. named classes declared in some method) need three segments (because local classes with the same name may be declared in different blocks). When there's no good name for a segment, a number is used instead; the numbers start from 1 and go up (when necessary to keep the whole name of every defined class unique) as the source is being parsed.

Overall, it seems possible to pry open the black box and get the class file names generated from a given source - provided we have a Java parser. Fortunately, Antlr is Java-based and (like any self-respecting parser) parses its own programming language, so a utility for this task is within reach.

Java source dependencies

When parsing Java sources, it is naturally also possible to notice the used class names and construct a dependency graph. The hard part is determining what is a class name (as opposed to, say, field name - see Section 6.5 of the Java spec [5]). Basically, a dependency analyzer must know about all classes (on the `classpath`, *and* classes that may not exist yet, but are defined in the project's sources) and use that information to determine the meaning of a name. Admittedly, the most complicated scenarios should be rare - perhaps it would pay to cheat a bit... Also, since classes accessible across packages have one-to-one correspondence between their source file and class file, ignoring inter-package dependencies (and always recompiling all sources in an updated package) would considerably simplify the analysis.

A proof of this concept (including a demonstration of problems with file-level granularity) is werken.javad [6]. In my view, it suffers from being a `make` add-on, but it certainly can serve as a basis to build from - all code I wrote to research this article (and don't present it - it's just not good enough) is based on werken.javad.

Does anybody care?

So, technically, it seems possible to build a Java-aware build tool - yet none exists. And I'm as guilty of it as anyone - I used `make` to build my experimental code, and it did not work right. New `make` replacements like Apache Ant [7] don't do dependency management at all. Is the problem of building Java applications solvable, but just too hard to bother solving? IDEs do try to solve it, but not very well (JBuilder, to pick my favorite whipping-boy, does have some dependency management, but it's unreliable - it doesn't handle files being removed from the project, for example), and there is no commercial build tool which isn't an all-singing, all-dancing, all-its-own IDE. I believe the problem is too hard for home-made solutions, but also that there's more Open Source Java projects than any one company has, and that these projects would profit from a Java-aware build tool. I believe they could profit as much as the GNU [8] projects profit from autoconf [9] - a piece of software whose development couldn't have been justified by competing Unix vendors precisely *because* their customers find it so useful (to mix and match their software).

Any takers?

Vaclav Barta
vbar@comp.cz

References

[1] GNU `make` manual: `http://www.gnu.org/manual/make/html_node/make_toc.html`
[2] CONS - A Software Construction System `http://www.dsmit.com/cons/stable/cons.html`
[3] The Java Virtual Machine Specification: `http://java.sun.com/docs/books/vmspec/`
[4] The ANTLR website: `http://www.antlr.org/`
[5] The Java Language Specification: `http://java.sun.com/docs/books/jls/`
[6] The Werken Digital website: `http://code.werken.com/`
[7] The Jakarta Project website (Apache Ant): `http://jakarta.apache.org/ant/`
[8] GNU website: `http://www.gnu.org/`
[9] GNU `autoconf` information and resources: `http://www.gnu.org/software/autoconf/`

An Overview of C#.NET

By Jon Jagger

.NET rests on the Common Language Infrastructure (CLI). Microsoft, Intel, and Hewlett-Packard have jointly submitted the CLI as an ECMA standard. The CLI is designed for strongly typed languages and the CLI proposal has 5 partitions. Part 1 specifies the CLI foundation: the CTS, the VES, and the CLS. The Common Type System (CTS) specifies two CLI fundamental types: value types and reference types. Compiling a C# program does not create a regular executable file. Instead it creates a program in Common Intermediate Language (CIL, specified in partition 3). A compiled C# program also contains a block of metadata (data about the program itself) called a manifest (specified in partition 2). This metadata allows reflection and effectively eliminates the need for the registry. The job of the Virtual Execution System (VES) is to translate the CIL into native executable code (which can be done just-in-time or at installation). The Common Language Specification (CLS) is a set of rules designed to allow language interoperability. For example, unsigned integer types are *not* in the CLS so your C# modules must not expose unsigned integers if you want them to be fully interoperable.

Hello World

The obligatory console Hello World in C# looks like this.

```
class HelloWorld {
  static void Main() {
    System.Console.WriteLine(
                        "Hello, world!");
  }
}
```

C# has a sensibly limited preprocessor. There are no macro functions. What you see is what you get. A C# source file is not required to have the same name as the class it contains. Identifiers should follow the camelCasing or PascalCasing notation depending on whether they are private or non-private respectively. Hungarian notation is officially *not* recommended. C# is a case sensitive language so Main must be spelled with a capital M. A C# program exposing two identifiers differing only in case is not CLS compliant. The CLS supports exception handling and C# accesses these features using the try/catch/finally keywords. Exceptions are used extensively in the .NET framework classes. C# also supports C++ like namespaces as a purely logical scoping/naming mechanism. You can write using directives to bring the typenames in a namespace into scope.

```
using System; // System.Exception

class HelloWorld {
  static void Main() {
    try {
      NotMain()
    }
    catch (Exception caught) {
      ...
    }
  }
  ...
}
```

C# Fundamentals

Numeric Types

C# supports 8 integer types (not all of which are CLS compliant) and three floating point types. The floating point literal suffixes for these three types are F/f, D/d, and M/m (think m for money).

Type	bits	CLS?	signed?	sig figs
byte	8	yes	no	
ushort	16	no	no	
uint	32	no	no	
ulong	64	no	no	
sbyte	8	no	yes	
short	16	yes	yes	
int	32	yes	yes	
long	64	yes	yes	
float	32	yes		7
double	64	yes		15
decimal	128	yes		28

Figure 1: C# Integer Types

C# expressions follow the standard C/C++/Java rules of precedence and associativity. As in Java, the order of operand evaluation is left to right (in C/C++ it's unspecified), an expression must have a side effect (in C/C++ it needn't) and a variable can only be used once it has definitely been assigned (not true in C/C++).

Checked Arithmetic

The CLS allows expressions or statements that contain integer arithmetic to be checked to detect integer overflow. C# uses the checked and unchecked keywords to access this feature. An integer overflow throws an `OverflowException` when checked. (Integer division by zero *always* throws a `DivideByZeroException`.) Floating point expressions never throw exceptions (except when being cast to integers). For example:

```
class Overflow {
  static void Main() {
    try {
      int x = int.MaxValue + 1;
                // wraps to int.MinValue
      int y = checked(int.MaxValue + 1);
                // throws
    }
    catch (System.OverflowException
                                  caught) {
      System.Console.WriteLine(caught);
    }
  }
}
```

Control Flow

C# supports the `if`/`while`/`for`/`do` statements familiar to C/C++/Java programmers. As in Java, a C# boolean expression must be a genuine boolean expression. There are *never* any conversions from a built in type to `true`/`false`. A variable

introduced in a `for` statement initialization is scoped to that `for` statement. C# supports a `foreach` statement, which you can use to effortlessly iterate through an array (or any type that supports the correct interface).

```
class Foreach {
  static void Main(string[] args) {
    foreach (string arg in args) {
      System.Console.WriteLine(arg);
    }
  }
}
```

The C# switch statement does not allow fall-through behavior. Every `case` section (including the optional `default` section) must end in a `break` statement, a `return` statement, a `throw` statement, or a `goto` statement. You are only allowed to switch on integral types, bools, chars, strings and enums (these types all have a literal syntax).

Methods and Parameters

C# does not allow global methods; all methods must be declared within a struct or a class. C# does not have a C/C++ header/source file separation; all methods must be declared inline. Arguments can be passed to methods in three different ways:

- *copy*. The parameter is a *copy* of the argument. The argument must be definitely assigned. The method cannot modify the argument.
- *out*. The parameter is an *alias* for the argument. The argument need not be definitely assigned. The method must definitely assign the parameter/argument.
- *ref*. The parameter is again an *alias* for the argument. The argument must be definitely assigned. The method is not required to assign the parameter/argument.

The ref/out keywords must appear on the method declaration *and* the method call. For example:

```
class Calling {
  static void Copies(int param) { ... }
  static void Modifies(out int param)
                                  { ... }
  static void Accesses(ref int param)
                                  { ... }

  static void Main() {
    int arg = 42;
    Copies(arg); // arg won't change
    Modifies(out arg); // arg will change
    Accesses(ref arg);
                // arg might change
  }
}
```

C# supports method overloading but not return type covariance. Unlike Java, C# does *not* support method throw specifications (all exceptions are effectively unchecked).

Value Types

C# makes a clear distinction between value types and reference types. Value type instances (values) live on the stack and are used directly whereas reference type instances (objects) live on the heap and are used indirectly. C# has excellent language support for declaring user-defined value types (unlike Java which has none).

Enums and Structs

You can declare `enum` types in C#. For example:

```
enum Suit {Hearts, Clubs, Diamonds, Spades}
```

You can also declare a user-defined value type using the `struct` keyword. For example:

```
struct CoOrdinate {
  int x, y;
}
```

Unlike C++, the default accessibility of `struct` fields is private. You control the initialization of `struct` values using constructors. You use the `static` keyword to declare shared methods and shared fields. The `readonly` keyword is used for fields that can't be modified and are initialised at runtime. The `const` keyword is used for fields (and local variables) that can't be modified and are initialised at compile time (and is therefore restricted to `enum`s and built in types). As in Java, each declaration must repeat its access specifier.

```
struct CoOrdinate {
  public CoOrdinate(int initialX,
                    initialY) {
    x = rangeCheckedX(initialX);
    y = rangeCheckedY(initialY);
  }
  public const int MaxX = 600;
  public static readonly CoOrdinate
           Empty = new CoOrdinate(0, 0);
  ...
  private int x, y;
}
```

The built in value type keywords are in fact just a notational convenience. The keyword `int` (for example) is an alias for `System.Int32`, a `struct` called `Int32` that lives in the `System` namespace. Whether you use `int` or `System.Int32` in a C# program makes no difference.

Operator Overloading

C# supports operator overloading. `Enum` types automatically support most operators but `struct` types do not. For example, to allow `struct` values to be compared for equality/inequality you must write == and != operators:

```
struct CoOrdinate {
  public static bool operator==(
        CoOrdinate lhs, CoOrdinate rhs) {
    return lhs.x == rhs.x &&
           lhs.y == rhs.y;
  }
  public static bool operator!=(
        CoOrdinate lhs, CoOrdinate rhs) {
    return !(lhs == rhs);
  }
  ...
  private int x, y;
}
```

Operators must be public static methods. Operator parameters can only be passed by copy (no `ref` or `out` parameters). One or more of the operator parameter types must be of the containing type so you can't change the meaning of the built in operators. The increment (and decrement) operator can be overloaded and works correctly when used in either prefix and postfix form. C# also supports conversion operators which must be declared using

the implicit or explicit keyword. Some operators (such as simple assignment) cannot be overloaded.

Properties

Rather than using a Java Bean like naming convention, C# uses properties to declare read/write access to a logical field without breaking encapsulation. Properties contain only get and set accessors. The get accessor is automatically called in a read context and the set accessor is automatically called in a write context. For example (note the `x` and `X` case difference):

```
struct CoOrdinate {
  ...
  public int X {
    get { return x; }
    set { x = rangeCheckedX(value); }
  }
  ...
  private static int
            rangeCheckedX(int argument) {
    if(argument < 0 || argument > MaxX) {
      throw new ArgumentOutOfRange("X");
    }
    return argument;
  }
  ...
  private int x, y;
}
```

Indexers

An indexer is an operator like way to allow a user-defined type to be used as an array. An indexer, like a property, can contain only get/set accessors. For example:

```
struct Matrix {
  ...
  public double this [ int x, int y ] {
    get { ... }
    set { ... }
  }
  public Row this [ int x ] {
    get { ... }
    set { ... }
  }
  ...
}
```

Reference Types

Classes

Classes allow you to create user-defined reference types. One or more reference type variables can easily refer to the same object. A variable whose declared type is a class can be assigned to `null` to signify that the reference does not refer to an object (struct variables cannot be assigned to `null`). Assignment to `null` counts as a Definite Assignment. Classes can declare constructors, destructors, fields, properties, indexers, and operators. Despite identical syntax, classes and structs have subtly different rules and semantics. For example, you can declare a parameterless constructor in a class but not in a struct. You can initialise fields declared in a class at their point of declaration, but struct fields can only be initialized inside a constructor. Here is a class called `MyForm` that implements the GUI equivalent of Hello World in C#.NET.

```
using System.Windows.Forms;
class Launch {
  static void Main() {
    Application.Run(new MyForm());
  }
}
class MyForm : Form {
  public MyForm() { Text = captionText; }
  private string captionText
                        = "Hello, world!";
}
```

Variables whose declared type is a class can be passed by `copy`, by `ref`, and by `out` exactly as before.

```
class WrappedInt {
  public WrappedInt(int initialValue)
    { value = initialValue; }
  ...
  private int value;
}
class Calling {
  static void Copies(WrappedInt param)
    { ... }
  static void Modifies(out
              WrappedInt param) { ... }
  static void Accesses(ref
              WrappedInt param) { ... }
  static void Main() {
    WrappedInt arg = new WrappedInt(42);
    Copies(arg); // arg won't change
    Modifies(out arg); // arg will change
    Accesses(ref arg); // arg might change
    }
}
```

Strings

C# string literals are double quote delimited (`char` literals are single quote delimited). Strings are reference types so it is easy for two or more string variables to refer to the same string object. The keyword `string` is an alias for the `System.String` class in exactly the same way that `int` is an alias for the `System.Int32` struct.

```
namespace System {
  public sealed class String : ... {
    ...
    public static bool operator==(
          String lhs, String rhs) { ... }
    public static bool operator!=(
          String lhs, String rhs) { ... }
    ...
    public int Length { get { ... } }
    public char this[int index]
      { get { ... } }
    ...
    public CharEnumerator GetEumerator()
      { ... }
    ...
  }
}
```

The `String` class supports a readonly indexer (it contains a `get` accessor but no `set` accessor). The C# string type is an immutable type (just like in Java). The string equality and inequality operators are overloaded but the relational operators (`< <= > >=`) are not. The `StringBuilder` class is the mutable companion to string and lives in the `System.Text` namespace. You can iterate through a string expression using a `foreach` statement.

Arrays

C# arrays are reference types. The size of the array is not part of the array type. You can declare rectangular arrays of any rank (Java supports only one dimensional rectangular arrays).

```
int[]  row;
int[,] grid;
```

Array instances are created using the `new` keyword. Array elements are default initialised to zero (`enum`s and numeric types), `false` (`bool`), or `null` (reference types).

```
row  = new int[42];
grid = new int[9,6];
```

Array instances can be initialised. A useful initialisation shorthand does not work for assignment.

```
int[] row = new int[4]{1, 2, 3, 4};
                                  // longhand
int[] row = { 1, 2, 3, 4 };  // shorthand
row = new int[4]{ 1, 2, 3, 4 };   // okay
row = {1, 2, 3, 4}; // compile time error
```

Array indexes start at zero and all array accesses are bounds checked (`IndexOutOfRangeException`). All arrays implicitly inherit from the `System.Array` class. This class brings array types into the CLR (Common Language Runtime) and provides some handy properties and methods:

```
namespace System {
  public abstract class Array : ... {
    ...
    public int Length { get { ... } }
    public int Rank { get { ... } }
    public int GetLength(int rank) { ... }
    public virtual IEnumerator
                  GetEnumerator() { ... }
    ...
  }
}
```

The element type of an array can itself be an array creating a so called "ragged" array. Ragged arrays are not CLS compliant. You can use a `foreach` statement to iterate through a ragged array or through a rectangular array of any rank:

```
class ArrayIteration {
  static void Main() {
    int[] row = { 1, 2, 3, 4 };
    foreach (int number in row) { ... }
    int[,] grid = { { 1, 2 }, { 3, 4 } };
    foreach (int number in grid) { ... }
    int[][] ragged = { new int[2]{1,2},
                       new int[4]{3,4,5,6} };
    foreach (int[] array in ragged) {
      foreach (int number in array) { ... }
    }
  }
}
```

Boxing

An object reference can be initialised with a value. This does not create a reference referring into the stack (which is just as well!). Instead the CLR makes a copy of the value on the heap and the reference refers to this copy. The copy is created using a plain bitwise copy (guaranteed to never throw an exception). This is called boxing. Extracting a boxed value back into a local value is called unboxing and requires an explicit cast. When unboxing the CLR checks if the boxed value has the exact type specified in the cast (conversions are not considered). If it doesn't the CLR throws an `InvalidCastException`. C# uses boxing as part of the params mechanism to create typesafe variadic methods (methods that can accept a variable number of arguments of any type).

```
struct CoOrdinate {
  ...
  private int x, y;
}
class Boxing {
  static void Main() {
    CoOrdinate pos;
    pos.X = 1;
    pos.Y = 2;
    object o = pos; // boxes
    ...
    CoOrdinate copy = (CoOrdinate)o;
                        // cast to unbox
  }
}
```

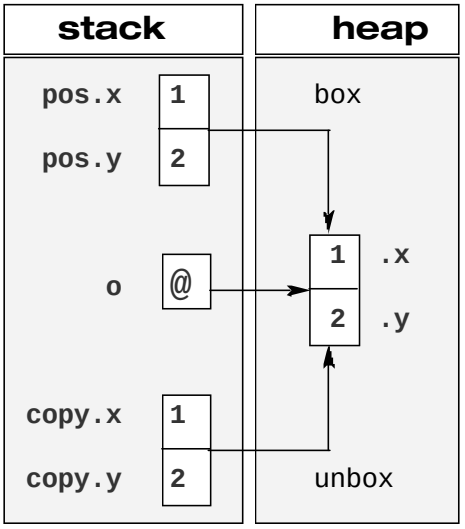

Figure 2: Boxing

Type Relationships

Inheritance

C# supports the same single inheritance model as Java; a class can extend at most one other class (in fact a class always extends exactly one class since all classes implicitly extend `System.Object`). A struct cannot act as a base type or be derived from. A derived class can access non-private members of its immediate base class using the `base` keyword. Unlike Java (and like C++) by default C# methods, indexers, properties, and events are *not* virtual. The `virtual` keyword specifies the *first*

implementation. The `override` keyword specifies *another* implementation. The `sealed override` combination specifies the *last* implementation.

```
class Token {
  ...
  public virtual CoOrdinate Location {
    get {
      ...
    }
  }
}
class LiteralToken : Token {
  ...
  public LiteralToken(string symbol) {
    ...
  }
  public override CoOrdinate Location {
    get {
      ...
    }
  }
}
class StringLiteralToken : LiteralToken {
  ...
  public StringLiteralToken(string
                  symbol) : base(symbol) {
    ...
  }
  public sealed override
                    CoOrdinate Location {
    get {
      ...
    }
  }
}
```

Interfaces

C# interfaces contain only the names of methods. Method bodies are not allowed. Access modifiers are not allowed (all methods are implicitly public). Fields are not allowed (not even static ones). Static methods are not allowed (so no operators). Nested types are not allowed. Properties, indexers, and events (again with no bodies) are allowed though. An interface, struct, or class can have as many base interfaces as it likes.

```
interface IToken {
  ...
  CoOrdinate Location { get; }
}
```

A struct or class must implement all its inherited interface methods. Interface methods can be implemented implicitly or explicitly.

```
class LiteralToken : IToken {
  ...
  public CoOrdinate Location {
  // implicit implementation
    get {
      ...
    }
  }
}
class LiteralToken : IToken {
  ...
  CoOrdinate IToken.Location {
  // explicit implementation
    get {
      ...
    }
  }
}
```

You use the `abstract` keyword to declare an abstract class or an abstract method (only abstract classes can declare abstract methods). You use the `sealed` keyword to declare a class that cannot be derived from. The inheritance notation is positional; base class first, followed by base interfaces.

```
interface IToken {
  ...
  CoOrdinate Location {
    get;
  }
}

abstract class DefaultToken {
  ...
  protected DefaultToken(CoOrdinate
                                  where) {
    location = where;
  }
  public CoOrdinate Location {
    get {
      return location;
    }
  }
  private readonly CoOrdinate location;
}

sealed class StringLiteralToken
                : DefaultToken, IToken {
  ...
}
```

Runtime type information is available via the `is`, `as`, and `typeof` keywords as well as the `object.GetType()` method.

Resource Management

You can declare a destructor in a class. A C# destructor has the same name as its class, prefixed with a tilde (~). A destructor is not allowed an access modifier or any parameters. The compiler converts your destructor into an override of the `object.Finalize` method. For example, this:

```
public class StreamWriter : TextReader {
  ...
  ~StreamWriter() {
    Close();
  }
  public override void Close() {
    ...
  }
}
```

is converted into this: (You can use the ILDASM tool to see this transformation in CIL.)

```
public class StreamWriter : TextReader {
  ...
  protected override void Finalize() {
    try {
      Close();
    }
    finally {
      base.Finalize();
    }
  }

  public override void Close() {
    ...
  }
}
```

You are not allowed to call a destructor or the `Finalize` method in code. Instead, the generational garbage collector (which is part of the CLR) calls `Finalize` on objects sometime after they become unreachable but definitely before the program ends. You can force a garbage collection using the `System.GC.Collect()` method. C# does not support struct destructors (although CIL does). However, C# does have a `using` statement which you can use to scope a resource to a local block in an exception safe way. For example, this:

```
class Example {
  void Method(string path) {
    using (LocalStreamWriter exSafe =
                new StreamWriter(path)) {
      StreamWriter writer =
                exSafe.StreamWriter;
      ...
    }
  }
}
```

is automatically translated into this:

```
class Example {
  void Method(string path) {
    {
      LocalStreamWriter exSafe =
                new StreamWriter(path);
      try {
        StreamWriter writer =
                exSafe.StreamWriter;
        ...
      }
      finally {
        exSafe.Dispose();
      }
    }
  }
}
```

which relies on `LocalStreamWriter` implementing the `System.IDisposable` interface:

```
public struct LocalStreamWriter
                          : IDisposable {
  public LocalStreamWriter(StreamWriter
                             decorated) {
    local = decorated;
  }

  public static implicit operator
       LocalStreamWriter(StreamWriter
                             decorated) {
    return new
           LocalStreamWriter(decorated);
  }

  public StreamWriter StreamWriter {
    get {
      return local;
    }
  }

  public void Dispose() {
    local.Close();
  }

  private readonly StreamWriter local;
}
```

Program Relationships

Delegates and Events

The `delegate` is the last C# type. A `delegate` is a named method signature (similar to a function pointer in C/C++). For example, the `System` namespace declares a `delegate` called `EventHandler` that's used extensively in the `Windows.Forms` classes:

```
namespace System {
  public delegate void EventHandler(
        object sender, EventArgs sent);
  ...
}
```

`EventHandler` is now a reference type you can use as a field, a parameter, or a local variable. Calling a `delegate` calls all the `delegate` instances attached to it.

```
namespace Not.System.Windows.Forms {
  public class Button {
    ...
    public EventHandler Click;
    ...
    protected void OnClick(EventArgs
                                 sent) {
      if (Click != null) {
        Click(this, sent); // call here
      }
    }
  }
}
```

All delegate types implicitly derive from the `System.Delegate` class. You use the `event` keyword to modify the declaration of a delegate field. Event delegates can only be used in restricted, safe ways (for example, you can't call the delegate from outside its class):

```
namespace System.Windows.Forms {
  public class Button {
    ...
    public event EventHandler Click;
  }
}
```

You create an instance of a delegate type by naming a method with a matching signature and you attach a delegate instance to a matching field using the += operator.

```
using System.Windows.Forms;

class MyForm : Form {
  ...
  private void initializeComponent() {
    ...
    okButton = new Button("OK");
    okButton.Click += new
             EventHandler(this.okClick);
             // create + attach
  }

  private void okClick(object sender,
                       EventArgs sent) {
    ...
  }
  ...
  private Button okButton;
}
```

Assemblies

You can compile a working set of source files (all written in the same supported language) into a .NET module. For example, using the C# command line compiler:

```
csc /target:module /out:ratio.netmodule *.cs
```

The default file extension for a .NET module is `.netmodule`. A .NET module contains types and CIL instructions directly and forms the smallest unit of dynamic download. However, a .NET module cannot be run. The only thing you can do with a .NET module is add it to an assembly. An assembly contains a manifest (a module does not). The manifest is metadata that describes the contents of the assembly and makes the assembly self describing. An assembly knows:

- the assembly identity
- any referenced assemblies
- any referenced modules
- types and CIL code held directly
- security permissions
- resources (eg bitmaps, icons)

You create a .NET DLL (an assembly) using the /target:library option from the command line compiler (there are various other options for adding modules and referencing other assemblies):

```
csc /target:library /out:ratio.dll *.cs
```

You create a .NET EXE (an executable assembly) using the `/target:exe` options on the command line compiler (one of the structs/classes must contain a `Main` method).

```
csc /target:exe /out:ratio.exe *.cs
```

Assemblies come in two forms. A private assembly is not versioned, and is used only by a single application. A shared assembly is versioned, and lives in a special shared directory called the Global Assembly Cache (GAC). Shared assembly version numbers are created using an IP like numbering scheme:

```
<major> . <minor> . <build> . <revision>
```

Shared applications that differ only by version number can co-exist in the GAC (this is called side-by-side execution). The particular version of an assembly that an individual application uses when running can be controlled from an XML file. For example:

```
...
<BindingPolicy>
    <BindingRedir
            Name="ratio" ...
            Version="*"
            VersionNew="6.1.1212.14"
            UseLatestBuildRevision="no"/>
</BindingPolicy>
...
```

You can edit this config file to choose your binding policy. For example:

- Safe: exactly as built
- Default: major.minor as built
- Specific: major.minor as specified.

Attributes

You use attributes to tag code elements with declarative information. This information is added to the metadata, and can be queried and acted upon at translation/run time using reflection. For example, you use the `[Conditional]` attribute to tag methods you want removed from the release build (calls to conditional methods are also removed):

```
using System.Diagnostics;

class Trace {
  [Conditional("DEBUG")]
  public static void Write(string
                                message) {
    ...
  }
}
```

You use the `[CLSCompliant]` attribute to declare (or check) that a source file conforms to the Common Language Specification:

```
using System;

[assembly:CLSCompliant(true)]
...
```

You can use the **`[MethodImpl]`** attribute to synchronize a method:

```
using System.Runtime.CompilerServices;

class Example {
  [MethodImpl(MethodImplOptions.
                            Synchronized)]
  void SynchronizedMethod() {
    ...
  }
}
```

The attribute mechanism is extensible; you can easily create and use your own attribute types:

```
public sealed class DeveloperAttribute
                  : Attribute {
  public DeveloperAttribute(string name) {
    ...
  }
}
...
[Developer("Jon Jagger")]
public struct LocalStreamWriter
                           : IDisposable {
  ...
}
```

Summary

C# programs compile into Common Intermediate Language (CIL). CIL types that conform to the CLS (Common Language Specification) can be used by any .NET language. For example, the types in the System namespace are implemented in the mscorlib.dll assembly. Programs written in C#, in VB.NET, or in managed C++, can all use this assembly (there isn't one version of the assembly for each language).

CIL programs are translated into executable programs either at installation time or just-in-time as they are executed by the VES (Virtual Execution System). The CLI (Common Language Infrastructure – the CTS, the VES, the CLS, and the metadata specification) is an ECMA standard and efforts are already underway to implement the CLI on non Windows platforms (eg `http://www.go-mono.com`).

C# is a modern general purpose programming language. It has clear similarities to Java (reference types, inheritance model, garbage collection) and to C++ (value types, operator overloading, logical namespaces, by default methods are not virtual). It has no backward compatibility constraints (as C++ does to C) and avoids/resolves known problems in Java. The CTS (Common Type System) makes a clear distinction between value types and reference types. The more I use C# the more I like it and the more I appreciate the careful and consistent decisions taken during its design. C# is my language of choice for .NET development. In roughly keeping to the allotted word count I have necessarily omitted numerous important aspects of C#. Nevertheless I hope this article has given you a flavour of C# and its relationship to .NET.

Jon Jagger
jon@jaggersoft.com

Letter To The Editor

Hi,

There was no mention in Overload 48 that my article, "Function Follows Form", was previously published online as part of the CUJ C++ Experts Forum online. The omission is not major, but it is worth pointing out that it appeared originally for November 2000 following the summer break after the untimely demise of C++ Report. Its prior publication is relevant because in the nearly two years since it was written - and much longer for the code and basic design - a couple of things, albeit minor, have changed in my thinking:

1. I used the name '`function_ptr`' to represent the zero-argument arbitrary function and function object adaptor with smart pointer semantics. I now prefer the name '`any_function`', which is more in keeping with my use of the name '`any`' for any arbitrary value, and the prefix '`any_`' for other wrappers in a similar style, e.g. '`any_iterator`' and '`any_string`'.

2. In the article I mentioned that the technique of using a non-templated base class to provide uniform access to a common family of variations expressed as a derived class template was based on the External Polymorphism pattern. I was careful to ensure that I said "is based on" rather than "is" because at the time I was not comfortable that the pattern's intent was appropriate, even though its structure was. Well, following some tentative repetition of idea and the "three strikes and you're out" approach, I have decided that it is categorically not the right pattern to reference. To the best of my knowledge, the pattern has not been properly documented elsewhere, although it is used extensively. I plan to document the pattern more thoroughly at some point, either under the name Polymorphic Wrapper, which captures the fact that it converts one form of polymorphism (templating) to another (virtual functions), or as Parameterized Derived Class, a more accurate albeit prosaic name. Generally it preserves the degree of polymorphism, but reduces the compile-time polymorphism by one axis to introduce one of runtime polymorphism.

Kevlin Henney
kevlin@curbralan.com

The scoping problem

By Allan Kelly

It is a cliché of software development that every project is understaffed with a deadline that is too tight, and maybe under-resourced in terms of hardware and software too. I mean, have you ever worked on a project where there where too many programmers?

Jim McCarthy [1] suggests that every project is bounded by three factors: features, time and resources (i.e. developers) – these form the scope of the project: what do we need to do? how long have we got? who's doing it? Their product is a constant: if you want more features you must either increase the time allowed or the number of developers. Likewise, if you want a delivery sooner you must either reduce the number of features or increase the number of developers. But, as we know from Brooks' law [2], increasing the number of developers makes a project later; unfortunately Brooks' law is not true in reverse: reducing the number of developers will also make the project later. Brooks' law can only be applied to a given point in time.

I'd like to add two observations of my own to McCarthy and Brooks:

Observation 1: Developers want to exceed the project requirements. Sometimes this means they want to see a feature which isn't in the spec, sometimes it means they want to do frivolous things like refactor the code, or make it more readable, heaven forbid, some of them actually want to make the code re-usable! Truth is, most developers believe they know best. Some of us are even known to claim that this is a professional asset.

Observation 2: Developers are more likely to underestimate the time taken to do a piece of work than they are to overestimate; this is especially true when a developer wants to do a piece of work.

I think from these observations flows the axiom: software projects are naturally over-committed in terms of time and resources.

If my observations have not convinced you, consider what would happen if you turned around at your next project meeting and said: "Well then chaps, looks like we've got four more weeks work and then we'll be finished a whole month early, well done." Would the reaction be:

- Launch a month early : pull that deadline back!
- Add in some more features, fix some more bugs : back to work!
- Fire the contractors, we can save some money and still get it done on time.
- Congratulations: take a months paid leave.

I'd like to be bold and suggest *Kelly's first law of project complexity*:

Project scope will always increase in proportion to resources.

You see, I think projects are bit like programs: they will expand to fill all the time and human resources available. This may be a generalised case of Brooks' law, in his case he adds more developers, now the project scope increases to cover training new developers too.

Every time a deadline is pushed back to allow more bugs to be fixed you are increasing the scope too. Even if the project introduced those bugs in the first place, fixing them was never on the feature list.

With a loose deadline there is a tendency to try some new beta OS, upgrade your compiler, or experiment with generic programming or many other things. Yes, these all have a place. I'm just saying you have to recognise the cost of all these things. Introduce a breathing space between projects to do these things.

I don't want to argue for arbitrary deadlines either. A deadline picked from the sky and imposed is just as bad. Most projects have a rough deadline before they ever start, given this the team needs to look at the feature set and the resources, and balance all these things.

A word of caution: a developer joining a project at mid-point may not be aware of these discussions, it is important to explain the schedule to them and even adapt it after hearing their views.

If all of this puts you in mind of short cycles you're right. Every time I work on a large project I am convinced that much of the project could be achieved with less code, complexity and effort.

This leads me to *Kelly's second law of project complexity*:

Inside every large project there is a small one struggling to get out.

Take a look at your project, why is it so big? Would you write it the same if you were doing it again? Chances are you would not, you have learned from what you have done, and you would re-write it using less complexity and less code.

Writing a program is teaching the machine how to do something. In doing this you come to understand the problem intimately and inevitably see better ways of doing it. So much complexity is unavoidable, but I suggest that much of it is avoidable if you merely reduce the scope of the project.

OK, how do we reduce the scope of the project? Well, firstly aim for minimalism [3] in design. Secondly, design extendable systems. Write a core system to which you can add code, your project may end up being big but it will still contain a micro-kernel like element which make it easier to get to grips with: embrace the words of John Vlissides [4] *"A hallmark.... of good object oriented design is that you can modify and extend a system by adding code rather than hacking it."*

Next: keep your project and your team focused. Each project iteration should have some *Big Idea* which you can drive towards. This is good for morale too and you always know when you have arrived.

And try this on your management: for every day the project is finished early, everyone on the team gets half a day's paid holiday.

Allan Kelly
Allan.Kelly@bigfoot.com

References

[1] Jim McCarthy, *Dynamics of Software Development*, Microsoft Press, 1995

[2] Fred Brooks, *Mythical Man-Month*, Addison-Wesley, 1974

[3] Kevlin Henney, minimalism : omit needless code, *Overload 45*, October 2001

[4] John Vlissides, *The C++ Report*, February 1998