contents

credits & contacts

Overload Editor:
Alan Griffiths
overload@accu.org
alan@octopull.demon.co.uk

Contributing Editor:
Mark Radford
mark@twonine.co.uk

Advisors:

Phil Bass
phil@stoneymanor.demon.co.uk

Thaddaeus Frogley
t.frogley@ntlworld.com

Richard Blundell
richard.blundell@metapraxis.com

Advertising:
Chris Lowe
ads@accu.org

Overload is a publication of the ACCU. For details of the ACCU and other ACCU publications and activities, see the ACCU website.

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

Information and Membership:
Join on the website or contact

David Hodge
membership@accu.org

Publications Officer:
John Merrells
publications@accu.org

ACCU Chair:
Ewan Milne
chair@accu.org

Editorial: A Glimpse Behind the Scenes

This summer I went walking in the Lake District with my girlfriend, and on one of the days we found ourselves walking through a mist that reduced visibility to around 10 metres. Unable to use other mountains as reference points (or, indeed, much of the mountain we were on) we were reliant on a compass and dead reckoning. Despite our best efforts we eventually found ourselves on a scree slope with no sign of a path. We were not exactly lost – we had a bearing to follow and would eventually reach landmarks and a path leading us to our destination, but the going was slow and difficult.

While we were debating where we'd missed the path a voice came from the mists hiding the top of a near vertical rock face "you shouldn't be down there – the path is up here". While that gave a point of reference, it didn't solve the problem of how to proceed from where we had arrived. We had gone wrong, but we had company: another group of people arrived and by all sharing the results of exploration we found a way forward that eventually rejoined the path.

This story does have relevance to Overload but, before explaining the connection, I'm going discuss a little of what happens behind the scenes to produce the copies of Overload that arrive for you to read every couple of months. You have probably guessed a part of it, but the issue that I want to address is one that rarely makes an explicit appearance in the journal:

- people write articles (volunteers to do this are always welcome);
- the articles are reviewed by the panel of advisors (this usually leads the authors to revise the articles); and,
- the editor chooses the articles to accept, pass to C Vu or which to reject completely.

Usually the review by the advisors is uncontroversial, but occasionally an article generates real debate. When considering Jeff Daudel's article "A Unified Singleton System" (Overload 56) there was such a concern over the content of the article that the editor (then John Merrells) was moved to include an editorial comment. And, in the next issue, one of the advisors (Mark Radford) wrote a balancing article "Singleton - The Anti-Pattern!" (Overload 56).

Such debate is rare, most articles present material that the whole editorial team is happy with. However, it has happened again (but on a lesser scale). One of the articles published in the last issue moved one of the advisors, Phil Bass, to email me re-iterating the concerns that he expressed about the article when first presented. I could have relegated this to the "letters" page but, by implication, it raises issues of editorial policy that I feel should be discussed.

First here is what Phil Bass has to say:

When Allan Kelly's article entitled "The Encapsulate Context Pattern" was circulated for review I expressed considerable disquiet. This is what I said in an email on 23rd August:

"It actually presents an Anti-Pattern as if it were a real Pattern giving the impression that it represents good practice when, in fact, the opposite is true.

Also, I believe the mention of Kevlin Henney, Frank Buschmann and EuroPLoP gives the article an unwarranted air of authority.

I would prefer this not to be published. If it is published I think it should be accompanied by further discussion so that dissenting views can be expressed.

You were not convinced by my argument, no-one else expressed any reservations and the article was published without comment. In spite of that I remain convinced that publication of this article did more harm than good and I would appreciate the opportunity to argue my case in the pages of Overload."

I will come back to Encapsulate Context in a moment, but first I would like to make an analogy:

Pattern 24 - Swap Colours

Context You are playing chess.

Problem You are down to one bishop and it's on a black square. Unfortunately, your opponent's major pieces are all on white squares and you are struggling to create an effective attack.

Forces

1. The Bishop is a useful piece. You don't want to lose it.
2. The Bishop needs to be defended, but you would prefer to use your pieces to attack your opponent's position.

Solution Change the colours of the squares on the board. Black squares become white and white squares becomes black. Now you have a bishop on a white square.

I won't try to push the analogy any further because this "pattern" is quite obviously complete nonsense. It describes a genuine problem and presents a false solution. Ignoring, for the moment, that changing the colours of the squares is against the rules, it actually doesn't address the real problem.

I believe "Encapsulate Context" has all the hallmarks of "Swap Colours". In my opinion, it describes a genuine problem and presents a false solution. I will try to explain why...

I think it is fair to summarise "Encapsulate Context" as follows:

Encapsulate Context

Context Many systems contain data that must be generally available to divergent parts of the system.

Problems Using global data is poor engineering practice. Using long parameter lists in order to avoid global data has an adverse effect on maintainability and object substitutability.

Solution Long parameter lists can be avoided by collecting common data together into a Context object and passing that in parameter lists.

Resulting Context Using a Context object improves substitutability, encapsulation and coupling, and reduces the need for copying data.

First of all I am not at all convinced that the pattern context occurs in well-designed systems. Allan presents an example that purports to illustrate how the context arises, but I don't find it at all convincing. Personally, I would not be tempted to design a stock exchange trading system like that example, so it looks like a straw man to me. However, let's give Allan the benefit of the doubt and work on the assumption that such systems do exist.

I agree that global data is a Bad Thing. And I agree that overly long parameter lists are a sign of a problem. But long parameter lists are only a symptom of a deeper problem. Coalescing several parameters into one changes the interface in only a trivial way. The same information is passed from the client code to the function, so coupling and cohesion are not significantly different.

Now, if we consider many different functions spread throughout the system and a Context object that contains all the global information that any of those functions might need, things definitely get worse. Where a function actually needs only one item of global data we provide it with all the global data items, introducing unnecessary coupling. And where that function originally had direct access to the one item of global data it needed, now it must extract it from a Context object, increasing the complexity of the interface.

The resulting context is a significantly more coupled system. As far as I can see, the effect on maintainability is minimal, there is no effect on substitutability, and encapsulation has suffered. We have gone from a Bad Thing to an even Worse Thing. That is the opposite of a design pattern; it is an Anti-Pattern.

I will add just one further comment. If you find yourself designing a system with significant amounts of global data and/or passing data between distant parts of a system, you have a problem. It may be a deep-seated problem. Think carefully about what this data is, where it is needed and how the system is partitioned. The problem can only be solved by a fundamental restructuring of the system. There probably isn't a single design pattern that will do that. "Encapsulate Context" won't help you. Don't use it.

OK, here we can see real concerns being expressed about the article. So why then did I choose to publish it without comment? Well, partly because the other advisors and I don't share these concerns.

While we agree with Phil that it would be better to have avoided arriving at the problem, people do get to this point and don't always have the opportunity for a "fundamental restructuring of the system". It is more helpful to facilitate the communication between such people than to take the part of the anonymous voice saying "you shouldn't be down there". It may serve as a warning to avoid retracing the steps that lead to the situation, but doesn't address the tactical problem of dealing with it.

Maybe I was wrong. I can imagine that some Overload reader somewhere is facing the problems described by Allan Kelly. They probably know they have problems, but upon reading the article they may get the impression that the solution presented solves the underlying problem and not only the symptomatic one. This too is a pattern "Shifting The Burden" – described by Peter Senge in "The Fifth Discipline" (an excellent book on organisational problems). "Shifting The Burden" is a bad practice or "anti-pattern" largely because by failing to address the true causes of a problem, but giving the appearance of fixing it, it encourages the problem to grow to proportions that make an eventual solution problematic.

Maybe I was right. I've used "Encapsulate Context" in the past – albeit long before reading the article. I took an application that had, scattered throughout its code, snippets of code that went to various sources of configuration data (the Windows registry, the command line, several databases, configuration files) and moved all of these snippets into a context object. This made it far simpler to test components (we didn't need to set up copies of all the sources of configuration data for each test). When additional configuration data was required by some new code, it was clear how to obtain it – from the context object. And the difference between live and test configurations became easy to parameterise. Even the author of the original code was impressed by the changes. That system is still running and being developed and – as of the last report – without suffering the high maintenance/low testability problems that are a risk of such highly-coupled approaches.

There is a fine line to be drawn here and, while I think that we got this one right, it is necessary to be vigilant. The role of the Overload team is to help the author present tools and techniques that may be of use to the reader and let the reader be the judge of when they are appropriate. I expect readers to use their judgement when considering the application of such ideas – they should know that there are always trade-offs. Knives and forks can be dangerous, but I don't insist my children eat with blunt sticks.

And the lesson I take from my experience in the Lake District? While it is good to know where one should be it is also worth listening to the people solving the same problem you have encountered where you are.

Alan Griffiths
overload@accu.org

Copy Deadlines

All articles intended for publication in *Overload 65* should be submitted to the editor by January 1st 2005, and for *Overload 66* by March 1st 2005.

Evolution of the Observer Pattern

by Phil Bass

Back in the early 1990s four wise men began a voyage of discovery. They were trying to trace the origins of good software design. They crossed great deserts of featureless software and fought through almost impenetrable jungles of code. What they found changed the way we think about software development. Eric Gamma, Richard Helm, Ralph Johnson and John Vlissides discovered a plethora of Design Pattern species falling into 23 separate genera.

The "Gang of Four", as they have become known, published their findings in 1995 [1] and "patternologists" have been studying these creatures ever since. I have taken a particular interest in the Observer pattern and I have evidence for changes in its morphology and behaviour. The Observer pattern is evolving rapidly and this paper presents some recent and, I believe, important changes.

Historical Perspective

In their description of the Observer pattern the Gang of Four assumed that a Subject has a single point of attachment and that Observers attach directly to their Subjects. I have argued ([2], [3]) that Subjects can have multiple attachment points (Events) and that Observers may attach indirectly via structures known as Callbacks. Indeed, it is possible to understand the Observer pattern purely in terms of Events and Callbacks. From this perspective, a Subject is anything that publishes Events and an Observer is anything that attaches Callbacks to Events.

Formally, a Subject provides registries of event handlers, while Observers add handlers to these registries and subsequently remove them again. State change events in a Subject are communicated to the Observers by calling the registered handlers. In this paper, however, we will use 'Event' to mean "registry of event handlers" and 'Callback' to mean "registered event handler". This terminology is slightly unusual, but it simplifies the discussion considerably.

In the Gang of Four pattern, detaching an Observer required the Subject to search its collection of attached Observers. I proposed a more efficient mechanism in which the Observer stores an iterator returned when a Callback is attached to an Event and passes the iterator back to the Event when the Callback is detached. Although more efficient, this approach requires more client code and is more vulnerable to programmer error. (I call this the correctness vs. efficiency problem.)

Until recently, my own research has been confined to Events based on lists of pointers to a polymorphic callback base class. For example, here is a complete specimen of the common Event (*Eventus Vulgaris*)[1] as described in [3].

```
// Abstract Function interface class.
template<typename Arg>
struct AbstractFunction {
  virtual ~AbstractFunction() {}
  virtual void operator() (Arg) = 0;
};

// Event class template.
template<typename Arg>
struct Event : list<AbstractFunction<Arg>*> {
  void notify(Arg arg) {
    typedef AbstractFunction<Arg> Func;
    for_each(begin(), end(),
          bind2nd(mem_fun(&Func::operator()),
                arg));
  }
};
```

Specimen 1 – An Event from early 2003

We will compare this with several specimens of a newly discovered species (*Eventus Adaptabilis*) that is able to use different types of container and different types of callback. These adaptive changes allow *E. Adaptabilis* to thrive in a much wider range of programming environments. They also provide a solution to the correctness vs. efficiency problem, as we shall see shortly.

A Remarkable New Species

The key to the greater adaptability of *E. Adaptabilis* is a second template parameter, which specifies the type of container and, hence, the type of the callback pointers stored within it.

```
template<
    typename Arg,
    typename Container =
        std::list<Callback::Function<Arg>*> >
struct Event;
```

Specimen 2 – The defining feature of Eventus Adaptabilis.

This tiny fragment suggests answers to several questions that have puzzled patternologists. It explains, for example, why most *E. Adaptabilis* individuals are almost indistinguishable from their *E. Vulgaris* cousins. Because, by default, *E. Adaptabilis* uses the same container type as *E. Vulgaris* the two species often have identical external appearance and behaviour. It is also clear from this fragment how *E. Adaptabilis* is able to adapt so easily to different environments. Simply by specifying a different `Container` argument *E. Adaptabilis* can acquire any of the characteristics of that `Container`.

It is not clear, however, from Specimen 2 whether *E. Adaptabilis* acquires its behavioural traits through inheritance, nor can we deduce which types of container constitute valid parameters to the *E. Adaptabilis* template. To answer such questions we must look inside the Event. Our next specimen is instructive, here.

```
template<typename Arg, typename Container>
struct Event : Container {
  struct notify_function {
    notify_function(Arg a) : arg(a) {}
    typedef typename
            element<Container>::type pointer;
    void operator()(const pointer& ptr)
                                {(*ptr)(arg);}
    Arg arg;
  };

  // ...  indistinct features

  void notify(Arg arg) {
    // for_each(begin(), end(),
    //          notify_function(arg)); ???
  }
};
```

Specimen 3 – Some internal structure of *E. Adaptabilis*.

1 The `std::` prefix has been omitted to improve the layout on the printed page.

Unfortunately, the details of the `notify()` function have not been preserved. When this specimen was first discovered we assumed that the `notify()` function is similar to that of *E. Vulgaris*, as shown by the comment. In fact, this assumption turned out to be incorrect, but Specimen 3 does clearly show several interesting features. It is immediately clear, for example, that *E. Adaptabilis* inherits all the characteristics of its `Container`.

The most striking feature of Specimen 3, however, is the nested function object class, `notify_function`. It is perfectly adapted to its assumed role in the `notify()` function. It provides exactly the right interface for the `for_each()` algorithm and yet makes only the minimum assumptions about the container element types. Where *E. Vulgaris* is restricted to using `std::list<>`, *E. Adaptabilis* is free to use vectors, lists, sets, user-defined containers, etc. And where *E. Vulgaris* requires the container element type to be a built-in pointer to an `AbstractFunction<Arg>`, *E. Adaptabilis* accepts built-in pointers and smart pointers to ordinary functions and function objects of any type that can be called with an argument convertible to `Arg`.

It is interesting to note that the `notify_function` is public and, therefore, available to client code. This seems to be a violation of encapsulation, but it also provides benefits, as we shall see later.

Another note-worthy feature of the `notify_function` is the `element<Container>` meta-function. The implementation of this meta-function was missing from Specimen 3, but an intact sample was discovered later and is shown here as Specimen 4.

```
template<typename Container>
struct element {
  typedef typename Container::value_type type;
};
```

Specimen 4 – The `element` meta-function.

In itself this is an unremarkable structure. It just extracts the `value_type` from a container that conforms to the standard requirements. In evolutionary terms, however, its existence is quite interesting. *E. Adaptabilis* can only benefit from the `element<>` meta-function when it uses a non-standard container and only then if the meta-function is specialised for that container. As yet, there are no known cases of *E. Adaptabilis* and non-standard containers co-existing like this in the wild. It must be a matter of speculation, therefore, whether this feature has any real benefit.

The internal structure of the `notify()` function was a surprise. Instead of the ubiquitous `for_each()` function it uses a hitherto unknown algorithm, `slither()`. The `notify()` function can be seen in Specimen 5 and the `slither()` algorithm itself is shown in Specimen 6.

```
template<typename Arg, typename Container>
struct Event : Container {

  // ...

  void notify(Arg arg) {
    slither(this->begin(),
            this->end(),
            notify_function(arg));
  }
};
```

Specimen 5 – *E. Adaptabilis* `notify()` function.

```
template<typename Iterator, typename Function>
void slither(Iterator first, Iterator last,
             Function function) {
  if(first != last) {
    for(Iterator next = first;
        ++next != last;) {
      function(*first), first = next;
    }
    function(*first);
  }
}
```

Specimen 6 – The `slither()` algorithm.

Like `for_each()`, `slither()` applies a given function to the result of dereferencing every iterator in the range [`first`, `last`). However, `slither()` uses two iterators. At the start of the for loop body `first` points to the function about to be called and `next` points to the next one. At that point the function is called, `first` is moved forward to `next` *by assignment*, and `next` is incremented. The loop then proceeds to the next iteration or terminates. The overall effect is the same as `for_each()` and yet the algorithm is more complex.

Patternologists puzzled over this for a long time. Natural selection is a powerful mechanism for reducing the costs associated with unnecessary complexity. It should prefer `for_each()` over `slither()`. And yet here was an example of evolution proceeding in the wrong direction. Several explanations were proposed to account for this anomaly. Perhaps `slither()` is just a transient mutation that hasn't yet been weeded out by competition with `for_each()`. Or, perhaps there is some hidden benefit to `slither()` that more than compensates for the cost.

I have made a crude attempt to measure the relative costs of `for_each()` and `slither()`. As is often the case when measuring speed of execution I found the result surprising. There was no difference in speed between the two algorithms. (I used GCC 3.2.3 on Linux Red Hat 9 with full optimisation.) In fact, my attempt to extend the running time by increasing the number of function pointers in the container just consumed all the available RAM and triggered swapping, which made further measurements meaningless. However, I saw approximately 200 million loop iterations per second on my 700 MHz PC before swapping kicked in. I tentatively concluded, therefore, that there is no significant cost associated with the `slither()` algorithm.

The negligible cost of `slither()` may explain how it manages to compete with `for_each()`, but it doesn't explain why it came into existence. For that we need to look at *E. Adaptabilis* in a hostile environment. Consider the following sample program:

```
#include "Event.hpp"

// Callback that detaches itself from the
// event when called.
struct Disconnect : Callback::Function<int> {
  Disconnect(Event<int>& e)
    : event(e),
      position(e.insert(e.end(),this)) {}
  void operator()(int i) {
    event.erase(position);
  }
  Event<int>&          event;
  Event<int>::iterator position;
};
```

```
int main() {
  Event<int> event;
  Disconnect disconnect(event);
  event.notify(7);   // !
  event.notify(7);
  return 0;
}
```

Specimen 7 – *E. Adaptabilis* in a hostile environment.

Here we have a callback that connects itself to an event in its constructor and disconnects itself when it is called. Such callbacks are extremely poisonous to *E. Vulgaris*, but *E. Adaptabilis* is immune. To see why, consider the `Event::notify()` call. *E. Vulgaris* iterates through its list of callback pointers using `for_each()` which (invariably) increments a single iterator. When the iterator reaches a `Disconnect` callback `for_each()` invokes the callback, which erases itself from the list, invalidating the iterator. The `for_each()` algorithm then tries to increment the invalid iterator and continue the sequence of function calls, typically with disastrous results. *E. Adaptabilis*, however, uses the `slither()` algorithm. When it gets to the `Disconnect` callback it invokes the callback, which erases itself from the list, invalidating the iterator as before. But `slither()` doesn't increment the invalid iterator, it simply assigns a new value to it. This is, of course, a valid operation, so the algorithm completes normally and *E. Adaptabilis* lives to notify another event.

Together these features provide an answer to the question of what constitutes a valid `Container` argument to the *E. Adaptabilis* template, `Event<Arg,Container>`. The `Container` must be a class with `begin()` and `end()` member functions returning `Forward` iterators. It must contain a nested typedef, `value_type`, that defines the type of the container elements, or it must provide a specialisation of the `element<>` meta-function for that purpose. The element type must define a dereference operation. And the result of dereferencing an element must be a function or function object that can be called with an argument of type `Arg`.

These are very general requirements. They can be summarised informally as:

- An *E. Adaptabilis* event is a *container* of *pointers* to *functions.*
- In this context, a *pointer* is any type that can be dereferenced;
- And a *function* is any type that can be called with an argument of the right type.

The Correctness vs. Efficiency Problem

Programmers working with *E. Vulgaris* must remember to store the iterator returned when a callback is attached and pass it back to the event when the callback is disconnected. This is tedious and it is tempting to leave the callback attached "for ever" to avoid having to manage the iterator. This usually leads to disaster and is always frustrating for the hapless programmer.

The need to store the iterator can be removed by searching the list of pointers before disconnecting the callback. However, in *E. Vulgaris* this technique carries a significant performance penalty because the `std::list<>` it uses only supports search algorithms with linear complexity. With *E. Adaptabilis*, however, there is an alternative. Specimen 8 provides a good example.

```
#include <iostream>
#include <set>
#include "Event.hpp"

using namespace std;

void log(int i) {
  clog << "void log(" << i << ")" << endl;
}

int main() {
  typedef std::set<void (*)(int)> container;

  Event<int,container> event;

  event.insert(log); // no need to store an
                     // iterator
  event.notify(8);
  event.erase(log); // efficient search and
                    // erase
  return 0;
}
```

Specimen 8 – *E. Adaptabilis* using a `std::set<>`

This variant uses a `std::set<>` of function pointers as its container. The insertion, removal and iteration operations are all "fairly efficient". By that I mean that, for most applications, efficiency is not an issue. And for very demanding applications it is always possible to use a variant of *E. Adaptabilis* based on specialised containers. It's even possible to use a specialised iteration algorithm thanks to the public access of the nested `notify_function` class.

Summary and Conclusion

This paper has described a recently discovered species of Event (*Eventus Adaptabilis*) with a remarkably wide range of habitats. *E. Adaptabilis* is closely related to the more common *Eventus Vulgaris* but is more adaptable in the following ways:

1. It accepts callbacks taking parameters convertible to its own argument type;
2. It accepts callbacks of ordinary function types or function object types;
3. It can store built-in pointers or smart pointers to callbacks;
4. It can use any of the standard containers and many other container types;
5. It is immune to callbacks that disconnect themselves from the event;
6. It allows user-defined iteration algorithms to be used.

It achieves all this without sacrificing efficiency and without forcing the programmer to store iterators. A rare specimen indeed.

Phil Bass
phil@stoneymanor.demon.co.uk

References

1. Gamma, Helm, Johnson and Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, ISBN 0-201-63361-2.
2. Phil Bass, "Implementing the Observer Pattern in C++ – Part 1", *Overload 52*, December 2002.
3. Phil Bass, "Implementing the Observer Pattern in C++ – Part 2", *Overload 53*, February 2003.

An Experience Report on Implementing a Custom Agile Methodology on a C++/Python Project

by Giovanni Asproni, Alexander Fedotov and Rodrigo Fernandez

In this article we'll describe our experience in implementing an agile methodology in a C++/Python project.

The promise of agile methodologies is to make software development faster and, at the same time, have higher code quality, higher customer satisfaction, and happier programmers. This is very appealing! However, their implementation is not easy, especially when it is the first one in the organisation, and the main language used is C++.

In fact, introducing a new methodology – and more generally, any kind of change – in an organisation may create several technical, human, and political problems. We'll describe the choices we made in order to minimise their occurrence, and how we managed to solve the ones we faced.

We'll start by briefly introducing the project. Then, we'll explain what agile software development means and why we decided to use an agile approach.

We'll describe the methodology with the rationale behind it, and we'll show the reasons why we decided to not pick an out of the box one like Extreme Programming [3].

Finally, we'll give an assessment of the results of our approach, including what worked well and what we would do differently the next time in similar circumstances.

Project Overview

The project consisted of the development of a family of C++ and Python programs used to produce, store, manage and distribute biological data to and from an Oracle database. These were developed for internal use.

The purpose was to substitute a family of old C programs that used text files as a storage medium. During the years the amount of data had increased so much that the usage of text files became a serious bottleneck in the data production process – the usage of files, and the way the programs were written, forced a serial process that was becoming too time consuming.

The usage of a relational database seemed the perfect solution to this: it could allow the execution of several production tasks at the same time, having the certainty that the data was kept consistent thanks to the isolation and synchronisation given by the transactional model. Implementing transactions in the C programs in order to keep the data in text files, would have been a very difficult task, and an obvious waste of time.

Six years before the three of us started to work on the project, some people already understood the limitations of those C programs, and started to work on their substitutes: a family of C++ programs that used an Oracle database (Python was something the three of us decided to introduce). During those years, the team size varied from a minimum of one to a maximum of five programmers. As far as we know, not all of them worked full-time on the project.

When we were assigned to the project, we inherited a code base of about fifty thousand lines of C++ code that had not been put into production yet. At that point, the project had become of critical importance to the business, and our task was to make it production ready in about one year – this meant fixing the bugs as well as implementing several new requirements.

When we started our work, the programming team was composed entirely of the three of us. None of us had been involved in the project before – all members of the original team either left the company, or were assigned to other tasks. From time to time, when some specialists joined the team, its size increased to four or five, to go back to three after a while.

Here is where our story begins.

The Beginnings

Unfortunately, the code we inherited was not in a very good shape – it was very difficult to modify and extend, there was not a single test and it was bug-ridden. Furthermore, there were a big number of (very volatile) new requirements to implement.

However, at first, we tried to improve it – part of the code-base was shared with another project that was maintained by a team with members in our office and in a room nearby, so we had the opportunity to ask questions and understand the code better. We added some tests, refactored some parts, and rewrote some others. But, after two months spent doing this, we realised that we were not making any real progress: the code was too tightly coupled in totally unpredictable ways. Fixing a bug caused others to show up from nowhere – and the number and the volatility of new requirements caused even more trouble. Furthermore, the code shared with the other project was in common for the wrong reasons – trying to exploit commonality where there was none – and that caused maintenance problems for the other team as well.

We analysed the situation, and decided that rewriting everything from scratch was a much better option: the basic logic of the programs was simple, and one year, if well used, was more than enough.

We had a few problems to solve though

- Convince our boss
- Convince our customer. This was the person that had the ultimate responsibility on the requirements, and also one of the users of the programs
- Organise our activities properly

To solve these problems, we decided to use an agile methodology. Before explaining why, we'll summarise what Agile Development is about.

Agile Software Development

"Agility is more attitude than process, more environment than methodology." Jim Highsmith [9]

The term Agile Software Development is a container that includes all the methodologies that share the values and principles stated on the *Agile Manifesto* (see sidebar on next page). The main characteristic of all these methodologies is that they are *people-centric* – i.e., they emphasise teamwork, face-to-face communication, and short feedback-loops. Furthermore, the processes and tools used have to be suited to the needs of the people involved: they may differ for specific methods, but the general recommendation in the agile community is to use the simplest ones that help in solving the problem at hand, and dismiss or substitute them as soon as they are not useful anymore.

That said, the following practices are becoming de-facto standards in all agile methods (and most of them are standard also on several non-agile methods as well)

- Extensive unit-testing
- Test-driven development (i.e., write the tests before the code)

- Short iterations (never longer than two months, usually 1-2 weeks)
- Merciless refactoring
- Continuous integration
- Configuration management
- Automatic Acceptance testing

Agile methodologies are especially suited for projects with highly volatile requirements, tight schedules and up to 10-12 developers. In fact, with more people involved, communication may become difficult. However some techniques to apply them to large projects are starting to appear [8].

The most important goal that this kind of methodologies tries to achieve is to deliver value to *all* the stakeholders of the project, including the *developers*.

The value delivered to the customer may be obvious: software that is fit for its intended purpose on time and within budget.

The value delivered to the developers is much less obvious: job satisfaction and self-motivation. In fact, the usage of an agile methodology can be a big help in increasing and keeping developers' motivation [2], and, more interestingly, self-motivation is the most important factor influencing developers' productivity [5]. An important consequence of these facts is that, delivering value to developers has a direct positive impact on the value delivered to the customer.

To learn more about agile development, a good starting point is the AgileAlliance web site [10].

The Agile Manifesto

The Agile Manifesto has been taken from [4].

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- **Individuals and interactions** over processes and tools
- **Working software** over comprehensive documentation
- **Customer collaboration** over contract negotiation
- **Responding to change** over following a plan

That is, while there is value in the items on the right, we value the items on the left more.

Principles behind the Agile Manifesto

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective methodology of conveying information to and within a development team is face-to-face conversation.
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
- Continuous attention to technical excellence and good design enhances agility.
- Simplicity-the art of maximising the amount of work not done-is essential.
- The best architectures, requirements, and designs emerge from self-organising teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Why an Agile Methodology?

At this point, many of the reasons why we decided to use an agile methodology should be clear.

First of all, we wanted to organise our development activities properly making the most out of the time and resources available. Our company didn't mandate the use of any particular methodology so we had total freedom of choice.

We wanted a methodology that was able to cope with unstable requirements. In our case, they were changing very often – sometimes from one day to the next – and we wanted to be able to respond to these changes as smoothly as possible.

We wanted to have early results and short feedback loops, so we could demonstrate our progress early and often.

We wanted to write high quality software with a clear architecture, no code duplication, and fully tested.

Each of us wanted to be involved in every development activity, from discussing the requirements to coding the solution.

We wanted to have fun, to learn new things, and to write code we could be proud of.

We didn't want to have too much overhead: we were on a tight schedule, so we wanted to work only on what was essential to achieve the goal.

The methodology had to be simple to implement but effective.

Finally, the methodology shouldn't get in to our way. It had to be natural to follow its practices.

Given these requirements, and the fact that one of us (Giovanni) had already experience with Extreme Programming and agile development, it seemed natural to use an agile methodology. However, we didn't choose one out of the box (e.g., Extreme Programming), because, doing that, there could have been the risk of focusing more on the methodology itself than on the goals of the project. What we did instead, was to follow a "methodology-per-project" approach [6]: we created our own one based on our needs and experience, in other terms we used our common sense.

The Methodology

Our methodology was based on some values: communication, courage, feedback, simplicity, humility, and trust. They were never stated explicitly as such – i.e., we never sat around a table to discuss them – but they were always present on the background.

We always strove to have open and honest communication among us and with the customer. We tried to make every assumption explicit – sometimes asking seemingly obvious questions – in order to avoid misunderstandings and wrong expectations.

We had the courage to take decisions and to accept accountability for them. We had the courage to never give up to pressure to deliver on unrealistic schedules – of course we always explained why, and tried to offer alternative solutions. We had also the courage to be honest, not hiding problems from the customer, or from our boss.

We had always the humility to recognise our own limitations, so we always listened to other people's opinions and advice, or asked for help when necessary.

We always looked for simple and clean solutions to the current problems. Our requirements were so unstable, that was pointless to try to predict the evolution of the system for a time period longer than one month.

We trusted each other; we knew that each of us had the skills and the motivation to make the project succeed.

The methodology itself was quite simple. Actually was easier to implement than to explain

- As every modern method (agile or not), it was iterative and incremental, with, usually, one-week long iterations. This was to give the customer – and us – a better feeling of progress and better control of the project. In fact the short feedback loop allowed her to change her ideas on some functionality without losing too much development time, and allowed us to know how well we were doing
- Team meeting every day before starting our activities. The meetings were usually ten minutes long. They were organised in a Scrum like fashion [11] [12]: everybody said what he did the day before, what problems he encountered, and what was going to do that day. Their purpose was to put everybody in synch with the others, and to decide how to solve the problems encountered along the way
- Metaphor. This was the very high level architecture of the system. Its purpose was to communicate with the customer, and to drive our development activities. It helped us in making all important design decisions. Every time we had a doubt on how to do something, the metaphor had a fundamental role in helping us finding a solution
- Coding standards. They were agreed at the beginning of the project. Their purpose was to allow us to guess the precise meaning of a name, to read code faster, and to give guidance on where to look for classes, functions, etc.
- Collective code ownership. Every member of the team had the power to change anything in the code to accommodate new requirements, or refactoring odd-looking code. Its main purpose was to avoid the typical problems created by "personal" code ownership, such as being dependent on a particular developer for the modifications in his or her parts of the system. It helped also in enforcing the coding standards.
- Collective accountability. If someone in the team made a mistake, the entire team would have been accountable for it in front of the customer and the boss. Its main purpose was to avoid scapegoating and finger pointing. Of course, in order to make this work, trust among us was essential
- Direct face-to-face communication was preferred to other forms of more indirect communication (i.e., written documentation, phone calls, e-mails, etc.). Its purpose was to make communication faster, and more precise
- Everybody was involved in all phases of development. Its purpose was to allow us to have fun, learn new things, and keep self-motivation high. It was also instrumental in improving the design of the system, since we always had brainstorming sessions before taking any important decision that could affect the architecture of the system
- Requirements prioritisation. Its purpose was twofold: to allow the customer to have the certainty of having the most important requirements implemented, and to give us a clear goal for each iteration
- Configuration management. Its purpose was to keep track of the modifications made to the system, rollback changes, and to be able to rebuild a specific version if we needed to. We managed the configuration of every artefact: source code, documentation, build scripts and also the external tools and libraries we used
- Extensive testing. Code was not finished until we had tests for it. Is purpose was to verify that the system worked as expected, and to allow us to make changes to the code with the confidence that we hadn't broken any existing functionality. As a side effect, it helped in improving our development speed as we proceeded because we had to spend less time hunting for bugs
- Merciless refactoring. Its main purpose was to keep the code clean and supple. One of its side effects (in concert with extensive testing) was to allow us to develop a set of reusable libraries. These, in turn, allowed us to increase our speed substantially, since, as we went along, the development of new functionality became more assembling of reusable assets and less development of new code
- Automation of all repetitive tasks-and of code generation when possible. Repetitive tasks are boring and slow if done manually, and bored people make silly mistakes. Enough said
- Never, ever, compromise design or code quality for any reason. The expected lifetime of the code we produced was in the order of several years, so it had to be maintainable and extensible. In conjunction with refactoring and testing, it allowed us to go faster as we proceeded
- Modify the methodology if necessary. Its purpose was to not get stubbornly stuck to practices or tools that were not useful anymore. For example, some times we modified the length of an iteration to fit our needs

In order to make our methodology work properly, we had to do also some complementary things right from the beginning of the project

- Define very clearly the responsibilities of the different stakeholders. This was fundamental to reduce problems and finger pointing in case of problems
- Ask our boss to not put any of us on other teams working on other projects at the same time. This would have been a teamicide [7], would have lead to time fragmentation, and would have made estimating impossible
- Strong emphasis on teamwork. We asked the customer to address her requests to the team, not individuals; for example, if she wanted to ask something by e-mail she had to address it to each of us
- During an iteration, the customer was not allowed to change the functionality being implemented. Its purpose was to avoid chaos, and had also the nice side effect of forcing the customer to think in more depth about her needs and priorities
- When possible, we used available code and libraries instead of writing our own – e.g., boost, CppUnit, Log4cplus, etc. This reduced development time, reduced the code base to maintain, and improved overall code quality
- Colocation in the same room, this had the purpose of increasing the communication bandwidth, and avoiding any of us being out of synch with the others – luckily we hadn't had to ask for this because we were already in the same office

From time to time we had to ask some "specialists" (i.e., Oracle database experts) for help. As far as we were concerned, they were akin to external consultants, and, since they worked on the project only for limited amounts of time, the fact that they were not located in our room was not a problem.

Every time we dealt with one of these specialists, we managed to learn enough from them so that, after a while, we could take over their tasks and carry on without their help. We acted as what Scott Ambler calls "generalizing specialists" [1]: each of us had some strong skills – e.g., C++ programming, or Object Oriented Design, etc. – but had also the ability and the will to learn and apply new ones.

We deliberately decided not to enforce the following practices used by Extreme Programming and other agile methods:

- Pair programming
- Test driven development (TDD)
- Planning game and story cards
- Automated acceptance tests written in collaboration with the customer

Pair programming all the time could be tough. It is a very intense experience, and it requires a bit of time to get used to. Apart from that, we thought that, in our specific case, we could work faster without pairing. However, from time to time, we used this technique, especially for working on some particularly difficult tasks.

Test driven development does not come naturally to everybody, and, since we wanted to implement the methodology in a way that was natural for each of us, we didn't mandate – or forbid – the usage of this practice. However, as time passed we started to use it more and more.

The planning game, story cards, and automated acceptance tests written with the user, were never used because we already introduced many changes, and we thought that introducing more could have shifted too much the attention from the product to the process, or overwhelm our customer. In fact she was very keen on manual acceptance testing, and also more traditional requirements gathering meetings.

To (partially) overcome the absence of automated acceptance tests, we wrote tests for high level functionality using CppUnit. This was less than ideal, but better than nothing.

Implementing the Methodology

"People hate change...
and that's because people hate change...
I want to be sure that you get my point.
People really *hate change.*
They really, really *do." Steve McMenamin as quoted in [7]*

Introducing any kind of change in an organisation can be very difficult. Humans are creatures of habit and tend to despise change. Furthermore, some people may perceive a change as a threat to their position of power. This fact can cause several political and technical problems.

In our case, in order to minimise the impact of the potential problems we implemented our methodology with great discretion. We never mentioned it explicitly – actually, we didn't even give it a name – we only explained our practices to our boss and our customer presenting them as simple common sense. The reason for doing that was to minimise the chance of having endless (and useless) methodological discussions.

In order to make the project succeed, we had to perform four concurrent, but strongly connected, activities:

- To manage the customer
- To manage the boss
- To carry out the technical tasks
- To have fun

These activities were all equally important. In fact, the failure of performing any of them could have made the project fail.

In the following paragraphs we'll show how our methodology helped us in carrying out these activities, leading to the successful completion of the project.

Managing the Customer

Managing the customer and her expectations was the most challenging and time-consuming activity. In many respects, she was a typical customer: she had the tendency to give us solutions, instead of explaining the problems; she had the very common "everything has top priority" mentality and, finally, we had to buy her trust.

All practices of our methodology had an important role in helping us to change her attitude, but face-to-face communication, incremental delivery, and short iterations were the ones that had a more immediate and direct effect.

Face to face communication is the fastest and more effective channel for exchanging information. Part of this information – that cannot be easily conveyed by other means of communication – is composed by the feelings, and emotions that are transmitted through gestures, and facial and body expressions. This last point is particularly important, for human beings are not entirely rational, and often make choices based on feelings or fears instead of logic or facts.

Being aware of that, we always presented facts to substantiate our choices, decisions, and proposals, but also took great care to be reassuring and look confident. This was helpful on several occasions.

We preferred face to face communication to written documentation to the point that, usually, we answered our customer's e-mails by going to her office and talking to her directly. This kind of answer was much more powerful than answering by e-mail for a couple of reasons: ambiguity and misunderstandings could be dealt with much more easily, and showing up in person sent a clear message of commitment to the project.

Finally, it was of great help in keeping her focused on the business side of the project, was instrumental in eliminating ambiguities from the requirements, solve problems more easily, helping her in prioritising the requirements, and in avoiding implicit assumptions and expectations by making every aspect of the project explicit and visible.

Incremental delivery and short iterations helped greatly in buying her trust. She could verify early and often our real progress by running the programs delivered at the end of each iteration. Furthermore, she could estimate the completion dates based on real data.

After a while, she started to realise the advantages of all the other practices as well, and to see how they reinforced each other.

Unfortunately, not everything went as well as we wanted. A couple of times we had to discuss some technical requirements with her (and our boss), and agree to implement an inferior solution for political reasons. Of course, when we did that we always made them aware of the drawbacks in order to avoid finger pointing in case of problems.

Managing the Boss

Managing the boss was easier than managing the customer. As long as the customer was happy, he was happy as well.

Initially, he was dubious about our decision of starting from scratch, and also about our approach. To convince him to give us a try, Giovanni – the officially appointed technical leader – accepted to link his career advancements and salary increase to the successful outcome of the project. This apparently careless gamble was actually a calculated risk. We had enough experience and domain knowledge to know that the risk of failure was quite low.

In fact, after a few months of work, the results were so good that our boss started to promote our way of working around the company. He also convinced us to give some seminars about configuration management and testing to other development teams.

Managing the Technical Activities

During our purely technical tasks, e.g., design, coding, and testing, we had been able to apply the methodology without effort. In fact it felt very natural to work in that way.

Since the languages used were C++ and Python, we hadn't the possibility of having any refactoring tool, but we never had any special problems for this reason. We managed to do merciless refactoring by hand quite well.

One of the main purposes of using Python was to reduce our coding effort. All C++ code used to access the database was generated by a Python program we wrote for that purpose. We accessed all the tables with the same pattern, so we could generate the C++ code directly from the SQL used to create the database. This saved us a huge amount of time and reduced the occurrence of bugs.

As the project went on, we were able to increase our development speed, due to the number of tests we had, to the absence of duplication, and to the refactoring that allowed us to improve our architecture and create a set of reusable libraries.

This last point deserves some explanation. A very common approach to create reusable libraries is to work bottom-up: trying to write them as reusable artefacts right from the start. Unfortunately, this approach doesn't always give the results intended, because trying to predict all possible uses is very difficult. Our approach, instead, was "use before reuse". When we had a specific problem to solve, the first thing we did was to look if we already solved a similar one. If we found it we would refactor the solution to an abstraction that could solve both of them; otherwise we wrote just the code for the specific case. Working in this way, we created several reusable libraries that helped in increasing our productivity quite a lot: we reached a point in which most programming tasks were mainly assembling components in our libraries.

The methodology was also instrumental in helping us solve some technical problems we had along the way.

Most of them were caused by our target operating system: Tru64 Digital Unix. Unfortunately, this wasn't a very well supported platform either by the software houses, or by the open source community. We had to write our own compilation scripts for boost, CppUnit, and other libraries. And the particular configuration of our hardware was less than optimal for developing software: as the code base grew, the compilation process became a painfully slow experience – from forty minutes to four hours, depending on the load of the machines.

To solve this we decided to port our code base under Linux and Windows – for which we had much faster machines that were also under our control – and compile under Digital Unix (usually once per day) only to run the tests when everything was already working on Windows and Linux.

Thanks to our extensive test suite, the full automation of the build process, and our obsession in keeping the code clean, the porting proved to be a painless experience. It had also a nice side effect: we ended up with truly portable software, with fewer defects. Some bugs that were missed from a compiler were almost certainly found by one of the others.

Having Fun

Fun is a fundamental part of work. It comprises all the things that make work something to look forward to. It comprises all the factors that motivate us in doing a job in the best possible way.

We decided, right from the start, that we wanted to really enjoy the project, and our methodology was strongly influenced by this decision.

Our strongest motivating factors (in no particular order) were

- Doing a quality job
- Delivering value
- Involvement
- Learn new things
- Having a challenge

The first three were directly addressed by our methodology.

The fourth factor was only partially addressed by it – i.e., learning a new approach to software development. So, we made some very deliberate choices to learn even more: we explicitly decided to learn more about Oracle, Python, and some modern (but not arcane) C++ techniques, and also made some design experiments.

The fifth was actually created by our boss, and our customer, with their initial belief that we were going to fail.

It was OK to experiment and make mistakes – we had configuration management and lots of tests to protect us from their consequences.

We certainly managed to enjoy the project, and this showed up in the final product, and in the satisfaction of the customer.

Conclusion

The first production quality version of the programs was released in about nine months. It had a very low number of bugs and no memory leaks.

Before releasing to production, our customer decided that she wanted even more functionality implemented, so the first production version was released in sixteen months after we started to work on it. So far there has been only one bug – that was easily fixed – in more than five months of operation.

Compared with the system we had to maintain when we were assigned to the project, the new one had much more functionality implemented, the performances were five to ten times better than the previous one, and was much easier to maintain,

The last point deserves some more explanation. Maintainability is something that is usually difficult to achieve, and even more difficult to quantify. So how can we claim to have written maintainable code?

We certainly put great care in keeping the architecture simple and tidy, in localising responsibilities as much as possible, and in writing clean self-documenting and well-tested code. But, more importantly, soon after the production release, there have been several changes and additions in the implemented functionality that really put maintainability under test: so far, the implementation of each of them has been quite straightforward, causing, at most, very localised changes in the code-base.

Even if the project has been quite successful, there are some things that, with hindsight, we would do differently. In particular, we would use TDD right from the start; we would push to have automated acceptance tests written in conjunction with the customer; and we would try to fight harder to avoid political compromises for deciding on purely technical issues.

Manual acceptance testing had been the cause of many problems, being a very time consuming and error prone activity. Sometimes the customer simply hadn't the time to run her tests, and, because of that, we couldn't proceed to work on the next iteration. For this reason, about four months out of sixteen have been wasted. Furthermore, it was easy to make mistakes during the tests' execution: several times, the customer came to us claiming, wrongly, to have found a bug in the code, when what had really happened was that she had made an error in the testing procedure.

[concluded at foot of next page]

Yet Another Hierarchical State Machine

by Stefan Heinzmann

Most of you are probably familiar with state machines. Finite state machines (FSMs) are widely used in both digital electronics and programming. Applications include communication protocols, industrial control or GUI programming. UML includes a more flexible variant in the form of statechart diagrams. Here, states can contain other states, making the corresponding state machine hierarchical.

Hierarchical state machines (HSMs) can be converted to ordinary (flat) state machines, so most implementors concentrate on the implementation of FSMs, and many implementations exist. The conversion however tends to lose the original structure of the HSM, making it difficult to make the connection between a statechart and its implementation. Clearly, it would be nicer if the HSM could be implemented directly in source code.

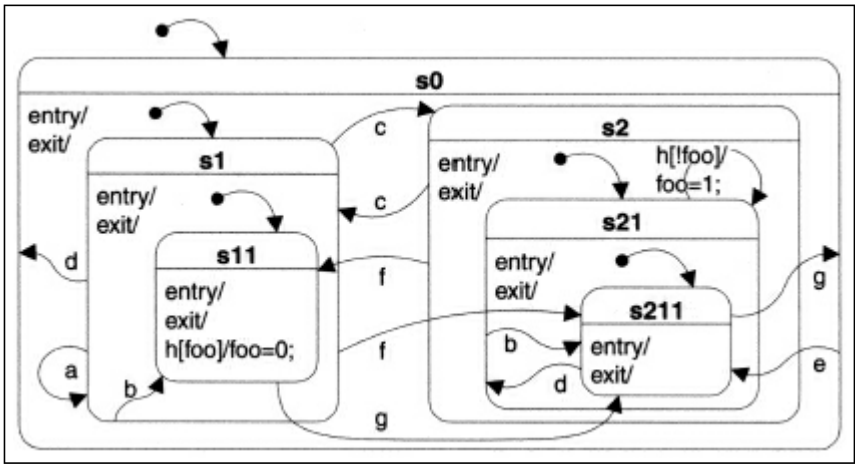
Direct implementations of HSMs seem to be comparatively rarely published. Miro Samek provides a prominent example in his book [1]. See the sidebar (next page) for some more information about his approach, which provided the motivation for the approach I will present here. UML tools are available that generate source code directly from statechart diagrams (example: Rhapsody [5]). More information about HSMs and Statecharts can be found in [1],[2],[3] and – of course – through Google.

The implementation I'm presenting here allows you to code your HSM in C++ directly, without the need for special code generation tools or wizards, although such a wizard could ease development further by converting the statechart automatically. Template techniques are used to enable the compiler to optimize the code through inlining, and I believe the resulting code ranks among the fastest you can get, provided you have a good up-to-date optimizing compiler (which you'll need anyway because of the templates).

Only a subset of UML statecharts are supported at present. If you need advanced features like concurrent states or history pseudo-states, you need to expand the solution yourself (tell me if you do so, it would be nice to have a complete implementation).

The TestHSM Example

It is best to describe how a hierarchical state machine works when you have an example. I lifted the statechart shown below from Miro's book. It specifies the state machine for the test example he implemented in his book ([1] page 95). We'll implement the same here. The example is artificial and only serves the purpose of providing a number of test cases for checking the correct implementation of the state machine. Its advantage is that it is quite small and therefore well suited for demonstration. It is driven by keypresses on the keyboard.



[continued from previous page]

The few technical compromises we had to make for political reasons always caused the problems we forecasted, that regularly gave rise to long and painful discussions about the reasons of the problems with our boss and with the customer.

Finally, we would keep the possibility of dynamically adapting the methodology to the needs of the project. We are aware of the fact that there is no one-size-fits-all methodology, but, from our experience, we are convinced that, sometimes, a methodology cannot fit even a single project if it is not flexible enough to change along with the needs of the project.

We don't claim that our methodology can work for every team, or project – in particular, we don't recommend anybody to put his or her career at stake as Giovanni did. However, we think that our approach can be used by other teams to develop their own methodology. In fact the most important assumption behind our choices (and behind the agile manifesto) was that the process and tools should fit the needs of the people involved in the project, not the other way round. In fact, happy programmers produce better software.

Giovanni Asproni
aspro@acm.org
Alexander Fedotov
fedotov@ebi.ac.uk
Rodrigo Fernandez
rodrigo.fernandez@bnpparibas.com

References

[1] Ambler, S., *Generalizing Specialists: Improving Your IT Career Skills*, http://www.agilemodeling.com/essays/generalizingSpecialists.htm

[2] Asproni, G., "Motivation, Teamwork, and Agile Development", *Agile Times Vol. 4*, 2004 http://www.giovanniasproni.com/articles

[3] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison Wesley, 1999

[4] Beck, K., et al., *The Agile Manifesto*, http://www.agilemanifesto.org

[5] Boehm B. W., *Software Engineering Economics*, Prentice Hall, 1981

[6] Cockburn, A., *Agile Software Development*, Addison Wesley, 2002

[7] DeMarco, T., Lister, T., *Peopleware: Productive Projects and Teams*, Dorset House Publishing, 1999

[8] Eckstein, J., *Agile Software Development in the Large: Diving into the Deep*, Dorset House Publishing, 2004

[9] Highsmith, J., *Agile Project Management*, Addison Wesley, 2004

[10] N.A., *AgileAlliance*, http://www.agilealliance.org

[11] Schwaber, K., *Scrum*, http://www.controlchaos.com

[12] Schwaber, K., Beedle, M., *Agile Software Development with Scrum*, Prentice Hall, 2002

An HSM is a state machine where states can be grouped into a composite state. Actions defined for such a composite state then apply automatically to all states contained therein. This allows a considerable simplification of the state diagram. In our diagram, you can easily see this. Each state is drawn as a box with rounded edges and bears a unique name. Composite states are those that contain other states. States that contain no other states are leaf states. The only leaf states in our example are `s11` and `s211`. Arrows represent the transitions between the states, they are labeled with an event that causes this transition to be taken. In the example, the transitions are annotated with the key that needs to be pressed to invoke the transition.

If a transition originates from a composite state, it is taken whenever a substate does not handle the corresponding event itself. A state can thus pass on the handling of a specific event to its enclosing state. For example, pressing the `e` key causes a transition to state `s211` regardless of the currently active state. Rather than cluttering the diagram with numerous transition arrows, it suffices to introduce an all-encompassing top-level state `s0` and handle the `e` key there.

This does not just simplify the diagram, it also points out where code reuse is possible. A statechart implementation should use this opportunity. We therefore need a possibility to pass unhandled events to the encompassing state. If no state handles the event, it is ultimately discarded. Each state can have special exit and entry actions associated with it. Those actions are executed whenever a transition leads out of or into a state, respectively. This is called an external transition. By contrast, an internal transition does not execute any exit and entry actions. Our state machine implementation needs to do the necessary bookkeeping to call the actions in the right order. In particular, the actions associated with a transistion are executed after all exit actions and before any entry action associated with the relevant states are executed.

During operation the state machine is always in a leaf state. So transitions ultimately lead from one leaf state to another. If a transition is drawn to target a composite state, the composite state needs to specify one of its substates as the initial state. In our example, the initial state of composite state `s0` is specified to be state `s1`. As this state is also composite, it needs an initial state, too, in this case `s11`. The effect is that any transition targeting state `s0` is in reality targeting state `s11`. A composite state that does not specify an initial state can not be the target of a transition. Our example diagram only contains two action specifications. In our code we will additionally print out trace messages for illustration, but the diagram does not show this. The action specifications shown are:

- The transition from `s21` to itself (a self-transition). This is an example of a transition that has a guard (in brackets `[]`) and an associated action (after the slash `/`). The guard is a condition that must evaluate to true to enable the transition. If it evaluates to false, the transition is not taken and none of the actions are executed. A self-transition exits and reenters the state, hence the associated exit and entry actions are executed.

Miro Samek's HSM Implementation

If coding state machines is one of your favourite pastimes you will surely have come across Miro Samek's book "Practical Statecharts in C/C++" [1]. Chris Hills reviewed it for ACCU quite favourably a few months ago. I can second this, yet I'm still in the game for new state machine designs. Why is that?

Well, you may have noticed that Miro's way of implementing state machines isn't typesafe and requires quite a few typecasts, neatly tucked away in a set of convenience macros. His implementation of hierarchical state machines isn't the fastest either, because of his way of handling entry and exit actions. There is a strong reason for this: His implementation works with just about anything that calls itself a C++ compiler, even ancient versions like VC++1.5. That means he completely avoids the "newer" C++ features like templates. If you are programming for embedded systems this is a good thing because "full" C++ compilers are only slowly gaining ground here.

State machines are more widely applicable than that, however, and even in embedded systems you may have the luck to use a compiler that attempts to support the full language, for example g++. Hence I believe there is a "market" for state machine designs that use the full language in order to address the deficiencies of Samek's design. This is what motivated me.

Miro's implementation represents the current state with a member function pointer that points to the state handler function for this state. This is an efficient representation, but it means that the handler function has to do double duty in that it also handles the entry and exit actions. For this, special reserved event codes are used, and a transition leads to potentially quite a large number of function calls through member function pointers. This is especially annoying when you realize that a large fraction of those will do little or no real work.

It also restricts your freedom in the way in which you can represent events. You are forced to use the predefined event class defined by Miro's framework, and some events/signals are predefined. The code presented here assumes nothing about the event representation. This aspect is left entirely to the concrete case you're concerned with.

Another difference is that Miro needs a number of typecasts, which are mostly hidden in convenience macros. This is because of the C++ restrictions in automatic conversion of member function pointer types. Miro's code works efficiently, but lacks type safety.

Miro works out which entry and exit actions are to be called in a function `tran()`, which does the work at runtime. This is very flexible as it allows dynamic transitions that can change at runtime. This comes at a cost, however, as there are potentially many handler functions that must be called without much work being done in them. As most transitions are static, he implemented an optimization that does the calculation of the transition chain only once and stores the result in a static variable. The code presented here only supports static transitions and calculates the chain at compile time, allowing inlining the entire chain. The result is typically both faster and uses less storage than Miro's code. Also, Miro found it hard to obey the UML rule that the actions associated with a transition are executed after any exit actions and before any entry actions are executed. His implementation executes the transition actions as the first thing, followed by all exit and entry actions. This makes exit actions a lot less useful. My code avoids these drawbacks.

The flip side is that Miro's code is more portable because the demands on the compiler are low. This is most welcome in embedded systems, where compilers often don't even attempt to implement the whole C++ standard. His solution is thus more widely applicable than mine.

Both implementations lack support for some more advanced features of UML statecharts, such as concurrent states or history pseudo-states. It is as yet an open question how difficult they are to add to the solution that I presented here. If you find you need such features and have an idea how to include them in the code presented here, I'd be interested to hear from you.

- The internal transition inside `s11` is not drawn with an arrow. It merely specifies an action that is to be taken when a certain event occurs, but no transition to another state occurs, and no exit or entry actions are performed. In our case the internal transition has a guard, so the associated action (`foo = 0`) is only executed when the `h` key is pressed while `foo` evaluates to `true`.

Note the difference between a self-transition and an internal transition. The latter never changes the active state and doesn't execute any exit or entry actions. Note also that internal transitions can be specified in a composite state, too, although this isn't shown in our example.

Representing the State

Flat state machines often represent the current state using a variable of `enum` type. Other implementations use a function pointer that points to the handler function for the current state. This handler function is called whenever an event occurs. Still other implementations represent states with objects, so the current state is represented by a pointer to the current state's object instance. This latter implementation is suggested by the State design pattern [6]. This is also the approach taken here, with the additional feature that all states have unique types to allow compile-time algorithms based on the state hierarchy. Only instances of leaf states need to be present, as they are the only states that can be active at any time. Composite states only exist as types, they are abstract and can therefore not be instantiated. The relationship between a composite state and its substates is modelled through inheritance. A composite state is the base class of its substates. All states derive from a top-level state class.

Often, entry or exit actions are empty or consist of trivial statements. I wanted to use the benefits of inlining as much as possible to allow the compiler to optimize away the overhead associated with functions that don't do much. I was prepared to dismiss the possibility of determining the target state at run time. If all transitions go from a source state to a target state, both of which are known at compile time, the compiler can figure out the entry and exit functions that need to be called and inline all of it into a single optimized string of code. My goal was to use templates to implement this compile-time task.

I chose therefore to represent the states as instances of class templates. Leaf states and composite states have separate templates. Each different state in the diagram is thus represented by a different instantiation of a predefined class template. Implementing state handlers and entry/exit actions is done by specializing member functions from this class template. If you don't specialize it, an empty default is automatically taken.

Here's the definition of the `CompState` and `LeafState` class templates:

```
template<typename H>
struct TopState {
  typedef H Host;
  typedef void Base;
  virtual void handler(Host&) const =0;
  virtual unsigned getId() const =0;
};
template<typename H, unsigned id,
         typename B> struct CompState;
template<typename H, unsigned id,
       typename B=CompState<H,0,TopState<H> > >
struct CompState : B {
  typedef B Base;
  typedef CompState<H,id,Base> This;
  template<typename X> void handle(H& h,
      const X& x) const { Base::handle(h,x); }
  static void init(H&); // no implementation
  static void entry(H&) {}
  static void exit(H&) {}
};
template<typename H>
struct CompState<H,0,TopState<H> > :
                                  TopState<H> {
  typedef TopState<H> Base;
  typedef CompState<H,0,Base> This;
  template<typename X> void handle(H&,
                           const X&) const {}
  static void init(H&); // no implementation
  static void entry(H&) {}
  static void exit(H&) {}
};
template<typename H, unsigned id,
      typename B=CompState<H,0,TopState<H> > >
struct LeafState : B {
  typedef B Base;
  typedef LeafState<H,id,Base> This;
  template<typename X> void handle(H& h,
      const X& x) const { Base::handle(h,x); }
  virtual void handler(H& h) const
                          { handle(h,*this); }
  virtual unsigned getId() const { return id; }
  static void init(H& h) { h.next(obj); }
                     // don't specialize this
  static void entry(H&) {}
  static void exit(H&) {}
  static const LeafState obj;
};
template<typename H, unsigned id, typename B>
const LeafState<H, id, B> LeafState<H, id,
                                    B>::obj;
```

And here's how you use this to specify the states of our example:

```
typedef CompState<TestHSM,0>     Top;
typedef CompState<TestHSM,1,Top>  S0;
typedef CompState<TestHSM,2,S0>     S1;
typedef LeafState<TestHSM,3,S1>       S11;
typedef CompState<TestHSM,4,S0>     S2;
typedef CompState<TestHSM,5,S2>       S21;
typedef LeafState<TestHSM,6,S21>        S211;
```

I used indentation to indicate state nesting. Each state bears a unique numeric ID code, starting with `0` for the top-level state which is outside of the all-encompassing `s0` state of our example. The ID code ensures that all states are distinct types. Except for the top-level state you have to specify the enclosing state for each state. This is how the hierarchy is defined. It leads to a corresponding class inheritance pattern, i.e. `Top` is a base class for all other states.

The `TestHSM` class is where the current state is held (it corresponds loosely to Miro's `QHsmTst` class). This class hosts the state machine. Actions are typically implemented as member functions of this class. As the states all have different types, we can only represent the current state through a pointer to the top-level state, from which all states are derived. Dispatching an event to the current state handler calls the `handler()` member function of the

current state through that pointer. The `handler()` member function thus needs to be virtual. All states that can actually be the current state, that is all leaf states, contain nothing but a vtbl-Pointer. So, ironically, they are objects without state.

The current state of the state machine is represented by a pointer to the corresponding state object.

```
const TopState<TestHSM>* state_;
```

Invoking the handler of the current state in response to an event is called dispatching, and it is done simply like this (assuming we're in a member function of `TestHSM`):

```
state_->handler(*this);
```

Note that only `LeafState` has a static member `obj`; `CompState` does not need it because it can't be instantiated anyway, as it does not implement the pure virtual functions inherited from `TopState`. The `LeafState` and `CompState` templates provide empty implementations for entry and exit actions. If you provide specialized functions yourself, they will be taken instead. This is how you implement your own entry and exit actions. More on this later.

Representing Events

An event is something that triggers actions and state transitions in the state machine. Without events, the state machine is sitting still and doing nothing. State machines are reactive systems. Events are not represented by anything predefined in this state machine implementation. You are essentially free to provide what you see fit for this task. The only thing you need to do is to call the event dispatcher shown above whenever an event happens. In order for the state handlers to determine what happened, you will also need to provide access to data associated with the event. For example, you could store an event ID-code in a member variable of the state machine's host class and have the state handler functions interrogate it to find out about the particular event at hand. Here's how our `TestHSM` class does it:

```
enum Signal { A_SIG,B_SIG,C_SIG,D_SIG,
              E_SIG,F_SIG,G_SIG,H_SIG };
class TestHSM {
public:
  TestHSM();
  ~TestHSM();
  void next(const TopState<TestHSM>& state)
                          { state_ = &state; }
  Signal getSig() const { return sig_; }
  void dispatch(Signal sig)
       { sig_ = sig; state_->handler(*this); }
  void foo(int i) { foo_ = i; }
  int foo() const { return foo_; }
private:
  const TopState<TestHSM>* state_;
  Signal sig_;
  int foo_;
};
```

Here, the event is represented by `enum` values corresponding to the actual key pressed on the keypad. On each keypress, the surrounding system needs to call `dispatch()` to invoke the dispatcher. In our example, we do it like this:

```
int main() {
  TestHSM test;
  for(;;) {
    printf("\nSignal<-");
    char c = getc(stdin);
    getc(stdin); // discard '\n'
    if(c<'a' || 'h'<c) {
      return 0;
    }
    test.dispatch((Signal)(c-'a'));
  }
}
```

You can see how the state machine is driven from the outside by calling `dispatch()` repeatedly. This is the essence of driving a reactive system. You call it each time something interesting happens. This is also easy to integrate with the message pump or event loop of a typical GUI, although I don't show this here (I would have to commit to a specific GUI, making it more difficult for you to try the code if you use a different system).

Your representation of events may be completely different from that in the example, and it is neither necessary to store it in a single member variable nor indeed do you need to store it in the state machine host class at all. You just need to make sure the handler functions can somehow get at it. This is easiest when it is stored in the host class, as a reference to the host object is always passed to the handlers.

Handlers and Actions

Implementing the handler functions is the central element of implementing the statechart. Here are the handler functions for our example. You may want to cross-check with the diagram while browsing through this source code. We implement a function template specialization for each state.

```
template<> template<typename X> inline void
    S0::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case E_SIG: { Tran<X,This,S211> t(h);
                  printf("s0-E;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
    S1::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case A_SIG: { Tran<X,This,S1> t(h);
                  printf("s1-A;"); return; }
    case B_SIG: { Tran<X,This,S11> t(h);
                  printf("s1-B;"); return; }
    case C_SIG: { Tran<X,This,S2> t(h);
                  printf("s1-C;"); return; }
    case D_SIG: { Tran<X,This,S0> t(h);
                  printf("s1-D;"); return; }
    case F_SIG: { Tran<X,This,S211> t(h);
                  printf("s1-F;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
   S11::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case G_SIG: { Tran<X,This,S211> t(h);
                  printf("s11-G;"); return; }
```

```
    case H_SIG: if(h.foo()) {
                  printf("s11-H;");
                  h.foo(0); return;
                } break;
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
    S2::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case C_SIG: { Tran<X,This,S1> t(h);
                  printf("s2-C;"); return; }
    case F_SIG: { Tran<X,This,S11> t(h);
                  printf("s2-F;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
   S21::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case B_SIG: { Tran<X,This,S211> t(h);
                  printf("s21-B;"); return; }
    case H_SIG: if(!h.foo()) {
                  Tran<X,This,S21> t(h);
                  printf("s21-H;"); h.foo(1);
                  return;
                } break;
    default: break;
  }
  return Base::handle(h,x);
}
template<> template<typename X> inline void
  S211::handle(TestHSM& h, const X& x) const {
  switch(h.getSig()) {
    case D_SIG: { Tran<X,This,S21> t(h);
                  printf("s211-D;"); return; }
    case G_SIG: { Tran<X,This,S0> t(h);
                  printf("s211-G;"); return; }
    default: break;
  }
  return Base::handle(h,x);
}
```

This is about as straightforward as is gets. Let's look at the last handler: `S211::handle()` as an example. If you check with the diagram, you can see that the `s211` state handles transitions associated with two events: Pressing `d` causes a transition to state `s21`, while pressing `g` causes a transition to state `s0`. Each of the transitions print a log message. The function `S211::handle()` implements this behaviour, and you should have no trouble making the connection between the diagram and the code. This simple handler function illustrates 3 points:

1. The event (key code) is retrieved from the host object using the `getSig()` function. A `switch` discriminates amongst the different events that are relevant for this state. The default case forwards the unhandled event types to the parent state. The `CompState`/`LeafState` class templates contain helpful typedefs to make this convenient. If no state handles the event, it ends up in the handler for the top state, where it is silently discarded by default. If you want a different behaviour, you may specialize the `handle()` function template for the `Top` state.
2. Actions are implemented as ordinary function calls, for example to member functions of the host class. In our example handler, the action is simply a call to `printf()`, which prints a log message.
3. Transitions are managed by yet another class template: `Tran`. The details of this are explained later, suffice to say that a `Tran` object is created on the stack in much the same way as a scoped lock object, and it is destroyed automatically at the end of the scope. At construction time all relevant exit actions associated with the state transition are called, and at destruction the relevant entry actions are performed. Also, the host object's state pointer is made to point at the new state. In between construction and destruction of this `Tran` object you can call any actions that are associated with this particular transition.

The UML statechart formalism allows a few more variations. It allows conditional transitions, that is transitions that are only executed when a guard condition holds true. This can be acommodated easily by testing the guard condition with an `if`-statement inside the corresponding `switch` case. The handler function `S21::handle()` illustrates this in `case H_SIG`. For an internal transition, you don't construct a `Tran` object. This is what is done in `case H_SIG` of `S11::handle()`.

The implementation of exit and entry actions is similarly straightforward:

```
// entry actions
template<> inline void S0 ::entry(TestHSM&)
                { printf("s0-ENTRY;"); }
template<> inline void S1 ::entry(TestHSM&)
                { printf("s1-ENTRY;"); }
// and so on...

// exit actions
template<> inline void S0 ::exit(TestHSM&)
                { printf("s0-EXIT;"); }
template<> inline void S1 ::exit(TestHSM&)
                { printf("s1-EXIT;"); }
// and so on...
```

Can it get any simpler? Here we just call the action routine that needs to be executed whenever a state is exited/entered. Again, we just print a log message, but anything could be done here. The only thing missing is the init routines, which are necessary for each state that has an initial transition. This initial transition may have an associated action, but usually just points to a substate.

```
// init actions (note the reverse ordering!)
template<> inline void S21 ::init(TestHSM& h)
     { Init<S211> i(h); printf("s21-INIT;"); }
template<> inline void S2 ::init(TestHSM& h)
     { Init<S21> i(h); printf("s2-INIT;"); }
// and so on...
```

As before, the action is the printing of a log message. Another special template `Init` is used to specify the transition to the initial substate. Please crosscheck with the diagram. In most practical cases, action routines will be members of the host class. This is hinted at in our example with the function `foo()`. This is where you put the actual code that implements the actions. The handlers only have the task of selecting the right action and state transition and invoke them in the right order. Try to keep detailed action code out of the handlers.

The Magical Tran Template

The most interesting part is the last: the `Tran` template that figures out which entry and exit actions to call:

```
template<typename C, typename S, typename T>
// Current,Source,Target
struct Tran {
  typedef typename C::Host Host;
  typedef typename C::Base CurrentBase;
  typedef typename S::Base SourceBase;
  typedef typename T::Base TargetBase;
  enum { // work out when to terminate
         // template recursion
    eTB_CB = IsDerivedFrom<TargetBase,
                           CurrentBase>::Res,
    eS_CB = IsDerivedFrom<S,CurrentBase>::Res,
    eS_C = IsDerivedFrom<S,C>::Res,
    eC_S = IsDerivedFrom<C,S>::Res,
    exitStop = eTB_CB && eS_C,
    entryStop = eS_C || eS_CB && !eC_S
  };
  // We use overloading to stop recursion. The
  // more natural template specialization
  // method would require to specialize the
  // inner template without specializing the
  // outer one, which is forbidden.
  static void exitActions(Host&, Bool<true>) {}
  static void exitActions(Host& h, Bool<false>){
    C::exit(h);
    Tran<CurrentBase,S,T>::exitActions(h,
                           Bool<exitStop>());
  }
  static void entryActions(Host&, Bool<true>) {}
  static void entryActions(Host& h,Bool<false>){
    Tran<CurrentBase,S,T>::entryActions(h,
                          Bool<entryStop>());
    C::entry(h);
  }
  Tran(Host& h) : host_(h)
        { exitActions(host_,Bool<false>()); }
  ~Tran() { Tran<T,S,T>::entryActions(host_,
            Bool<false>()); T::init(host_); }
  Host& host_;
};
```

It uses a gadget described in Herb Sutter's GotW #71 [4]. It is used to test at compile time whether a class `D` is derived from a class `B` either directly or indirectly. This is an important ingredient in the mechanism that figures out the exit/entry actions to call. Here it is:

```
template<class D, class B>
class IsDerivedFrom {
private:
  class Yes { char a[1]; };
  class No { char a[10]; };
  static Yes Test( B* ); // undefined
  static No Test( ... ); // undefined
public:
  enum { Res = sizeof(Test(static_cast<D*>(0)))
               == sizeof(Yes) ? 1 : 0 };
};
```

So how does `Tran` work? I explained already that all the exit actions are called when a `Tran` object is constructed and all entry actions are called when it is destructed again. Our states are all different types, so `Tran` needs to be a template. Its template parameters are the type of the current state (which is always a leaf state), the source state (where the transition arrow originates, which may be a composite state that contains the leaf state either directly or indirectly) and the type of the target state.

`Tran` now needs to walk up in the inheritance hierarchy of the current state (C) until it finds the common base class of current and target state (C and T), but it must not stop before the source state (S) was reached. From there it needs to descend the hierarchy down to the target state T. While ascending, it needs to call the exit actions of the states along the way, and when descending it needs to call the entry actions of the states along the way.

Ascending uses template recursion in `exitActions()`, and descending uses a similar recursion in `entryActions()`. The additional `Bool` parameter of these functions is used to terminate the recursion at the right point via overloading. Finding the right point is where Herb's gadget enters the picture.

The point where the recursion needs to terminate is at the first state that is common to both source and target state, in other words a common base class of both states. So when you are ascending from the source state you will eventually encounter a base class of the target state, and there's where the recursion must end. Similarly when ascending from the target state you will eventually encounter a state that is a base class of the source state.

So we know how to ascend from both ends towards the common base class, but we actually need to descend towards the target class once we have ascended from the source state, so it appears as if we've got it the wrong way up. But this is not a problem, as we can ensure the correct order of the entry routines by just swapping the recursion point with the invocation of the action as seen in `entryActions()`. The effect is that in `exitActions()`, the actual exit actions are invoked as we drill recursively into the inheritance hierarchy, while in `entryActions()` the entry actions are invoked as we work ourselves back out of the hierarchy.

You can now also see why there is a member function template `handle()` in the `CompState`/`LeafState` class template. Since `Tran` needs to know the current state in order to work out which entry and exit actions to invoke, it is necessary to pass it up the inheritance hierarchy in the default case of each handler function's `switch` statement. If we didn't do that, transitions handled in the handler for a composite state would miss the exit actions of its substates.

Finally, the handling of the initial state of a composite state deserves explanation. Remember that targetting a composite state with a transition leads you to the initial state specified within the composite state. The init action of a target state is always executed after executing the entry action. The init action of a leaf state is to announce itself to the host class as the new state. This behaviour shouldn't be changed. A composite state by default has no init action. So if you target a composite state with a transition, you will get a compile-time error, unless you specifically provide an init function for this composite state. Inside such an init function, you use the following `Init` class template to specify the initial substate.

```
template<typename T>
struct Init {
  typedef typename T::HostClass Host;
  Init(Host& h) : host_(h) {}
  ~Init() { T::entry(host_); T::init(host_); }
  Host& host_;
};
```

A Test Run

When you compile all the code for our example, you may run a little test to see whether the actions are called in the right order. Here's what I got:

```
top-INIT;s0-ENTRY;s0-INIT;s1-ENTRY;s1-INIT;s11-ENTRY;
Signal<-a
s11-EXIT;s1-EXIT;s1-A;s1-ENTRY;s1-INIT;s11-ENTRY;
Signal<-e
s11-EXIT;s1-EXIT;s0-EXIT;s0-E;s0-ENTRY;
                     s2-ENTRY;s21-ENTRY;s211-ENTRY;
Signal<-e
s211-EXIT;s21-EXIT;s2-EXIT;s0-EXIT;s0-E;
            s0-ENTRY;s2-ENTRY;s21-ENTRY;s211-ENTRY;
Signal<-a

Signal<-h
s211-EXIT;s21-EXIT;s21-H;s21-ENTRY;s21-INIT;
                                        s211-ENTRY;
Signal<-h

Signal<-x
```

You'll notice that Miro's implementation renders a different result (page 100 in [1]). Notably, the actions associated with a transition are executed before any exit actions in Miro's version. This violates the UML rules, but Miro explains that this was deliberate, as obeying this rule would have made his implementation significantly more complicated. My code obeys the rule with no noticeable hit on code performance.

Furthermore, note that when pressing the `e` key, `s0` is exited and reentered. This is not immediately obvious from the way the diagram is drawn. In fact, Miro's code doesn't show this behaviour. The UML definition seems to specify the behaviour of my code, although this isn't entirely clear to me. It certainly is more consistent with the behaviour of self-transitions. If my interpretation turned out to be correct, it would be clearer to draw transition arrows in UML statecharts such that they always leave the source state boundary towards the outside, and also enter it from the outside. Hence the exit action of the source state is always called, even when the target is a substate of the source state. Likewise the entry action of the target state is called even when the source state is one of its substates.

Efficiency of the Generated Code

With such a lot of templates, you might worry about the kind of code generated. Templates are still being accused of causing code bloat, a reason why embedded programmers in particular still hesitate to use them. If used wisely, however, templates in conjunction with inlining can actually reduce the amount of code produced. So how does the system presented here fare in this respect?

Given a good quality compiler and a sensible setting of compiler switches, all `handle()`, `init()`, `entry()` and `exit()` functions will be inlined into the virtual `handler()` function for a state. As a result, you get as many handler functions as there are leaf states. A good compiler will also be able to fuse the `switch` statement of a `handle()` function with those from the `handle()` functions of the base classes, so that you effectively get a larger `switch` incorporating all cases that need to be considered in a state. The result is the same as if you had converted the hierarchical state machine into a flat one, taking all entry and exit actions into account, and implemented the handler function for each flat state manually. The code generated literally is exactly the same. The templates flatten the hierarchical state machine into a simple state machine and generate the code for that.

In particular, empty actions don't produce any code at all, not even a call to an empty procedure. If the entry and exit actions associated with a particular transition are also empty, the transition simmers down to a single assignment to the host class's state variable, which on many processors is just one or two instructions.

The result is probably about as fast as it gets, but there is no code sharing between states. This is the reason why you should not include a lot of action code in the handlers, but rather call a corresponding action function in the host class. This is particularly true for handler functions of composite states.

Afterword

Templates can be used to advantage here in order to allow the compiler to thoroughly optimize the code. It is even fairly readable and requires neither liberal casting nor preprocessor macros as the solution described by Miro Samek. It does require a dose of template metaprogramming, and this may challenge your compiler.

So we've got a balance of advantages and drawbacks:

+ Transitions are worked out at compile time, allowing generation of very efficient inlined code
+ Malformed statecharts are caught at compile time
+ Stylized code lends itself well to automatic code generation
+ The code is typesafe and doesn't need casts
+ Complete flexibility in representing events
– We need full template support in the compiler
– All transitions must be static (known at compile time)
– Only a subset of the functionality of UML statecharts is supported

[concluded at foot of next page]

boost::fsm

The well-known boost library [7] is about to acquire statechart support. Andreas Huber has developed a library that aims to cover the entire functionality of UML statecharts, and it should appear in one of the next official releases of boost. Until it is accepted, you may have a look in boost's sandbox: `http://boost-sandbox.sourceforge.net/libs/fsm/doc/index.html` is the entry point to the documentation accompanying `boost::fsm`.

Some of the design goals of `boost::fsm` match mine. Both facilitate direct coding of statecharts in C++ without the aid of special code generation tools. Such tools ought to be pretty straightforward in both cases, however. Both solutions are type safe and detect malformed statecharts at compile time.

The support of `boost::fsm` for the complete UML semantics makes it less efficient, although it should still surpass the efficiency of Miro Samek's implementation in many cases. In particular, entering a state is done through construction of a state object. It gets destructed again when exiting the state. As states are allocated using `operator new`, the heap manager is excercised unless you overload `operator new` and `operator delete`. This is done so that you can include your own extra data members with a state.

No dynamic allocation happens in the solution I introduced here. As noted, the resulting code should be fast and consume little memory. The downside is of course that some major facilities of UML statecharts aren't supported.

The benefit is yours: You've got a choice between a restricted, efficient solution and a flexible, universal solution.

A Mini-project to Decode A Mini-language – Part Two

by Thomas Guest

Part 1 of this two-part article [15] described the preliminary stages of a mini-project to write a codec for a mini-language, delivering:

- a rough specification of the codec,
- a suite of test data,
- some prototype code,
- three implementation strategies.

Part 2 continues the project and presents the final implementation.

Motivation

Part 1 of this article drew inspiration from "The Art of UNIX Programming", by Eric Raymond [13]. Part 2 continues to draw from this same source, which applies as readily to implementation as it did to design.

At this point, I can reveal a second motivating source, "The Tale of a Struggling Template Programmer", Stefan Heinzmann [7], which served to remind me how frustrating software development can be: sometimes the tools are to blame, sometimes the languages appear faulty, and sometimes the poor programmer takes a wrong turn. More personally, it reminded me that I ought to experiment with modern C++[1].

Anyone familiar with both sources will appreciate there's a degree of tension between them. In what follows, I document my attempts to resolve this tension. Along the way, we shall revisit the world of MPEG video encoding and get started with the Boost Spirit library.

Project Recap

To briefly recap, then, our goal is to write a tool to convert the binary format used in MPEG-2 digital video broadcasting into a textual form and back again – to write a `dvbcodec`. For example, we would like to convert a section of the Program Association Table (PAT), whose syntax is as follows:

```
program_association_section() {
  table_id                    8
  section_syntax_indicator    1
  '0'                         1
  reserved                    2
  section_length             12
  transport_stream_id        16
  reserved                    2
  version_number              5
  current_next_indicator      1
  section_number              8
  last_section_number         8
  for(i=0; i<N; i++) {
    program_number           16
    reserved                  3
    if(program_number == '0') {
      network_PID            13
    }
    else {
      program_map_PID        13
    }
  }
  CRC_32                     32
}
```

[ISO/IEC 13818-1] Table 2-26 – Program association section

The numerical values here represent field widths in bits: the first byte of the section encodes the `table_id`, the next bit the `section_syntax_indicator`, and so on until the final four bytes which encode the cyclic redundancy check.

The PAT is just one of the tables we would like to decode. There are many others, the next three most important being the the Conditional Access Table, the Program Map Table and the Network Information Table (CAT, PMT and NIT).

The textual output format we decided on should reflect the syntax description as follows:

```
program_association_section() {
  table_id                    8 = 0x0
  section_syntax_indicator    1 = 0x1
  '0'                         1 = 0x0
  ...
  CRC_32             32 = 0xcae52d9f
}
```

We came up with three possible implementation strategies for our `dvbcodec`:

- Implement a pat-codec. Then implement a cat-codec, then a pmt-codec, etc.
- Implement a general codec which understands the full bitstream syntax and can use it to parse an arbitrary section format. All that then remains is to prime this codec with the required section formats.
- Devise a code generator which, given a section format, will generate a program to encode/decode that particular format.

Towards a Solution

The first strategy holds little appeal: it risks being a recipe for cut-and-paste code and boring repetition. We reject it.

1 My job involves writing portable C++ to run on embedded platforms. The compilers supplied for these platforms often do not support "modern" C++ features such as templates.

[continued from previous page]

If you find this approach useful, have improvements or comments on offer or bugs to fix, I'd like to hear from you.

I'd like to thank Miro Samek and Andreas Huber for discussion and advice as well as for their work on HSM implementations. My work wouldn't exist without theirs. Thanks also to the Overload reviewers.

Stefan Heinzmann

References

[1] Miro Samek, *Practical Statecharts in C/C++*, CMP Books 2002. There's a companion website with additional information: `http://www.quantum-leaps.com`

[2] Booch, Rumbaugh, Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley

[3] Rumbaugh, Jacobson, Booch, *The Unified Modeling Language Reference Manual*, Addison-Wesley 1999

[4] `http://www.gotw.ca`

[5] `http://www.ilogix.com`

[6] Gamma, Helm, Johnson, Vlissides, *Design Patterns*, Addison-Wesley 1995

[7] `http://www.boost.org`

The second and third strategies look to have more going from them, particularly since we have restricted our scope to a subset of the full bitstream syntax. Although these strategies appear rather different, they both require us to parse syntax descriptions of the general form exemplified by the `program_association_section`.

So, we need a parser. We need one capable of handling conditionals and loops: one capable, that is, of handling a Turing-complete mini-language. Raymond [13] can advise. In general terms, he suggests:

- Where possible, reuse. Look for a proven, documented, supported, portable, parser. (He argues these criteria pretty much imply an open source solution.)
- Prefer scripting languages such as Python and Perl. These facilitate rapid development and are less prone to memory management bugs. You may not need the raw performance offered by C/C++, and the library support offered by these languages is often superior.

On the more specific subject of parsers, Raymond recommends `lex` and `yacc` though, in keeping with the Unix philosophy of documenting weaknesses, he admits these tools are not perfect. He also suggests:

> *"If you can implement your parser in a higher-level language than C (which we recommend you do ...) then look for equivalent facilities like Python's PLY ..."*

I tend to agree with Raymond, but I'm not convinced PLY is the way to go here. Of course, it won't get me very far with my aim of finding out about modern C++, but it's also not part of the standard Python distribution. In fact, a web search reveals several other Python parser frameworks – it's unclear which will prevail.

The C++ standard library doesn't provide a parser either. We might make some progress tokenising our data with `std::strtok` or even `std::sscanf`, but this won't suffice. `lex` and `yacc` are of course a tried and tested solution, but I'd rather not have to learn two more mini-languages.

The next place to look is in the next best thing to the C++ standard library, namely Boost [3]. Three clicks from the front page takes us to the Spirit parser, which claims to be a scalable parser framework written in C++. We trust the source, the documentation is good, the examples compile: let's try some code.

Getting Started With Spirit

The code below is a complete program to recognise lines of the form:

```
reserved 2 = 0x3
```

this being the format we arrived at for fields of our text sections.

```
#include <boost/spirit/core.hpp>
#include <iostream>
#include <string>

using namespace boost;

/**
 * @brief Parse a string representing a field
 * @returns True if the field matches the
 * format: <field_name> <bitwidth> = <value>,
 * false otherwise.
 */
bool parseField(std::string const & str) {
  return spirit::parse(
      str.begin(),
      str.end(),
      spirit::lexeme_d[+spirit::graph_p]
      >> spirit::uint_p
      >> '='
      >> spirit::hex_p,
      spirit::space_p).full;
}
int main() {
  std::cout << "Enter text.\n"
        << "Lines will be matched against: \n"
        << "<field_name> <bitwidth> = "
        << <hexvalue>\n"
        << "Type 'q' to quit\n";

  std::string str;
  std::string const quit("q");

  while(std::getline(std::cin, str) &&
        str != quit) {
    std::cout << (parseField(str)
                        ? "hit" : "miss")
              << std::endl;
  }
  return 0;
}
```

Here, the action is concentrated in the function `parseField()`, which wraps a call to `spirit::parse()`. `spirit::parse()` accepts as arguments:

- two iterators marking the start and end of the data to be parsed,
- a parser,
- a skip parser.

We have used `spirit::space_p` directly as our skip parser: this primitive parser recognizes whitespace and tells `spirit::parse()` which characters it should skip past in the input. A more sophisticated skip parser might be used to skip comments.

Operator Overloading

The parser itself is a sequence of sub-parsers which can be read: recognise input consisting of a block one or more printable characters, followed by an unsigned integer, followed by the equals sign, followed by a hexadecimal integer.

Operator overloading is used by Spirit to make such expressions into readable approximations of EBNF syntax descriptions (see also [1] for more on this technique). Here, we see that `operator<<()` has been overloaded as a sequencing operator, prefix `operator+()` has been overloaded to mean "one or more of", and `operator[]()` is overloaded to adapt the behaviour of a sub-parser – in this case using `spirit::lexeme_d` to turn off whitespace skipping.

Parser Generators

I should also mention that the `'='` sub-parser is a shorthand for `spirit::ch_p('=')`, which in turn is a parser generator returning the character literal parser `spirit::chlit<char>('=')`.

Similarly, `spirit::hex_p` and `spirit::uint_p` are parser generator functions which return suitable specialisations of the `spirit::uint_parser` template struct. The full template parameters of this struct are as follows:

```
template<typename T = unsigned,
         int Radix = 10,
         unsigned MinDigits = 1,
         int MaxDigits = -1>
struct uint_parser { /* */ };
```

The helper functions `hex_p` and `uint_p` are often good enough, but it's also useful to have the full flexibility of the base parser. For example, if we need to match larger hex values, and `long long` is available, we could create an alternative hex parser:

```
uint_parser<unsigned long long, 16> const
  long_long_hex_p
    = uint_parser<unsigned long long, 16>();
```

In fact, the `uint_parser` should work with any user defined scalar type.

(You've probably noticed I'm now working in the `boost::spirit` namespace. I continue to do so for the remainder of this article.)

One thing I cannot do with the hex parser, unfortunately, is get it to accept the `0x` we've used to prefix hex digits. We can fix the bug in our program by introducing a new parser rule.

```
with_base_hex_p
  =  lexeme_d
     [
       as_lower_d["0x"]
       >> hex_p
     ];
```

Note here:

- the `as_lower_d` directive, which converts all characters from the input to lowercase, and therefore recognising both `0x` and `0X`.
- the rather unusual code layout. I have tried to follow the Spirit style guide [17] when writing parser grammars. This will become increasingly more important when we develop a more substantial grammar.
- the string literal `"0x"`, which in this context becomes yet another parser.

Semantic Actions

Simply recognising fields is not enough: we need to act on their contents. That is, we must associate semantic actions with the sub-parsers. This can be done using another overload of `operator[]()`, which enables us to link an action to a parser.

Here, then is an encoder which will convert text versions of sections to binary. I have omitted `#include` directives etc. for brevity. The full implementation is available with the source distribution for this article [8].

```
typedef std::string::const_iterator iter;

/**
 * @brief Put the input value to the output
 *  stream using the specified bitwidth
 */
void putBits(std::ostream &, unsigned w,
             unsigned v) { /* */ }
/**
 * @brief Parse a field of the form:
 *  <field_name> <bitwidth> = <value>
 */
bool parseField(iter const & begin,
                iter const & end,
                unsigned & bitwidth,
                unsigned & value) {
  return parse(
      begin,
      end,
      lexeme_d[+graph_p]
      >> uint_p[assign_a(bitwidth)]
      >> '='
      >> lexeme_d
         [
            ! as_lower_d["0x"]
            >>  hex_p[assign_a(value)]
         ]
      ,
      space_p).full;
}
int main() {
  std::string str;
  int line = 0;
  try {
    while(std::getline(std::cin, str)) {
      ++line;
      unsigned bitwidth, value;
      if(parseField(str.begin(), str.end(),
                    bitwidth, value)) {
        putBits(std::cout, bitwidth, value);
      }
    }
  }
  catch(std::exception & exc) {
    std::cerr << "Error parsing line " << line
              << "\n" << str << "\n"
              << exc.what() << std::endl;
    return -1;
  }
  return 0;
}
```

Note here:

- I have used typedefs for the iterators passed into the parser. This will ease switching to another forward iterator type, if required.
- I decided to make the `0x` preceding hexadecimal values optional, using Spirit's overload of `operator!()`
- The use of the `assign_a` actor for our semantic action. We could have used any function accepting an unsigned integer or any functor providing `operator()(unsigned int)`. Again, it's simpler to use one of Sprit's off-the-peg actors.
- The program implements a classic Unix filter. This makes it suitable for use in a Unix pipeline. See [13] for more on this. Unfortunately, I'm not sure this is a great idea for binary output: I haven't found a portable way to reset `std::cout` for binary.

Exceptions in Parsers

Another important point to note about our simple encoder is the way it handles failure conditions using C++ exceptions rather than C-style error codes. There are plenty of failure conditions to handle: a value might not fit in the available bitwidth, the output stream might not be in a suitable state, and so on.

In this simple parser we might equally well have passed error codes around, but a more complex parser is likely to involve recursion and/or nested function calls. Exceptions perform well in both the simple and the complex case, offering a scalable solution.

The Spirit parser framework itself uses exceptions internally for similar reasons. To quote the documentation:

> *"C++'s exception handling mechanism is a perfect match for Spirit due to its highly recursive functional nature. C++ Exceptions are used extensively by this module for handling errors."*

Like our program, Spirit should not leak any such exceptions to its users.

Weaknesses

The simple encoder presented above follows Postel's prescription, to a degree [11]. It doesn't mind too much about whitespace; it allows any sequence of printable characters as a field name; and it isn't fussy about the presentation of hexadecimal numbers.

Its main flaw is that it does not look at the text format of our sections as a whole: it simply skips the lines which close blocks or start loops, for example. This means the encoding will quietly do the wrong thing given input where a new-line has gone missing, or where the data has been truncated. This is dangerous. It also means the encoder cannot check the integrity of our text data – for example, to confirm the `section_length` field contains the actual section length, or to validate a CRC.

When we start thinking along these lines, we realise that perhaps the encoder should calculate the values of these fields for us. We'll need a CRC generator anyway – why not embed it in the encoder?

These are important points. However, we never considered data validation when we planned our codec and I'm not going to worry about it just yet – I need to get started on the decoder. Data validation, though valuable, would need to be optional since an encoder must let us generate broken data for test purposes.

Also, Raymond [13] encourages us to limit options whenever possible: if we can release code earlier then our users can tell us which options they really want. Ideally, he suggests we make the release open-source, and allow users to (submit patches which) implement these options.

Progress Review

We've used Spirit to write a micro-parser to drive the encoder. We're ready to start on the decoder. Spirit's scalability will be tested.

The Decoder

I decided to attempt the second implementation strategy: to develop a codec which understands the bitstream syntax and can use it to parse an arbitrary section format. I had no good reason for preferring this to the code-generator strategy.

As already noted, this is a parsing task. We will use Spirit to define the grammar used by the bitstream syntax. We can then parse our static program data – the section formats we're interested in – which gives us the basis we need to parse the run-time program inputs, that is, actual instances of binary encoded sections.

Grammar Definitions and Parse Trees

I do not propose to dwell on the practical use of Spirit for much longer: we've already seen enough of what it can do, so for full implementation details please refer to [16] and the codec source distribution [8].

For the decoder, note that simply parsing the data once and associating semantic actions to the various lexical elements is not enough. For instance, to process descriptor loops we need to revisit the same parser node several times. Spirit provides abstract syntax trees for exactly this purpose.

I do think it is worth presenting here a portion of the section grammar. To me, this is a quite remarkable application of C++. For even more remarkable transcriptions of EBNF syntax definitions into Spirit grammars – including a C++ tokenizer and a C parser – I would recommend a visit to the Spirit Applications Repository [14].

```
/**
 * @brief MPEG-2 Section grammar defined
 * using Boost Spirit.
 * Reference: - ISO/IEC 13818-1, MPEG-2
 *              Transport Stream
 */
struct Section :
  public boost::spirit::grammar<Section> {
  template <typename ScannerT>
  struct definition {
    definition(Section const & /*self*/) {
      section_
        =  section_ref_
           >> section_body_
        ;
      section_ref_
        =  text_id_
           >>  '('
           >>  ')'
        ;
      text_id_
        =  leaf_node_d[
             lexeme_d[
               alpha_p
               >> * (alnum_p | '_')
             ]
           ]
        ;
      quoted_binary_
        =  leaf_node_d[
             lexeme_d[
               '\''
               >>  bin_p
               >> '\''
             ]
           ]
        ;
      section_body_
        =  ch_p('{')
           >> *(    field_
                  |  loop_
                  |  conditional_
                  |  section_ref_
                )
           >> '}'
        ;
      field_
        =  identifier_
           >>   uint_p
        ;
      identifier_
        =  text_id_
           |  quoted_binary_
        ;
      conditional_
        =  str_p("if")
           >> condition_
```

```
            >> section_body_
            >>  ! (str_p("else")
                   >>  section_body_)
          ;
        condition_
          =  inner_node_d['('
               >> text_id_
               >> "=="
               >> quoted_binary_
               >> ')'
             ]
          ;
        loop_
          =  loop_control_
             >> section_body_
          ;
        loop_control_
          =  leaf_node_d[str_p("for")
               >>  '('
               >>  * (anychar_p - ')')
               >>  ')'
             ]
          ;

      }
      ...
      boost::spirit::rule<ScannerT> const &
      start() const {
         return section_;
      }
    };
  };
```

Decisions Taken

Many of the decisions taken when writing our naive encoder scale up to the decoder: limited use options; exceptions used in preference to error codes; Spirit style guide for grammar definitions; typedefs for iterators.

Some decisions were ones we haven't yet faced. The main one was: where should we put section format definitions (for the PAT, CAT, PMT and NIT)? There are two obvious alternatives:

- create a C++ source file containing these definitions – perhaps as an array of string literals,
- place them in a text file in a known location, and have this file read when the decoder starts up.

The second alternative is perhaps most faithful to our original aims. Program logic and program data are nicely separated, and extending the decoder to handle other sections is a simple matter of editing the text file. No recompilation necessary.

Despite these attractions, I went for the first option – partly because it's easier to implement and partly because I didn't want to work out where to put the text file.

The other corner I cut concerns determining how and when to exit loops. The issue is fully described in the first part of this article (see the subsection headed "Complications"). My resolution was to notice that loops always exit when we've used up the number of bytes encoded in a `length` field – with the single exception of the outermost loop, which may end four bytes early in order to leave space for a CRC. So, the decoder maintains a stack of `length` fields, testing the top value after each loop iteration, popping it on loop exit. The first item to be stacked may need adjusting to allow for the four byte CRC. Again, the source distribution should make this clear.

I could find no official statement regarding what could be used as a field name in the section format definitions: inspection suggested that these names were rather similar to C identifiers, with the important addition of quoted binary values for fixed fields (e.g. the `'0'` which is the third named field in the `program_association_section` format definition).

A few more parse tree node directives might have resulted in a leaner decoder, but I wanted the syntax grammar to be as simple as possible. I am inexperienced in writing grammars and preferred a small amount of extra code in my application. The application is quite compact anyway.

Ship Happens [1]

Raymond [13] has lots of practical advice on how to ship a source code distribution, going down to details of file naming conventions. Some of the suggestions I have followed are:

- choose a suitable license
- include a README
- set up a project homepage [8]
- include a BUGS file, listing known defects and limitations
- include platform/compiler details
- include self-test code

The BUGS file is a strangely satisfying thing to write, particularly if you've ever delivered software which doesn't admit to defects, let alone limitations (or indeed if you've ever used such software). In this case it is essential to document both.

Version 0.1 of the `dvbcodec` features the naive encoder described in the preceding – really only of use for system testing (we can check that decoding then encoding a file recreates the original file). The decoder is rather better – in fact, I've extended it beyond the original specification to handle a few more section formats: `dvbcodec -l` gives details.

Having done the hard work and shipped our beta release, the rest of this article is dedicated to some closing thoughts.

Is C++ the Right Tool for the Job?

My criteria for language selection were somewhat artificial. If I had been allowed a free hand I almost certainly would have been biased towards Python [12]. However, having gone the C++ route – the modern C++ route, even – it would seem a good point to step back and review my selection.

Raymond [13] has reservations about C++, which I summarise here:

1 By not automating memory management it fails to address C's biggest shortcoming; and backwards compatibility with C has compromised the language's design.
2 Object oriented software design isn't all it's cracked up to be. All too often it leads to shaky object hierarchies, unnecessary abstractions and code which is hard to maintain,
3 C++ is so complex that no one programmer can be expected to know it all,
4 If C++ really were superior to C, it would now dominate it.

(Incidentally, as already mentioned, Raymond is not knocking C++ to promote C. His recommendation is to adopt scripted and mixed language solutions.)

To fully address all these points is outside the scope of this article. Instead I shall look at each in the context of the development of our codec:

1 By using standard library containers – `map`, `vector`, `stack`, `string` etc – I have avoided a single direct call to `operator new`. If I've got my exception handling and my use of Spirit right, there should be no leaks. Regarding C compatibility, I have benefited from the C standard library in a few places. Converting from C string literals to C++ strings is convenient.
2 The application code (as opposed to the Spirit framework code) uses only two concrete classes. I have resisted the temptation to make these generic, or to make them derive from an abstract class. The RAII class idiom is usefully employed. The Spirit framework itself has moved with the times: what was "implemented with run-time polymorphic classes" is now "a complete rewrite ... using expression templates and static polymorphism".
3 Agreed! Spirit's fine documentation includes examples which have served as a basis for my own application. When I deviated from these examples too far the result was a barely comprehensible torrent of compiler errors. Typeless programming in a strongly typed language can be tough[2].
4 C is far more portable. I believe our codec is standards compliant, so maybe in the long term it will be portable. However, at the moment (September 2004) the list of compilers which cope with Spirit is small. So our codec isn't really portable. Not one of the compilers I use at work could cope with this program. This reflects my experience with C++ over the years: to get the good stuff, either you wait, or you work around compiler limitations. Bear in mind too that of two types of compiler bugs – incorrect error messages, no object code; no error messages, incorrect object code – the second is far more insidious and dangerous: and the presence of the first naturally leads a programmer to suspect the existence of the second.

Despite all this, Spirit has proven itself flexible, scalable, capable of expressing grammars clearly and of writing efficient parsers without the need to step outside C++. Indeed, it could never have been done without C++. I am sure I will use the Spirit parser framework again.

Open Source

The future of Unix is Linux is open source. Raymond is passionate about software quality and argues convincingly that open source the best way to deliver the highest quality software. I do not propose to rehearse these arguments here: Raymond's writings are available both in print form and online. (See, for example [4]).

What does interest me is that I cannot see how the full power of Spirit could be realised using anything other than a full source code distribution. It's all done with header files. Maybe with some reworking the implementation could be delivered in pre-built libraries, multiplied up by the various operating system, platform, version permutations – but wouldn't this make it even harder for compilers to build applications based on Spirit?

Of course, open source means more than just access to source: but it's still notable that this style of C++ favours open source distribution[3].

And Finally

I'm still not sure if it would have been better to write a code generator, to generate our codec from the section formats.

Maybe I'll try using Spirit and C++ to generate a C codec.

Thomas Guest
`thomas.guest@ntlworld.com`

References

[1] I borrowed this section header from Andrei Alexandrescu's homepage. `http://www.moderncppdesign.com/main.html`. It's funny because it's true.
[2] Frank Antonsen, "Stream-Based Parsing in C++", *Overload 56*, August 2003
[3] Boost: `http://www.boost.org`
[4] Eric Raymond, `http://www.catb.org/~esr/writings/cathedral-bazaar/`
[5] Greg Colvin, *In the Spirit of C*, `http://www.artima.com/cppsource/spiritofc.html`
[6] *Digital Video Broadcasting (DVB); Specification for Service Information (SI) in DVB systems*
[7] S. Heinzmann, "The Tale of a Struggling Template Programmer", *Overload 61*, June 2004
[8] Homepage: `http://homepage.ntlworld.com/thomas.guest`
[9] INFORMATION TECHNOLOGY - GENERIC CODING OF MOVING PICTURES AND ASSOCIATED AUDIO: SYSTEMS Recommendation H.222.0 ISO/IEC 13818-1
[10] PLY: `http://systems.cs.uchicago.edu/ply/`
[11] The canonical form of Postel's prescription, according to the Jargon File (`http://www.catb.org/~esr/jargon/`) is: "Be liberal in what you accept, and conservative in what you send."
[12] Python: `http://www.python.org`
[13] Eric S. Raymond, *The Art of UNIX Programming*, Addison-Wesley 0-13-142901-9
[14] The Spirit Applications Repository is available at `http://spirit.sourceforge.net`
[15] Thomas Guest, "A Mini-project to Decode a Mini-language – Part One", *Overload 63*, October 2004
[16] Spirit: `http://www.boost.org/libs/spirit/index.html`
[17] Spirit Style Guide: `http://www.boost.org/libs/spirit/doc/style_guide.html`

Credits

This article and the accompanying source code was developed using the GNU emacs integrated development environment (GNU emacs, GNU make, GCC, find, grep, etags, PCL-CVS etc), JASSPA Microemacs (for its superb text mode and binary editor), all running on the – sorry Eric, thanks Cygwin – Microsoft Windows operating system.

2 The "Techniques" section of the Spirit documentation [16] describes an extraordinary method for obtaining an object's type: "... Try to compile. Then, the compiler will generate an obnoxious error message ... THERE YOU GO! You got its type! I just copy and paste the correct type."
Elsewhere, the Spirit documentation mentions Dave Abrahams' proposal to reuse the `auto` keyword for type deduced variables.
See also Colvin [5] for a radical take on this issue.

3 The case for access to source code is even stronger for the libraries built using popular scripting languages such as Python and Perl.