

overload 86

AUGUST 2008 £3

Performitis

Therapeutic patterns and treatment programmes for sick software

Globals, Singletons and Parameters

We deal with issues arising from the PfA pattern

Triangle of Constraints

Seven important lessons for the management of software projects

OVERLOAD 86**August 2008**

ISSN 1354-3172

Editor

Ric Parkin
overload@accu.org

Advisors

Phil Bass
phil@stoneymanor.demon.co.uk

Richard Blundell
richard.blundell@gmail.com

Simon Farnsworth
simon@farnz.co.uk

Alistair McDonald
alistair@inrevo.com

Roger Orr
rogero@howzatt.demon.co.uk

Simon Sebright
simon.sebright@ubs.com

Paul Thomas
pthomas@spongelava.com

Anthony Williams
anthony.ajw@gmail.com

Advertising enquiries

ads@accu.org

Cover art and design

Pete Goodliffe
pete@cthree.org

Copy deadlines

All articles intended for publication in Overload 87 should be submitted to the editor by 1st September 2008 and for Overload 88 by 1st November 2008.

ACCU

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

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

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

4 DynamicAny, Part 1

Alex Fabijanic implements dynamic typing in C++.

10 Performitis, Part 2

Klaus Marquardt prescribes some treatments.

17 Globals, Singletons and Parameters

Bill Clare finds ways to parameterize code.

20 Exceptions Make For Elegant Code

Anthony Williams compares error handling techniques.

24 Divide and Conquer: Partition Trees and Their Uses

Stuart Golodetz introduces a powerful data structure.

29 On Management

Allan Kelly starts a new series looking at software management.

Copyrights and Trade Marks

Some articles and other contributions use terms that are either registered trade marks or claimed as such. The use of such terms is not intended to support nor disparage any trade mark claim. On request we will withdraw all references to a specific trade mark and its owner.

By default, the copyright of all material published by ACCU is the exclusive property of the author. By submitting material to ACCU for publication, an author is, by default, assumed to have granted ACCU the right to publish and republish that material in any medium as they see fit. An author of an article or column (not a letter or a review of software or a book) may explicitly offer single (first serial) publication rights and thereby retain all other rights.

Except for licences granted to 1) Corporate Members to copy solely for internal distribution 2) members to copy source code for use on their own computers, no material can be copied from Overload without written permission from the copyright holder.

It's good to talk...

Writing code is only part of developing software.

Consider a typical work day – what interactions do you have? Unless you only ever write software on your own for your own needs exclusively, at some point you will talk to someone. But what sort of interactions are they? In other words, who, what, how, and how often?

For example, here are some of my own recent experiences. First thing on Monday morning I chat with the person at the next desk about what we did over the weekend, which morphs into a discussion about a bug we were tackling at the end of the previous week. After a few code experiments, we update bugzilla with what we have learned. I use skype to check with the support team to find out how important a fix is to the customer, and then talk with the project manager about the approaches we could use and the risks involved. As a treat, I have a quick read of *accu-general* and the newsgroups, which gets an answer to a tricky template question. After a pub lunch with some new starters, I attend the daily stand-up team meeting to see what everyone is up to, and update the white board with the release schedule on it, after which I update the current branch diagram and publish to a remote monitor so that everyone can see it. While getting a coffee I bump into a firmware developer and check on a detail of a system we've just finished that had been queried. Then it's a scheduled code review, where I go through my comments with the recipient, then email the details to them. After checking my assigned bug list, I study some logs from a customer's machine and fix the bug, commenting in code why there had been a problem. I get a quick review of the change before checking it in and waiting for build server to mail everyone the build report. To round off the day there's a big company meeting involving Seattle ringing in and Palo Alto on videophone to round up what's happening in sales, marketing, engineering, finance and HR.

The range of interactions can be remarkably diverse.

The frequency can go from constantly during an intensive pair programming session, once a day at a team meeting, once a week at a department or company meeting, every few months when people visit from another office, to a one off client meeting.

The media range from a simple chat, a 'second pair of eyes' check of code, a formally arranged meeting, email, newsgroups, blogs, phone call, instant messaging, video conferencing, presentations, bug reports, debug logs, design documents, whiteboards, code comments, check in reports.

The type of person varies too: programmers, architects, project managers, product managers, customer support, customers, external contractors, sales, marketing. The immediate usefulness of the information they have will vary



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

too, so the initial level of detail will often mirror the likelihood of it being of interest, with the option of going into further detail if needed.

And there's a time lag aspect too – a real time conversation is immediate; email has a noticeable roundtrip delay; a comment in code is often communicating with a very important person – myself in six months time when I've forgotten the details; and a debug log is a result of the development team sending 'help' messages in a bottle to their future selves trying to diagnose problems.

There's a lot of talk going on.

I think this is for several reasons: almost always writing software is doing something new (at least to you!), which requires generating ideas, followed by implementing, checking, and backtracking as you try them out; you have to find out what to do in the first place, and checking you're still doing the right thing as you learn; and letting people know how things are going so that problems can be spotted and plans adjusted in a timely manner.

Thus developing software is not so much a technical problem as a communication one. To be successful, all of it has to work well – you can have the best development team in the world, deliver above spec, early, and under budget, and yet it never gets used because despite it doing what was asked for, that turned out to be not useful to the customer in practice.

I'll go further – the best teams are the ones who can move information such that the right level of detail gets to the right people at the right time. This is a tough challenge: collecting the information, knowing who needs it, in what form, and correctly prioritised. One of the real difficulties here is that the answers to these questions will be different for the sender and the recipient. Think of it as an economic problem – how to organise this information-processing system so that the maximum value is extracted.

But things aren't hopeless – we've been solving these sorts of organisation problems since humans got together into groups, so we must have learnt something along the way. Two key ideas are already implicit in my commentary above – specialisation, and teams.

A good all-rounder is very useful, and essential for a successful small company, but if you gather a group of people all of whom are experts in their own area, they can be much more productive than the same number of all-rounders. The downside is that a group of people now have to cooperate to achieve a goal, and that requires coordination of aims and delivery, and that requires communication between the members. But as a group grows, you end up with a combinatorial problem – if everyone talks to everyone else, the number of possible communication channels

explodes, so that you end up all your time talking to people and not actually doing anything.

Some solutions are fairly natural: Organise people so they talk to the people they need to easily, and it's harder to talk to the others. The inputs are from looking at the value generated from all possible interactions, and finding which are most beneficial. This can usually be done quite efficiently on an ad hoc basis, but a formal analysis could be an interesting exercise. (I'm reminded of the book *Notes On the Synthesis Of Form* [Alexander] which gives the mathematics of finding tractible design sub-problems given a network of dependencies. This looks analogous to finding the communication sub-networks of most value.)

For example, I can think of two very different ways that can work: organise similar roles into teams then have representatives do the inter-team communication; or do a vertical slice, where a team has representatives from each role and they work as an almost self-contained unit. Which is more appropriate depends on your situation, and can even be mixed.

Great, we've worked out our team structure, where shall we put them? Think carefully, and compare your answer to your current structure and locations.

I think a good rule of thumb is 'physical proximity should reflect desired communication intensity'. In English: be near the people you need to talk to most. This is similar to Conway's Law [Conway], in that the resulting structure will reflect the relative communication channels, and that is dominated by physical location.

As Churchill remarked, 'First we shape our buildings and afterwards our buildings shape us' [Churchill43]. This was in the context of rebuilding the Houses of Parliament, which had been damaged in a bombing raid. He understood that the government/opposition design of the chamber and its inability to fit everyone in at once actively shaped the political process, and should be retained.

New technologies warp this a little: from the telegraph to the phone to an instant message chat, we can interact remotely, but it is a different sort of communication to a face to face conversation. An example: in two minutes talking to someone with a customer support issue who'd popped in to talk to someone else, we cleared up more than the previous hour of back and forth email.

Ah, a serendipitous ad hoc meeting! A strikingly effective communication channel for those rare but valuable times you need to talk outside of your normal day-to-day interactions. But these 'chance' meetings need some way of being allowed to happen. A classic solution is a kitchen area with (a good) coffee machine, water coolers, comfy seats, and whiteboards. Steve Jobs apparently understood this, and put the Pixar bathrooms in what

initially seemed an awkward location in a central atrium so everyone could bump into each other [Bird]. Stuart Brand also discusses building and office design to make these things happen, and how bad buildings prevent them [Brand94].

All these dynamics need to be understood in a successful organisation and acted upon - the organisation should reflect the desired communication of information value, and physical locations chosen to encourage it. And it should be reviewed and adjusted as the world changes around you - nothing stays the same.

Communication changes

And there are indeed changes happening in how we communicate with you. Up till now CVu and Overload have been produced every two months, and sent to you in one mailing. But from now on they will alternate and you will be sent one magazine each month - the next Overload will be delivered in October as normal, but will be followed by CVu in November, then Overload in December etc. While seemingly a small change, this will mean you'll get a magazine twice as often, but without the information overload [sic] of two magazines to read all at once.



References

- [Alexander] Alexander, Christopher (1974) *Notes on the Synthesis of Form*, Harvard University Press: USA (ISBN 0-674-62751-2).
- [Bird] <http://gigaom.com/2008/04/17/pixars-brad-bird-on-fostering-innovation/>
Lesson Six: Steve Jobs Says 'Interaction = Innovation'
- [Brand94] Brand, Stewart (1995) *How Buildings Learn: What Happens After They're Built*, Penguin Books (ISBN 978-0140139969).
- [Churchill43] House of Commons (meeting in the House of Lords), 28 October 1943, http://www.winstonchurchill.org/i4a/pages/index.cfm?pageid=388#Shape_our_Buildings
- [Conway]: 'Any piece of software reflects the organizational structure that produced it.'
It was actually an observation on how committees work: <http://www.melconway.com/research/committees.html>
And there's some empirical evidence for it: <http://www.hbs.edu/research/pdf/08-039.pdf>

DynamicAny, Part I

Alex Fabijanic presents a class hierarchy providing dynamic typing in standard C++.

God created the integers, all the rest is the work of man.
Leopold Kronecker

Dynamic and static typing are competing forces acting upon the programming languages domain. The strong typing system, such as the one in C++ can be a bulletproof vest or a straitjacket, depending on the context. While C++ strong static typing is well-justified and useful, sometimes it is convenient or even necessary to circumvent it. Over time, various ways around it have been devised, on both high and low ends of the abstraction spectrum¹. Additionally, standard C++ offers dynamic and static polymorphism.

Dynamic languages have no notion of variable type. Values have type, while variables are type-agnostic. Hence, the type of a variable can change during its lifetime, depending on the value assigned to it. Clearly, in addition to static type-safety loss, there is also a performance penalty associated with this convenience. There are, however, scenarios where a relaxed, dynamic type system is a desirable feature, even in a statically typed language like C++. An example that comes in mind first is a retrieval of structured data from an external source. Typically, the data will arrive in a variety of types. In a statically typed world, this implies the requirement of knowing the exact data types at compilation time. Additionally, every time the data types or layout changes, the code must change as well. A way around this obstacle is through dynamic typing support.

This article describes the approach taken by the C++ Portable Components [POCOa] framework ('POCO' in further text) to implement safe and efficient dynamic typing capabilities within the confines of standard C++.

Any

Boost Libraries [Boost] contain multiple classes meant to alleviate the pains associated with static typing. Our focus here is on `boost::any` and a solution building on its design. Through clever construction and type erasure, `boost::any` is capable of storing any type. Both built-in and user-defined types are supported. A code example of `boost::any` usage is shown in Listing 1.

However, `boost::any` is implemented in the type-safe spirit of C++. Run-time efficiency and strong typing are the constraints behind its design. Although it provides a mechanism for dynamic type discovery (Listing 2.), `boost::any` does not itself get involved into such activity, nor is it willing to cooperate in implicit conversions between values of different types. Moreover, an attempt to extract a type other than the one held, results in either an exception being thrown or a null pointer returned (in the manner of standard C++ `dynamic_cast`).

A `boost::any` object is really convenient when one wants to pass around a variable of arbitrary type without having to worry about what type it actually is. The most common use is storing diverse types in an STL

```
std::list<boost::any> a1;
int i = 0;
std::string s = "1";

a1.push_back(i);
a1.push_back(s);
```

Listing 1

```
bool isInt(const boost::any& a) {
    return a.type() == typeid(int);
}
```

Listing 2

container. However, the true nature of `boost::any` is static and, at the actual value extraction place, it is necessary to know precisely what type is held inside the `any`. While very 'soft' on the assignment side, on the extraction side `boost::any` is even more rigid than built-in C++ types – it is only possible to cast it back to its original type. The `boost::any` class has been ported to POCO (with some additions²), where it is known as `Poco::Any`. The design of this class has served as a foundation for development of its dynamic cousin, `Poco::DynamicAny`, which is the main theme of this article.

DynamicAny

As mentioned above, `Poco::Any` is a handy class for storing variety of types behind a common interface offering strongly typed cast mechanism and support for querying the held data type. `Poco::DynamicAny` extends `Any` functionality by providing full-blown runtime dynamic typing functionality within an ANSI/ISO C++ compliant framework. `DynamicAny` builds on the heritage of `Any` by adding the following features:

- runtime checked value conversion and retrieval
- non-initialized state support
- implicit conversion to target type when possible and safe
- seamless cooperation with POD types
- seamless cooperation with `std::string`
- `std::map` wrapper (a.k.a. `DynamicStruct`)
- `std::vector` specialization (array-like semantics support)
- date/time specializations
- binary large object specialization
- basic JSON support.

1 unions, void pointers, Microsoft COM Variant, `boost::variant`, `boost::any`, `boost::lexical_cast`
2 Added `RefAnyCast` operators returning reference and `const` reference to stored value.

Aleksandar Fabijanic Alex is a C++ and POCO enthusiast. He is using POCO at work for industrial automation and process control software development. Alex spends a lot of his free time contributing, supporting and managing the project. Contact him at alex@appinf.us

deciding what is true or false seems like an easy task until it is actually attempted

The class goes a long way to provide intuitive and reasonable conversion semantics and prevent unexpected data loss, particularly when performing narrowing or signedness conversions of numeric data types. One of the challenges during the design process was to come up with a set of intuitive conversion and behaviour rules. Of course, many conversions attempts will throw an exception because they make no sense (e.g. converting "abc" to a numeric type). Additionally, deciding what is true or false seems like an easy task until it is actually attempted. The final verdict was that anything resembling either explicit falsehood (string "false", bool false) or 'nothingness' (empty string, integer zero, min. float value ...) shall be **false**, everything else is **true**. This decision is compatible with C and C++, where zero integer is **false** and everything else is **true**. Also, "false" and "true" strings behave as expected in a case-insensitive manner.

The rules governing the behavior of **DynamicAny** are³:

- An attempt to convert or extract from a non-initialized ('empty') **DynamicAny** variable shall result in an exception being thrown
- Loss of signedness is not permitted for numeric values. An attempt to convert a negative signed integer value to an unsigned integer type storage results in an exception being thrown.
- Overflow is prohibited; attempt to assign a value larger than the target numeric type size can accommodate results in an exception being thrown.
- Precision loss, such as in conversion from floating-point types to integers or from double to float on platforms where they differ in size (provided double value fits in float min/max range), is permitted.
- String truncation is allowed – it is possible to convert between string and character when string length is greater than 1. An empty string gets converted to the char '\0', a non-empty string is truncated to the first character.

Boolean conversions are performed as follows:

- A string value "false" (not case sensitive), "0" or "" (empty string) evaluates to **false**; any string not evaluating to **false** evaluates to **true** (e.g. "hi" → **true**).
- All integer zero values are **false**, everything else is **true**.
- Floating point values equal to the minimal floating point representation on a given platform are **false**, everything else is **true**.

3 Some of the features are scheduled for the next release and are currently available from the development code repository [POCOb]

4 For conversion from type T1 to type T2 to be possible, a **DynamicAnyHolderImpl<T1>::convert(T2&)** must be defined.

5 The commented line does not compile with g++ (MSVC++ and Sun Studio compile it successfully).

```
// Values are interchangeable between
// different types in a safe way
DynamicAny any("42");
int i = any; // i == 42
any = 65536;
std::string s = any; // s == "65536"
char c = any; // too big, throws RangeException
```

Listing 3

```
// Conversion operators for
// basic types are available
DynamicAny any = 10;
int i = any - 5; // i == 5
i += any; // i == 15
i = 30 / any; // i == 3
bool b = 10 == any; // b == true
```

Listing 4

```
// DynamicAny can be incremented or
// decremented when holding integral value
DynamicAny any = 10;
any++; // any == 11
--any; // any == 10
any = 1.2f; // make it float
++any; // throws InvalidArgumentException
```

Listing 5

```
// Workaround for std::string
DynamicAny any("42");
std::string s1 = any; //OK
// std::string s2(any); //g++ compile error
std::string s3(any.convert<std::string>()); //OK
```

Listing 6

The added value and benefit of **DynamicAny** is in relieving the programmer from type-related worries for all the fundamental C++ types and some POCO framework objects. **DynamicAny** allows storage of different data types and transparent conversion between them in the fashion of dynamic languages.⁴

Some **DynamicAny** usage examples are shown in listings 3–6.

There are some conversions that require 'workarounds' with some compilers as illustrated in the code snippet in Listing 6⁵.

storage and extraction of an arbitrary user-defined type are supported out-of-the-box

DynamicAny implementation

In the manner of `boost::any`, storage and extraction of an arbitrary user-defined type are supported out-of-the-box. In addition to that, `DynamicAny`'s conversions are fully extensible. In order to provide the support for conversion to other types, the `DynamicAnyHolder<Type>` must be specialized for the `Type` with appropriate `convert()` function overloads being defined.

The structure outline of the `DynamicAny` and `DynamicAnyHolder` class hierarchy is shown in Listing 7. `DynamicAny` owns a pointer to

```
class DynamicAny
{
public:
    DynamicAny();
    // Creates an empty DynamicAny.

    template <typename T> DynamicAny(const T &val):
        _pHolder(new DynamicAnyHolderImpl<T>(val))
    // Creates the DynamicAny from the given value.
    { }

    // ...

    DynamicAnyHolder* _pHolder;
};

class DynamicAnyHolder
{
public:
    virtual ~DynamicAnyHolder();
    // ...
    virtual void convert(Int8& val) const
    { throw BadCastException(
        "Can not convert to Int8"); }

    virtual void convert(Int16& val) const
    { throw BadCastException(
        "Can not convert to Int16"); }

    // ...

    virtual void convert(std::string& val) const
    { throw BadCastException(
        "Can not convert to string"); }

    // ...
protected:
    DynamicAnyHolder();
    // ...
};
```

Listing 7

```
template <typename T>
class DynamicAnyHolderImpl: public
    DynamicAnyHolder
// template for arbitrary user-defined types
{
public:
    DynamicAnyHolderImpl(const T& val): _val(val) {
    } ~DynamicAnyHolderImpl() { }

    const std::type_info& type() const
    { return typeid(T); }

    DynamicAnyHolder* clone() const
    { return new DynamicAnyHolderImpl(_val); }

    const T& value() const
    { return _val; }

private:
    DynamicAnyHolderImpl();
    // ...
    T _val;
}

template <>
class DynamicAnyHolderImpl<Int8>:
    public DynamicAnyHolder
// Int8 specialization
{
public:
    DynamicAnyHolderImpl(Int8 val): _val(val) { }

    // ...

    void convert(Int8& val) const
    { val = _val; }

    void convert(Int16& val) const
    { val = _val; }

    // ...

    void convert(std::string& val) const
    { val = NumberFormatter::format(_val); }

    // ...

private:
    DynamicAnyHolderImpl();
    Int8 _val;
};
```

Listing 7 (cont'd)

it took several iterations of safe conversion check versions to reconcile with all supported compilers and platforms

DynamicAnyHolder. The default zero pointer indicates that variable has not been initialized yet. In the non-initialized state, attempt for extraction or conversion triggers an exception. At assignment time, the **DynamicAnyHolder** storage is allocated on the heap and the address stored in the pointer. The storage is automatically released at destruction time by virtue of the C++ RAII mechanism.

Support for various data types is achieved through polymorphism – **DynamicAnyHolderImpl** is a template class inheriting from **DynamicAnyHolder** and only specializations of this class do the conversion work. The direct extraction of the original data type depends on the template and specializations having **value()** member function returning the held value. Although it may be viewed as a questionable design decision, for efficiency sake **value()** has intentionally not been made virtual. This decision has provided the value extraction performance comparable to that of **boost::any**.

The most commonly used data types (all fundamental data types, **std::string**, **std::vector<DynamicAny>**, **DateTime**, **Timestamp**, **BLOB**) are specialized within the POCO framework and ready for immediate use. The mentioned set of data types covers the majority of cases where automatic conversion is frequently needed. All other data types are covered by the generic **DynamicAnyHolderImpl<T>** template and, like **boost::any**, allow only extraction of the held type, while an attempt to convert the value results in exception. When needed, a specialization for user-defined types is possible. A definition of sample UDT (a social security number formatter), with specialization and usage example code is shown in Listing 8.

As seen in the example, **DynamicAny** readily holds **SSN** and smoothly converts it to supported values. Assignment from **DynamicAny** to **SSN** is also possible. **DynamicAnyHolder** provides the dynamic behaviour through polymorphism by virtue of its descendant specializations – the actual value resides in **DynamicAnyHolderImpl** specialization. This value is converted through the overloaded **convert()** virtual function call for the appropriate data type. Were the specialization not present, only extraction of the original type (in the fashion of **boost::any**'s **any_cast** functionality) would have been possible.

The main challenges encountered during the design were making **DynamicAny** coexist in harmony with built-in types and compilers as well as implementing specializations and safe conversions for most commonly used types. The first attempt for operator overloading was template-based, but that has proved to be painting with too broad a brush, resulting in obscure compile errors on some platforms. To fix the problem, operators on both sides (member and non-member ones) have been re-implemented as overloaded functions. Also, it took several iterations of safe conversion check versions to reconcile with all supported compilers and platforms. The POCO community contribution in the process was instrumental.

DynamicAny in real world

Surely, all this is not without meaning [Melville51]. The code sample shown may be a clever data formatter, but was it worth going through such

```
class SSN
    /// a user-defined type
{
public:
    SSN(const SSN& ssn): _ssn(ssn._ssn) { }

    SSN(const DynamicAny& da): _ssn(da) { }

    operator SSN ()
    { return *this; }

    std::string sSSN() const
    { return format(); }

    int nSSN() const
    { return _ssn; }

private:
    std::string format() const
        /// format integer as SSN
    {
        std::string tmp;
        std::string str =
            NumberFormatter::format(_ssn);
        tmp.clear();
        tmp.append(str, 0, 3);
        tmp += '-';
        tmp.append(str, 3, 2);
        tmp += '-';
        tmp.append(str, 5, 4);
        return tmp;
    }

    int _ssn;
};

// Sample usage:

SSN udt1(123456789);
DynamicAny da = udt1;
std::string ssn = da;
std::cout << ssn << std::endl;
SSN udt2 = da;
std::cout << udt2.nSSN() << std::endl;

// Output:

123-45-6789
123456789
```

Listing 8

Static and dynamic data typing are contrasting solutions, each with its own advantages and drawbacks

effort only to provide conversion and formatting between numbers and strings? A code snippet using `DynamicAny` in a realistic scenario is shown in Listing 9. The added value that `DynamicAny` brings in this case is:

- shield against the compile-time data type and layout knowledge requirement
- shield against conversion data loss

To achieve the desired `RecordSet` capabilities, class `Row` was introduced. By utilizing `DynamicAny`'s dynamic typing facilities, `Row` conveniently wraps a row of data and, through `RowIterator`, works seamlessly in conjunction with STL algorithms to provide functionality for a flexible `RecordSet` class, as shown in Listing 10. The details are outside of the scope of this article, but suffice it to say that the code shown works for any given SQL statement (i.e. any given column count/datatype combination) thanks to dynamic typing provided by `DynamicAny`.

As demonstrated in Listings 9 and 10, `DynamicAny` comes handy as a 'mediator' between type aware data storage (e.g. database) and a type

relaxed representation (e.g. web page). Displaying data from database is easy by simply converting all the values to string and embedding them into HTML, for example. However, the data coming back from the web page shall all be strings. In a scenario proposed by a POCO contributor, `DynamicAny` had been extended having two callback functions being called before and after a value assignment or change. The callbacks allow changes in the value to be instantly reflected in a XML structure and then transformed to any representation on demand. String value coming from a response XML could be put in a `DynamicAny` then bound to database query without explicit type conversion. Full details are beyond the scope of this article, but the basic outline is laid out in Listing 11. Currently, this is not a part of mainstream code base and a discussion is going on about whether and how to integrate this functionality into the framework.

Conclusion

Static and dynamic data typing are contrasting solutions, each with its own advantages and drawbacks. While dynamic typing affects runtime performance, in certain scenarios (e.g. fetching data from a remote database) the performance hit is dwarfed by the time spent on other operations.

As illustrated in the examples, `DynamicAny` is useful whenever performance requirements are loose and/or data types involved are unknown at compile time. However, as will be shown in part II of the article, performance concern was not a design afterthought. `DynamicAny` class is part of C++ Portable Components framework Foundation library with extensive use in the Data library. Additionally, some experimenting is underway with `DynamicAny` used as a 'bridge' between C++ and scripting languages [POCOc].

In the next installment of this article, more details about internal implementation of `DynamicAny` will be given, as well as some comparison tests between different C++ data type conversion mechanisms and classes. ■

```
using namespace Poco::Data::Keywords;
using Poco::Data::Session;
using Poco::Data::Statement;
using Poco::Data::RecordSet;

// create a session
Session session("SQLite", "sample.db");

// a simple query
Statement stmt(session);
stmt << "INSERT INTO Person VALUES ('Bart', 12)",
      now;

// create a RecordSet
RecordSet rs(session, "SELECT Name,
                    Age FROM Person");

int i = rs[1]; // OK
std::string s = rs[1]; // OK, too
i = rs[0]; // throws, can't convert 'Bart' to int
```

Listing 9

```
Session session("SQLite", "sample.db");
std::cout << RecordSet(session, "SELECT * FROM
Person");

// This is how streaming is achieved under
// the hood:
// copy(begin(), end(),
//       ostream_iterator<Row>(cout));
```

Listing 10

```
// retrieve from a database
RecordSet rs(session, "SELECT Name,
                    Age FROM Simpsons");
// type is known here
DynamicAny age = rs[1];
// output as string (eg. in some html)
// ...
// assign from a string (e.g. from some html form)
age = "14";
// bind, implicitly casting age to int
session <<
    "INSERT INTO Simpsons VALUES('Bart', ?)",
    use(age), now;
```

Listing 11

One of the challenges during the design process was to come up with a set of intuitive conversion and behaviour rules

Acknowledgements

Kevlin Henney is the originator of the idea and author of the `boost::any` class. Kevlin has provided valuable comments on the article.

Peter Schojer has ported `boost::any` to POCO, implemented major portions of `DynamicAny` and provided valuable comments on the article.

Günter Obiltschnig has written majority of the POCO framework and provided valuable comments on the article.

Laszlo Keresztfalvi has provided valuable development and testing feedback, sample usage code as well as valuable comments on the article.

References

- [Boost] Boost `any` library: <http://www.boost.org/doc/html/any.html>
- [Henney00] Henney, Kevlin (2000) 'Valued Conversions', *C++ Report*, July–August 2000.
- [Melville51] Melville, Herman (1851) *Moby Dick*, Harper & Brothers Publishers
- [POCOa] C++ Portable Components: <http://poco.sourceforge.net>
- [POCOb] C++ Portable Components development repository: <http://poco.svn.sourceforge.net/viewvc/poco/>
- [POCOc] `Poco::Script`: <http://poco.svn.sourceforge.net/viewvc/poco/sandbox/Script/>
- [Stroustrup97] Stroustrup, Bjarne (1997) *The C++ Programming Language*, Addison-Wesley.
- [Sutter07] Sutter, Herb (2007) 'Modern C++ Libraries', *Proceedings, SD West*.

IMPORTANT NOTICE!

ACCU's journal publication dates are changing...

The next issue of *CUu* will be out in November, after which it will be published on a bi-monthly schedule as before, but on alternate months to *Overload*. This means that if you subscribe to both journals, you'll receive an ACCU journal every month!

accu... it's a code thing

Performitis – Part 2

Software problems have much in common with diseases. Klaus Marquardt has a diagnosis and offers some treatments.

To recap, what is PERFORMITIS, or performance bloat?

Every part of the system is directly influenced by local performance tuning measures. There is either no global performance strategy, or it ignores other qualities of the system such as testability and maintainability

In the first part of this article we learned about a project one would never like to be associated with. Certainly it is more fun to read about a doomed project than to live within one. However, PERFORMITIS is not doom, it is a disease that can be cured.

The worst thing about PERFORMITIS is that many colleagues probably consider it a solution rather than a problem. Performance is a key issue of the system, and it is being cared for. Thus, when the evaluation of the symptoms indicate that the project suffers from PERFORMITIS, it may be wise not to spread the news bluntly. Even without naming it or any of its non-technical implications, there are many things to do that resonate with sound engineering practices. PERFORMITIS infected projects normally do not follow these – but the developers should be reasonably familiar with them, or able to understand them, so that the suffering is limited.

Does that help? Well, yes. It reduces the immediate risk: not being able to ship at all. However, the next project likely faces similar problems. In terms of the medical metaphor, we have then soothed the pain and treated some symptoms, but we have not cured the disease.

While PERFORMITIS appears to be a technical diagnosis at first, its real causes are with people and their socialization. The technical symptoms can be attacked by some therapeutic measures or suppressed by extensive processes. These require continued effort and can at best maintain a state of remission. Curative therapies need to address the pathogen.

The first set of therapeutic measures addresses the technical symptoms:

- MEASUREMENT-BASED TUNING can become a relief from thinking too much about tuning up-front and in the least relevant places. Plus, it goes together nicely with tuning attempts in later stages of development.
- Selecting the most efficient places for performance tuning is the topic of ARCHITECTURE TUNING.
- Separating the PERFORMANCE-CRITICAL COMPONENTS is the basic architectural technique to avoid performance bloat – spreading mediocre performance optimization techniques all over the code.

PERFORMITIS has root causes in the team culture and value system. Looking at the techniques to apply, PERFORMITIS-infected teams may not be willing to tackle them without intense discussion. These therapies help to establish a broader view on technical measures:

- The effort expended on performance can be limited and controlled by explicitly assigning room for performance in the development process. VISIBLE QUALITIES allows making a case for performance, but opens room for other qualities as well.
- In teams and organizations that do not define or assign an architect's role, DEDICATED ARCHITECT is a prerequisite

Both therapies change the way the team cooperates and communicates. Discussions about roles and responsibilities, and finally the way to ensure performance and other intrinsic qualities will arise. Speaking in the tongue of the medical metaphor, these changes cause a fever. Like an infection in a living organism, new ideas introduced become attacked. Hot discussions cause friction, like a fever – and finally result in a learning process that helps to improve the balance between different system qualities.

The feverish therapies can and need to be combined with any other therapy of choice. Apart from the stated overdose effects, no combination of the suggested therapies can be harmful to the project. They have mutually increasing effect. However, too many therapeutic changes at the same time might break morale and the team structure. It is worth taking the time to introduce one after another, and anticipate which one brings the most effect in the current situation.

Therapy overview

Table 1, overleaf, gives an overview on the applicability of the individual therapies, and how they can best be combined. Some medical terms are used:

- preventive – keep a disease from happening;
- palliative – reducing the violence of a disease, soothing the symptoms so that the quality of life is maximized;
- remission – relief from suffering, while the disease is still present;
- curative – healing from a disease.

Before treating a system with some proposed therapy, due diligence includes learning about its mechanism, and its overdose and side effects. The education of doctors is taken seriously.

Klaus Marquardt is a technical manager and system architect with Dräger Medical in Lübeck, Germany. His experience includes life support systems and large international projects. Klaus is particularly interested in the relations between technology, organization, people and process. He has contributed sessions to many conferences including OOP, JAOO, ACCU, SPA and OOPSLA. Klaus can be contacted at pattern@kmarquardt.de

Caveat

There are rare systems where the absolute dominance of performance is not pathological but a conscious and justifiable decision. Hold on a minute – in all likelihood this is not your situation. To find out, make your priorities of different qualities explicit as in the VISIBLE QUALITIES therapy. Even if you are there, these technical therapies offer some suggestions for improvement: PERFORMANCE-CRITICAL COMPONENTS, ARCHITECTURE TUNING and MEASUREMENT-BASED TUNING.

Whenever you think there is a problem, turn your assumption into knowledge

	Applicability	Effect	Related therapies
MEASUREMENT-BASED TUNING	Any time during the project.	Palliative. Preventive when applied early.	Works best with a DEDICATED ARCHITECT.
ARCHITECTURE TUNING	Any time during the project.	Remission possible.	Most successful with PERFORMANCE-CRITICAL COMPONENTS already in place.
PERFORMANCE-CRITICAL COMPONENTS	Early in the project.	Preventive; remission possible.	Works best with a DEDICATED ARCHITECT.
VISIBLE QUALITIES	Preferably early in the project.	Preventive; remission possible.	Works best with a DEDICATED ARCHITECT.
DEDICATED ARCHITECT	Preferably early in the project.	Remission possible.	Boosts the application of other therapies.

Table 1

Pattern form and approach

Both diagnoses and therapies follow their own forms, including sections that contribute to the medical metaphor.

With each diagnosis, symptoms and examination are discussed and concluded by a checklist. A description of possible pathogens and the etiology closes the diagnosis.

Each diagnosis comes with a brief explanation of applicable therapies. This includes possible therapy combinations and treatment schemes that combine several therapies. These are suggested starting points for a successful treatment of the actual situation.

Therapies are measures, processes or other medications applicable to one or several diagnoses. Their description includes problem, forces, solution, implementation hints and an example or project report. Their initial context is kept rather broad since most can be applied for different diagnoses.

In addition to the common pattern elements, therapeutic measures contain additional, optional sections of pharmaceutical information. These are introduced by symbols:

-  the mechanisms of a therapy and how it works;
-  the involved roles and related costs;
-  counter indications, side and overdose effects;
-  cross effects when combined with other therapies.

MEASUREMENT-BASED TUNING

Applies to projects in domains that require a high system responsiveness, especially when the team is only vaguely familiar with the domain and its specific requirements.

Every developer knows that tuning the system for performance is necessary. The key decisions are when to take measures, and which tuning measures to initiate.

- Measures taken early are typically most effective, ... but measures taken on assumptions instead of proper knowledge are often inefficient and compromise other system qualities.
- Being afraid is always bad advice, ... but being aware of possible problems is wise.

Therefore, measure where the actual performance bottlenecks are, and start tuning measures there. Do not take preventive measures against assumed performance problems. Spend your performance tuning effort where you know it is most effective.

Whenever you think there is a problem, turn your assumption into knowledge. Critical architectural issues can be clarified by spike solution projects [Beck99] or prototypes [Cockburn98] with the sole purpose of identifying the actual performance issues. These spikes are most useful when you have established load profile scenarios or performance budgets.

Where you cannot gain knowledge for some reason, follow sound practices and 'proactively wait'¹ [Marquardt99]. Resist the temptation to begin with micro tuning. Instead, focus on other qualities, especially on testability and maintainability. Most performance tuning measures on architecture or

1 To apply the right amount of waiting is an important virtue of a medical doctor. While the symptoms are not severe and the patient does not suffer, a lot of diseases are left to mere observation until a significant change occurs.

make sure that the architecture itself helps the system responsiveness

design level require a clear distribution of responsibilities anyway, and you can spend the structure clean-up effort now, when it hurts least.



MEASUREMENT-BASED TUNING is not directly effective in a curative way, but frees attention and effort to be put on relevant topics of the project.



All development team members and technical management needs to be involved with MEASUREMENT-BASED TUNING. Interestingly, the costs of MEASUREMENT-BASED TUNING are often negative. It prevents effort being put into misled measures, and enables you to proceed faster during initial development as well as during performance tuning phases. The costs spent on convincing other stakeholders of the validity of this approach, and of constant observation are typically low.



There are no counter indications to MEASUREMENT-BASED TUNING. The side effects are desired: the team does not inefficiently start tuning, and potentially pays more attention to other qualities. Measures are taken in an informed manner. Overdose effects would be to ignore the obvious common sense in your technical domain, or to wait too long before you take corrective action.



MEASUREMENT-BASED TUNING has the highest chances to succeed if the architecture has prepared for separation of concerns. One of the key practices to prepare for late tuning measures is to separate PERFORMANCE-CRITICAL COMPONENTS.

To 'proactively wait' is an important, but difficult virtue. Key to this technique is not to miss the time when decisive action is necessary. This requires self-consciousness and constant observation. At some time the performance problems become noticeable, and then it is necessary to dig deeper. The threshold of when to take action is typically subject to personal taste and working style and requires significant experience. However, discussing them openly with colleagues helps not to miss important indications, and the rarely absent lack of time prevents from being overly responsive.

When we first used an object oriented database, we put all data in it that needed to be shared between different clients. This design led to a highly consistent system. Unfortunately, it was also horribly slow. We lived with that fact for some time, hoping that increasing our knowledge about the OODBMS would provide us with counter measures. After a handful of iterations, the GUI team decided to stub the database and leave the process of continuous integration. This was a severe warning, and we immediately checked the database performance.

It turned out that the most expensive data the OODBMS was occupied with was transient data; it was distributed among different clients but did not require persistency. We had naively not separated these aspects, hoping the OODBMS would be sufficiently fast.

Two concrete actions were initiated. First, the distribution mechanism became separated from the database access. Second, for the sake of consistent class interfaces, the classes meant to become persistent were no longer derived from the OODBMS base class. Instead we provided a distinct persistence service that we passed the objects to, and maintained the database schema by generating the persistent classes from the application's class model.

ARCHITECTURE TUNING

Applies to projects that need performance tuning.

A development team experiences severe performance problems, and needs to decide how to tackle them.

- Local tuning efforts based on profiling help to improve the system's responsiveness,
 - ... but you'd need to have a lot of local improvements to push the overall performance by orders of magnitude.
- Performance can be attacked on every level of development,
 - ... but initiatives on architectural level likely have the most impact.

Therefore, tune the system at the architectural level. Before you initiate any other improvement efforts, make sure that the architecture itself helps the system responsiveness, and exhibits no obvious flaws or 'black holes' in which processing power vanishes.

Ignoring the architecture level might get you lost in an endless sequence of severe battles, each saving in the sub-percent range of CPU load or other system resources. Your actual goal is not only to win the performance war but to win your peace with performance.

Looking at the scope of the architecture, performance is mostly decreased due to one or more of the following mechanisms:

- Random and inefficient use of system resources or infrastructure.
- Inappropriate locking of shared resources, or inappropriate transaction granularity.
- Multiple executions of identical calls without functionality gain.
- Network or inter process communication in many small messages, and amongst many different components.
- Blocking operations in tasks supposed to be responsive.
- Inappropriate distribution of responsibilities among clients and servers, for example how query result sets are transported and kept.
- Inappropriate database schemas, or example inadequate data types, or too little or too much normalization.
- Extensive error checking, tracing and logging.
- Error handling strategies that pollute the standard flow of operation.
- Interfaces requiring multiple data copying or data format conversion.
- Doing everything doable as soon as possible, or as late as possible.

factor out the Performance-Critical Components, and limit preventive tuning measures to these

A few changes to the architecture or top-level design can increase the performance by orders of magnitude. Performance tuning typically follows a few fundamental principles [Marquardt02a].



ARCHITECTURE TUNING is a process to find curative technical solutions to technical symptoms. If the related diagnosis indicates that the pathogen is beyond the technical scope, it can lead to remission.



ARCHITECTURE TUNING involves the architect and every developer assisting in analysis and implementation of the performance tuning. Its cost are hardly predictable, they depend on the necessary refactoring effort and the actual implemented changes to the architecture. However, in large projects they are lower than the costs of numerous attempts for local optimizations, and it is more likely to be effective.



There are no counter indications to ARCHITECTURE TUNING – given that you do not consider abandoning tuning effort, or the project at all. The side effect is a well-structured system. No overdose effect is currently known.



ARCHITECTURE TUNING comes with the least cost when you separate the PERFORMANCE-CRITICAL COMPONENTS early in the project.

If your system does not allow you to implement the changes you have identified being necessary, you need to refactor it in advance. Typical refactorings for tuning the architecture are the introduction of shared technical components, separation of resource maintenance from resource usage, and separation of performance critical tasks into distinct components.

While the system is developed, it is good practice to make these late changes as convenient as possible. As PERFORMANCE-CRITICAL COMPONENTS explains, this is most efficiently done by maintaining a design with a clear separation of responsibilities, and a dependency structure with few (if any) compromises. Such a structure also helps during development with respect to testing and task assignment, and to maintenance in later project phases.

A contractor had managed to become the mind monopole at one of my customers. He motivated his queer data model with reasons such as ‘using DB/2, comparing integers shows higher performance than comparing strings’. The effect exists, but can be neglected compared to the costs of disk access or joins.

When the system went productive, it needed 1.7 seconds per transaction, which would have caused annual operation costs of several 100,000 €. Tuning measures saved around 10–20%, but to reach a performance comparable to similar systems a factor of 100 would have been necessary. I was asked to evaluate the architecture for optimisation possibilities, and found sufficient opportunities to save a factor of 1000 – starting from simple measures like skipping consistency checks of large data structures with every internal call (the much too large structures contained

several 100 sets of data), up to fundamental changes to the architecture.

PERFORMANCE-CRITICAL COMPONENTS

Applies to projects in domains that require high system responsiveness.

A development team that is aware of performance tuning needs to prepare for tuning measures before they are actually done.

- It is hard to foresee where changed will become necessary later in the project,
 - ...but preparation early in development saves restructuring effort and time later.
- Performance can be attacked on every level of development,
 - ...but initiatives at the architectural level likely have the most impact.
- Incremental development can proceed fastest when the user-visible functions are developed independently,
 - ...but shared libraries and common infrastructure can become more mature and efficient than dispersed items all over the system.

Therefore, factor out the performance critical components, and limit preventive tuning measures to these. Start by dividing them in a way that business logic, display, distribution and technical infrastructure are independent of each other and can be tuned individually. Ensure that all applications use the same infrastructure so that central tuning measures become possible.

In systems that have been found to waste performance, these areas were good candidates where a responsiveness gain by an order of magnitude was possible:

- Inefficient use of system resources or infrastructure.
- Multiple repetitions of identical calls without functionality gain.
- Badly designed or agreed interfaces requiring multiple data copying or data format conversion.
- Extensive tracing and logging; error handling strategies that pollute the standard flow of operation.

To avoid these pitfalls, your system needs to be prepared to change internal access strategies and rearrange call sequences. Make sure that you can start with ARCHITECTURE TUNING whenever you need to. Once you start tuning, refactor to the degree that the planned tuning related changes can be made easily [Fowl99]. To avoid large and late refactoring efforts, start with a piece-meal growth approach and a ‘clean desk’ policy that includes refactoring in the daily work.

Within each of the components above, again distinguish between published and internal functionality. Factor everything specific to your particular environment into distinct components, like services, calls to technical services, database queries, and handle acquisition. Make sure that the responsibilities among all components are clear and concise.

Separate the logical contents from the physical execution, with few explicit linkage points

Separate the logical contents from the physical execution, with few explicit linkage points.



PERFORMANCE-CRITICAL COMPONENTS is a preventive therapy that increases the system's adaptivity to further therapies such as ARCHITECTURE TUNING. It fosters many qualities, but does not directly affect performance in and by itself.



The entire development team and technical management need to be involved. PERFORMANCE-CRITICAL COMPONENTS increase the costs you would spend on architectural decomposition. Though the initial costs will pay off if you really run into performance problems, consider PERFORMANCE-CRITICAL COMPONENTS as a risk reduction strategy.



Do not apply PERFORMANCE-CRITICAL COMPONENTS when your system is very small, or applies standard technology only. Though you gain valuable experience, the costs hardly pay off in these cases. Another counter indication is a weak position of the architect in the project. Side effects include an improved logical structure of your system. Overdose effects would be DESIGN BY SPLINTER [Marquardt01] if you drive the separation to an unbalanced extreme, or a micro-architecture similar to micro-management violating your DEFINED NEGLIGENCE LEVEL. [Marquardt02b].



PERFORMANCE-CRITICAL COMPONENTS is the preventive strategy to later ARCHITECTURE TUNING. It can be one of your risk reduction measures applied with VISIBLE QUALITIES.

Start with a useful separation that is most likely to support your actual needs. Keep the performance critical components as small as possible by iteratively repeating the division.

From the experience the team has gathered in the domain, both application and technology domain, you already know which parts of the system are likely to become the bottlenecks. Run a retrospective to uncover which areas have been performance relevant in previous projects. You will experience a fair amount of support when you separate these from the remainder of the system, and apply EXPLICIT DEPENDENCY MANAGEMENT [Marquardt02b].

In a real time patient information system, all transient data was stored in shared memory. This was hidden from the clients to this data; some opaque access classes provided an interface independent of implementation issues. Tuning the performance of data access and locking granularity was located within the access layer classes.

VISIBLE QUALITIES

Applies to projects whose team focuses all work and thoughts on a few essential ideas, but ignores all other issues that might also be or become important to the project's success.

In a development team that focuses on a particular quality of the product, you need to address further important system qualities that are essential to adequately manage the system architecture.

- Neglecting internal qualities can cause a large system to break under its own weight,
 - ...but the value they add to the software is hidden and becomes visible only in the long term.
- Internal qualities can be crafted intentionally into the software,
 - ...but they are hardly visible from a bird's eye perspective.

Therefore, make your system's internal qualities visible. Similar to sound risk management practice, maintain a list of your top five qualities. Define measures to achieve them, and determine frequently to what extent you have reached your goal.

The key issue is to raise awareness for the existence of these qualities and their relative importance in the team and in management. Especially when the internal system qualities are unbalanced, ask the team for a list of possible qualities and discuss their value and advantages. The team should order them according to their priority. Do not mind if your favourites are not the topmost – you will go through the list every week or two and re-evaluate.

Do the same process with management, and make both lists visible. While it is often not possible to resolve any conflict and come to consensus, the fact that all qualities are there and considered important leads to awareness, a more careful balancing and to an architecture and design that addresses different qualities explicitly.

You need to maintain the lists, find criteria how to evaluate whether a specific quality has been achieved, and define appropriate actions [Weinberg92]. This could become a part of a periodically scheduled team meeting. Especially the evaluation criteria would be a tough job, as most qualities show only indirect effects. Try to define goals that appear reasonable to the project. If you or the team fails to define criteria, leave that quality at the end of the list for the time being.



VISIBLE QUALITIES is effective through creating attention and a positive attitude. The attention achieved by the top-five list causes second thoughts, awareness, and potentially actions, while the measurable achievement fosters a positive attitude that in itself already could improve the quality of work.



The work and initial costs are with the architect, but VISIBLE QUALITIES requires involvement of the entire team. In the mid term, the effort required is comparable to mentoring or coaching, while in the long term it pays off through improved development practices.

The architect becomes responsible for creating a common vision of the system



There are no real counter indications to VISIBLE QUALITIES, but if your team is resilient to learning other therapies might be more cost effective for your project at hand. You might experience negative side effects if you fail to explain the importance of different qualities, and a continuous neglect of specific qualities might finally break a large system. Prevent this by establishing a veto right on certain priorities. An overdose could be injected if the team does not get the idea at all, or is disgusted by the somewhat formal process. Use the drive for discussion to come to an adequate dosage.



Bringing a mentor to the project could reduce the ceremony level introduced by VISIBLE QUALITIES. Otherwise, they are successfully accompanied by ARCHITECT ALSO IMPLEMENTS [CoHa04].

For motivation of the team and management, the testability quality often is a good starter. Its benefits towards risk reduction and customer satisfaction are obvious, and it can be verified with concrete actions, namely implementing the tests. For testability, the achievement criterion could be ‘all classes are accompanied by at least one unit test’ or, if you introduce unit tests late in the project, ‘every fixed defect has to be accompanied by at least one new test case’. If for some reason the unit testability is hard to achieve, this is a potential hint for a design fault. To get away with a rule violation, a developer should need to convince the architect. There are situations e.g. in GUI development that are hard to unit test, but improvement suggestions may enable to test at least parts of the functionality, e.g. after a class has been split into distinct parts.

It is not important to maintain the list for a long time. If you introduce it, and hold it up often enough so that the developers know that you are serious about it, you might neglect the list and only check it at the start of a new iteration or release period. The check to what degree you have reached the internal qualities never becomes obsolete, but can be reduced to one check with each iteration or release.

Some qualities are hard to measure by numbers, but for others there are commercial tools available. As an example, the software tomograph [Roock06] supports a quantitative evaluation of the internal software structure.

The team was new to object-oriented design, so we discussed a lot about the promised qualities it should deliver. We started to do joint design at the white board, and explored some examples how a high extensibility could be reached, how testability could be increased, and what amount of decoupling required what effort.

When the team size increased, design reviews became an essential part of the project. Initially I participated in most, and we established an ordered catalogue of criteria to check. With this catalogue, the process was accepted and carried by the team. Closer to the end of the project, the team decided to focus on other issues and reduce the formality of the design reviews. By that time, the project lasted for more than two years; all team members had significant expertise and shared a common sense.

DEDICATED ARCHITECT

Applies to projects that have no dedicated architect and experience trouble with their architecture, either in quality of the architecture itself, in incoherent visions, or in uncovered effort.

In a development team that has an informal design and architecture process without a dedicated role assignment, the lack of a dedicated architect can cause one or more of the following problems:

- The development focus is on management goals only. A technical focus is not present, or is randomly selected by individual developers or managers.
- Important internal system qualities that are essential to adequately manage the system are not addressed.
- Developers who take over significant parts of the architectural tasks fall behind their schedule.
- Different people address different expert developers for technical issues.
- Questions concerning the architecture are not consistently answered.

These forces are present in the situation:

- An acknowledged architect has less time available for real programming, and is potentially expensive,
 - ...but dealing with inconsistencies and the subsequent system failures is even more expensive,
- Small projects can come along without much effort on architecture,
 - ...but building a large or reuseable system requires attention to issues that hardly matter in smaller systems; neglecting internal qualities can cause a system to break under its own weight.
- Any architect’s experience and view on the software world is limited,
 - ...but an architect has the broadest view of the developers, and he can still delegate.
- Newly assigning an architect within a given team might cause personal conflicts,
 - ...but each such conflict would be present anyway, and would otherwise express itself in technical inconsistencies.

Therefore, ensure that an architect’s role becomes defined and assigned to a key developer. The architect becomes responsible for creating a common vision of the system, ensuring technical consistency, broadening the architectural view, re-balancing the different forces on the architecture, and coaching the development team on the internal qualities (the ‘~ilities’) that are essential to crafting large software systems. In turn, all developers, managers and technical leads pass their decision competency in these areas to the dedicated architect, and provide sufficient resources – namely the working time of the architect.

this means that the team creates the architecture by consensus or accident, though with the best of intentions

An architect that is expected to initiate significant change will need dedication, explicit empowerment, and time to become accepted among his peers. Much less time can be devoted to 'real work' such as coding, and this needs to be reflected in the project schedule and the performance review criteria. While most project situations can live with one or more developer informally taking parts of the architect's role, as advocated by agile development methods [Beck99, Agile01], architectural tasks may require significant effort and time. You can compromise on how the role is called, but an architect needs to be able to spend significant effort without troubling his boss or his career.

The obvious candidate for the assignment is the informal architect. Convince your manager to establish the architect's role by indicating the risk reduction it could bring to the project, and by comparing the consequences of not having a consistent architecture against the costs of having an architect. This process will likely take some time since the problem needs to be perceivable to management. Try to get support from other developers in advance, including external contractors in case they join the project team. Their opinions might be considered more significant than those of employees.

When the team has several informal architects, the one of them who is most frequently asked is the right candidate. A team of architects can also work when each member has a distinct key area. However, one person must have the final decision.

If there is no informal architect, this means that the team creates the architecture by consensus or accident, though with the best of intentions. In this case you should consider asking for an outsider to join. The same applies if there are too many architects in the team, and picking one of them would break the project.



The mechanism behind DEDICATED ARCHITECT is acknowledgement by management. Only an acknowledged architect is able to devote sufficient time, and receive sufficient respect from the team.



DEDICATED ARCHITECT involves management, the architect, and potentially all team members. The costs can become significant because you need to dedicate time to architectural issues as well as to establishing the new role in the first place. Contracted architects are even more expensive than internal ones. However, the costs for a good architect will reduce the project risk and likely pay off several times, while the costs for a bad one will lead to further cost explosion.



DEDICATED ARCHITECT has one counter indication: when the team would not accept any architect. Its side effects are on the workload that the team can manage. It will decrease in the short term, but eventually increase in the mid to long term. Another side effect is the positive influence on the career of the assigned architect.



The trust in the architect is likely fostered by ARCHITECT ALSO IMPLEMENTS [Coplien04], when the developers perceive that the architect still knows how to express ideas in code.

The doctor wants to see you again

For a system suffering from Performitis, the major engineering practices have been introduced. They should reduce the problems to an acceptable level, and can be applied after considering their effects and costs. Other therapies, VISIBLE QUALITIES and DEDICATED ARCHITECT, facilitate the introduction of technical practices.

While this is sound practice and probably useful advice, it doesn't address the heart of the problem. Your team is likely to fall back into Performitis as soon as these therapies are no longer applied strictly. And poor compliance with prescribed therapies is something that all doctors suspect of their patients...

By next time, the therapies should have become effective. We will then reflect on the successes and see what we can do about the non-technical issues. We can then avoid Performitis in your next project. ■

References

- [Agile01] <http://www.agilemanifesto.org>
- [Beck99] Beck, Kent (1999) *Extreme Programming Explained: Embrace Change*, Addison-Wesley.
- [Cockburn98] Cockburn, Alistair (1998) *Surviving Object-Oriented Projects*, Addison-Wesley.
- [Coplien04] Coplien, James and Harrison, Neil (2004) *Organizational Patterns of Agile Software Development*, Prentice-Hall.
- [Fowler99] Fowler, Martin (1999) *Refactoring*, Addison-Wesley.
- [Marquardt99] Marquardt, Dr. Kerstin (1999) private communication.
- [Marquardt01] Marquardt, Klaus (2001) 'Dependency Structures. Architectural Diagnoses and Therapies' in *Proceedings of EuroPLoP 2001*.
- [Marquardt02a] Marquardt, Klaus (2002) 'Principles of Performance Tuning' in *Proceedings of EuroPLoP 2002*. See <http://www.kmarquardt.de/performance> for this and other papers.
- [Marquardt02b] Marquardt, Klaus (2002) 'Patterns for the Practicing Software Architect' in *Proceedings of VikingPLoP 2002*.
- [Roock06] Roock, Stefan and Lippert, Martin (2006) *Refactorings in Large Software Projects: Performing Complex Restructurings Successfully*, Wiley.
- [Weinberg92] Weinberg, Jerry (1992) *Software Quality Management Series: First-Order Measurement*, Dorset House.

Globals, Singletons and Parameters

One size rarely fits all. Bill Clare considers different approaches to parameterization.

The purpose of this article is to outline a design approach for allowing data to be shared where needed and to not be visible elsewhere. The issue becomes particularly interesting when not all users who have visibility of a shared capability are actually provided with the same implementation of that capability.

Background

Kevlin Henney [Henney07] has recently written a series of articles ('The PFA Papers: Context Matters' et al.) about the history and advantages of a pattern termed PFA or PARAMETERIZE FROM ABOVE. The pattern involves passing environment variables to applications as parameters rather than through use of globals or singletons.

Allan Kelly [Kelly04] addressed many of the same issues by developing a Pattern for a shared context 'The Encapsulated Context Pattern'. A spirited set of responses was later given (Overload 65, February 2005).

Many have written about the use and misuse of singletons.

Problem to be addressed

Before addressing approaches to this, it is well to state carefully the issues involved.

OO methodology provides strong support for encapsulation of the behaviour of a single or a related set of concepts. By itself, though, it provides little guidance about how these concepts interact and communicate. Various language features, patterns and approaches address this issue.

Here we consider approaches where:

- There are multiple users or clients that share data from multiple external capabilities.

Clients can include components, services, threads, algorithms, objects or functions.

Capabilities can include services, characteristics, resources, error and exception handlers, and values.

Also capabilities can be clients of higher level capabilities, which in turn are external to them.

- Different users, under different circumstances may need a different version or implementation of a particular capability or service.

Here we can partition uses by nested scopes in the function invocation hierarchy, where invocation includes both function calls and routing of work to different servers.

This provides a separation of capabilities, that are dependent on their environment, from the base functionality of the clients, which are environment independent. The approach is to allow clients to be adapted to a rich set of environment based capabilities without code change to the client. However, it is worth noting that this notion of an 'environment' boundary can be somewhat flexible for many applications.

This notion also supports some of the concepts of Aspect Oriented programming, where common capabilities are 'woven' into client users.

The emphasis there is on compile-time binding, while here it is on runtime binding.

Objectives

The basic objectives here are to suggest a framework where:

- Sharing of a common capability does not create dependencies among the clients.
- Implementations for capabilities can be specified globally to meet overall system requirements and adapted locally to meet specific internal scope requirements.
- New capabilities can be introduced without impacting existing code.
- New implementations of a capability can be introduced without impact to existing code.
- New scopes can be introduced and modified without impacting existing code.
- Code to access capabilities is independent of their implementation or of their tailoring to a scope.

General approach

An approach to this can be based on considerations of an overall environment with particular implementations of shared capabilities to be used within certain scopes. Also there are related considerations for capabilities that have interdependencies, for resource control, for establishing concurrent processing, for data sharing, for establishing controls externally and for testing.

Environment

An environment is specified through a set of function or function object pointers that provide client access to capabilities.

Capability implementations

Capability implementations can be maintained and specified:

- globally and accessed through maps indexed by scope;
- locally to the scope that uses them; or
- in some combination of these.

Roles

Ordinarily the external capabilities are orthogonal not only to their clients but also to each other. Where there are interdependencies among the

Bill Clare Bill Clare has retired from a 40 year career with IBM and Lockheed Martin, which focused largely on software engineering, architecture and design. This work included both product development and support, as well as development of large software systems for U.S. Government agencies.

If it is not known how many objects will be needed within the scope, then an empty container, with possibly a factory, can be set up

capabilities, it is useful to view them as satisfying a common role. Examples of role based capabilities include:

- A set of mathematical and physical constants that are needed to provide consistent values and precision for a particular algorithm.
- A logging capability which collects, filters, formats, routes and records data.
- Here different clients may need different versions of some of these particular functions.

Scopes

Values of a capability access pointer are stacked by scopes within the execution hierarchy. This can occur in two phases:

- Before the scope is invoked.
This is useful to establish external consistency and to meet external requirements.
- Within scope initialization.
This is useful to establish internal consistency and to meet internal requirements.

In both cases, resources may be obtained and setup, and appropriate configurations initialized when the scope is entered. When the scope is exited, finalization routines release resources, and restore the previous external state.

Access to other capabilities, that do not need to be adapted to the scope, are left untouched and thus are directly inherited.

Resource allocation and management

Data and other resources for shared capabilities needs to be actually created at some time and within some scope. The scope concept suggests a framework for management of resource instances. In many cases, this can provide more structured support for data sharing and control than through use of so called smart pointers.

Alternatives here include:

- Resources instances are created as needed, and released after use.
- Resources instances are created when first needed, and then cached and reinitialized as needed.
Actual release of the resource can occur at some higher level of scope, based on tradeoffs of memory and processor usage.
- Resources can simply be initialized in advance of use, possibly at startup, and never explicitly released.
- If it is not known at the scope level if data or a resource will be needed, then a Singleton can be set up or the Singleton can register with the scope manager. The resource is released if necessary at the scope end.
- If it is not known how many objects will be needed within the scope, then an empty container, with possibly a factory, can be set up and the resources released, if necessary, at the scope end.

Implementation instances can be specified globally through maps indexed by a scope ID to appropriate functions or objects. Alternatively, implementations can be specified through local scopes where they are needed. This is based on design trade-offs between a co-ordinated global environment and configuration management on the one hand, and independent support for lower level functions on the other.

Concurrent processing

For concurrently executing threads or processes and for remote executions, the current environment is copied to initialize the new thread, process or remote execution environment.

For applications that queue requests to a separate thread or process, the queue manager can propagate access pointer references to the execution environment or environments of the request processing.

Data sharing

Where data needs to be shared for both read and update, the usual issues of data sharing remain, independently of scope management. These issues can be addressed with the usual techniques of:

- Providing separate data instances, possibly using copy-on-write techniques.
- Use of locking.
- Queuing requests to a data owner thread or process.

External environment parameters

Parameters for adaptation can be supplied in environment variables and files that are accessed by initialization routines. With this, adaptation and tailoring of services for particular environments can be accomplished without code modification.

Test requirements

Testing occurs at several levels.

- Capabilities can be tested independently of clients.
- Clients can be unit tested with adapted versions of the capabilities they invoke.
- As always, integration testing is needed to verify interactions among separate capabilities.

Accessing capabilities

Actual access to capabilities needs to be consider from two perspectives:

- Scope managers to setup client access to particular capability implementations.
- Client access to actually make use of the capability.

Scope setup

Scope managers set up client access to particular capability implementations. They have visibility to implementations only to create

Scopes have internal and external capabilities with many of the external capabilities shared to differing degrees with other components

and locate them, and then to set pointers. There are several mechanisms possible for this access.

- The most direct is through global variables for the pointers.

In particular, these globals could be isolated in an Environment Namespace.

- Singletons are sometimes advocated as an alternative to the use of globals.

Singletons combine concepts of global access and creation when needed. This creation on the fly, when and if the object is needed, is the advantage of a Singleton. However, it is also its disadvantage, in that it leaves open the issue of when the resource should be deleted, if ever.

Here two approaches are possible:

- A single Singleton could provide the basis for access to all implementation instances.
- Singletons could be used to create shared data resources as needed. Again scope management could be used to release the resources at appropriate times.

Ultimately however, the Singleton creator itself must be global, and unless there is some reason to postpone this creation, it does not appear to provide added value.

- An alternative is for scope managers to pass parameters, or parameter blocks, to their immediate functions, which then pass these down the call hierarchy.

This can provide some encapsulation by combining the internal and external capabilities within a single function, but it introduces its own complexity.

None of these techniques is exclusive of the others, and they are independent of the requirements of scope management. Thus, they can be combined as needed, especially for code obtained from different sources.

Client access

Actual use of a capability within client code can be much simpler. Here, the actual base application code can look like:

```
pi ( );           // retrieve value with
                 // consistent precision for
                 // this scope
```

or perhaps:

```
math ( ).pi ( ); // retrieve role and property
```

The implementation mechanism at the client level can determine if these routines that access `pi` or `math` are:

- set up as globals

- defined within the local scope
- functions or properties of the local class.

In particular, this is independent of whether the actual values are derived from globals, singletons or passed parameters.

Logging example

Turning back to the logging example above, particular scopes may need specific logging capabilities, and so may substitute their own or just add to a global capability. Logging within a particular scope can in turn have specific functions for filtering, formatting, or routing. For test environments, or even as deployed, logging needs may vary considerably for individual scopes and circumstances.

Applications can be instrumented with a considerable amount of internal trace calls. Such code usually has considerable overhead, so it is desirable to be able to dynamically adjust the amount of detailed logging within separate scopes. Each trace call can provide parameters that specify a level of detail at which output should be recorded. The amount of output for specific tests, or to debug specific problems, can then be adjusted through external parameters that specify trace levels for different scopes. With appropriate templates, some of this tailoring can occur at compile time, with calls being completely eliminated through redefinition of the call templates.

Summary

A program's environment consists of a set of nested scopes which can be more or less global. Scopes have internal and external capabilities with many of the external capabilities shared to differing degrees with other components. A capability may have different implementations for use in different scopes under different conditions.

Judicious use of scope concepts allows common capabilities to:

- be global where needed, but limited otherwise
- support managed resources
- allow controlled data sharing
- be flexible, in terms of adaptation and of being introduced into an evolving code base.

Within this framework, applications can, without code impact, use capabilities derived from globals, singletons or parameters or, where necessary, combinations of such techniques. ■

References

- [Henney07] Henney, Kevlin (2007) 'The PfA Papers: Context Matters' in *Overload 82*, December 2007 (and other articles in the same series in subsequent volumes).
- [Kelly04] Kelly, Allan (2004) 'The Encapsulated Context Pattern' in *Overload 63*, October 2004.

Exceptions Make for Elegant Code

Anything that can go wrong, will go wrong. Anthony Williams compares ways of dealing with errors.

On episode 8 of the Stack Overflow podcast that he does with Jeff Atwood [Spolsky], Joel Spolsky comes out quite strongly against exceptions, on the basis that they are hidden flow paths. Whilst I can sympathise with the idea of making every possible control path in a routine explicitly visible, coming back to writing C code for a recent project after many years of coding in C++ has driven home to me that this actually makes the code a lot harder to follow, as the actual code for what it's really doing is hidden amongst a load of error checking.

Whether or not you use exceptions, you have the same number of possible flow paths. With exceptions, the code can be a lot cleaner than with exceptions, as you don't have to write a check after every function call to verify that it did indeed succeed, and you can now proceed with the rest of the function. Instead, the code tells you when it's gone wrong by throwing an exception.

Exceptions also simplify the function signature: rather than having to add an additional parameter to hold the potential error code, or to hold the function result (because the return value is used for the error code), exceptions allow the function signature to specify exactly what is appropriate for the task at hand, with errors being reported 'out-of-band'. Yes, some functions use `errno`, which helps by providing a similar out-of-band error channel, but it's not a panacea: you have to check and clear it between every call, otherwise you might be passing invalid data into subsequent functions. Also, it requires that you have a value you can use for the return type in the case that an error occurs. With exceptions you don't have to worry about either of these, as they interrupt the code at the point of the error, and you don't have to supply a return value.

Listing 1 shows three implementations of the same function using error code returns, `errno` and exceptions.

Error recovery

In all three cases, I've assumed that no recovery is required if `do_blah` succeeds but `do_flibble` fails. If recovery was required, additional code would be required. It could be argued that this is where the problems with exceptions begin, as the code paths for exceptions are hidden, and it is therefore unclear where the cleanup must be done. However, if you design your code with exceptions in mind I find you still get elegant code (see my blog entry [Williams#2] for some considerations on elegance in software). `try/catch` blocks are ugly: this is where deterministic destruction comes into its own. By encapsulating resources, and performing changes in an exception-safe manner, you end up with elegant

```
int foo_with_error_codes(some_type param1,
    other_type param2, result_type* result)
{
    int error=0;
    intermediate_type temp;
    if ((error=do_blah(param1,23,&temp)) ||
        (error=do_flibble(param2,temp,result))
        )
    {
        return error;
    }
    return 0;
}

result_type foo_with_errno(some_type param1,
    other_type param2)
{
    errno=0;
    intermediate_type temp=do_blah(param1,23);
    if(errno)
    {
        return dummy_result_type_value;
    }
    return do_flibble(param2,temp);
}

result_type foo_with_exceptions(some_type param1,
    other_type param2)
{
    return do_flibble(param2,do_blah(param1,23));
}
```

Listing 1

code that behaves gracefully in the face of exceptions, without cluttering the 'happy path'. See Listing 2.

In the error code cases, we need to explicitly cleanup on error, by calling `cleanup_blah`. In the exception case we've got two possibilities, depending on how your code is structured. In `foo_with_exceptions`, everything is just handled directly: if `do_flibble` doesn't take ownership of the intermediate data, it cleans itself up. This might well be the case if `do_blah` returns a type that handles its own resources, such as `std::string` or `boost::shared_ptr`. If explicit cleanup might be required, we can write a resource management class such as `blah_cleanup_guard` used by `foo_with_exceptions2`, which takes ownership of the effects of `do_blah`, and calls `cleanup_blah` in the destructor unless we call `dismiss` to indicate that everything is going OK.

Real examples

That's enough waffling about made up examples, let's look at some real(ish) code. Here's something simple: adding a new value to a dynamic array of `Data` objects held in a simple `dynamic_array` class. Let's

Anthony Williams Anthony is the Managing Director of Just Software Solutions Ltd, where he spends most of his time developing custom software for clients. Anthony is also the maintainer of the Boost Thread Library, and has considerable experience with developing and maintaining high-quality multi-threaded software in C++. He can be contacted at anthony@justsoftwaresolutions.co.uk.

this is where the problems with exceptions begin, as the code paths for exceptions are hidden

```
int foo_with_error_codes(some_type param1,
    other_type param2,result_type* result)
{
    int error=0;
    intermediate_type temp;
    if(error=do_blah(param1,23,&temp))
    {
        return error;
    }
    if(error=do_flibble(param2,temp,result))
    {
        cleanup_blah(temp);
        return error;
    }
    return 0;
}

result_type foo_with_errno(some_type param1,
    other_type param2)
{
    errno=0;
    intermediate_type temp=do_blah(param1,23);
    if(errno)
    {
        return dummy_result_type_value;
    }
    result_type res=do_flibble(param2,temp);
    if(errno)
    {
        cleanup_blah(temp);
        return dummy_result_type_value;
    }
    return res;
}

result_type foo_with_exceptions(some_type param1,
    other_type param2)
{
    return do_flibble(param2,do_blah(param1,23));
}

result_type foo_with_exceptions2(some_type param1,
    other_type param2)
{
    blah_cleanup_guard temp(do_blah(param1,23));
    result_type res=do_flibble(param2,temp);
    temp.dismiss();
    return res;
}
```

Listing 2

assume that objects of `DataType` can somehow fail to be copied: maybe they allocate memory internally, which may therefore fail. We'll also use a really dumb algorithm that reallocates every time a new element is added. This is not for any reason other than it simplifies the code: we don't need to check whether or not reallocation is needed.

If we're using exceptions, that failure will manifest as an exception, and our code looks like Listing 3. On the other, if we can't use exceptions, the code looks like Listing 4.

It's not too dissimilar, but there are a lot of checks for error codes: `add_element` has gone from 10 lines to 17, which is almost double, and there are also additional checks in the `heap_data_holder` class. In my

```
class DataType
{
public:
    DataType(const DataType& other);
};
class dynamic_array
{
private:
    class heap_data_holder
    {
        DataType* data;
        unsigned initialized_count;
    public:
        heap_data_holder():
            data(0),initialized_count(0)
        {}
        explicit heap_data_holder(unsigned max_count):
            data((DataType*)malloc(
                max_count*sizeof(DataType))),
            initialized_count(0)
        {
            if(!data)
            {
                throw std::bad_alloc();
            }
        }
        void append_copy(DataType const& value)
        {
            new (
                data+initialized_count) DataType(value);
            ++initialized_count;
        }
        void swap(heap_data_holder& other)
        {
            std::swap(data,other.data);
            std::swap(initialized_count,
                other.initialized_count);
        }
    }
};
```

Listing 3

```

unsigned get_count() const
{
    return initialized_count;
}
~heap_data_holder()
{
    for(unsigned i=0;i<initialized_count;++i)
    {
        data[i].~DataType();
    }
    free(data);
}
DataType& operator[](unsigned index)
{
    return data[index];
};
heap_data_holder data;
// no copying for now
dynamic_array& operator=(
    dynamic_array& other);
dynamic_array(dynamic_array& other);
public:
dynamic_array()
{}
void add_element(DataType const& new_value)
{
    heap_data_holder new_data(data.get_count()+1);
    for(unsigned i=0;i<data.get_count();++i)
    {
        new_data.append_copy(data[i]);
    }
    new_data.append_copy(new_value);
    new_data.swap(data);
}
};

```

Listing 3 (cont'd)

```

class DataType
{
public:
    DataType(const DataType& other);
    int get_error();
};
class dynamic_array
{
private:
    class heap_data_holder
    {
        DataType* data;
        unsigned initialized_count;
        int error_code;
    public:
        heap_data_holder():
            data(0), initialized_count(0),
            error_code(0)
        {}
        explicit heap_data_holder(unsigned max_count):
            data((DataType*)malloc(
                max_count*sizeof(DataType))),
            initialized_count(0), error_code(0)
        {
            if(!data)
            {
                error_code=out_of_memory;
            }
        }
    };
};

```

Listing 4

```

int get_error() const
{
    return error_code;
}
int append_copy(DataType const& value)
{
    new (
        data+initialized_count) DataType(value);
    if(data[initialized_count].get_error())
    {
        int const error=
            data[initialized_count].get_error();
        data[initialized_count].~DataType();
        return error;
    }
    ++initialized_count;
    return 0;
}
void swap(heap_data_holder& other)
{
    std::swap(data, other.data);
    std::swap(initialized_count,
        other.initialized_count);
}
unsigned get_count() const
{
    return initialized_count;
}
~heap_data_holder()
{
    for(unsigned i=0;i<initialized_count;++i)
    {
        data[i].~DataType();
    }
    free(data);
}
DataType& operator[](unsigned index)
{
    return data[index];
}
};
heap_data_holder data;
// no copying for now
dynamic_array& operator=(dynamic_array& other);
dynamic_array(dynamic_array& other);
public:
dynamic_array()
{}
int add_element(DataType const& new_value)
{
    heap_data_holder new_data(data.get_count()+1);
    if(new_data.get_error())
        return new_data.get_error();
    for(unsigned i=0;i<data.get_count();++i)
    {
        int const error=
            new_data.append_copy(data[i]);
        if(error)
            return error;
    }
    int const error=
        new_data.append_copy(new_value);
    if(error)
        return error;
    new_data.swap(data);
    return 0;
}
}

```

Listing 4 (cont'd)

experience, this is typical: if you have to explicitly write error checks at every failure point rather than use exceptions, your code can get quite a lot larger for no gain. Also, the constructor of `heap_data_holder` can no longer report failure directly: it must store the error code for later retrieval. To my eyes, the exception-based version is a whole lot clearer and more elegant, as well as being shorter: a net gain over the error-code version.

Error safety

I expect most of you are familiar with the Abrahams Exception Safety Guarantees [Abrahams], but these could realistically be termed Error Safety Guarantees: we want code that is robust in the face of errors in general, not just exceptions. The only reason that exceptions are ‘special’ is that people are less familiar with how to write code in the presence of exceptions. It is providing sensible guarantees for the code that leads us to the structure of the example code above; something that remains essentially the same even when using error codes. Creating a copy of a structure ‘off to the side’ and then swapping it with the original is a useful technique whichever error handling mechanism you use, but it really comes into its own with exceptions.

Structural changes

The lack of the ability for the constructor of `heap_data_holder` to abort on failure means that the way objects are written must change: the ‘invariants’ of the class must be extended to allow for it to be in an ‘invalid’ state due to the constructor failing. Similarly, the signatures of some functions must change to allow for failures: if your function returns a reference then this can pose a problem if the function failed: you don’t necessarily have an object to return a reference to, and must instead return a pointer, which can therefore be `NULL`. This subtle shift in the design of the code now means that any code that calls this function has to be prepared for a `NULL` pointer to be returned where before it could rely on there being an object, since there is no such thing as a `NULL` reference.

The lack of exceptions actually makes it hard to pass the result of one function as a parameter to another altogether: because the function will return normally even if it failed, the second function has to handle whatever the first returns on error without causing serious problems. We saw this back in the first example where the call to `do_blah` was separated from the call to `do_flibble` in order to check the error code, whereas the exception version had `do_blah` called directly in the call for `do_flibble`. If you apply this to operators it gets even worse: operators don’t have any means of returning an error code directly, so they have to resort to techniques such as the use of `errno`, and you essentially lose any benefits of writing an operator in the first place. With exceptions, we can write:

```
std::string foo(std::string const& s)
{
    return "hello " + s + " goodbye";
}
```

where the second call to `operator+` will only happen if the first succeeded. If we don’t have exceptions then the first call to `operator+` has to return something, and the second call to `operator+` has to handle the case that one or more of its arguments is an ‘invalid’ object.

Conclusion

I guess it’s a matter of taste, but I find code that uses exceptions is shorter, clearer, and actually has fewer bugs than code that uses error codes. Yes, you have to think about the consequences of an exception, and at which points in the code an exception can be thrown, but you have to do that anyway with error codes, and it’s easy to write simple resource management classes to ensure everything is taken care of. Without exceptions you often have to contort the design to handle the error checking. ■

The Abrahams Exception Safety Guarantee

These guarantees were first documented by Dave Abrahams when the C++ Standards committee were working on the 1998 C++ Standard. The idea is that code should provide one of the three guarantees – if it doesn’t, then an exception occurring in your code will result in leaked resources or corrupt data structures or both. The guarantees are:

The no-fail (or no-throw) guarantee

This is the strongest of all guarantees. A function that provides this guarantee will not throw any exceptions, and will not fail. All destructors should provide this guarantee, as should important operations like swap which provide the building blocks for the code that uses them to provide suitable exception safety guarantees.

The strong guarantee

A function that provides this guarantee is all or nothing: if it fails, then any effects are rolled back so the state of the data structure is the same as it was on entry. This requires that the function doesn’t do anything irreversible (like perform I/O), and that there are suitable operations that provide the no-fail guarantee which can be used to commit or roll back the changes.

The basic guarantee

This is the basic level you should strive for in all code: if a function fails, then it must leave the data structures in a valid state, even if that state differs from the original. For example, failure to insert a new item into a container must leave the container in a valid state, even if all the existing items have been deleted.

References

- [Abrahams] Abrahams, D., ‘Exception Safety in Generic Components’, http://www.boost.org/community/exception_safety.html
- [Spolsky] ‘Stack Overflow’ (podcast) – available from: <http://blog.stackoverflow.com/index.php/2008/06/podcast-8/>
- [Williams07] ‘Elegance in Software’, <http://www.justsoftwaresolutions.co.uk/design/elegance-in-software.html>

This article is based on a blog entry with the same title at <http://www.justsoftwaresolutions.co.uk/design/exceptions-make-for-elegant-code.html>

Divide and Conquer: Partition Trees and Their Uses

The world is a complex place. Stuart Golodetz introduces a powerful technique to describe it.

Partition trees are a useful data structure for maintaining a hierarchy of partitions of an entity, e.g. the world, an image, or even a pizza. In this article, I want to explain how they work and describe some of their uses.

A *partition* of an entity is a division of it into mutually disjoint parts which together make up the whole entity. (Put mathematically, a partition of E is a set $S = \{E_1, \dots, E_n\}$ of sub-entities of E such that $\text{Union}(S) = E$ and for all i, j in $\{1, n\}$, $i \neq j \Rightarrow \text{intersect}(E_i, E_j) = \emptyset$.) For example, we could partition a pizza into twelve slices: none of the slices overlap, and together they make up the whole pizza (until someone starts munching, at any rate).

A partition tree is a way of representing a hierarchy of these partitions of an entity. The way it works is as follows:

- Each node in the tree represents a part of the whole entity. In particular, the root node of the tree represents the entity in its entirety.
- As is usual with trees in computer science, a node is either a branch node or a leaf node. (The distinction is that a branch node has child nodes, whereas a leaf has none.) In a partition tree, the children of a branch node represent a partition of the branch node (properly speaking, the parts of the entity represented by the children of the branch node represent a partition of the part of the entity represented by the branch node, but continually distinguishing between nodes and the sub-entities they represent is tedious).
- Each layer in the hierarchy represents a partition of the entire entity.

Using our pizza analogy, we could imagine first dividing our pizza into three portions, one for each person at the table. Each person's portion is then further sub-divided into four slices. Each person's slices are a partition of their portion, and the portions are a partition of the pizza as a whole. Furthermore, if you take all the slices, or all the portions, together, you have the entire pizza.

As a graphical example, take a look at Figure 1, which shows a partition tree for the square image shown at the bottom-right. The image pieces at each node are colour-coded for easy identification in the images for each layer of the tree on the right. They are clearly mutually disjoint, and their union in each case is the whole image. This example with images illustrates one potential use for partition trees (and indeed the reason I'm interested in them at the moment, as those of you who saw my previous articles on image segmentation [Golodetz, 08b] may have guessed), but it is very much one among many.

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

Binary space partitioning

We'll return to image partition trees later, but first I want to look at a different domain in which partition trees are useful, that of partitioning 2D or 3D space. Space partitioning is an extremely useful tool in the arsenal of 3D graphics programmers, particularly in the context of games, which was where I first encountered the technique.

The general idea is to start with an infinite space (e.g. the game world) and recursively divide it into pieces. This division can happen in a number of ways, depending on the type of tree, but we'll start by discussing *binary space partitioning*.

Binary space partitioning, as its name suggests, is the process of recursively dividing a world into two. We start with our infinite world, split it across a plane, then split the newly formed sub-spaces on each side across other planes, and carry on until we run out of sub-spaces we want to split. These terminal sub-spaces become the leaves of our tree. (Figure 2 shows an example of the BSP construction process.) Each branch node stores the plane which was chosen to split its sub-space into two (the *splitter*, or *split plane*), and the tree generally satisfies the property that every node in the sub-tree rooted at the left child of a branch node represents a sub-space in front of that branch's splitter, and conversely for nodes in the right sub-tree. One consequence of the way the splitting process works is that the sub-spaces represented by the nodes are all convex (inductive proof: the initial world can be viewed as convex, and splitting a convex space across a plane gives you two convex sub-spaces as a result).

As an example of this, consider Figure 3. In (a) we see the recursive partitioning of a 2D world consisting of two rectangular rooms joined by a short corridor, whilst (b) shows its equivalent partition tree. Branch nodes in the tree are marked with the number of the wall they correspond to in (a), while leaf nodes are marked either \perp or \top (along with a Greek letter indicating the convex subspace in (a) to which they correspond). Leaves marked \perp are called *empty* leaves, and represent the parts of the world which are inside the rooms/corridor (the bits a player could walk around in, were this a game). Leaves marked \top are called *solid* leaves, and represent the opposite.

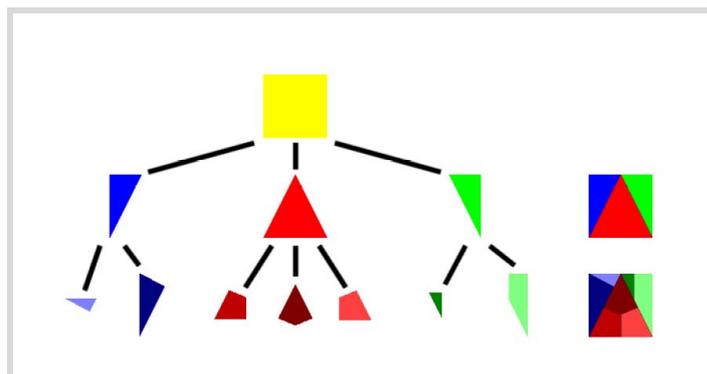


Figure 1

a tree like this can easily answer the question of whether any given point is in empty or solid space

The BSP construction process for a simple L-shaped room: at each stage, a split plane is chosen to split the world in two, according to a metric m based on the weighted linear combination of the balance of polygons on either side of the splitter and the number of polygons which will be split if that splitter is chosen. The plane of a polygon with the lowest value of m is chosen. The value of m in the figure is calculated as $8 \times \text{front} - \text{back} + 1 \times \text{straddle}$, where *front* is the number of polygons completely in front of the plane, *back* is the number completely behind, and *straddle* is the number straddling the plane. The weights 8 and 1 are somewhat arbitrary, and the ratio between them can be varied to produce alternative trees.

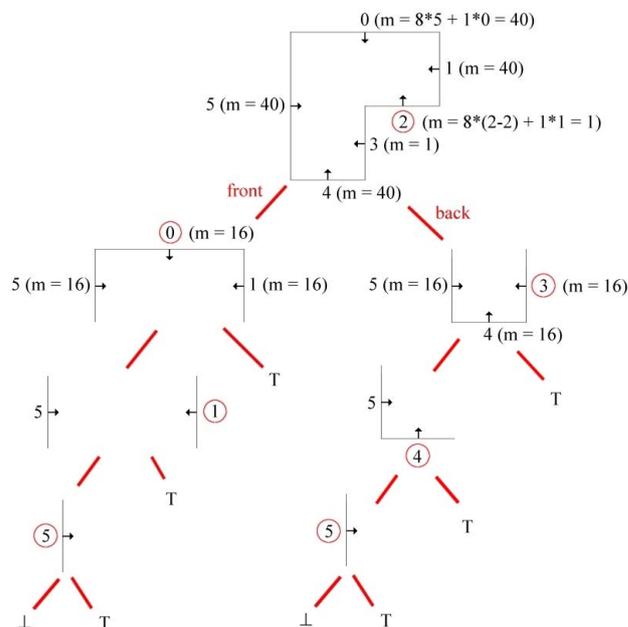


Figure 2

So why is this representation useful? Well, for a start, a tree like this can easily answer the question of whether any given point is in empty or solid space. As those of you who can remember your maths will recall, the equation of a plane is:

$$\hat{n} \cdot \vec{x} - d = 0$$

where \hat{n} is the unit normal of the plane, x is an arbitrary point on the plane and d is the plane distance value. The plane normal indicates the facing of the plane. Plugging a specific point x into this equation gives us a positive value if x is in front of the plane, a negative value if it's behind, and 0 if it lies in the plane (when we call it a *coplanar* point).

We can therefore classify a point in terms of where it lies in relation to a plane. To determine whether a point lies in empty or solid space, we start by classifying it against the split plane for the root node of the tree. If it lies in front of the plane, we recurse to the left child of the root (the child representing the sub-space in front of the plane) and classify the point again; if behind, we recurse to the right child. (If it lies on the plane, we

have a choice of what to do: if we recurse down the front side of the tree, then coplanar points will be classified as being in empty space; if we recurse down the back side, they'll be in solid space. It's up to us in this case.)

Eventually we will recurse into a leaf, which has no split plane. At this point, we return whether or not the leaf is solid as the result. This method provides a simple (if very naive) way of doing collision detection for small objects: we move an object, check whether its centroid is in solid space, and perform collision resolution on it if so. (There are much better ways, however!)

Another simple technique can be used to find the first 'transition point' of a half-ray, i.e. the first point at which the half-ray first goes from empty space to solid space, or vice-versa (the first point at which it hits a wall). (Specifically, given a ray $r(\lambda) = \text{start} + \lambda(\text{dir})$, $\lambda \geq 0$, the algorithm will return the transition point closest to *start*.) The process essentially works in a similar way to a line segment classifier, but it prioritises the side of the tree nearest the start of the ray. Starting from the root of the tree, the

They provide us with a way to render our polygons in back-to-front order when viewed from an arbitrary position in the world

algorithm classifies the ray against the split plane at the current node (via classifying its endpoints against the plane). If it is entirely on one side of the plane, it is passed down the relevant side of the tree. If it straddles the plane, it is split in two at the plane and the two halves are passed down their respective sides of the tree, with the half nearest the start of the ray being processed first and the other half only being processed if no transition point has been found in the first half. Finally, there are a number of annoying coplanar special cases: I don't want to get into those here, but the interested reader can find working code for the whole algorithm in my transfer report [Golodetz, 08a]. The find first transition algorithm provides a way of testing 'line-of-sight' in a game world (although it actually gives you back more information than you strictly need, as line-of-sight is a boolean value): this can be useful for AI players when deciding whether to try and shoot you, for example. It can also provide a (slightly!) better

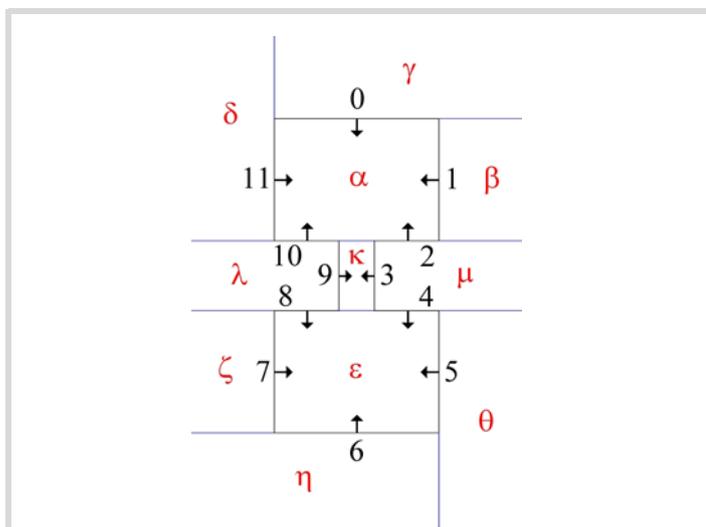
method of collision detection than the point-checking method above: we check whether the ray from the object's old location to its new location intersects a wall, and move it back to somewhere near the first transition point if it does (I say 'somewhere near' because we're usually dealing with objects that have extent, rather than point objects).

As a brief aside, I'll quickly mention a reasonably good BSP-based collision detection scheme you can use for objects with extent. This was used in id Software's *Quake III Arena* [van Waveren], with good results. The basic idea is a *configuration space* one. They determine an axis-aligned bounding box for each class of object in the game, then generate a special collision BSP for each size of bounding box (not necessarily one for each class of object, though this can happen if all the classes have different bounding boxes). *Quake III* worlds are built up from a large number of convex building blocks (brushes). In order to build a collision BSP for a particular size of bounding box, they push the planes which define each brush along their respective normals by a certain amount (thus expanding the brush *anisotropically*). Each plane is moved along its normal by the amount which would take it to the centre of the bounding box when the box was just touching the plane (see Figure 6.4 in [van Waveren] for a picture). The collision BSP is then built from the expanded brushes. What this whole exercise achieves is the generation of a tree that defines where the centroid of bounding box is allowed to go (i.e. the configuration space for the centroid): an annoying bounding box collision detection problem has thus been reduced to a point problem. We can then use any scheme we like (e.g. the find first transition approach described above) to perform the actual point-based collision detection. It should be noted that there are a few complications to the whole approach, which are described in [van Waveren], but this is the basic principle involved.

Collision detection aside, BSP trees are also useful for rendering. They provide us with a way to render our polygons in back-to-front order when viewed from an arbitrary position in the world. This was especially useful in the days when z-buffers were still quite costly to use, but it remains a useful technique even now when rendering transparent polygons. The rendering algorithm is once again recursive:

- Starting from the root node, classify the position of the viewer against the split plane.
- If it's in front of the plane, recurse down the back half of the tree and render all the polygons behind the plane, then render the polygon on the plane itself, then recurse down the front half of the tree and render all the polygons in front of the plane.
- If it's behind the plane, do the opposite, except that the polygon on the plane itself isn't rendered (we're behind it, so it gets back-face culled).

One very useful thing you can do with BSP trees (which saw use in id Software's *Quake* [Abrash]) is to precalculate something called the potentially visible set (PVS) of each leaf in the tree. This basically means calculating and caching which leaves can be viewed from which other leaves. The way this is done is a little bit complicated, but the first step is to calculate all the portals (doorways) between the various leaves: an interesting and useful process in itself. Portal calculation essentially



A partition diagram of an example 2D world (above) and its partition tree (below).

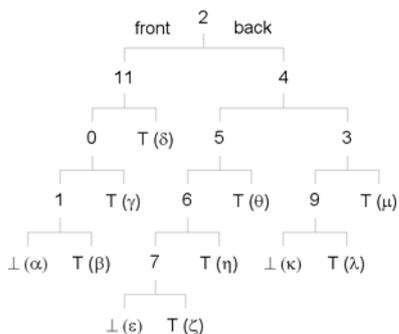


Figure 3

by combining two simple squares together, we can create a hollow square using a difference operation

involves clipping polygons to the tree. We build massive polygons (bigger than our game world) on each unique plane in which one of our world polygons lies, then clip them to the tree to get the portals. Caching the PVS for each leaf makes rendering substantially (e.g. an order of magnitude) faster. Instead of having to render all the polygons in the game world, we now only have to render a small subset of them. This was one of the key techniques that made *Quake* run so fast.

A final BSP-related technique worth mentioning is that of constructive solid geometry (CSG). This is a completely general technique which involves combining primitive solid objects into more complicated ones using set operations (i.e. union, intersection and difference) and doesn't necessarily have to be implemented using BSP trees, but doing CSG using BSP trees works rather well. Figure 4 shows the idea: by combining two simple squares together, we can create a hollow square using a difference operation. This sort of approach works extremely well in game level editors, since it allows the user to create complicated worlds from a very

simple set of initial objects (e.g. cuboids, cylinders, cones and spheres). For more details, see my undergraduate project report [Golodetz, 06].

Quadtrees and octrees

For now, let's look at a completely different way of partitioning the world. Quadtrees (in 2D) and octrees (their 3D analogue) are a way of subdividing the world along axis-aligned planes. Consider Figure 5, in which we imagine applying a quadtree approach to a 2D map of a river running through a landscape. The idea is basically as follows: at each stage, we divide the space into four along planes in the x and y directions. We have some sort of termination criteria to tell us when to stop. For instance, we could stop dividing a square when it contains $> 95\%$ river or land, say.

A simple example of a CSG difference operation

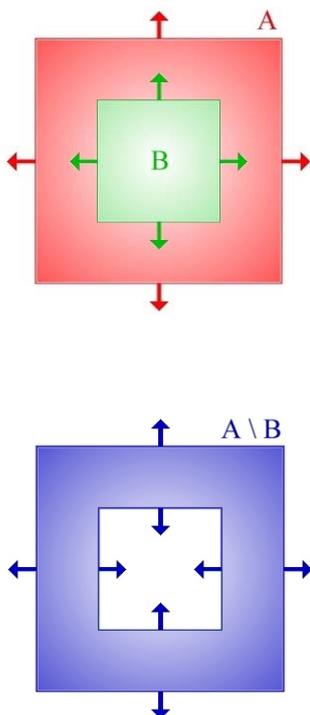


Figure 4

An example quadtree for a river – only the first 5 levels are shown.

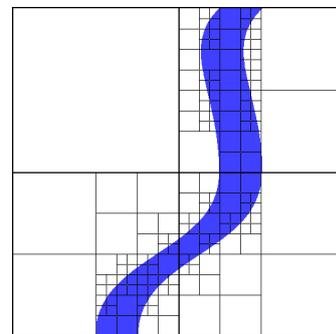


Figure 5

So why is this useful? One obvious application is mesh generation: if we've got large homogeneous areas of land, such as in the north-west part of the image, we don't want to generate a fine mesh for them because it's wasteful. If there's a lot of detail, however, such as in the north-east of the image, we don't want to miss it by having too coarse a mesh. Quadtrees provide one solution to this problem (another would be a binary triangle tree algorithm like ROAM – real-time optimally adapting meshes [Duchaineau]) by effectively adapting the grid size to the local terrain.

As with BSP trees, quadtrees can be used to classify a point in the world: in this case, we can easily determine whether a given point is part of the river or part of the terrain by recursively classifying the point against the tree. 'Big deal!', you might think, since we have the original image and can just look it up in that. But quadtrees are a much more space-efficient representation of the world than the original image: this allows us to capture the *useful* information in the original image without keeping it continually hanging around in memory.

There are lots of other uses of quadtrees besides dividing the terrain. One such application is collision detection (see Figure 6). The idea of Figure 6 is basically that objects can only collide with objects in the quadrants they

Using a quadtree to avoid doing unnecessary collision detection – the square remains within its quadrant, so it will never collide with the stationary circles and doesn't need to be tested against them.

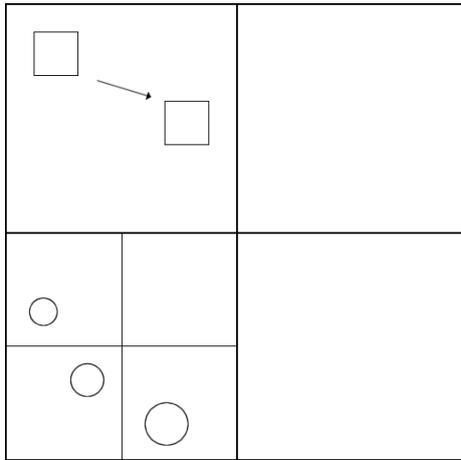


Figure 6

are in. So for instance, the square object won't collide with the stationary circles elsewhere in the world, so we don't need to check for that. There's more to it than that, of course, such as how to change the tree when the objects move around, but that's another story.

Image partition trees

For now, I want to conclude this brief overview of partition trees by returning to the image partition trees (IPTs) we started with in the introduction. So far, the partition trees we've seen have been constructed using a top-down splitting approach. This is certainly possible for IPTs, but for the purposes of this article, I want to describe an alternative bottom-up approach based on region merging.

The basic plan is as follows. We start by somehow generating a large number of small regions in the image, then combine some of them to form the next layer up in the tree, then iteratively repeat this process until we run out of regions to merge (i.e. we get a single region at the top of the tree: the root). At a more concrete level, the algorithms I described in my previous articles [Golodetz, 08b] can be directly put to use here: the watershed transform will give you a very fine initial partition of the image into small regions, and the waterfall algorithm can be used for the actual merging process, thereby generating a hierarchy of partitions of the image that get coarser the further up the tree you get. (Other approaches can also be taken here: this is merely by way of an 'existence proof' that it can be done.)

Having generated an IPT, what can we do with it? Its main use (i.e. its advantage over the simple sequence of partitions generated as the normal output of the waterfall algorithm) is in representing how regions were merged: it's very useful to know that (say) a kidney and two bits of liver were merged into a larger region, since if we identify the kidney (a process which will remove it from consideration) then we know that the remainder of the larger region represents the liver. It's also useful for knowing about parent relationships: if two regions are good candidates for being identified as a kidney, it's very important to know if one is the parent of the other, since it helps us distinguish between there being two separate kidneys, and there being two different versions of the same kidney.

Conclusion

Partition trees are a hugely useful information representation in at least two different domains. In this article, I've done my best to give you a broad overview of the various uses of spatial partition trees for things like collision detection, rendering, constructive solid geometry, mesh generation, and the use of image partition trees for feature identification

in images. In future articles, I hope to return to a few of the topics, such as portal and PVS generation, and level building using CSG, in more detail. Till then... ■

References

[Abrash97] Abrash, M., *Graphics Programming Black Book* (Special Edition), Coriolis Group Books, July 1997.

[Duchaineau97] Duchaineau, M., et al., 'ROAMing Terrain: Real-time Optimally Adapting Meshes', *IEEE Visualization Journal*, 1997.

[Golodetz06] Golodetz, S., *A 3D Map Editor* (undergraduate project report), May 2006: <http://compsci.gxstudios.net/project.pdf>.

[Golodetz08a] Golodetz, S., *Segmentation of Abdominal Organs and Growth Modelling of Tumours in Renal Cancer Patients*, (transfer report), p.59, May 2008: <http://dphil.gxstudios.net/transfer2.pdf>.

[Golodetz08b] Golodetz, S. 'Watersheds and Waterfalls', *Overload 83/84*, February/April 2008.

[VanWaveren01] Van Waveren, J.M.P., *The Quake III Arena Bot*, (MSc Thesis), p.25 onwards, June 2001: http://www.kbs.twi.tudelft.nl/docs/MSc/2001/Waveren_Jean-Paul_van/thesis.pdf.



C++ Libraries and Tools to Simplify Your Life

- > **POCO C++ Libraries:** free open source libraries for internet-age cross-platform C++ applications
- > **POCO Remoting:** the easiest way to distributed objects and SOAP/WSDL Web Services in C++
- > **POCO Open Service Platform:** create high performance component-based, manageable and dynamically extensible applications in C++
- > and much more: Fast Infoset, WebWidgets, ...
- > available on many platforms – highly portable code
- > scalable from embedded to enterprise applications



appliedinformatics

Free Download and Evaluation: appinf.com/simplify

On Management

Management is a vital part of software development. Allan Kelly starts a new series by balancing some constraints.

As usual, Fred Brooks [Brooks75] got here first: In many ways, managing a large computer programming project is like managing any other large undertaking – in more ways than most programmers believe. But in many other ways it is different – in more ways than most professional managers expect.

A few years later [Brooks95] he pointed out how important management is:

Some readers have found it curious that *The Mythical Man Month* devotes most of the essays to the managerial aspects of software engineering, rather than the many technical issues. This bias ... sprang from [my] conviction that the quality of the people on a project, and their organization and management, are much more important factors in the success than are the tools they use or the technical approaches they take.

Managing software development is a big topic. It is a mistake to equate the management of software development efforts with *project management*. There are project management aspects to the topic but they are a subset of the whole. Indeed, the discipline of project management openly acknowledges this. For example, the UK Government backed PRINCE 2 project management techniques excludes all human resources aspects of management.

PRINCE 2 defines a project [Commerce05] as:

A temporary organisation that is needed to produce a unique and predefined outcome or result at a prespecified time using predetermined resources.

While I'm sure this describes the situation some readers find themselves in, I'm also sure that many many more of you find yourselves in a different type of organization. You are working on something that doesn't have an end date, or if it does there will be another 'project' starting on the same code base the next day.

Rather than call these efforts *projects*, a better term is *products*. Products, unlike projects, go on and on. This introduces a longer time perspective and emphasises the need to produce something tangible from the work.

Product Management is a discipline in its own right. One that is understood much better in Silicon Valley and the US than it is in the UK and Europe. You can replace the word Project with the word Product but you can't replace a Project Manager with a Product Manager as the roles are different. More importantly the skills needed for each, and the training given to each, are different.

Then there is all the other management stuff: recruitment, retention, assessment, business strategy, etc., etc. In other words: there is a lot to be said about management in the software development arena.

Unfortunately, a lot of people have come to believe that 'project management' is the way to manage all IT. It isn't. There is a lot more to 'software development management' than managing the project. Limiting our view of management to 'project management' risks harming our work.

So I have decided Overload needs a new series, 'On Management'. We'll start with Project Management, move through into Product Management and take in some of the other stuff along the way. No time scales, no promises, no defined route, design will be emergent.

In this, and future, articles, I will not hide my agreement with Agile and Lean thinking. Indeed I will take many of the Agile practices as given. Agile is a brand, a powerful brand, and a brand that gets most things right. But it is also a brand that gets people's backs up. It's also a brand that doesn't go far enough in some respects.

When it comes to management most Agile management practices are just plain *good management*. I know not everyone agrees with Agile ideas – and I don't agree with every word ever written about Agile development – but at present I think Agile represents the current state of the art.

Product management, strategy, IT strategy, financing, human resources – recruitment, retention, objective setting, compensation, succession planning, and more – much more. There is plenty of material here. So best to get started...

Triangle of constraints

All software is developed under constraints but there are three which are more important than others: time, resources and features [McCarthy95].

Others could be added, money being the obvious: Money is, economists like to tell us, fungible. Which is another way of saying it can be exchanged for other things very easily. Money can be exchanged for resources such as a new developer, thereby increasing our resources. Or money may pay for overtime working thereby increasing the time we have on a project.

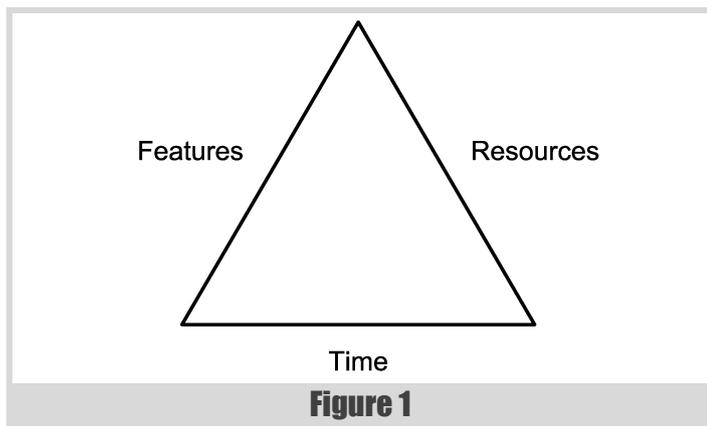
The net result is that introducing money complicates things. Since (almost) everything can be reduced, or replaced, by money, this analysis leaves money to one side. Rather it is better to regard cost as a function of time and resources, and revenue as a function of features. If we increase the time or resources then costs will increase, and if cost needs to be reduced then resources and/or time needs to be reduced.

Resources is a rather elastic word as well and can include just about anything. In the name of simplicity, in this context resources is taken to mean people (developer, testers, etc.) and the tools they need to do their job.

These three parameters can be thought of as a triangle, as shown in Figure 1.

Allan Kelly After years at the code-face Allan realised that most of the problems faced by software developers are not in the code but in the management of projects and products. He now works as a consultant and trainer to address these problems by helping teams adopt Agile methods and improve development practices and processes. He can be contacted at allan@allankelly.net and maintains a blog at <http://allankelly.blogspot.net>.

The future is uncertain, and the degree of uncertainty increases in proportion to the length of time considered



Lesson 1: Time, resources and features are the critical factors that require managing. But they are not the only factors.

All software development takes place within such a triangle. As with any triangle it is not possible to change one of the three parameters without changing another:

- More features must be accommodated by either increasing the amount of time available or adding more resources.
- Delivering a project in less time requires more resources or a reduction in features.
- Adding more people (resources) to the project, in theory, either reduces the amount of time it will take or allows for more features – except...

The people issue

That last bullet point sounds OK, doesn't it? Except the way it usually works is that adding more people invokes Brooks' Law [Brooks75]:

adding more people to a project a late software project makes it later

Adding people to a project comes at a cost. New people need time to come up to speed on the system being developed, the requirements, the existing code base, the technology, etc., etc. Consequently, in the short run the resources on a project are effectively fixed and adding more people will delay the work.

In the long run people can be added to a project, and they can increase the capacity to undertake work but they come at a cost. Therefore, as Brooks' Law states, if the project is late, adding more people will make it later.

However, if a project is not late, or rather if the project is managed actively, people may be added to the project without too much detriment. Projects which plan to add people can do it in an orderly fashion.

Lesson 2: Adding people to a project needs to be done in an orderly fashion.

In fact, it is essential to add people to a project over time because there is a natural tendency for people to leave a project. People get offered better jobs, people take time off for health and personal reasons, overseas workers decide to go home, and people retire.

Lesson 3: Active management seeks to slowly expand a team to compensate for natural loss.

Obviously there are times when this is inappropriate, such as when a project is winding down. There are also occasions where it is more important to add people.

The net result of these forces is that, for any project, in the short term the resources available are fixed or even reducing. (The short term may be as short as three months or as long as a year.) Only in the long term can resources be increased and even then major increases in resources are not possible.

Consequently, managing software development becomes an exercise in:

- Human resource management: motivating people, retaining people, hiring people and training people.
- Managing the time v. feature trade off.

Neither of these trade-offs is, strictly speaking, a Project Management task. Project management techniques like PRINCE 2 explicitly exclude managing people. While a Project Manager may be able to offer advice on time considerations, the decision on whether to include or sacrifice a feature is a job for someone well versed in business need. This is a job for a Product Manager or a Business Analyst.

Lesson 4: Human Resource Management is not part of Project Management. However, when managing a project many of the issues are people issues.

Time v. features

It's obvious really: the more time a team has, the more work it can do and the more features it can implement. However, the longer a piece of work is scheduled to last the greater the expectations and the greater the risk.

The future is uncertain, and the degree of uncertainty increases in proportion to the length of time considered. Next week is more uncertain than tomorrow, and next year more so. A competitor may launch a product and steal the market, legal changes may limit the product's application – as happened to some online gaming companies – or economic changes may render the software unprofitable.

Neither is it just risk that increases with time: technology advances. New operating systems, new chips, new discoveries may undermine the software under development or require re-work.

A late product, no matter how well engineered it may be, is often worthless

The maintenance phase corollary

Most people who have formally studied software development and engineering will have been taught that 80% of the cost and effort expended on software occurs not in the *development* phase but rather during the later life time of the software, the *maintenance* phase.

But far fewer people appreciate the corollary of this. If this rule holds for all software, it follows that 80% of a developer's career will be spent maintaining existing software, or possibly that 80% of developers will spend their entire career maintaining software.

Given that, it might be reasonable to assume that 80% of the research into software development considers the maintenance phase, or that 80% of the publication relate to maintaining software. Yet neither seems to be the case.

Lesson 5: The further you look ahead the greater the uncertainty.

In order to cope with these difficulties – and others – it is necessary to consider shorter time frames. There is significantly less risk attached to product development which lasts six months than one lasting two years.

Less risk equates to less cost but there is also revenue to consider. A product that ships in six months will start earning revenue for the builder in a quarter of the time it takes the longer project. This means cash will start flowing that much sooner – especially useful for start-up companies.

However, shipping a product in a reduced time frame creates two problems, one technical and one social.

A technical problem

Technically software engineers are taught to, well, engineer. To design systems that are resilient to change and will stand the test of time. To stand like a bridge for a hundred years. But software faces different economics to bridges and buildings.

Unlike most construction projects, most of the costs of software occur after it is initially released – what is euphemistically called the maintenance phase. It is hard to foresee the changes that are required during this phase.

A building may be designed by one individual, or by a small group of individuals. It is then constructed by another, larger, group of people. However, there is little design, innovation or problem solving during this phase. Much of the work is performed to industry standards. Therefore the final structure mostly resembles the original design.

Design, innovation and problem solving occur at every step of software development. Deciding whether to divide a piece of work into several classes each with one function or one class with several functions is a design decision left to individual developers. The scale of the task is such that the designer, or architect, cannot have sight of all the decisions unless they actually perform the work themselves.

Consequently software is the ongoing work of many minds rather than a few. Naturally there will be differences of opinion and approach.

Software development is often opportune: if released at the right time the software can fill a market need and make profit. Releasing the same software later may miss the opportunity. Therefore the pressure to 'get something' delivered is high.

A late product, no matter how well engineered it may be, is often worthless. But a timely product, no matter how bad, may be worth millions. This dilemma creates the conditions for adverse selection. Poorly engineered or designed products may often be better positioned to win. This problem has been called *worse is better* [Gabriel90].

These problems bedevil software developers. Software engineers have yet to find ways of developing software that allow for *good design* without imposing excessive economic costs. Test-driven design, rough up-front design and refactoring are part of that solution but not the entire solution.

Prioritisation

The second problem a reduced timeframe creates is the need to decide which features are included and which are left out. According to our triangle, with fixed resources, if we reduce time we must reduce the feature set.

Unfortunately this requires tough choices. Development projects are often like trains. They don't leave the station very often and when they do you are either on it or you are not. People will pay a lot of money to be on a train, or squeeze themselves into a small space rather than wait for the next one. Worse still, with software projects it is not always clear that there will be another one.

Consequently lots of people want their requests included in a software project. Since including a request is relatively cheap there is little incentive not to include it. Indeed, not including a request risks offending or upsetting someone, therefore there is an understandable momentum for including it.

At some point decisions about which features are included and which are not need to be made. Postponing these decisions is bad for the development team because they have to consider all requests – or at least read the documents – and most likely spend time discussing requirements with stakeholders.

Postponing decisions makes sense not only from a social point of view but also from a business point of view. The option to develop a new feature, or not to develop a feature, is exactly that: an option. Economics, again, shows that options are valuable. (If you want to know the details read up on Real Options which apply ideas from financial options to real life problems.)

What is needed is a clear prioritisation process for the development team. The team need to know what work is required for the next development period and which is not. They should then ignore all other requests, to consider any element would unbalance the economics.

To date software engineering has done developers a disservice by allowing engineering to become top heavy

In order to have clear prioritisation somebody – or some group of people – must be able to make a decision. This individual needs to have all the information necessary to make the decision, they must be trusted by the organization and they must be empowered to make these decisions and make them at the right time.

This role is that of Product Manager. Not all organizations have Product Managers in name – they have different titles, like Business Analyst or Product Owner – but many organizations simply do not have Product Managers at all.

Lesson 6: Product Managers are needed to decide what goes in, and what does not go in, each software release.

In some organizations Project Managers fill this function. The problem here is that Project Managers are trained in a different skill set. They are trained for estimating, project scheduling, risk assessment, issue and progress tracking, reporting and such. They are not trained to gather information from disparate sources and make business value judgements.

Priorities should be communicated to development teams in unambiguous terms. The simplest way to do this is to prioritise requests as 1, 2, 3, and so on where no two items are allowed to have the same priority. So there is only one number one priority, one number two and so on.

Lesson 7: Priorities need to be unambiguously spelt out to teams

Some organizations use the so called ‘MoSCoW’ rules to categorise items as Must Have, Should Have, Could Have and Will Not Have (or Would like to Have). Such prioritisations are an abdication of responsibility on the part of the business. Asking a team to develop five ‘Must have’ features turns over the decision to the development team – when this happens the business loses its right to complain about the result.

Conclusion

The triangle of constraints governs all software development. Add to it Brooks’ Law and all decisions come down to questions of how long a project will take, and which features are included.

To date software engineering has done developers a disservice by allowing engineering to become top heavy. New engineering techniques are needed that can be used in short cycles.

The business side of work also faces a challenge: to straighten out the prioritisation process. There is one ready made answer: to embrace Product Management but unfortunately too few organization are using these techniques. Neither is this any guarantee, product management can be done badly or it can be done well.

And this is just the tip of the iceberg when it comes to managing software development. A future article will discuss the role of Product Management in depth, but before then, the next instalment will discuss quality, time-boxing and focus. ■

Acknowledgements

Thanks to Ric Parkin and the Overload team for comments and suggestions.

References

- [Brooks75] Brooks, F., 1975, *The mythical man month: essays on software engineering*, Addison-Wesley.
- [Brooks95] Brooks, F., 1995, *The mythical man month: essays on software engineering*, Anniversary edition Edition: Addison-Wesley.
- [Commerce05] Commerce, Office of Government, 2005, *Managing Successful Projects with PRINCE2*, Fourth Edition. London: TSO (The Stationary Office), page 7.
- [Gabriel90] Gabriel, R.P., 1990, ‘Worse is Better’ in *EuroPAL*. Cambridge.
- [McCarthy95] McCarthy, J., 1995, *Dynamics of Software Development*, Microsoft Press.