Change

We said in the introductory lecture that systems spend more time being maintained than being built

How do we plan a system to take account of this?

In the next lectures we'll think about "planning for change"

- What sorts of things might change? How can we plan to deal with them? How can we minimise their impact?
- Make software change-resilient and re-usable

Systems again

Not just the software in hand

- Processor and other hardware
- Operating system
- Programming language
- User interface
- Data format

A computer *system* includes all these factors, and more

An important application must be able to accommodate a change in one or more factors

- Portability – move to a new operating system version
- Extensibility – accommodate a new feature
- Re-usability – use in a different set of circumstances

What needs to survive change?

Application

- Maintain a service for users over changes in platform
- Extend to encompass new features
- New user interfaces and interconnections with other tools

Data

- Often forgotten – but data that can't be accessed is dead
- It's often more important for data to survive than applications
- Consider a space mission: all the spacecraft design data from the start needs to be accessible at the end, fifteen years later. The design tools may be long gone by then
- Other examples include: customer information and product stocks

A system meeting these criteria will remain useful longer, and therefore be more valuable to its users

Legacy systems - 1

What operating systems might you reasonably expect to encounter?

- MS-DOS, CP/M, Pick
- Windows 3.1/95/98/2000/NT/XP, OS/2
- Unix – Linux, FreeBSD, SunOS, Solaris, HP-UX, Ultrix, AIX
- VMS, OS/360

What programming languages?

- C, C++, C#, Ada, Pascal, Modula, Java
- Perl, Python, Visual Basic
- Lisp, Prolog, Erlang, Smalltalk, Forth
- Fortran, COBOL

Not to mention the third -party applications…

Legacy systems - 2

An application could encounter *all* of these – and not just shielded by the Internet

- Lots of legacy code still running on mainframes
- Increasing need to interwork between Windows and Unix (Linux)
- Some hardware devices only have DOS drivers
- Even Internet (Java) code may need to access C libraries

You can *never* control the environment of an application

- Close integration kills you
- Code which can cope with lots of environments can be used more widely

"You're using it to do *what*???"

Really useful software is a real pain – people use it in ways you never considered, and expect you to fix it

- The World Wide Web evolved as a tools for co-operative authoring and project planning amongst nuclear physicists
- Java started out as a language for programming cellphones
- Perl evolved as a way of using Usenet news to support a programming team distributed across the US

So re-use occurs where you least expect it

- Never underestimate the ingenuity of a systems programmer with a problem…

Consequences and approaches

Design software to be change-resistant

- Application lifetimes are longer than anything else in the system– except the data
- Minimise dependencies on features which might change
- Avoid making things unnecessarily complicated
- Adopt international standards – and failing that, the *de facto* standards
- Document everything, including failed or rejected alternatives – the rationale for their rejection might go away

"It is impossible to foresee the consequences of being clever – and one should therefore avoid it whenever possible"

Chris Strachey

Hard lessons

Forget performance, program for portability

- Low performance goes away for the most part, but change remains
- Strongly localise any optimisations away from the core functionality

Forget proprietary binary file formats

- Data will be needed long after the original application goes away
- Binary's advantage is compactness is less of an issue than it was

Decouple user interface from functionality

- Interface fashions change much faster than the core application functions

We'll look at data portability more thoroughly later in these lectures

We'll look at designing and coding for portability later in the course

Getting it right - 1

Something written for an IBM System/360 Model 30 in 1964 will run **unchanged** on an IBM System/390 Model 1C5 built in 1998

- Internal hardware and OS architectures may be radically different
- …but the programming abstraction is the same
- Same programming languages and libraries
- …but talking to completely different external products (Oracle rather than CODASYL)
- A stable base from which to extend

Think about how much commitment to users is implied by 34 years' continuity in the face of continuous change

- Portability is at least as much about the quality goals of the supplier as about technical solutions

Getting it right - 2

Re-use, portability and extensibility all start with the domain model

- What might change here? What might be needed? What can I use elsewhere?
- Understand the business and you understand where it might go

Specification and (to a lesser extent) requirements

- Exact details of a process/algorithm can be re-used
- …and will be portable, if written at the problem level

The earlier you start to consider re-use and portability, the more leverage you get

- …so code re-use is the least effective
- …but can be the most general, taking (admittedly constrained) functionality across domains
- A "learning organisation" will re-use models of its customers

Impediments to portability

Incompatibilities

- Operating system features – the lack of a proper shell (or tool set) on Windows 95/98/NT
- Numerical formats – 64- and 128-bit reals
- Compilers – may not be available for your language of choice, may differ in details

Embedded decisions

- The business will never acquire another warehouse
- The user will always use a windowing interface
- The company will never change database/programming language/network/…

Each of these decisions may be justified by the domain model – but probably not

Portability is the ability to recover from an outdated decision

Non-portability as marketing

It can sometimes seem to make sense for a company to "go its own way"

- Gain customer lock-in – once they buy, they can not move
- Kill the competition using "fear, uncertainty and doubt"
- Implement the accepted standard and add "extensions" which kill it
- Implement a different but functionally identical product

Many businesses want multiple suppliers, and will not buy from a company who is the sole source

Case study: Java

Java certainly addresses a lot of small-scale portability concerns

- ✓ "Write one, run anywhere" by providing an interpretive bytecode layer on top of the raw hardware
- ✓ A complete platform – the VM plus the standard libraries – insulating applications from tricky, variable systems-level stuff
- ✓ Few implementation-dependent parts of the language specification
- × Still evolving (and always will be) so things will change even within Java – older browsers can not handle Java 1.1 applets
- × Different capabilities on different device classes – personal Java, embedded Java
- × Does little or nothing to address data portability outside Java – and serialisation does not cope well with class changes

Impediments to re-use

Programmer ego – "I can do better"

Organisational mis-trust – the "not invented here" syndrome, unknown quality

Imagine **having** to re-use code you knew was flaky…

Retrieval – finding the code to re-use, ensuring that code really is re-usable

In software, similarity can be deceptive

Expense – building re-usable code is more expensive (at least 60%)

Costs appear in design, implementation, storage, retrieval, the downstream project, and the process itself

Ownership – who owns the results, who pays, who is liable for failures

See the case studies in chapter 7 of Stephen Schach, *Classical and object-oriented software engineering*, Addison-Wesley (1999)

How to do re-use

Not an add-on – needs to be integrated directly into the software process

- *E.g.* Brooks' "surgical team" with the librarian, spiral model's explicit search for alternatives, …

For every piece of software, think "where else can I use this?"

- Keep the models that gave rise to the code
- Document *all* the decisions and alternatives

Otherwise you've got a snapshot of where you ended up, with no idea how (or why) you went there

- Break things up into small units, and compose them
- Define libraries that do one thing well

Small-scale stuff within the larger context

Whole-system portability

Largely a matter of design, with some sensitivity to the possible business context changes

Application survival can be accomplished by avoiding over-commitment and localising dependencies

- Extra levels of indirection and abstraction

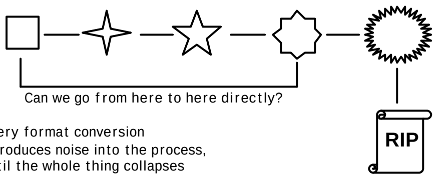
Data survival means making sure any data a user produces remains usable

- Avoid lock-in – destroys some commercial advantages, but only over the short term

Largely a matter of a determination to keep data accessible no matter what

Getting it wrong

- Define a proprietary data format, preferably binary
- Don't tell anyone what it is: just hard-code it into the application
- Let the users store data in it for a few years
- Change the format every time you release a new version, making sure the old format isn't a sub-set of the new one



Every format conversion introduces noise into the process, until the whole thing collapses

Binary formats

Store everything in exactly as many bytes as are needed – and no more

- Minimises the memory/communication footprint – integers are four bytes as binary and ten characters as text
- Less important as time goes on – bandwidth increases, processing power increases
- Make some operations easier – although even binary formats still need translation into in-core representations
- But, extending and re-using existing data can be tricky
- Might be useful to discourage third parties from manipulating data that is explicitly proprietary

Compromises

- Compress a text format using a standard compression algorithm

Plain text

The old favourite

- Lots of Unix file formats remain unchanged (and still readable) after twenty years
- New presentations from old formats – HTML man pages

Critique

- ✓ Easy to define
- ✓ Easy to create and fix – by hand, *in extremis*
- ✓ Easy to process when line-based
- × …but more complex formats can be tricky
- ✓ Portable and easy to evolve
- × Not friendly to non-English languages
- × Lack of structure can lead to messy formats
- × Can be tricky when there are lots of files to relate together

How plain is "plain"?

Even a "plain" text file has *some* structure

- One record per line, comma-delimited, …
- Java is just a plain old text file – but with a complex grammar to be respected for a document (program) to be *well-formed*

Critique

- ✓ Grammar defines a structure, so junk can be rejected quickly
- × Need to parse the grammar from the file into an internal representation for each application (and back again)
- × The tools for this are often language-specific
- ✓ …but the grammar itself is language-neutral
- ✓ Large variation in grammars

Emergence of *structured mark-up* reduces the number of grammars – sort of…

Structured mark-up

A text file whose structure follows some structure

- Designer isn't free to adopt arbitrary text forms, but follows some (loose) rules

Critique

- ✓ Easy to explain and write
- Something can be syntactically correct and not be a valid document
- Still need to parse the grammar from the file into an internal representation for each application (and back again)
- ✓ Single structure makes grammar descriptions more portable – "one tool parses all"
- ✗ May be harder to parse than a dedicated format
- ✓ Fairly easy to lose the mark-up and retrieve the data
- ✗ Verbose

HTML

The canonical example of structured mark-up

- Add structure and layout information
- Still text, so low authoring overhead

```
<HTML>
<HEAD>
<TITLE>Simple page</TITLE>
</HEAD>
<BODY>
<H1>Main page</H1>
<P>Some text and an image
<IMG SRC="hhh.gif">
</BODY>
</HTML>
```

An element of the document between corresponding start and end tags

Not all elements have end tags

Elements can bring in things not possible with plain text

Critique of HTML

Something that popular can't be *all* bad…

Grew rapidly and acquired features with no clear design

- Some tags are structural -- `<H1>`, `<EM>`
- Some are presentational -- `<IMG>`, `<I>`, `<CENTER>`
- Some are procedural -- `<APPLET>`

HTML documents will remain readable

- Strip the tags to get a plain text document – lose some meaning, but at least there's a chance…

Will become a legacy format

eXtensible Mark-up Language

The ISO-standard SGML without the silly bits

Defines a standard way of defining a document's structure

- A mixture of text and control elements
- Document Type Definition - what elements are allowed, how they can be nested, what can and must appear

Becoming extremely important

- An Internet/WWW standard
- Adopted by just about anyone wherever information is to be shared between applications

W3's site is a good source for the other XML standards mentioned

http://www.w3.org/TR/REC-xml

General form

Looks extremely like HTML

- Ordinary text with mark-up in angle brackets

```
<?xml version="1.0"?>
<order>
<customer>1234</customer>
<product quantity=100>5678</product>
</order>
```

This identifies the document as XML

Unlike HTML, XML is case-sensitive – which is going to cause some real problems…

Differences

- No pre-defined elements – no built-in capabilities, everything has to be defined and implemented by applications
- Strict rules on validity and well-formedness – applications can afford to be less tolerant

Openness and its consequences

XML has *no* in-built semantics or capabilities

- Everything to be represented has to be defined externally
- …and the definition has to be agreed by all parties who want to share the data

No silver bullet

- This is a data format like any other – it has *no* intrinsic advantages over any other text format, except for wide adaptation

Chicken-and-egg adoption cycle

- Not useful until widely adopted – and why should you adopt it?

Adoption brings usefulness

- Agreed XML formats for commonly shared data
- …and these are by definition shared across the community, even if they're not "open" in terms of participation in their definition
- Re-usable elements (namespaces) for common capabilities

Using XML

How to reduce your workload…

Is there a format already defined?

- May be possible to re-use it or extend it

Are any desirable capabilities already defined?

- Import them as namespaces to minimise work and maximise the regularity of DTDs

Use a standard parser to read/write documents

- Could use tree and DOM as the "real" in-core representation

Summary

Portability is a whole-system issue

- Make the applications live long, but make the data live longer

Code portability

- Localise dependencies into well-known modules

Data portability

- Avoid proprietary binary formats like the plague – valuable data will *always* live linger than any application
- Structured text formats like XML seem to be the way to go – built with longevity and modularity in mind