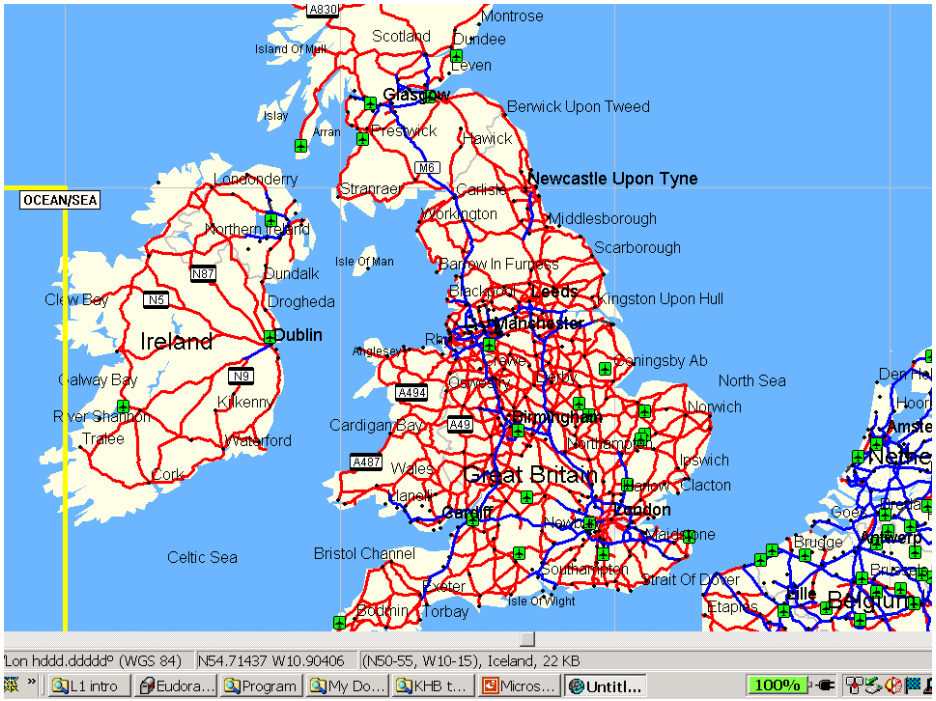
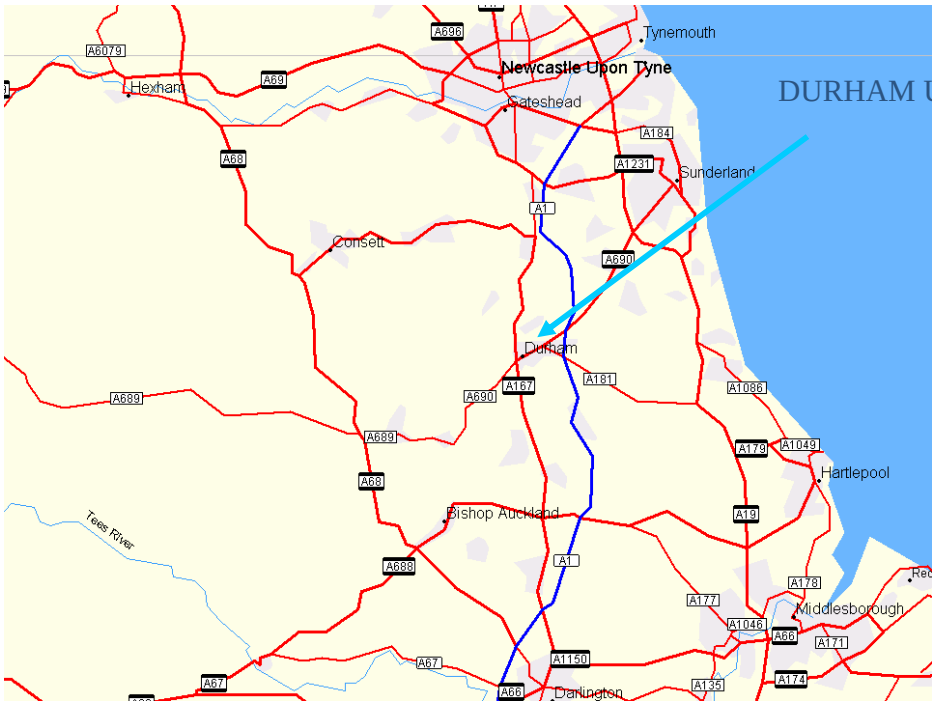
Software Evolution and the Future for Flexible Software

Keith Bennett
University of Durham

Pennine Research Group
(UMIST, Keele, Leeds, Durham)





www.service-oriented.com

keith.bennett@durham.ac.uk

Pennine = software engineering researchers from UMIST, Keele, Leeds and Durham.

Contents

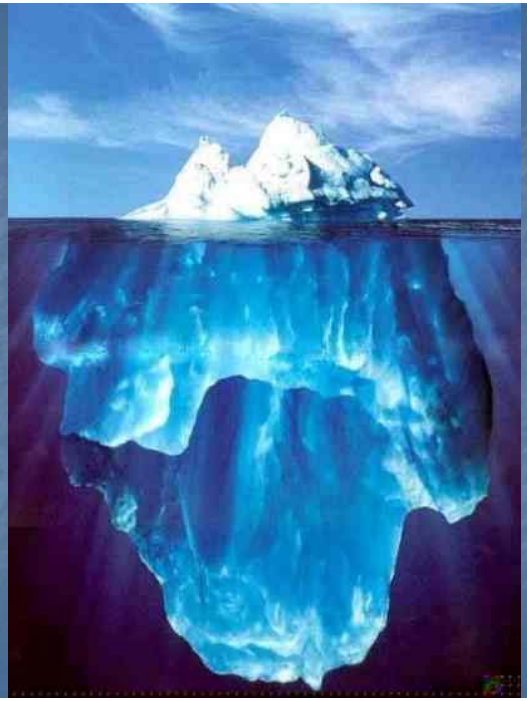
- 1. Current evolution SOTA
- 2. Meeting the challenge of building software systems which are easy to change.
- (the IBHIS project represents our experimental system)

What is evolution?

- It is activities performed on software after first delivery (IEEE): *The modification of a software product **after delivery** to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment.*
- (Old fashioned – reuse, components etc)
- Evolution and maintenance:
  - *maintenance* means general post-delivery activities
  - *evolution* to refer to a particular phase in the *staged model* where substantial changes are made to the software

Empirical data of software maintenance

- Software maintenance represents 67- 80 % of software costs
- Survey by Lientz and Swanson
  - late 1970s, very widely cited
  - maintenance activities divided into four classes:
    - Adaptive – changes in the software environment
    - Perfective – new user requirements
    - Corrective – fixing errors (21% of all changes)
    - Preventive – prevent problems in the future.
  - incorporation of *new user requirements* is the **core problem** for software evolution and maintenance (79% of all changes)



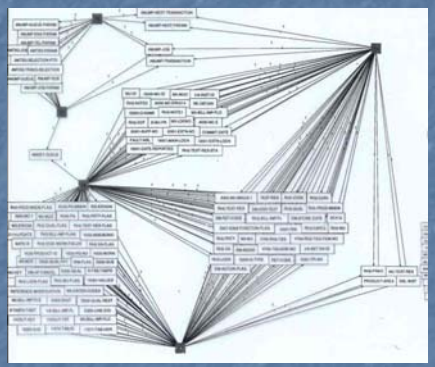
Challenge of software maintenance / evolution

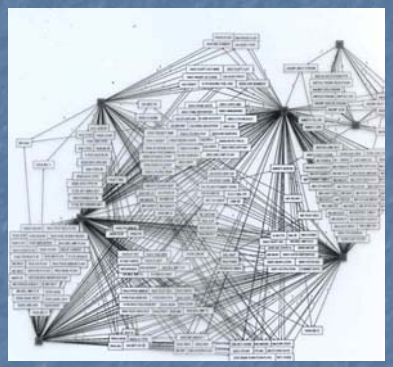
- incorporation of new user requirements quickly and reliably
- **If** changes can be anticipated at design time
  - they can be built in by a parameterization, encapsulations, etc.
  - the problem solved
- **However** 40 years of hard experience confirms:
  - many changes cannot be even *conceived* of by the original designers
  - inability to change software quickly and reliably means that business opportunities are lost

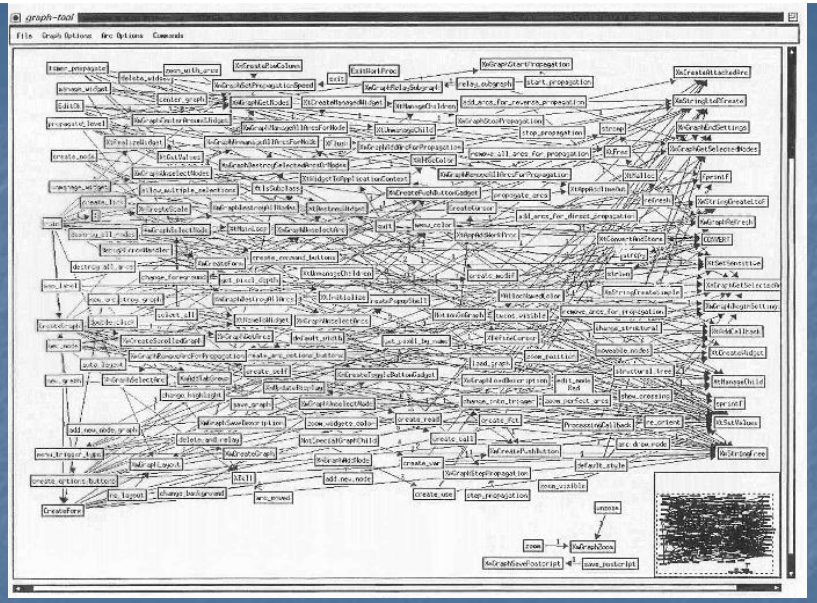
Graphs

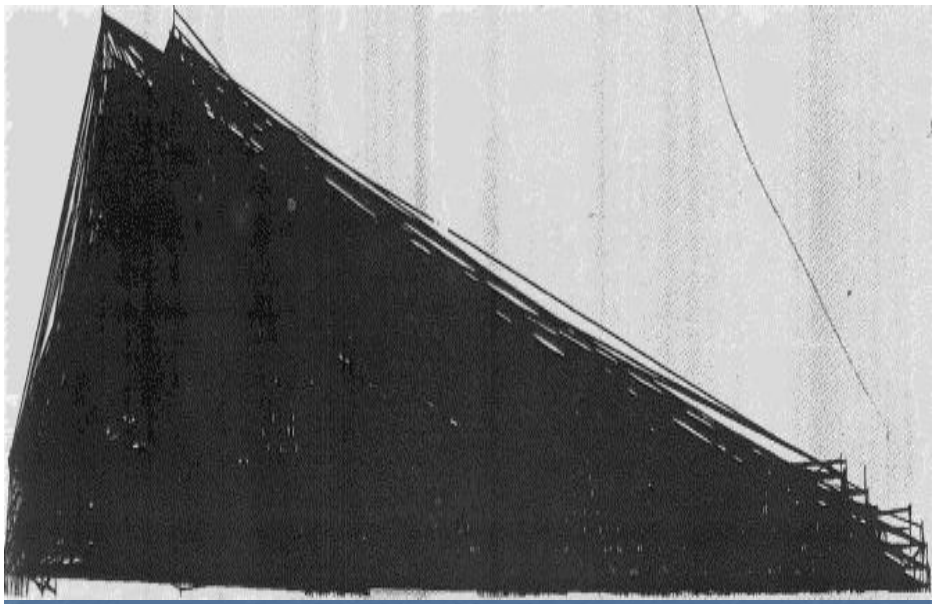
- Liz Burd
- www.dur.ac.uk/liz.burd/evolution.ppt
- The charts are copyright of Liz Burd.

Increase in interface complexity









Harrier examples

Reverse engineering

- Technical problems: equivalence of old and new (do we want exact equivalence, or modified requirements) e.g. CICS
- Business: we need to invest large amounts of capital to achieve unquantifiable gains e.g. in maintainability (!!) and new staffing problems
- Almost no interest from outsourcers

Delocalization of change

- The architecture does not support contemplated change, because the concepts of the application domain relevant to the change are delocalized in the code eg Harrier
- the solution may be to restructure first and to localize the concept in one location, and then to change it
- behavior preserving transformations do not change the behavior of the program, but change the architecture.
- In the case of delocalized changes, an advisable strategy could be:
  - to transform the architecture so that the change will be localized
  - then to make the change itself

Some fundamental issues

- Maintainability eg extreme prog (process) or or OO (product)
- CS thinks they have solved it. (thierry) no scientific evidence
- (Lehman, feedback systems)
- Modern software not green fields – reuse, components, integration
- Too slow a process
- We know very little about evolution, no theory

Issue: Maintainability

- Software will change in ways inconceivable to original designers (VME)
- We do not really understand this, and know how to quantify it
- It may not be a technical matter alone
- "Research by advocacy" and "leave to the reader" a real problem in CS research (example 2 weeks ago)
- One poor evolution activity can ruin it

Issue: Emergent systems

- Driven by business need, so "co-evolution" is a good term.
- No initial spec, design, code, test, maintain
- More like (for user, for developer) join, participate, disengage.
- Lehman E type software

Emergent organizations

- time-to-market for software has become the top priority for many business applications
  - A finance house may create a new financial product; it must be implemented and launched within 24 hours; and then has a life of only two more days.
- A group of senior software engineering academics and industrialists in UK met regularly to explore and frame visions of the future of software
  - level of abstraction of software engineering will continue to rise.
  - the focus of research will change from technology to the interface of the software with business
  - software will move from product oriented view to service oriented view

Issue: observations

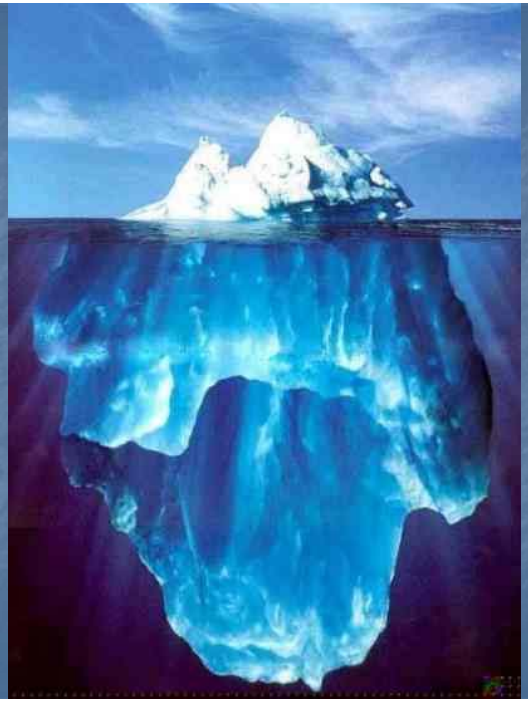
- Only one source of info – the source code cf assertions
- Documentation often write once only
- Maintainers will often clone and modify
- (e.g. ERNIE, Harrier)
- Evolution while system is running (modern distributed systems)
- Testing needs evolving too.

Issue: understanding evolution

- Lehman has derived 8 laws of evolution (example law 1). Done by observing an existing process and measuring
- Now trying to model by a feedback system
- Maybe a *pro-active* process model (maintenance which requires degradation of structure to be fixed) will produce other results.

Grand Challenge

- Right first time, every time.
- The problem with evolution is not that it is known to be a problem, but what the solution space looks like



Flexible software

Is there a solution?

Problem

What is the real problem?

It isn't simply a new way of representing requirements and then doing a delta on this

So let's look at the problem in more detail

Challenge

- Software needs to evolve, often rapidly, to meet and reflect business needs. In almost all cases the evolution is not predictable at design time.
- A system typically consists of various components of various ages, technologies etc.
- Example: health services, Ernie, Harrier

Problem

- How do we build and sustain software which is highly maintainable and flexible?
- And which can be revalidated/reverified easily?
- And properties like FT, dependability (and non-functional props) not damaged?

It seems important..

- Is software A more maintainable than software B?
- If we change software A, does it become less maintainable?
- We'd like to measure it

Metrics would help with lifecycle cost estimation

- Vendor A produces software which is very inexpensive to buy, yet is so unmaintainable that only the vendor can change it
- Vendor B produces expensive software which is easy for all to maintain

Emergence

- A hard challenge is raised by “emergent” applications and organisations
- Systems Domain
  - Well defined boundaries and requirements
- Business Domain
  - Emergent Organisations
    - “Organisations in a state of continual process change, never arriving, always in transition”

D. Truex, R.Baskeville and H.Klein, “Growing Systems in Emergent Organizations”, Comm.ACM, Vol.42, No.8, August 1999

System and emergence

- System
  - Complexity in algorithms and real time
  - Boundaries of concern tend to be fixed
  - Careful cautious process of evolution eg CICS
- Emergent
  - Change is continuous, not discrete
  - Very rapid rate of change, set by business
  - Eg e-businesses (NHS??!!)

Changing Nature of Business

- 40% of Fortune 500 companies in 1979 are no longer corporate entities
- 30% of firms under 10 employees generate 70% of EU turnover
- Competitiveness through time to market is major driver

Answer

Analogy

- Cities and buildings evolve in a way that is inconceivable to their original inhabitants e.g. Romans
- Is Durham “more evolvable” than Bari?
- Cities tend to evolve incrementally, reacting to short term goals and those in power, and making use of existing structures.
- Building Pompeii alongside Vesuvius less obvious after the eruption of AD79

Observations

- Mostly we have “research by advocacy”, which assumes all but one factor (typically requirements) remains constant, e.g. OO, extreme programming, WITHOUT evidence
- Software evolves – so does theory, tools, process, technology, environments, people, expectations, legal framework, trust….
- [How buildings learn – S Brand ISBN0753800500]

Typical problem

- A simple highly localised requirement change requires a substantial and highly distributed set of changes throughout the software (e.g. Harrier)

Industry

- Industry seems unwilling to pay for benefits which are not quantifiable (maintainability, retest costs): high cost for intangible benefit.
- Reverse engineering, other than some sorts of code restructuring, seems problematic.

Conclusions

- We don't understand flexible software
- We cannot measure it;
- Pragmatically, addressed by employing excellent (but low status) staff

Pause

- We understand the problem domain
- But what is the solution?
- A Grand Challenge: "Right first time every time"

Pennine approach

- We started with a user study in cognate disciplines (ref. CACM paper)
- Users do not like evolution (though it is no use having last year's tax tables)
- Fundamentally they do not like *cost of ownership*. They want to have the benefit of *using* software without its cost.

P.Brereton, D.Budgen, K.Bennett, M.Munro, P.Layzell, L.Macaulay, D.Griffiths and C.Stannett, "The Future of Software: Defining the Research Agenda", Comm. ACM, Vol.42, No.12, December 1999

Basic idea

- We have a marketplace of software services
- When we need a better/faster/cheaper service, we substitute the old service by a new one then and there (on the fly)
- No magic solutions (self evolving, autonomic etc.)

A service

- A business definition: something that is used, not owned (cf web services)

Idea for evolution

- Get the granularity of change right (currently tied up with MS marketing strategy)
- Users use services. They compose what they want to do out of current services at the instant of need!
- Then execute the composition
- Then (like a rental car) disengage.

Demand led

- Consider e.g. WORD. Constantly changing, in ways not required by most users
- Users become very hostile
- Of course, this is MS business model – updating to sustain revenue
- This is supply side, or vendor led change

Service**

- The user employs the best/most recent/cheapest/most dependable services
- When use complete, then disengage

Software as a Service

- A user (human or program) does not own software – they rent it
- At the point of need they compose the system they need to undertake a task (ultra late binding)
- They execute this
- And then disengage

Hypothesis

- In SaaS software functionality is delivered as a service
- Each time functionality is required, service elements are identified, terms and conditions are negotiated, executed and then “disengaged” – so there is no further obligation to that particular solution

How might evolution be addressed?

- The available services will evolve to meet user needs. If service providers evolve services to meet needs, they are rewarded; if not they are punished!
- That’s the property of a marketplace
- We are exploring this in the health service domain (IBHIS)
- Cf evolution of a city

Example: payroll**

- This week: paper payslips
- Next week: uses an email service

Is this a commodity market?

- No, not necessarily. Many CS people think of it like this, and see the matching problem as core. Essentially a pay-per-view model. See public UDDI sites for problems with WSDL and http://www.uddicentral.com/
- In business there are many other models eg Kaizen, preferred supplier.

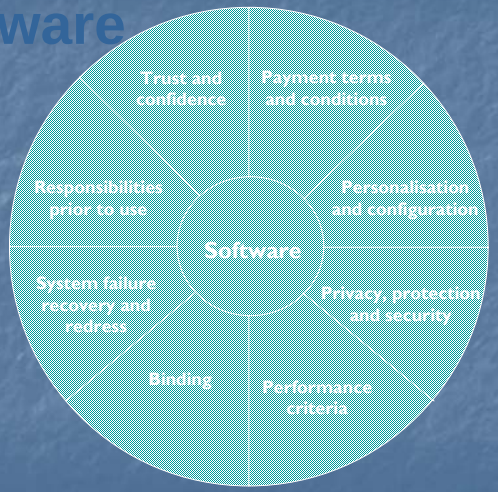
Is the composition hard?

- Compositions may be sold as services
- We don't see this otherwise as automatic (ie magic)
- A rather open problem for us is the process (links to Brian Warboys' group)

What are the hard problems?

- We assume that we have a reliable distributed infrastructure, across heterogeneous autonomous organisations, such as that provided by WS or Grid
- The hard issues then address business related problems like terms and conditions, negotiation protocols, legal frameworks, trust, dependability
- And, in particular, security
- Ultra late binding of T & C is interesting.

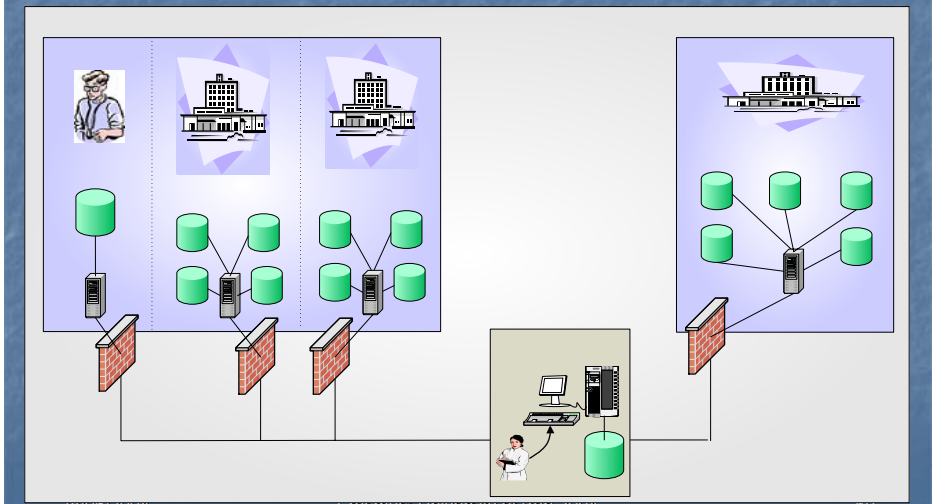
Serviceware



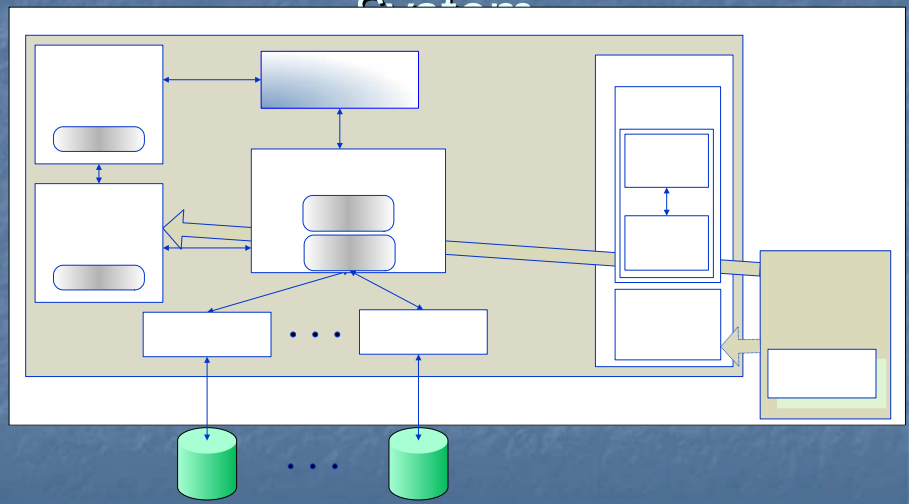
Summary

- Our proposal sees evolution only partly as a technical issue (ultra late binding)
- The rest is based on “socio-technical issues” aka people and markets
- We seek an evidence-based evaluation
- We are developing formal models

IBHIS Overview



Architecture: Operational System



Access

Learning about web services

- Version 1: static architecture, no UDDI, setup by administrator, completed december 2003
- Version 2: dynamic binding, using WSDL and UDDI

Web services 1

- We used RPC, but too static and unscaleable (too many proxies). Needs document style
- Websphere and its wizards worked well
- UDDI shows many limitations for describing resources, finding, binding.

Web services 2

- In V1 we used a global schema as our ontology. We need domain ontologies (eg MEDINFO)
- WSDL has very limited support for terms and conditions, for negotiation
- Security, privacy remain problematic. We have used RBAC
- Performance - needs detailed study

Conclusions

- We have presented a demand led possible solution to software evolution
- Functionally, the basic hypothesis seems to work (version 1)
- In version 2, more realistic experiments of evolution (much more dynamic testbed) will be undertaken
- Our work uses ws as infrastructure

end

Steve Schach's results

- At Vanderbilt University, Texas
- srs@vuse.vanderbilt.edu
- Three unexpected results with open source software

1. Who finds errors?

- Open source software tends to have many peripheral readers (find fixes, suggest repairs) and a few core people who can modify the software
- The tenet of the open source movement is: many os products are so successful because vast numbers of people on the periphery study the code, find problems, suggest fixes.
- "Given enough eyeballs, all bugs are shallow"

Results 1

- Tomcat Gnome and mozilla

	Core	Periphery
Tomcat 4	71.7	28.3
Gnome	81.1	18.9
Mozilla	89.6	10.4

Conclusions 1

- Most bugs are found and fixed by core members
- The large number of peripheral eyeballs have a small effect
- Steve's current work is suggesting that most enhancements come from the core team too.

Problem 2 Coupling

- Schach and his group studied coupling through global variables in 365 versions of LINUX (3M LOC each)

Results 2

- The number of couplings through globals rises exponentially with version number.
- The LOC grows linearly with version number.

Conclusions 2

- These would indicate that LINUX is becoming unmaintainable
- (global variable instances in kernel = 1022; outside kernel = 14688)

Problem 3 Lientz and Swanson

- Summary of results
- Questionable methods (survey based, not actual metrics, in CMM level 1 organisations)

Result 3

- For open source systems studied (GCC, LINUX kernel, a RT system), >40% is corrective maintenance.

Conclusions 3

- Schach argues that Lintz & Swanson are wrong for open source
- KHB: I'd ask if this were not what Lehamn has been saying for 25 years. The open source software is all S-type. L & S addressed applications which are basically E-type.

Overall

- Schach's fascinating results show the value of empirical work
- Open source is worthy of more investigation

The lifecycle

- Evolution costs for successful software
- What are the activities/types?
- Program comprehension
- Re-validation e.g. Harrier

What do we know?

- Types of software: Lehman's E, S and P types.
- Lehman's 8 laws e.g. Structural decay

Framework

- Rajlich and Bennett framework
- Bennett K. H. and Rajlich V. T. *A staged mode for the software lifecycle*. IEEE Computer, vol. 33, no. 7, pp. 66 –71, July 2000, ISSN 0018-9162.
- Analysis of empirical results

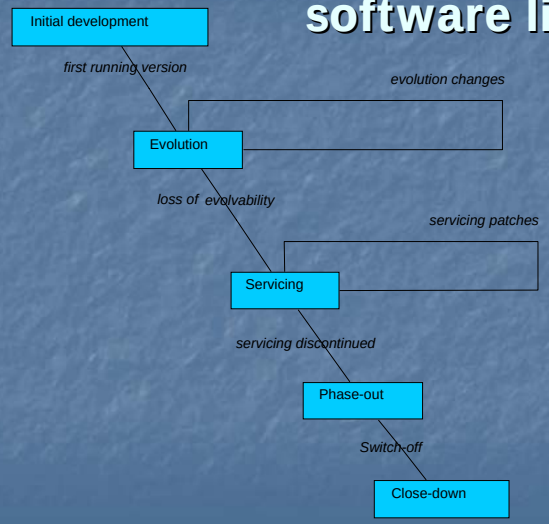
Evolution

- NOT simply "initial development" and then "maintenance"

Additional empirical data

- Cusumano and Selby reported that requirements during each iteration may change by 30% or more, as a direct result of the team learning process during the iteration
- Lehner described yearly variations in the frequency of the changes of a long lived system
  - frequency peaks, then declines
  - identified different phases
- Pigoski: similar data
- facts:
  - inability to predict changes
  - several different stages, not a uniform “maintenance”

Staged model of software lifecycle



Initial development*

- first version of the software system is developed
  - may be lacking some features
  - already possesses the *architecture* that will persist thought the rest of the life of the program
    - in one documented instance, we studied a program that underwent substantial changes during its 20 years of existence, but it still possesses the architecture of the original first version.
- programming team acquires the *knowledge* of
  - application domain, user requirements, role of the application in the business process, solutions and algorithms, data formats, strengths and weaknesses of the program architecture, operating environment, etc.
  - crucial prerequisite for the subsequent phase of *evolution*.

Research challenge*

- To build evolvable software
- assist the next (evolution) stage, not subsequent stages
- ‘design for change’ should be predominantly aimed at strategic evolution, not code level servicing
- in the evolvable architecture, ‘the cost of making the change is proportional to the size of the change, not to the size of the overall software system’
- evolvable software can handle unanticipated changes

Evolution*

- goals
  - to adapt the application to the ever-changing user and operating environment
  - to correct the faults in the application
  - to respond to both developer and user learning
- inevitability of evolution [Lehman]
- business setting of evolution
  - user demand is strong
  - the organization is supportive
  - return on investment is excellent
  - both software architecture and software team knowledge make evolution possible

Code decay*

- There is a positive feedback between the loss of software architecture coherence, and the loss of the software knowledge
  - less coherent architecture requires more extensive knowledge in order to evolve it
  - if the knowledge necessary for evolution is lost, the changes in the software will lead to a faster deterioration of the architecture
- Example of loss of knowledge:
  - loss of key personnel
- Research challenge: eliminate or slow code decay

Servicing*

- the program is no longer evolvable
- changes are limited to patches and wrappers
  - they are less costly
  - they further deteriorate the architecture.
- Senior designers and architects do not need to be available
- Tools and processes are very different from evolution
- A typical engineer will be assigned only part of the software to support
  - will have partial knowledge of the system.
- The process is stable, well understood and mature.
  - it is well suited to process measurement and improvement

Research issues in servicing*

- Making the change without unexpected additional effects
- Program comprehension
- Impact analysis and ripple effect management.
- Regression testing
- Concept identification, location and representation.
- Automated tool for code improvement
- Documentation management
- Delivery of service patches
  - Upgrading software without the need to halt it.
- Program health checkers

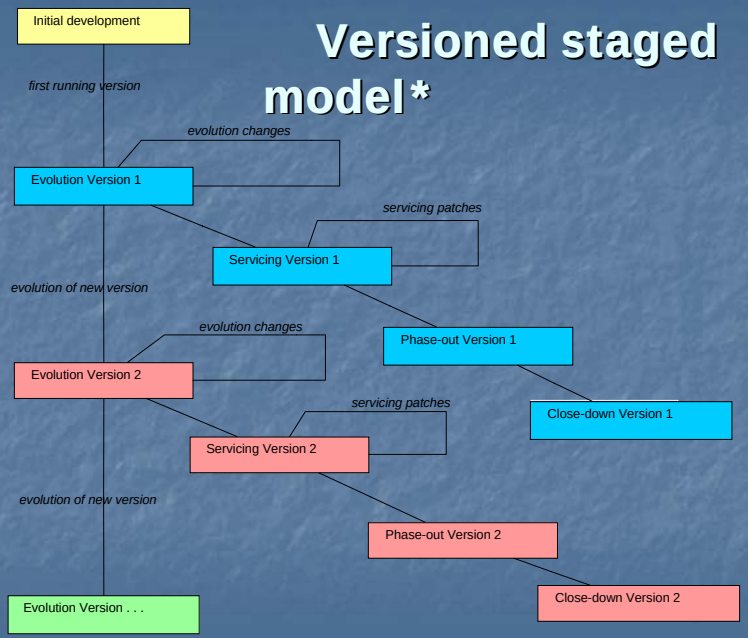
Reversal from servicing to evolution*

- worthy research goal
- in practice:
  - very hard, very rare
- not simply a *technical* problem
  - the *knowledge* of the software team must also be addressed
- for all practical reasons, the transition from evolution to servicing is irreversible

Phase-out and close down stages*

- phase-out
  - no more servicing is being undertaken, but the system still may be in production
  - the users must work around known deficiencies
- close-down
  - the software use is disconnected
  - the users are directed towards a replacement.
- business issues:
  - Can any of the software be re-used?
  - 'exit strategy' is needed.
    - once an organization commits to a system, changing to another is expensive, technically difficult, and time consuming.
    - Do data have to be preserved?

Versioned staged model*



Software change*

- basic operation of both software evolution and software servicing
- change mini-cycle consists of the following phases:

  Request for change
  Planning phase
  Program comprehension
  Change impact analysis
  Change implementation
  Restructuring for change
  Change propagation
  Verification and validation
  Re-documentation

Software change*

- Program comprehension is a prerequisite of any change
  - it has been reported that this phase consumes more than half of all maintenance time
- Change impact analysis assesses the extent of the change, i.e. the components that will be impacted by the change
  - it indicates how costly the change is going to be
- Change propagation
  - change may consist of several steps, each visiting one specific software component
  - if the visited component is modified, it may no longer fit with the rest
  - neighboring components may need to be changed

Delocalization of change*

- The architecture does not support contemplated change, because the concepts of the application domain relevant to the change are delocalized in the code
- the solution is to restructure first and to localize the concept in one location, and then to change it
- behavior preserving transformations do not change the behavior of the program, but change the architecture.
- In the case of delocalized changes, an advisable strategy is:
  - to transform the architecture so that the change will be localized
  - then to make the change itself

Redocumentation*

- change is not complete without the update of the program documentation
- if the documentation of the program is missing or incomplete, the end of the mini-cycle is the opportunity to record the comprehension acquired during the change
- program comprehension is a very valuable commodity (more than 50% of resources of software maintenance)
- in current practice, that value is thrown away when the programmer completes the change and turns his/her attention to new things
- in order to avoid that loss, incremental and opportunistic re-documentation effort is called for. After a time, substantial documentation can be accumulated

Summary

- Underpinning the field are two views:
- *Reactive*: we cannot predict new requirements, new technology, new markets, new processes, so evolution must always react to events
- *Proactive*: Evolution can be planned in advance Processes, tools, products)

Solutions to evolution

- Heavyweight processes e.g. CICS and avoid ad-hoc
- Good staff
- Heavy stress on revalidation
- Wrap legacy components
- Then try to improve each component of such heavyweight processes.

Reverse engineering

- Redesigning all or part of a software system to improve its quality (*Chikovsky*)
- Code improvement (within one stage of the staged lifecycle) may be viable

Reversal from servicing to evolution

- worthy research goal
- in practice:
  - very hard, very rare
- not simply a *technical* problem
  - the *knowledge* of the software team must also be addressed
- for all practical reasons, the transition from evolution to servicing is irreversible