Software Maintenance and Evolution

Massimo Felici

Room 1402, JCMB, KB

0131 650 5899

mfelici@inf.ed.ac.uk

Software Maintenance (vs. Evolution)

- **Maintenance to repair software faults:**
  - Coding errors are usually relatively cheap to correct
  - Design errors are more expensive as they may involve rewriting several program components
  - Requirements errors are the most expensive to repair because of the extensive system redesign that may be necessary
- **Maintenance to adapt the software to a different operating environment:**
  - This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating systems or other support software changes
  - The application system must be modified to adapt it to cope with these environmental changes
- **Maintenance to add to or modify the system's functionality:**
  - This type of maintenance is necessary when the system requirements changes in response to organizational or business change
  - The scale of the changes required to the software is often much grater that the other types of maintenance
- Software Maintenance differs from Software Evolution

Software Maintenance

- Types of **Maintenance**:
  - **Corrective**:
    - correcting faults in system behaviour
    - caused by errors in coding, design or requirements
  - **Adaptive**:
    - due to changes in operating environment
    - e.g., different hardware or Operating System
  - **Perfective**:
    - due to changes in requirements
    - Often triggered by organizational, business or user learning
  - **Preventive**:
    - e.g., dealing with legacy systems
- **Software re-engineering** is an approach to dealing with legacy systems through re-implementation

Development vs. Maintenance

- **Team Stability**
  - After a system has been delivered, it is normal for the development team to be broken up and people work on new projects
  - The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions
  - A lot of the effort during maintenance process is taken up with understanding the existing system before implementing changes to it
- **Contractual Responsibility**
  - The contract to maintain the system is usually separate from the system development contract
  - The maintenance contract may be given to a different company rather than the original system developer
  - No incentive to write easy maintainable software
  - Reducing development cost may increase maintenance cost
- **Staff Skills**
  - Maintenance staff are often relatively inexperienced and unfamiliar with the application domain
  - Maintenance has a poor image among software engineering
- **Program Age and Structure**
  - As programs age, their structure tends to be degraded by change, so they become harder to understand and modify

Some Maintenance Statistics

- **Maintenance** consumes **40%-80%** of total costs
- Typical developer's activity (from Lients and Swanston's review of 487 companies):
  - 48% Maintenance
  - 46.1% New Development
  - 5.9% other
- Huge quantities of legacy code:
  - US/DoD maintains more than 1.4 billion LOC (Line Of Code) for non-combat information systems, over more that 1700 data centres. Estimated to cost $9 billion per annum.
  - (in 1999) Boeing payroll system: approx 22 years old; 650K LOC Cobol.
  - Bell Northern Research's entire operation is maintenance of one system – telephone switching product line. 12 million LOC (assembly and "higher-level" languages), approximately 1 million LOC revised annually.

Distribution of Maintenance Effort

- Corrective (approx. 21%)
  - 12.4% emergency debugging
  - 9.3% routine debugging
- Adaptive (approx. 25%)
  - 17.3% data environment adaptation
  - 6.2% changes to hardware or operating system
- Perfective (approx. 50%)
  - 41.8% enhancements for users
  - 5.5% improve documentation
  - 3.4% other
- Preventive (approx. 4%)
  - 4.0% improve code efficiency

Maintenance is Hard

- Key **design concept** not captured
- Systems not robust under **change**
- Poor **documentation**
  - of code
  - of design process and rationale
  - of system's evolution
- "stupid" code **features** may not be so stupid
  - Work-arounds of artificial constraints may no longer be documented (e.g., Operating System bugs, undocumented features, memory limits, etc.)
- Poor management **attitudes** (**culture**)
  - Maintenance not "sexy"
  - It is just "patchy code"
  - Easier/less important than design (does not need similar level of support – tools, modelling, documentation, management, etc.)

Managing Maintenance

- **Corrective**:
  - Requires maintenance **strategy** preferably negotiated contract between supplier and customer(s)
  - Policies for reporting and fixing errors; auditing of process
- **Perfective**:
  - Should be treated as **development** (i.e., requirements, specification, design, testing, etc.)
  - **Iterative** (or evolutionary) development approach best suited
  - **Risks**: drift, shift, creep, ooze, bloat, etc.
  - When does design or development stop?
- **Adaptive** and **Preventive**:
  - Can anticipate, schedule, monitor and manage, etc.

Maintenance Management Case Study [1/3]

- Spring Mills Inc.: early 1970's
  - Programming shop runs 24 hours a day, 6 days a week
  - 3000+ programs in production
  - Approx. 700 new programs per year
- 1972, John Mooney assessed operation as:
  - Overworked programmers operating under stress
  - New systems typically over budget and late
  - No designated maintenance staff
  - Approx. 75 maintenance requests per week
  - Non maintenance strategy or planning
  - Developers time: 30% maintenance; 45% new development; 10% special; 14% admin

Maintenance Management Case Study [2/3]

- 1973, Mooney reorganizes shop and creates maintenance team
- Management strategy: requests logged, classified, evaluated, prioritised and assigned
- Team responsibilities: fast; good programming standards; regression testing of modified programs
  - Numerous incentives, including financial
  - Team responsible for all existing programs
  - New programs "signed over" to team when error- and change-free for 90 days - Sign-over activity becomes significant project landmark

Maintenance Management Case Study [3/3]

- Outcomes:
  - Maintenance team becomes "highly skilled, elite corps of multi-lingual experts"
  - Deep understanding of company's systems – particularly troublesome dependencies
  - Offer services as "system auditors" or "consultants" on difficult problems
  - De facto quality assurance stakeholders
- Leads to overall development time:
  - 20% Maintenance; 57.9 new development; 21.3% special and admin
- Previously, developers time:
  - 30% Maintenance; 45% new development; 24% special and admin
- Everybody happy…

Preventive Maintenance

- Accounts 4% of maintenance requests
  - **Pareto Principle** applies: 20% of causes responsible for 80% of effect. Proposed by Dr. Jodeph Juran (of Total Management fame), after Wilfredo Pareto – C19th economist and sociologist.
  - Legacy systems increasing problem
- **Software Migration** approaches:
  - **Redevelopment**: rebuilt system from scratch. Easier problem (initially) but costly and very high risk
  - **Transformation**: to (typically) new language/paradigm
    - **Restructuring**: e.g., refactoring
    - **Re-engineering** typically reverse-engineering followed by forward-engineering
    - **Design** recapture recreates design abstractions from code, documentation, personal experience, general problem and domain knowledge
  - **Encapsulation**: "Software Wrapping" – wrap up existing code as components

Software Wrapping Case Study [1/3]

- Sparkasse: German savings and loan organization
- 7 regional computing centres; client-server batch processing on conventional mainframe system; code (variously) in Assembrer, PL/1, Cobol and natural
- Legacy host systems highly integrated
- Desired to introduce OO and components
- Wrapping approach taken
  - Reuse S/W by encapsulating and controlling access via API's (Application Program Interfaces)
  - Reuse existing S/W without moving it to new environment
  - Legacy S/W remains, with minor changes. In native environment – yet is accessible to newer distributed OO components

Software Wrapping Case Study [2/3]

- 1997: Wrapping pilot-project undertaken
- 5 encapsulated levels
  - Job: remotely invoked batch-type job control procedures
  - Transaction: client-server transactions
  - Program: remotely invoked batch program
  - Module: native code modules (easiest to wrap – already "component-ish")
  - Procedure: individual procedure within legacy code (hardest to wrap)

Software Wrapping Case Study [3/3]

- Adaption of all subprograms necessary
- Server to host communication weakest link
- Character conversion, ASCII to EBCDIC, common
- Constant translation and re-translation
- Testing time-consuming due to high number of dependencies
- 5-step, bottom-up testing strategy
  1. Test adapted program in controlled test-harness
  2. Test wrapper software with driver for client and stub for wrapper code
  3. Test wrapper and wrapped code
  4. Integration testing: complete client-server transaction
  5. System test: multiple translations to test re-entrancy of wrapper and wrapped code

Maintenance Prediction

- **Maintenance Prediction**
  - Whether a system change should be accepted depends, to some extent, on the maintainability of the system components affected by that change
  - Implementing system changes tends to degrade the system structure and hence reduce its maintainability
  - Maintenance costs depend on the number of changes, and the cost of change implementation depend on the maintainability of the system components
- **Predicting Changes**
  - Evaluation of the relationship between a system and its environment
  - The number and complexity of system interfaces
  - The number of inherently volatile system requirements
  - The business processes in which the system is sued
- **Measuring Maintainability**
  - Number of requests for corrective maintenance
  - Average time required for impact analysis
  - Average time taken to implement a change request
  - Number of outstanding change requests

System re-engineering

- Re-engineering a software system has two advantages over more radical approaches to systems evolution
  - Reduced risk
  - Reduced cost
- A re-engineering process may involve
  - Source code translation
  - Reverse engineering
  - Program structure improvement
  - Program modularisation
  - Data re-engineering
- Factors affecting **re-engineering costs**
  - The quality of the software to be re-engineered
  - The tool support available for re-engineering
  - The extent of data conversion required
  - The availability of expert staff

Legacy System Evolution

- Four strategic options
  1. Scarp the system completely
  2. Leave the system unchanged and continue with regular maintenance
  3. Re-engineer the system to improve its maintainability
  4. Replace all or part of the system with a new system
- Legacy System Assessment
  - Low quality, low business value
  - Low quality, high business value
  - High quality, low business value
  - High quality, high business value
- Assessing the business value of the system
  - The use of the systems
  - The business processes that are supported
  - The system dependability
  - The system outputs

Environmental Assessment

- Supplier stability: Is the supplier is still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
- Failure rate: Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
- Age: How old is the hardware and software?
- Performance: Is the performance of the system adequate? Do performance problems have a significant effect on system users?
- Support requirements: What local support is required by the hardware and software?
- Maintenance costs: What are the costs of hardware maintenance and support software licences?
- Interoperability: Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Application Assessment

- Understandability: How difficult is it to understand the source code of the current system? How complex are the control structures that are used?
- Documentation: What system documentation is available? Is the documentation complete, consistent and current?
- Data: Is there an explicit data model for the system? Is the data used by the system up-to-date and consistent?
- Performance: Is the performance of the application adequate? Do performance problems have a significant effect on system users?
- Programming language: Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
- Configuration management: Are all versions of all parts of the system managed by a configuration management system?
- Test data: Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
- Personnel skills: Are there people available who have the skills to maintain the application?

Lehman's laws on Software Evolution

- **Continuing change**: A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
- **Increasing complexity**: As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
- **Large program evolution**: Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
- **Organizational stability**: Over a program's lifetime, its rate development is approximately constant and independent of the resources devoted to system development.

Lehman's laws on Software Evolution continued

- **Conservation of familiarity**: Over the lifetime of a system, the incremental change in each releases is approximately constant.
- **Continuing growth**: The functionality offered by systems has to continually increase to maintain user satisfaction
- **Declining quality**: The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
- **Feedback system**: Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Reading/Activity

- Please read: Manny Lehman, *Software's Future: Managing Evolution.* In IEEE Software, January-February 1998, pp. 40-44.

- Please read: Lutz and Mikulski, Operational anomalies as a cause of safety-critical requirements evolution. In the Journal of System and Software 65(2):155-161, 2003.

Summary

- Maintenance
  - Important, difficult and costly
  - Can, and should, be managed
  - Has a bad reputation, but can and should be challenging and rewarding
- Legacy systems a significant increasing problem
  - Number of approaches to dealing with legacy systems
  - Many involve transformation to OO and/or component based paradigms (e.g., Abstraction / high cohesion and Encapsulation / low coupling)
  - The business value of a legacy system and the quality of the application software and its environment should be assessed to determine whether the system should be replaced, transformed or maintained
- Software development and evolution should be a single, integrated, iterative process
- Looking at system evolution (in the long-term) provides insights on software evolution
- The cost of software maintenance generally exceed the software development costs
- The process of software evolution is driven by request for changes
- Software re-engineering is concerned with re-structuring and re-documenting software