Maintenance

Why maintenance?

All successful software gets changed. Two processes are at work. First, as a software product is found to be useful, people try it in new cases at the edge of or beyond the original domain. The pressures for extended function come chiefly from users who like the basic function and invent new uses for it.

Second, successful software survives beyond the normal life of the machine vehicle for which it is first written. If not new computers, than at least new disks, new displays, new printers come along; and the software must be conformed to its new vehicles of opportunity.

—Fred Brooks, No Silver Bullet

What is software maintenance

- Software maintenance is defined as the process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment . Software maintenance can consume as much as 90 percent of all the total effort expended on a system in its lifetime. Some computer scientists prefer to use the term evolution rather than maintenance to indicate that a product normally and naturally evolves over time.
- Software is the only technology where the equivalent of adding a new wing to a building is called "maintenance."

Program evolution dynamics

- Program evolution dynamics is the study of the processes of system change.
- After major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which applied to all systems as they evolved.
- There are sensible observations rather than laws. They are applicable to large systems developed by large organizations. Perhaps less applicable in other cases.

Lehman's Laws

Law	Description
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 of development is approximately constant and independent of the resources devoted to system development
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release 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.

Applicability of Lehman's laws

- Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.
  - Confirmed in more recent work by Lehman on the FEAST project (see further reading on book website).
- It is not clear how they should be modified for
  - Shrink-wrapped software products;
  - Systems that incorporate a significant number of COTS components;
  - Small organisations;
  - Medium sized systems.

Lehman's system types

- S-system: formally defined, derivable from a specification
- P-system: requirements based on approximate solution to a problem, but real-world remains stable
- E-system: embedded in the real world and changes as the world does

Software maintenance

- Modifying a program after it has been put into use.
- Maintenance does not normally involve major changes to the system's architecture.
- Changes are implemented by modifying existing components and adding new components to the system.

Maintenance is inevitable

- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be maintained therefore if they are to remain useful in an environment.

Examples of change during software development

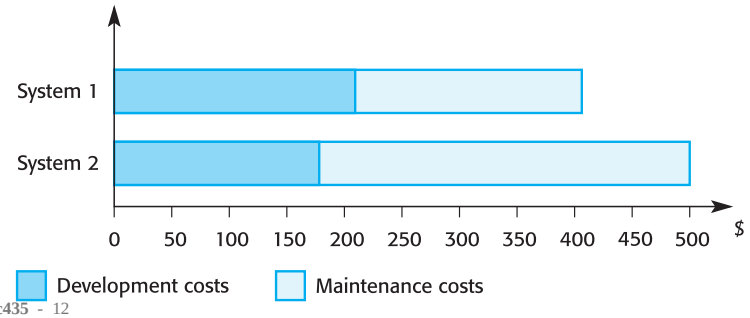
Activity from which initial change results	*Artifacts requiring consequent change*
Requirements analysis	Requirements specification
System design	Architectural design specification
	Technical design specification
Program design	Program design specification
Program implementation	Program code
	Program documentation
Unit testing	Test plans
	Test scripts
System testing	Test plans
	Test scripts
System delivery	User documentation
	Training aids
	Operator documentation
	System guide
	Programmer guide
	Training classes

Types of maintenance

- Corrective maintenance to repair software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- Adaptive maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Perfective maintenance to add to or modify the system's functionality or other system attribute
  - Modifying the system to satisfy new requirements.
- Preventive maintenance is maintenance to prevent problems before they occur

Maintenance costs

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- Aging software can have high support costs (e.g. old languages, compilers etc.).



System evolution vs. decline

- Is the cost of maintenance too high?
- Is the system reliability unacceptable?
- Can the system no longer adapt to further change, and within a reasonable amount of time?
- Is system performance still beyond prescribed constraints?
- Are system functions of limited usefulness?
- Can other systems do the same job better, faster or cheaper?
- Is the cost of maintaining the hardware great enough to justify replacing it with cheaper, newer hardware?

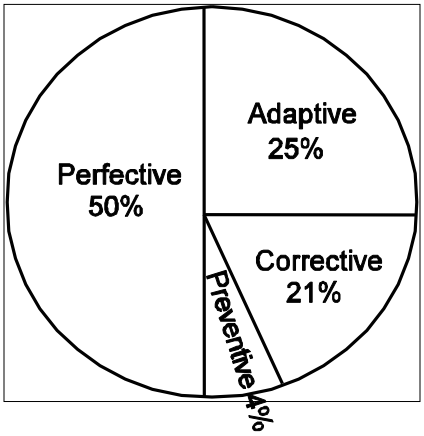
Laws of software evolution

- Continuing change: leads to less utility
- Increasing complexity: structure deteriorates
- Fundamental law of program evolution: program obeys statistically-determined trends and has invariants
- Conservation of organizational stability: global activity rate is invariant
- Conservation of familiarity: release content is invariant

Types of maintenance

- Corrective: maintaining control over day-to-day functions (fixing problems)
- Adaptive: maintaining control over system modifications (adding functionality)
- Perfective: perfecting existing functions - refactoring (redesign)
- Preventive: preventing system performance from degrading to unacceptable levels (adding additional internal checks)

Maintenance classes



- Williams uses the percentages 61%,18%, 17%, 4% instead of the numbers here. But the relative values are in the same order. Perfective maintenance dominates the cost.

Maintenance team responsibilities

- understanding the system
- locating information in system documentation
- keeping system documentation up-to-date
- extending existing functions to accommodate new or changing requirements
- adding new functions to the system
- finding the source of system failures or problems
- locating and correcting faults answering questions about the way the system works
- restructuring design and code components
- rewriting design and code components
- deleting design and code components that are no longer useful
- managing changes to the system as they are made

Maintenance problems

- Staff problems
  - Limited understanding
  - Management priorities
  - Morale
- Technical problems
  - Artifacts and paradigms
  - Testing difficulties

Factors affecting maintenance effort

- Application type
- System novelty
- Turnover and maintenance staff ability
- System life span
- Dependence on a changing environment
- Hardware characteristics
- Design quality
- Code quality
- Documentation quality
- Testing quality

Measuring maintainability

- Necessary data:
  - time at which problem is reported
  - time lost due to administrative delay
  - time required to analyze problem
  - time required to specify which changes are to be made
  - time needed to make the change
  - time needed to test the change
  - time needed to document the change
- Desirable data:
  - ratio of total change implementation time to total number of changes implemented
  - number of unresolved problems
  - time spent on unresolved problems
  - percentage of changes that introduce new faults
  - number of components modified to implement a change

Fog index

$$F = 0.4 \times \frac{\text{number of words}}{\text{number of sentences}} + \text{percentage of words of 3 or more syllables}$$

Automated maintenance tools

- Text editors
- File comparators
- Compilers and linkers
- Debugging tools
- Cross-reference generators
- Static code analyzers
- Configuration management repositories

Software rejuvenation

- Redocumentation: static analysis adds more information
- Restructuring (refactoring): transform to improve code structure
- Reverse engineering: recreate design and specification information from the code
- Reengineering: reverse engineer and then make changes to specification and design to complete the logical model; then generate new system from revised specification and design

Redocumentation

- Output may include:
  - component calling relationships
  - data-interface tables
  - data-dictionary information
  - data flow tables or diagrams
  - control flow tables or diagrams
  - pseudocode
  - test paths
  - component and variable cross-references

Four (unfortunate) truths

- Source code is king. [Why? Because…]
- Documentation is untrustworthy. [It is not kept up to date.]
- Bug tracking database stores knowledge.
- Reproduction is essential to obtaining a solution. [Reproducing is also important to ensure that the problem was a real problem and not a perceived one or caused by external sources such as software interactions.]

Key points

- Use a mini-development iterative enhancement cycle (analysis, design, code, test) when making and verifying any code changes. Maintenance is an error-prone process which can cause the injection of new faults – new faults which might cause more problems than the original problem – if the maintenance engineer is not careful and thorough.
- Avoid the quick-fix process for software maintenance to avoid the introduction of increased program complexity and unforeseen ripple effects. If you cannot avoid it, reanalyze the fix via a more systematic process once the crisis is over.
- In organizations that have change control boards, complete a software change request form and submit to this board for approval of proposed adaptive, perfective, and preventative maintenance.
- Complete a fault report for all needed corrective maintenance to describe a reported problem. Update this form with a detailed description of what was changed and why
- Use maintainability metrics to signal when maintenance costs are becoming excessive and when it is time for some preventative maintenance.