Lecture 06: Software Evolution

A case for Software Evolution

Broadly defined, *software evolution* refers to the study and management of the process of making changes to software over time. It comprises of: Development activities, Maintenance activities, and Reengineering activities.

Narrowly defined, *software evolution* is the process of developing software initially, and then repeatedly updating it for one reason or the other.

After deployment, systems will definitely have to change in order to remain useful:

- New requirements may emerge
- Existing requirements may change
- Business/ organisation changes may occur, and along with them a generation of new requirements for existing software
- Parts of the software may need modification to correct errors discovered during operation
- Modifications might be required to adapt software to a new platform
- Changes may need to be made to improve a software's performance

Importance of Software Evolution

- Organisations have huge investments in their software systems- they are critical business assets.
- To maintain the value of these assets to the business, they must be changed and updated.
- The majority of the software budget in large companies is devoted to evolving existing software rather than developing new software

The Software Evolution Process

Evolution processes depend on

- The type of software being maintained;
- The development processes used;
- The skills and experience of the people involved.

Proposals for change are the driver for system evolution. Change identification and evolution continue throughout the system lifetime

Existing requirements not implemented in previous release or there might be requests for new requirements, need for repairs of software.

The Software Evolution Process involves understanding the program that has to be changed, and then implementing these changes.

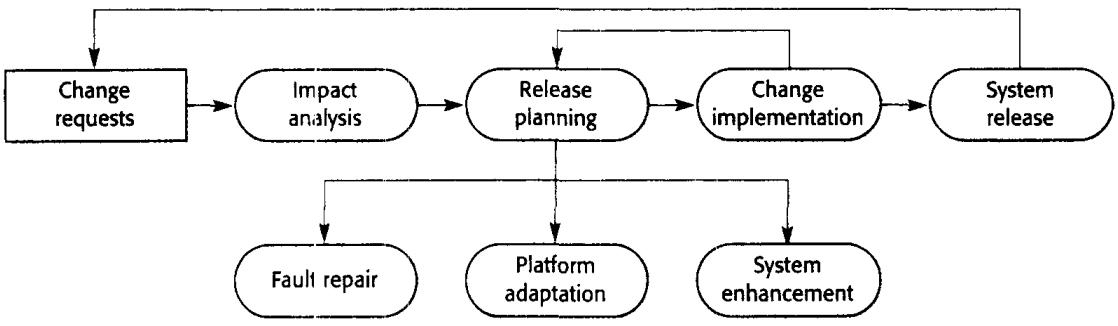


Figure 1: The Software Evolution Process

Lehman's Eight Laws of Software Evolution

The above cases all occur as postulated by Prof. Meir M. Lehman, when he described the behaviour in evolution of software in *Lehman's Eight Laws of Software Evolution:*

1. **Continuing change:** A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.

2. **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.

3. **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.

4. **Organizational stability:** Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

5. **Conservation of familiarity**: Over the lifetime of a system, the incremental change in each release is approximately constant.

6. **Continuing growth**: The functionality offered by systems has to continually increase to maintain user satisfaction

7. **Declining quality**: The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.

8. **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.

Software maintenance

Software maintenance is 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. [IEEE 1219]

It is the process of changing a system after it has been delivered/put to use in order to maintain its usefulness

Maintenance also involves activities like program understanding as the original developers may not be the same people carrying out the maintenance activities.

It has been observed that maintenance costs are usually high and this is attributed to the fact that it is more expensive to add functionality after the system is in operation than in it is to implement the same functionality during development.

Driven by change requests from users, it works in such a way that: modification requests are logged and tracked, the impact of proposed changes is determined, code and other software artefacts are modified, testing is conducted, and a new version of the software product is released.

Types of maintenance

Software maintenance is not just about 'fixing defaults'. Initially proposed as three types: corrective, perfective and adaptive maintenance by E.B Swanson, these have been updated to include the fourth type as preventive maintenance (formally included in corrective) in the standard ISO/IEC 14764:2006:

1. **Perfective maintenance:** This is maintenance carried out to add to or modify the system's functionality or modify the system to satisfy new requirements. These changes are usually required as a result of user requests (a.k.a. *evolutive* maintenance).

2. **Adaptive maintenance:** This is maintenance done to adapt software to a different operating environment which might be different from its initial environment of implementation. These changes are usually needed as a consequence of operating system, hardware, or DBMS changes.

3. **Corrective maintenance:** Maintenance to repair software faults which involves changing a system to correct deficiencies in the way meets its requirements. It primarily involves the identification and removal of faults in the software.

4. **Preventative maintenance:** A software product is modified after delivery to detect and correct latent faults in the software product before they become effective faults. Here the changes made to even before grave faults occur with the aim of making it more maintainable.

Note: The majority of SM is concerned with evolution deriving from user requested changes.

Software Maintenance Processes

- **Process Implementation**
  This involves software preparation and transition activities, e.g. the conception and creation of the maintenance plan, the preparation for handling problems identified during development, and the follow-up on product configuration management.

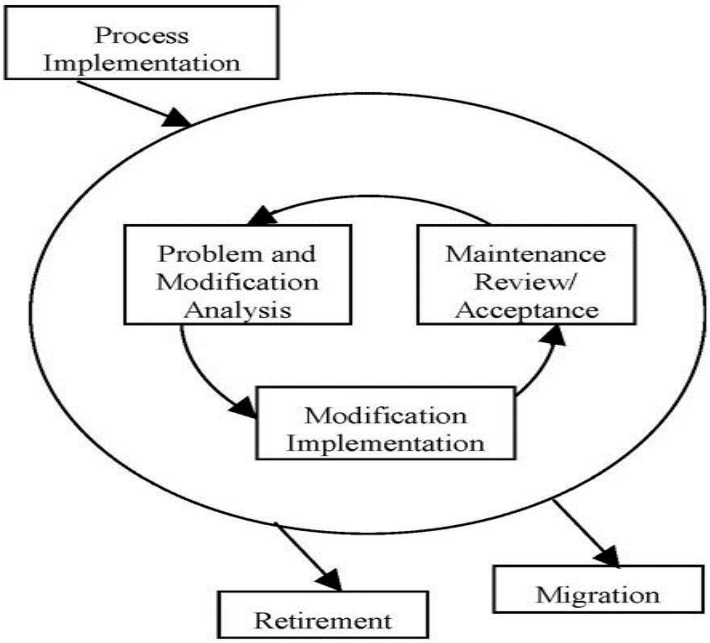
- **Problem and Modification Analysis**
  This process is executed once the application has become the responsibility of the maintenance group. The maintenance programmer must analyses each request, confirms it and checks its validity, investigates it and proposes a solution, documents the request and the solution proposal, and, finally, obtains all the required authorizations to apply the modifications.

- **Modification Implementation**
  This process entails considering the implementation of the modification itself.

- **Maintenance Review/Acceptance**
  This is the process of acceptance of the modification, by confirming the modified work with the individual who submitted the request in order to make sure the modification provided a solution.

- **Migration**
  The migration process is not part of daily maintenance tasks e.g. If the software must be ported to another platform without any change in functionality, this process will be used and a maintenance project team is likely to be assigned to this task.

- **Software Retirement**
  This is the last maintenance process, also an event which does not occur on a daily basis. It simply entails the retirement of a piece of software.



The Techniques for maintenance

1. Program Comprehension

There is need for software maintainers (programmers) to spend some time reading and understanding programs in order to be able to safely implement changes. This is also aided by availability of sufficient documentation to accompany the code.
Tools are available to aid in program comprehension e.g. Code browsers.

2. Reengineering

Reengineering is defined as the examination and alteration of software to reconstitute it in a new form (without changing functionality), and includes the subsequent implementation of the new form. It involves the reimplementation of all or key parts of important software systems. It is an approach to dealing with legacy systems through re-implementation.
The goal of re-engineering is to reduce maintenance costs and to improve software quality

Re-engineering involves the following activities:

- Source code translation
  This involves conversion of program code from the old language to a modern version of the same language of a new language.
- Reverse engineering
  The program is analyzed to understand it and extracting information from it in order to document its organization and functionality.
- Program structure improvement
  The control structure of a program is analyzed and modified to make it easier to read and understand.
- Program modularization
  The program structure is reorganized by grouping related parts together and removing redundancy where appropriate.
- Data reengineering
  This entails clean-up and restructure of system data to reflect program changes.

Re-engineering system software has two advantages over more radical approaches:

- Reduced risk: There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost: The cost of re-engineering is often significantly less than the costs of developing new software.

Reengineering cost factors

- The quality of the software (and supporting documentation) to be reengineered; the lower the quality the higher the re-engineering costs.
- The tool support available for reengineering; it is cheaper if one can use CASE tools to automate most of the program changes.
- The extent of data conversion required; higher volumes of data to mean increased cost.
- The availability of expert staff for reengineering; this can be a problem with old systems based on technology that is no longer widely used.

3. Reverse engineering

Reverse engineering is the process of analyzing software to identify the software's components and their interrelationships and to create representations of the software in another form or at higher levels of abstraction.

Reverse engineering is passive i.e. it does not change the software, or result in new software. The result of this activity is call graphs and control flow graphs from source code.

Reverse engineering difficulty increases with complexity of the software.

There are two types of reverse engineering:

a. Redocumentation: involves the creation or revision of a semantically equivalent representation within the same relative abstraction layer;

b. Design Recovery: involves identifying meaningful higher level abstractions beyond those obtained directly by examining the system itself.

The main motivation of both types of reverse engineering is to provide help in program comprehension.

Software Maintenance issues

Management

1. Alignment with organizational objectives.
   Software maintenance is resources consuming and it has no clear quantifiable benefit for the organization i.e. unclear Return-On-Investment (ROI).

2. Process issues
   Maintenance requires a number of additional activities not found in initial development e.g. help desk support, impact analysis and regression tests on the software changes are crucial issues which present challenges to management.

Technical

3. Limited Understanding.
   A lot of maintenance effort is usually devoted to understanding the software to be modified.

4. Domino Effect (a.k.a Ripple Effect)
   When a change is made to the code, there may be substantial consequential changes, not only in the code itself, but within documentation, design, and test suites.

5. Testing
   The cost of repeating a test on a piece of software can be significant time and cost wise.

Maintenance Cost factors

Maintenance costs are usually greater than development costs (depending on the application). Although they are affected by both technical and nontechnical factors (as proposed below), one should bear in mind that costs increase as software is maintained i.e. Maintenance corrupts the software structure so makes further maintenance more difficult.

- **Team stability**
  Maintenance costs are reduced if the same staff that developed the software is involved in maintenance.

- **Contractual responsibility**
  The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change (easy to change).

- **Staff skills**
  Maintenance staff is often inexperienced and have limited domain knowledge and quite often must learn the languages to maintain the system.

- **Program age and structure**
  As programs age, their structure is degraded by change and they become harder to understand and change further.

Importance of Maintenance

- 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.

Maintenance Prediction

Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs.

- Change acceptance depends on the maintainability of the components affected by the change;
- Implementing changes degrades the system and reduces its maintainability;
- Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Predicting Changes

Predicting the number of changes requires an understanding of the relationships between system and its environment, *for instance*: tightly coupled systems require changes whenever the environment is changed.

Factors influencing this relationship are:

- Number and complexity of system interfaces; the larger the number of interfaces and the more complex they are the more likely it is that demand for change will be made.
- Number of inherently volatile system requirements; *e.g.* requirements reflecting organisational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.
- The business processes where the system is used; evolution of business processes calls for a generation of change requests and the more business processes that use a system the more the demands for system change.

Complexity metrics

Predictions of maintainability can be made by assessing the complexity of system components. Complexity depends on

- Complexity of control structures;
- Complexity of data structures;
- Object, method (procedure) and module size.

Process metrics - Maintainability

Process measurements may be used to assess maintainability.

1. Number of requests for corrective maintenance: These might be in terms of failure reports recorded. However if these are still on the rise rather than decline even after numerous repairs, it may be an indication of a decline in maintainability.

2. Average time required for impact analysis: This time is an indication of the number of program components that are affected by the change request. An increase in this time implies that more components were affected and thus a decline in maintainability.

3. Average time taken to implement a change request: The time needed to carry out a system modification together with its documentation after impact analysis. An increase in this time indicates a decline in maintainability.

4. Number of outstanding change requests: almost related to the first metric, an increase in this number implies a decline in maintainability.

However, it is of great importance to the client that changes are accomplished *quickly* & *cost effectively*. This means that:

- Reliability should not degrade.
- Maintainability should not degrade.

Maintenance that becomes increasingly more expensive and difficult becomes known as a *legacy system*. However the legacy system may still be of essential importance to the organization.

Configuration management

Software configuration management (SCM) is the task of tracking and controlling changes in the software. Configuration management is concerned with managing evolving software systems. It aims to control the costs and effort involved in making changes to a system.

The software product and any changes made to it must be controlled. This control is established by implementing and enforcing an approved software configuration management (SCM) process, and this involves the development and application of procedures and standards to manage an evolving software product.

Configuration management practices include revision/version control and the establishment of baselines (starting point for further development), Build Management, Release Management, and Process Control.

Version control

In order to keep track of changes made to software, a version control system can be used. This is done as part of the Software configuration process.

Version control is simply the automated act of tracking the changes of a particular file over time. This is typically accomplished by maintaining one copy of the file in a repository, then tracking the changes to that file (rather than maintaining multiple copies of the file itself). This is done using check-in and check-out principles i.e. Each time someone needs to edit a file, they check it out for exclusive editing and then check it back in to the repository when finished, thus creating a new version.

A version control system thus maintains an organized set of all the versions of files that are made over time.
Version control benefits include:

- Enables roll back to a previous version of a given file
- Allows one to compare two versions of a file, highlighting differences
- Provides a mechanism of locking, forcing serialized change to any given file
- Creates branches that allow for parallel concurrent development
- Maintains an instant audit trail on each and every file: versions, modified date, modifier, and any additional amount of metadata your system provides for and you choose to implement.

Techniques for dealing with change

A. Impact Analysis

Impact analysis is the evaluation of the many risks associated with the change, including estimates of effects on resources, effort, and schedule.

Work product: any development artifact whose change is significant, e.g. requirements, design and code components, test cases, etc.

The quality of one can affect the quality of others.

Horizontal traceability

This approach traces the relationships of components across collections of work products.

Vertical traceability

This approach traces the relationships among parts of a work product.

Impact Analysis is needed:

- *To ensure that the change has been correctly and consistently bounded i.e. Making sure that all affected parts are changed correctly (**change propagation**);*
- *To identify all objects impacted by changes in the primary sector (**change impact analysis**).*

B. Restructuring/Refactoring

This activity involves improving the software structure or architecture without changing the behavior.

Software restructuring modifies source code and/or data in an effort to make it amenable to future changes.

C. Regression testing

This activity involves efficiently verifying that the change preserved the behavior of functionalities that should not be impacted.

Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects. Regression tests should be done every time a major change is made to the software.

Legacy Systems

A legacy system is a socio-technical system that is useful or essential to an organization but which has been developed using obsolete technology or methods.

As earlier mentioned, it is a system for which maintenance becomes increasingly more expensive and difficult. These systems are usually characterized by their problems – Legacy problems such that by definition a legacy system is typically:

- very old and large
- has been heavily modified
- based on the old technology
- documentation not be available
- none of the original members of the software development team may still be around
- often support very large quantities of live data
- the software is often at the core of the business and replacing it would be a **huge** expense.

Legacy System Evolution

Organisations that rely on legacy systems must choose a strategy for evolving these systems

- Scrap the system completely and modify business processes so that it is no longer required;
- Continue maintaining the system;
- Transform the system by re-engineering to improve its maintainability;
- Replace the system with a new system.

The strategy chosen by an organisation should depend on the system quality and its business value.

Categories of Legacy Systems

a. Low quality, low business value: these systems should be scrapped.
b. Low-quality, high-business value: these make an important business contribution but are expensive to maintain. These should be re-engineered or replaced if a suitable system is available.
c. High-quality, low-business value: these should be replaced with Commercial, off-the shelf (COTS), scraped completely or maintained.
d. High-quality, high business value: the best option for these is to continue in operation using normal regular system maintenance.

Automated maintenance tools

A software maintenance tool is an artifact that supports a software maintainer in performing a tasks. The use of tools for software maintenance simplifies the tasks and increases efficiency and productivity.

Classifying the maintenance tools based on the specific tasks they support:

A. Program understanding and reverse engineering tools.
   - Program slicer: Slicing is the mechanical process of marking all the sections of a program text that may influence the value of a variable at a given point in the program. Program slicing helps the programmers select and view only the parts of the program that are affected by the changes.
   - Static analyzer is used in analyzing the different parts if the program such as modules, procedures, variables, data elements, objects and classes. A static analyzer allows general viewing of the program text and generates summaries of contents and usage of selected elements in the program text, such as variables or objects.
   - A dynamic analyzer could be used to analyze the program while it is executing.
   - A data flow analyzer is a static analysis tool that allows the maintainer to track all possible data flow and control flow paths in the program. It allows analysis of the program to better outline the underlying logic of the program. It also helps display the relationship between components of the system.
   - A cross-referencer produces information on the usage of a program. This tool helps the user focus on the parts that are affected by the change.
   - A dependency analyzer assists the maintainer to analyze and understand the interrelationships between entities in a program. It tool provides capabilities to set up and query the database of the dependencies in a program. It also provides graphical representations of the dependencies.

B. Testing tools
   - Test simulator tools help the maintainer to test the effects of the change in a controlled environment before implementing the change on the actual system.
   - A test case generator produces test data that is used to test the functionality of the modified system.
   - A test path generator helps the maintainer to find all the data flow and control flow paths affected by the changes.

C. Configuration Management tools: code management, version control, and change tracking tools. Configuration management and version control tools help store the objects that form the software system. A source control system is used to keep a history of the files so that versions can be tracked and the programmer can keep track of the file changes.

D. Documentation tools: hypertext-based tools, data flow and control chart generators, requirements tracers, and CASE tools.

Software re-use

The goals of reuse during maintenance are to ***increase productivity***, ***increase quality***, ***facilitate code transportation***, ***reduce maintenance time and effort***, ***and improve maintainability***.

Software reuse is the reapplication of a variety of kinds of knowledge about one system to another similar system in order to reduce the effort of development or maintenance of that other system (Biigerstaff et al.[1989])

Software reuse is derived from the process, the personnel, and the product.

- The **process** is an activity or action performed by a machine or person. A methodology could be reused on different application problems. E.g. Object Oriented Design.
- The reuse of **personnel** consists of reusing the knowledge of the people that have faced and overcome issues in previous projects and applying this knowledge to the new projects.
- **Product** reuse consists of reusing the previously created projects. Data, design, and programs are all products that can be reused

Further Reading

Sommerville, I., *Software Engineering*, 8th ed., Pearson, 2006.
Myers, Glenford J., *The Art of Software Testing*, 2nd ed., Wiley, 2004
Pressman, Roger S., *Software Engineering-A Practitioner's Approach*, 5th ed., McGraw-Hill, 2001.