CH-1: Software Evolution and maintenance concepts

- The concept of ***software maintenance*** means preventing software from failing to deliver the intended functionalities by means of bug fixing.
- The concept of ***software evolution*** means a continual change from a lesser, simpler, or worse state to a higher or better state.

Bennett and Xu made further distinctions between the two as follows:

- All support activities carried out *after* delivery of software are put under the category of *maintenance*.
- All activities carried out to effect changes in requirements are put under the category of *evolution*.

...

Software maintenance comprises all activities associated with the process of changing software for the purposes of:

- fixing bugs; and/or
- improving the design of the system so that future changes to the system are less expensive.

Evolution of software systems means creating new but related designs from existing ones. The objectives include:

- supporting new functionalities
- making the system perform better and
- making the system run on a different operating system

Software Evolution

- Fundamental work in the field of software evolution was done by Lehman and his collaborators.
  - Based on empirical studies, Lehman and his collaborators formulated some observations and they introduced them as ***laws of evolution***.
  - The "laws" themselves have "evolved" from *three* in 1974 to *eight* by 1997.
- Those laws are the results of studies of the evolution of large-scale proprietary or closed source software (CSS) systems.
  - The laws concern a category of software systems called *E-type* systems. The eight laws are briefly explained as follows:

Software Evolution…

1. Continuing change: Unless a system is continually modified to satisfy emerging needs of users, the system becomes increasingly less useful.
2. Increasing complexity: Unless additional work is done to explicitly reduce the complexity of a system, the system will become increasingly more complex due to maintenance-related changes.
3. Self-regulation: The evolution process is self-regulating in the sense that the measures of products and processes, that are produced during the evolution, follow close to *normal* distributions.

Software Evolution…

4. Conservation of organizational stability: The average effective global activity rate on an evolving system is almost constant throughout the lifetime of the system. In other words, the average amount of additional effort needed to produce a new release is almost the same.
5. Conservation of familiarity: As a system evolves all kinds of personnel, namely, developers and users, for example, must gain a desired level of understanding of the system's content and behavior to realize satisfactory evolution. A large incremental growth in a release reduces that understanding. Therefore, the average incremental growth in an evolving system remains almost the same.

Software Evolution…

6. Continuing growth: As time passes, the functional content of a system is continually increased to satisfy user needs.
7. Declining quality: Unless the design of a system is diligently fine-tuned and adapted to new operational environments, the system's qualities will be perceived as declining over the lifetime of the system.
8. Feedback system: The system's evolution process involves multi-loop, multi agent, multi-level feedback among different kinds of activities.
    - ♦ Developers must recognize those complex interactions in order to continually evolve an existing system to deliver more functionalities and higher levels of qualities.

Software Maintenance

- There are defects in delivered software applications, because defect removal and quality control processes are not perfect.
  - Therefore, maintenance is needed to repair those defects in released software.
  - E. Burton Swanson initially defined three categories of software maintenance activities, namely, corrective, adaptive, and perfective.
- Those definitions were later incorporated into the standard software engineering–software life cycle processes–Maintenance and introduced a fourth category called preventive maintenance.
  - The reader may note that some researchers and developers view preventive maintenance as a subset of perfective maintenance.

Software Maintenance…

Types of Maintenance:

- Corrective:
  - correcting faults in system behavior
  - caused by errors in coding, design or requirements
- Perfective:
  - due to changes in requirements
  - Often triggered by organizational, business or user learning
- Adaptive:
  - due to changes in operating environment
  - e.g., different hardware or Operating System
- Preventive:
  - e.g., dealing with legacy systems

Software Maintenance…

- Swanson's classification of maintenance activities is intention based because the maintenance activities reflect the intents of the developer to carry out specific maintenance tasks on the system.
- In the intention-based classification of maintenance activities, the intention of an activity depends upon the motivations for the change.
  - Activities to make corrections: If there are discrepancies between the expected behavior of a system and the actual behavior, then some activities are performed to eliminate or reduce the discrepancies.
  - Activities to make enhancements: A number of activities are performed to implement a change to the system, thereby changing the behavior or implementation of the system.

Software Maintenance…

This category of activities is further refined into three subcategories:

- Enhancements that modify existing requirements;
- Enhancements that create new requirements; and
- Enhancements that modify the implementation without changing the requirements.

Software Maintenance…

- Chapin expanded the typology of Swanson into an evidence-based classification of **12** different types of software maintenance:

♦ Training	♦ Reformative	♦ Preventive	♦ Reductive
♦ Consultive	♦ Updative	♦ Performance	♦ Corrective
♦ Evaluative	♦ Groomative	♦ Adaptive	♦ Enhancive

- The three objectives for classifying the types of software maintenance are as follows:
  - It is more informative to classify maintenance tasks based on objective evidence that can be verified with observations and/or comparisons of software before and after modifications.
  - The granularity of the proposed classification can be made to accurately reflect the actual mix of activities observed in the practice of software maintenance and evolution.
  - The classification groups are independent of hardware platform, operating system choice, design methodology, implementation language…

Software Maintenance…

- Component based software systems (CBS): New components are developed by combining commercial off-the-shelf (COTS) components, custom-built (in-house) components, and open source software components.
- The major differences between component-based software systems (CBS) and custom-built software systems:
  - *Skills of system maintenance teams:* Maintenance of CBS requires specialized skills to monitor and integrate COTS products.
  - *Infrastructure and organization:* Running a support group for in-house products is necessary to manage a large product.
  - *COTS maintenance cost:* This cost includes the costs of purchasing components, licensing components, upgrading components, and training maintenance personnel.
  - *Larger user community:* COTS users are part of a broad community of users, and the community of users can be considered as a resource, which is a positive factor.

Software Maintenance…

- *Modernization:*
- *Split maintenance function:*
- *More complex planning:*

SOFTWARE EVOLUTION MODELS AND PROCESSES

- Software maintenance should have its own software maintenance life cycle (SMLC) model.
- A number of SMLC models with some variations are available in literature. Three common features of the SMLC models found in the literature are:
  - **Understanding the code;**
  - **Modifying the code; and**
  - **Revalidating the code.**
- Other models view software development as ***iterative*** processes and based on the idea of ***change mini-cycle*** as explained in the following:
  - **Iterative models**: Systems are constructed in builds, each of which is a refinement of requirements of the previous build. A build is refined by considering feedback from users.
  - **Change mini-cycle models**: These models consist of five major phases: change request, analyze and plan change, implement change, verify and validate, and documentation change.
    - In this process model, several important activities were identified, such as program comprehension, impact analysis, refactoring, and change propagation.

SOFTWARE EVOLUTION MODELS …

- A different kind of software evolution model, called *staged model of maintenance and evolution*, has been proposed by Rajlich and Bennett.
- The model is descriptive in nature, and its primary objective is to improve the understanding of how long-lived software evolves.
- The model considers four distinct, sequential stages of the lifetime of a system, as explained below:

1. Initial development: When the initial version of the system is produced, detailed knowledge about the system is fresh. Before delivery of the system, it undergoes many changes. Eventually, a system architecture emerges and soon it stabilizes.
2. Evolution: Significant changes involve higher cost and higher risk. In the period immediately following the initial delivery, knowledge about the system is still almost fresh in the minds of the developers.

- In general, for many systems, their lifespan are spent in this stage, because the systems continue to be of importance to the organizations.

SOFTWARE EVOLUTION MODELS …

3. Servicing: When the knowledge about the system has significantly decreased, the developers mainly focus on maintenance tasks, such as fixing bugs, whereas architectural changes are rarely effected.
4. Phase out: When even minimal servicing of a system is not an option, the system enters its very final stage. The organization decides to replace the system for various reasons:

   (i) it is too expensive to maintain the system; or

   (ii) there is a newer solution available.

SOFTWARE EVOLUTION MODELS …

- ***Software Maintenance Standards:*** A well-defined process for software maintenance can be observed and measured, and thus improved.
  - IEEE and ISO have both addressed s/w maintenance processes.
  - IEEE/EIA 1219 and ISO/IEC 14764 as a part of ISO/IEC12207 (life cycle process).
  - IEEE/EIA 1219 organizes the maintenance process in seven phases:
    - Problem identification, Analysis, Design, Implementation, System test, Acceptance test and Delivery.
    - for each phase, the standard identifies the input and output deliverables, the supporting processes and the related activities, and a set of evaluation metrics.

SOFTWARE EVOLUTION MODELS …

- ***Software Configuration Management:*** Configuration management (CM) is the discipline of managing and controlling changes in the evolution of software systems.
  - The goal of CM is to manage and control the various extensions, adaptations, and corrections that are applied to a system over its lifetime.
- An SCM system has four different elements, each element addressing a distinct user need as follows:
  - **Identification of software configurations**: This includes the definitions of the different artifacts, their baselines or milestones, and the changes to the artifacts.
  - **Control of software configurations:** controlling the ways artifacts or configurations are altered with the necessary technical and administrative support.
  - **Auditing software configurations:** Making the current status of the software system in its life cycle visible to management and determining whether or not the baselines meet their requirements.
  - **Accounting software configuration status:** providing an administrative history of how the software system has been altered, by recording the activities.

REENGINEERING

- Reengineering implies a single cycle of taking an existing system and generating from it a new system, Whereas evolution can go forever.
- To a large extent, software evolution can be seen as repeated software reengineering.
- Reengineering includes some kind of reverse engineering activities to design an abstract view of a given system.
  - The new abstract view is restructured, and forward engineering activities are performed to implement the system in its new form.
- The aforementioned process is captured by Jacobson and Lindst¨orm with the following expression:
- Reengineering = Reverse engineering+Δ+Forward engineering.

REENGINEERING…

- The first element "Reverse engineering" is the activity of defining a more abstract and easier to understand representation of the system.
  - For example, the input to the reverse engineering process is the source code of the system, and the output is the system architecture.
  - The core of reverse engineering is the process of examination of the system, and it is not a process of change.
- The third element "forward engineering" is the traditional process of moving from a high-level abstraction and logical, implementation-independent design to the physical implementation of the system.
- The second element "Δ" captures alterations performed to the original system.

LEGACY SYSTEMS

- A legacy software system is an old program that continues to be used because it still meets the users' needs, in spite of the availability of newer technology or more efficient methods of performing the task.
- It is the phase out stage of the software evolution model of Rajlich and Bennet described earlier.
- There are a number of options available to manage legacy systems. Typical solution include:
- Freeze: The organization decides no further work on the legacy system should be performed.
- Outsource: An organization may decide that supporting software is not core business, and may outsource it to a specialist organization offering this service.

LEGACY SYSTEMS…

- Carry on maintenance: Despite all the problems of support, the organization decides to carry on maintenance for another period.
- Discard and redevelop: Throw all the software away and redevelop the application once again from scratch.
- Wrap: It is a black-box modernization technique – surrounds the legacy system with a software layer that hides the unwanted complexity of the existing data, individual programs, application systems, and interfaces with the new interfaces.
- Migrate: Legacy system migration basically moves an existing, operational system to a new platform, retaining the legacy system's functionality and causing minimal disruption to the existing operational business environment as possible.

IMPACT ANALYSIS

- Impact analysis is the task of estimating the parts of the software that can be affected if a proposed change request is made.

- Impact analysis techniques can be partitioned into two classes:
  - Traceability analysis: In this approach the high-level artifacts such as requirements, design, code and test cases related to the feature to be changed are identified.
  - A model of inter-artifacts such that each artifact in one level links to other artifacts is constructed, which helps to locate a pieces of design, code and test cases that need to be maintained.
  - Dependency (or source-code) analysis: Dependency analysis attempt to assess the affects of change on semantic dependencies between program entities.
  - This is achieved by identifying the syntactic dependencies that may signal the presence of such semantic dependencies.
    - The two dependency-based impact analysis techniques are: call graph based analysis and dependency graph based analysis.

REFACTORING

- Refactoring: is the process of making a change to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.
- It is the object-oriented equivalent of restructuring.
- Refactoring, which aims to improve the internal structure of the code, achieve through the removal of duplicate code, simplification, making code easier to understand, help to find defects and adding flexibility to program faster.
- There are two aspects of the above definition:
  - It must preserve the “observable behavior” of the software system (through regression).
  - To improve the internal structure of a software system (improve maintainability).

SOFTWARE REUSE

- Software reuse was introduced by Dough McIlroy in his 1968 seminal paper:
  - The development of an industry of reusable source-code software components and the industrialization of the production of application software from off-the-shelf components.
- Software reuse is using existing artifacts or software knowledge during the construction of a new software system.
  - Reusable assets can be either reusable artifacts or software knowledge.
- Capers Jones identified four types of reusable artifacts:
  - data reuse, involving a standardization of data formats,
  - architectural reuse, which consists of standardizing a set of design and programming conventions dealing with the logical organization of software,
  - design reuse, for some common business applications, and
  - program reuse, which deals with reusing executable code.

SOFTWARE REUSE…

Benefits of Reuse

- Increased reliability
- Reduced process risk
- Increase productivity
- Standards compliance
- Accelerated development
- Improve maintainability
- Reduction in maintenance time and effort