Software Evolution and Maintenance

A Practitioner's Approach

Chapter 1
Basic Concepts and Preliminaries

Outline of the Chapter

1.1 Evolution Versus Maintenance

- The terms evolution and maintenance are used interchangeably.
- However there is a semantic difference.
- Lowell Jay Arthur distinguish the two terms as follows:

  "Software maintenance means to preserve from failure or decline."

  "Software evolution means a continuous change from lesser, simpler, or worse state to a higher or better state."
- Keith H. Bennett and Lie Xu use the term:

  "maintenance for all post-delivery support and evolution to those driven by changes in requirements."

1.1 Evolution Versus Maintenance

- Maintenance is considered to be set of planned activities whereas evolution concern whatever happens to a system over time.

- Mehdi Jazayer's view on software evolution:

  "Over time what evolves is not the software but our knowledge about a particular type of software."

1.1.1 Software Evolution

- In 1965, Mark Halpern used the term *evolution* to define the dynamic growth of software.
- The term evolution in relation to application systems took gradually in the 1970s.
- Lehman and his collaborators from IBM are generally credited with pioneering the research field of software evolution.
- Lehman formulated a set of observations that he called laws of evolution.
- These laws are the results of studies of the evolution of large-scale proprietary or closed source system (CSS).
- The laws concern what Lehman called E-type systems:

  "Monolithic systems produced by a team within an organization that solve a real world problem and have human users."

1.1.1 Software Evolution: Laws of Lehman

- *Continuing change* (1st) – A system will become progressively less satisfying to its user over time, unless it is continually adapted to meet new needs.
- *Increasing complexity* (2nd) – A system will become progressively more complex, unless work is done to explicitly reduce the complexity.
- *Self-regulation* (3rd) – The process of software evolution is self regulating with respect to the distributions of the products and process artifacts that are produced.
- *Conservation of organizational stability* (4th) – The average effective global activity rate on an evolving system does not change over time, that is the average amount of work that goes into each release is about the same.

1.1.1 Software Evolution: Laws of Lehman

- *Conservation of familiarity* (5th) – The amount of new content in each successive release of a system tends to stay constant or decrease over time.
- *Continuing growth* (6th) – The amount of functionality in a system will increase over time, in order to please its users.
- *Declining quality* (7th) – A system will be perceived as losing quality over time, unless its design is carefully maintained and adapted to new operational constraints.
- *Feedback system* (8th) – Successfully evolving a software system requires recognition that the development process is a multi-loop, multi-agent, multi-level feedback system.

1.1.1 Software Evolution: FOSS System

- In circa 1988, Pirzada pointed out the differences between the evolution of the Unix OS and system studied by Lehman
- Pirzada argued that the differences in academic and industrial s/w development could lead to a differences in the evolutionary pattern.
- In circa 2000, empirical study of free and open source software (FOSS) evolution was conducted by Godfrey and Tu.
- They found that the growth trends from 1994-1999 for the evolution of FOSS Lunix OS to be super-linear, that is greater than linear.
- Robles and his collaborator concluded that Lehman's laws, 3, 4, and 5 are not fitted to large scale FOSS system such as Linux.

"FOSS is made available to the general public with either relaxed or non-existent intellectual property restrictions. The free emphasizes the freedom to modify and redistribute under the terms of the original license while open emphasizes the accessibility to the source code."

1.1.2 Software Maintenance

- There will always be defects in the delivered software application because software defect removal and quality control are not perfect.
- Therefore, software maintenance is needed to repair these defects in the released software.
- E. Burton Swanson defined three type of software maintenance:

  Corrective, Adaptive & Perfective.
- It is based on the intent of the developer towards the system.

1.1.2 Software Maintenance

- Swanson definition was later updated in the standard for software engineering – ISO/IEC 14764.
- Introduced a fourth category called preventive maintenance.
- Some researchers and developers view preventive maintenance as a subset of perfective maintenance.

1.1.2 Software Maintenance

- Kitchenham described maintenance modifications in a hierarchical ways based on the concept of activity:
  - 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.

    This category is subdivided into three types:

    - enhancements that change existing requirement,
    - enhancements that add new system requirements, and
    - enhancements that change the implementation but not the requirements.

1.1.2 Software Maintenance

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

- Training
- Evaluate
- Updative
- Preventive
- Adaptive
- Corrective
- Consultive
- Reformative
- Groomative
- Performance
- Reductive
- Enhancive

1.1.2 Software Maintenance: COTS

The major differences between component-based software systems (CBS) and custom-built software systems:

- Skills of system maintenance teams.
- Infrastructure and organization.
- COTS maintenance cost.
- Larger user community.
- Modernization.
- Split maintenance function.
- More complex planning.

1.2 Software Evolution Models and Processes

- Software maintenance have its own software maintenance life cycle (SMLC) model.

- Three popular SMLC models:

    - Staged model of maintenance and evolution.
    - Iterative models.
    - Change mini-cycle models.

1.2 Software Evolution Models and Processes

Software Maintenance Standards

- 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.
- ISO/IEC 14764 describes s/w maintenance as an iterative process for managing and executing software maintenance activities.
- The activities which comprise the maintenance process are:
  - process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration and retirement.
- Each of these maintenance activity is made up of tasks. Each task describes a specific action with inputs and outputs.

1.2 Software Evolution Models and Processes

Software Configuration Management

- SCM is the discipline of managing and controlling change in the evolution of software system.

- It ensures that the released software is not contaminated by uncontrolled or unapproved changes.

- An SCM system has four different elements, each element addressing a distinct user need as follows:
    - Identification of software configurations.
    - Control of software configurations.
    - Auditing software configurations.
    - Accounting software configuration status.

1.3 Reengineering

- Reengineering is done to transform an existing “lesser or simpler” system into a new “better” system.

- Reengineering is the examination, analysis and restructuring of an existing software system to reconstitute it in a new form, and the subsequent implementation of the new form.

- Chikofsky and Cross II defines reengineering as:

  “the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form.”

1.3 Reengineering

- Jacobson and Lindstorm defined following formula:

  Reengineering = Reverse engineering + Δ + Forward engineering.

- Reverse engineering is the activity of defining a more abstract, and easier to understand, representation of the system
- The core of reverse engineering is the process of examination, not a process of change, therefore it does not involve changing the software under examination.
- The third element "forward engineering," is the traditional process of moving from high-level abstraction and logical, implementation-independent designs to the physical implementation of the system.
- The second element Δ captures alteration that is change of the system.

1.4 Legacy System

- A legacy system is an old system which is valuable for the company which often developed and owns it.

- It is the phase out stage of the software evolution model of Rajlich and Bennet described earlier.

- Bennett used the following definition:

  "large software systems that we don't know how to cope with but that are vital to our organization."

- Similarly, Brodie and Stonebraker: define a legacy system as

  "any information system that significantly resists modification and evolution to meet new and constantly changing business requirements."

1.4 Legacy System

- 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.
  - 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.

1.5 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.

1.5 Impact Analysis

- Two additional notions related to impact analysis are very common among practitioners:
  - Ripple effect.
  - Change propagation.
- Ripple effect analysis measures the impact, or how likely it is that a change to a particular module may cause problems in the rest of the program.
- Measuring ripple effect can provide knowledge about the system as a whole through its evolution:
  - how much its complexity has increased or decreased since the previous version,
  - how complex individual parts of a system are in relation to other parts of the system, and
  - to look at the effect that a new module has on the complexity of a system as a whole when it is added.
- The change propagation activity ensures that a change made in one component is propagated properly throughout the entire system.

1.6 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).

1.7 Program Comprehension

- Program understanding or comprehension is

  "the task of building mental models of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution and re-engineering purposes."

- A mental model describes a programmer's mental representation of the program to be comprehended.
- Program comprehension deals with the cognitive processes involved in constructing a mental model of the program.
- A common element of such cognitive models is generating hypotheses and investigating whether they hold or must be rejected.
  – Hypotheses are a way to understand code in an incremental manner.
  – After some understanding of the code, the programmer forms a hypothesis and verifies it by reading code.
  – By continuously formulating new hypotheses and verifying them, the programmer understands more and more code and in increasing details.

1.7 Program Comprehension

- Several strategies can be used to arrive at relevant hypotheses such as:
  - bottom up (starting from the code).
  - top down (starting from high-level goal).
  - opportunistic combinations of the two.
- A strategy is formulated by identifying actions to achieve a goal.
- Strategies guide two mechanisms, namely chunking and cross-referencing to produce higher-level abstraction structures.
  - Chunking creates new, higher level abstraction structures from lower level structures.
  - Cross-referencing means being able to make links elements of different abstraction levels.
- Chunking and cross-referencing helps in building mental model of the program under study at different levels of abstractions.

1.8 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.

- Other significant early reuse research developments include:
  - Program families (David Parnas).
  - Domain analysis (Jim Neighbors).
- Program families are sets of programs whose common properties are so extensive that it becomes advantageous to study the common properties of these programs before analyzing individual differences.
- Whereas domain analysis is an activity of identifying objects and operations of a class of similar systems in a particular problem domain.

1.8 Software Reuse

- 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,
  - arhitectural 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 of previously written code is a way to increase
  - software development productivity.
  - quality of the software.
- The cost savings during maintenance as a consequence of reuse are nearly twice the corresponding savings during development.

1.8 Software Reuse

- Reusability is a property of a software assets that indicates the degree to which it can be reused.
- For software component to be reusable, it must exhibit the following characteristics that directly encourage its use is similar situation:
  - Environmental independence - The components can be reused irrespective of the environment from which they were originally captured.
  - High cohesion - The components that implement a single operation or set of related operations.
  - Loose coupling - The components that have minimal links to other components.
  - Adaptability - The components that are adaptable so they can be customized to fit a range of similar situation.
  - Understandability - The components which are easily understandable that users can quickly interpret functionality.
  - Reliability - The components that are error-free.
  - Portability - The components that are not restricted in terms of the software or hardware environment they operate in.

1.8 Software Reuse

Benefits of Reuse

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