SE 811: Software Maintenance

Toukir Ahammed

Evolution of Software Systems

- The term *evolution* was used by Mark I. Halpern in circa 1965 to define the dynamic growth of software
- *Evolution* means a continuously changing software from a worse state to a better state.
- Ned Chapin defines software evolution as:
  - "the applications of software maintenance activities and processes that generate a new operational software version with a changed customer-experienced functionality or properties from a prior operational version, where the time period between versions may last from less than a minute to decades, together with the associated quality assurance activities and processes, and with the management of the activities and processes"

Evolution of Software Systems

Software evolution is like a computer program, with inputs, processes, and outputs

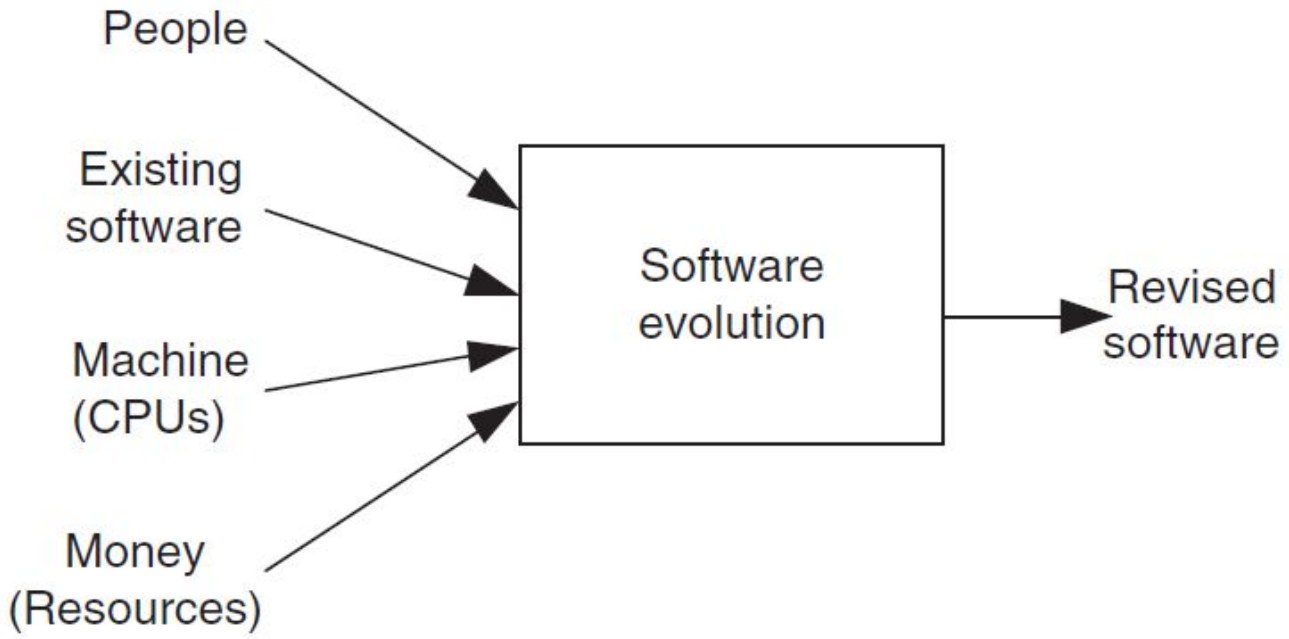

FIGURE 2.4 Inputs and outputs of software evolution. From Reference 26. © 1988 John Wiley & Sons

Evolution of Software Systems

Software evolution is studied with two broad, complementary approaches:

- *Explanatory (what/why)*
  - attempts to explain the causes of software evolution, the processes used, and the effects of software evolution
  - studies evolution from a *scientific* view point
- *Process improvement (how)*
  - attempts to manage the effects of software evolution by developing better methods and tools, namely, design, maintenance, refactoring, and reengineering.
  - studies evolution from an *engineering* view point

SPE Taxonomy

- SPE refers to

  S (Specified),

  P (Problem), and

  E (Evolving) programs

- The classification scheme is characterized by:

  (i) how a program interacts with its environment and

  (ii) the degree to which the environment and the underlying problem that the program addresses can change

S-type Programs

- S-type —programs that are derived from a well defined, probably formal, specification.

S-type programs have the following characteristics:

- All the nonfunctional and functional program properties that are important to its stakeholders are *formally* and *completely* defined.
- Correctness of the program with respect to its formal specification is the *only* criterion of the acceptability of the solution to its stakeholders.

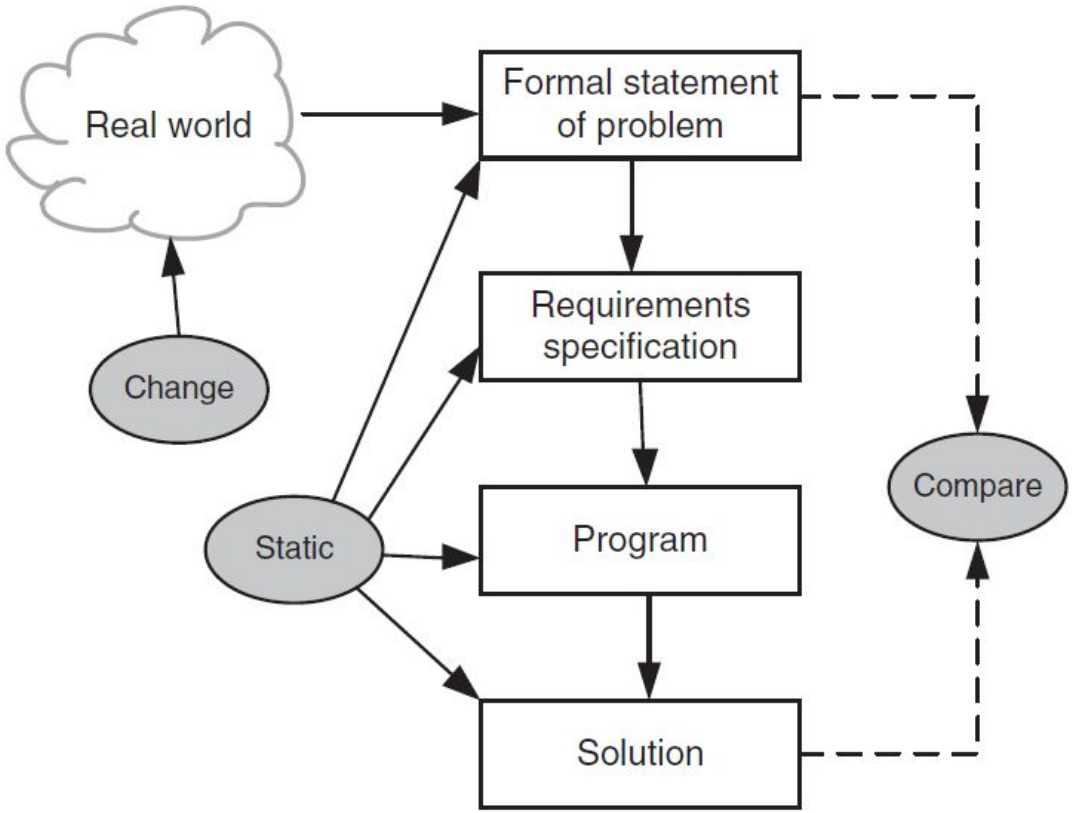

FIGURE 2.5 S-type programs

S-type Programs

- Examples of S-type programs include
  - calculation of the lowest common multiple of two integers and
  - perform matrix addition, multiplication, and inversion
- The problem is completely defined, and there are one or more correct solutions to the problem as stated.

P-type Programs

- P-type — programs that attempt to solve, or at least approximately solve, a real-world problem that cannot be solved formally in a pre-specified manner.
- Instead, an iterative process, that checks and successively refines the performance and possibly other properties of candidate solutions, must be employed.

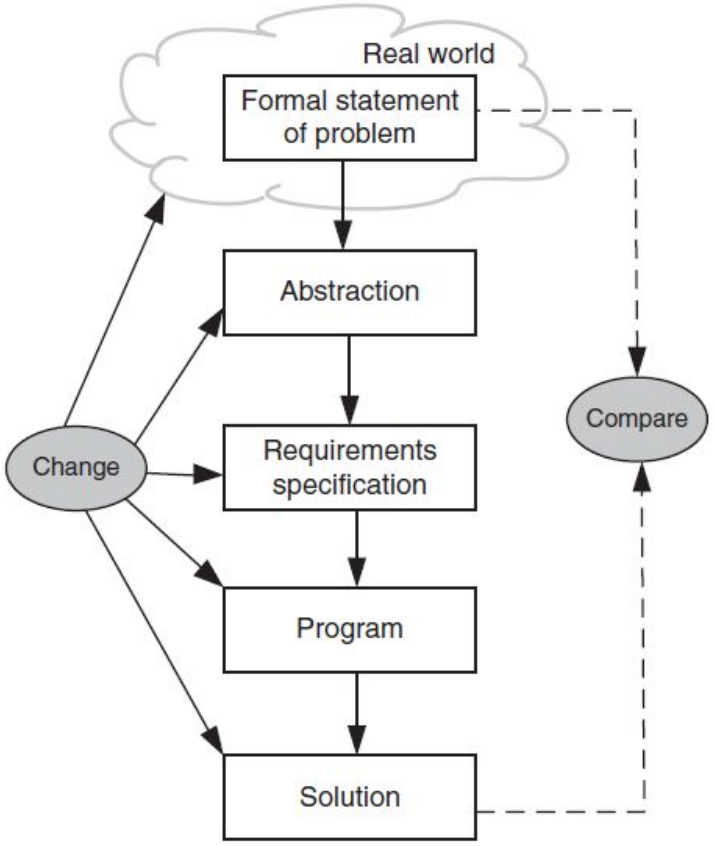

FIGURE 2.6 P-type programs

P-type Programs

- For example, consider a program to play chess.
- Since the rules of chess are completely defined, the problem can be completely specified.
- At each step of the game a solution might involve calculating the various moves and their impacts to determine the next best move.
- However, complete implementation of such a solution may not be possible, because the number of moves is too large to be evaluated in a given time duration.
- Therefore, one must develop an approximate solution that is more practical while being acceptable.

E-type Programs

- **E-type** —programs that are embedded in the real world and become part of it. These are programs that mechanize a human or societal activity, but by becoming part of the world, they also change this activity.
- The acceptance of an E-type program entirely depends upon the stakeholders' opinion and judgment of the solution.
- In fact, the activity and the program become parts of a closed-loop feedback system, feeding on and forcing changes on each other.

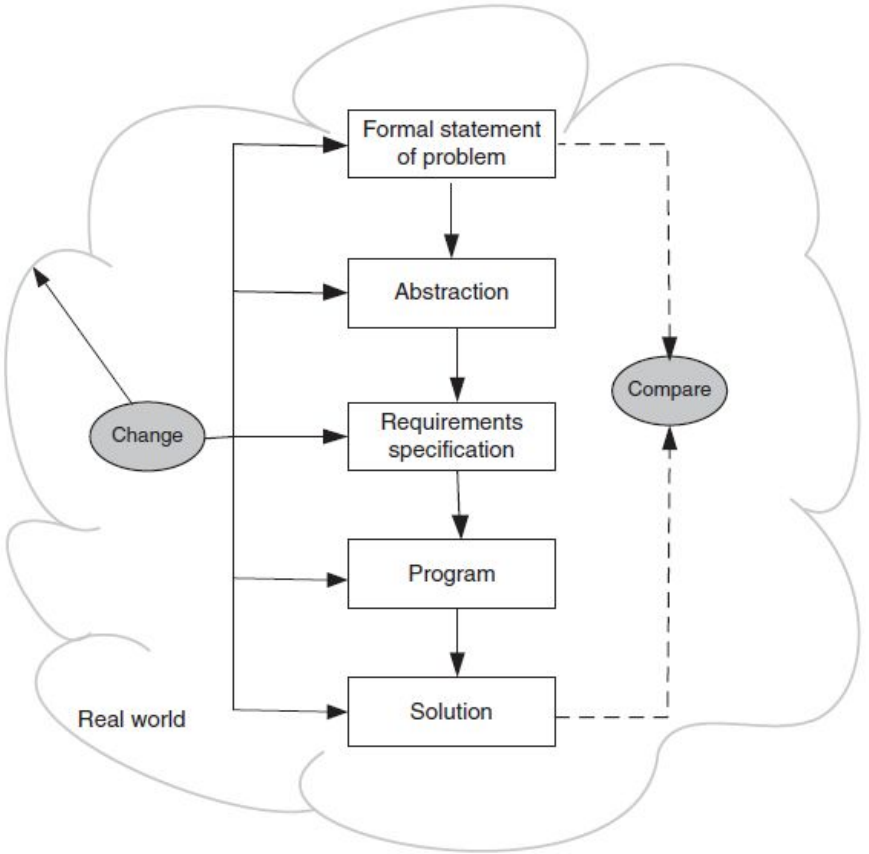

FIGURE 2.7 E-type programs

E-type Programs

- An E-type system is to be regularly adapted to:
  - (i) stay true to its domain of application;
  - (ii) remain compatible with its executing environment; and
  - (iii) meet the goals and expectations of its stakeholders

Laws of Software Evolution

TABLE 2.5 Laws of Software Evolution

Names of the Laws	Brief Descriptions
I. Continuing change (1974)	E-type programs must be continually adapted, else they become progressively less satisfactory.
II. Increasing complexity (1974)	As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.
III. Self-regulation (1974)	The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.
IV. Conservation of organizational stability (1978)	The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.
V. Conservation of familiarity (1978)	The average content of successive releases is constant during the life cycle of an evolving E-type program.
VI. Continuing growth (1991)	To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.
VII. Declining quality (1996)	An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.
VIII. Feedback system (1971–1996)	The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.

Source: Adapted from Lehman et al. [34]. ©1997 IEEE.

Law I - Continuing Change

E-type programs must be continually adapted, else they become progressively less satisfactory.

Law I - Continuing Change

- In case of E-type systems, using the program itself affects the environment, thereby driving the need for modifications.
- Many assumptions are embedded in an E-type program. A subset of those assumptions may be *complete* and *valid* at the initial release of the product; that is, the program performed satisfactorily even if not all assumptions were satisfied.
- As users continue to use a system over time, they gain more experience, and their needs and expectations grow. As the application's environment changes in terms of the number of sophisticated users, a growing number of assumptions become *invalid*.
- Consequently, new requirements and new CRs will emerge. In addition, changes in the real world will occur and the application will be impacted, requiring changes to be made to the program to restore it to an acceptable model.
- Unless a system is continually modified to satisfy emerging needs of users, the system becomes increasingly less useful.

Law II - Increasing Complexity

As an E-type program evolves, its complexity increases unless work is done to maintain or reduce it.

Law II - Increasing Complexity

- As the program evolves, its complexity grows because of the imposition of changes after changes on the program. In order to incorporate new changes, more objects, modules, and sub-systems are added to the system.
- This complements the first law. When the program is changed, the first concern is the needed functionality. Thus the changes are typically done as a patch, disregarding the integrity of the original design.
- The implication is that in order to keep the program operational, it is not enough to invest in changing it — one also needs to invest in repeatedly reducing the complexity again to acceptable levels, in order to facilitate, or at least simplify, subsequent changes.
- The only way to avoid this from happening is to invest in preventive maintenance, where one spends time to improve the structure of the software without adding to its functionality.

Law III - Self Regulation

The evolution process of E-type programs is self-regulating, with the time distribution of measures of processes and products being close to normal.

Law III - Self Regulation

- It reflects a balance between forces that demand change, and constraints on what can actually be done.
- It states that large programs have a dynamics of their own; attributes such as size, time between releases, and the number of reported faults are approximately invariant from release to release because of fundamental structural and organizational factors.
- In an industrial setup E-type programs are designed and coded by a team of experts working in a larger context comprising a variety of management entities, namely, finance, business, human resource, sales, marketing, support, and user process.
- Their actions control, check, and balance the resource usage, which is a kind of feedback-driven growth and stabilization mechanism.
- This establishes a self-controlled dynamic system whose process and product parameters are normally distributed as a result of a huge number of largely independent implementation and managerial decisions.

Law IV - Conservation of Organizational Stability

The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.

(invariant work rate)

Law IV - Conservation of Organizational Stability

- This law suggests that most large software projects work in a “stationary” state,
- Large organizations, that produce large software systems, have inertia, and tend to continue working in the same way which reflects the difficulty in moving staff, making budget changes, etc.
- Even the changes in resources or staffing have small effects on long-term evolution of the software. The average effective global activity rate on an evolving system is almost constant throughout the lifetime of the system.
- To a certain extent management certainly do control resource allocation and planning of activities.
- Circumstances may even arise where providing additional resources may actually reduce the effective rate of productive output as a result of increased communication and other overheads or decreases in process quality.

Law V - Conservation of Familiarity

The average content of successive releases is constant during the life cycle of an evolving E-type program.

Law V - 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.
- One of the factors that determines the progress of a software development is the familiarity of all involved with its goals. The more changes and additions in a particular release, the more difficult it is to for all concerned to be aware, to understand and to appreciate what is required of them.
- In practice, adding new features to a program invariably introduces new program faults due to unfamiliarity with the new functionality and the new operational environment.
- The law suggests that one should not include a large number of features in a new release without taking into account the need for fixing the newly introduced faults.
- Therefore, the average incremental growth in an evolving system remains almost the same. Alternatively, large releases are typically followed by small ones that are required in order to restore stability. Trying to make too many changes at once is very likely to lead to serious problems.

Law VI - Continuing Growth

To maintain user satisfaction with the program over its lifetime, the functional content of an E-type program must be continually increased.

Law VI - Continuing Growth

- As time passes, the functional content of a system is continually increased to satisfy user needs.
- It is useful to note that programs exhibit finite behaviors, which implies that they have limited properties relative to the potential of the application domain.
- Properties excluded by the limitedness of the programs eventually become a source of performance constraints, errors, and irritation.
- To eliminate all those negative attributes, it is needed to make the system grow.

Law VI - Continuing Growth

- It is important to distinguish this law from the first law which focuses on "Continuing Change."
- This is an extension of the first law: functionality is not only adjusted to changing conditions, but also augmented with totally new capabilities that reflect new demands from users, marketing people, and possibly others.
- The first law captures the fact that an E-type software's operational domain undergoes continual changes. Those changes are partly driven by installation and operation of the system and partly by other forces; an example of other forces is human desire for improvement and perfection.
- These two laws—the first and the sixth—reflect distinct phenomena and different mechanisms. When phenomena are observed, it is often difficult to determine which of the two laws underlies the observation.

Law VII - Declining Quality

An E-type program is perceived by its stakeholders to have declining quality if it is not maintained and adapted to its environment.

Law VII - 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.
- An E-Type program must undergo changes in the forms of adaptations and extensions to remain satisfactory in a changing operational domain. Those changes are very likely to degrade the performance and will potentially inject more faults into the evolving program. In addition, the complexity (e.g., the cyclomatic measure) of the program in terms of interactions between its components increases, and the program structure deteriorates.
- There is significant decline in stakeholder satisfaction because of growing entropy, declining performance, increasing number of faults, and mismatch of operational domains. The aforementioned factors also cause a decline in software quality from the user's perspective
- Therefore, it is important to continually undertake preventive measures to reduce the entropy by improving the software's overall architecture, high-level and low-level design, and coding.

Law VII - Declining Quality

- This is a corollary to the first law, and states the results of violating it. Specifically, if the software is not adapted, assumptions that were used in its construction will become progressively less valid, even invalid, and their latent effects unpredictable.
- This law directly follows from the first and the sixth laws.

Law VIII - Feedback System

The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.

Law VIII - Feedback System

- Several laws of software revolution refer to the role of information feedback in the life cycles of software.
- This law is based on the observation that evolution process of the E-type software constitutes a multi-level, multi-loop, multi-agent feedback system:
  (i) multi-loop means that it is an iterative process;
  (ii) multi-level refers to the fact that it occurs in more than one aspect of the software and its documentation; and
  (iii) a multi-agent software system is a computational system where software agents cooperate and compete to achieve some individual or collective tasks.
- Feedback will determine and constrain the manner in which the software agents communicate among themselves to change their behavior