Taxonomy

Idaho State University | Computer Science

Isaac Griffith

CS 4423 and CS 5523
Department of Computer Science
Idaho State University

ROAR

Outcomes

After today's lecture you will:

- Have an understanding of the SPE taxonomy
- Have an understanding of the laws of software evolution
- Have an understanding of software aging and decay
- Have an understanding of how the evolutions of FOSS and COTS systems differ from CSS

SME Taxonomies

CS 4423/5523

SPE Taxonomy

- The abbreviation SPE refers to
  - **S (Specified)**
  - **P (Problem)**, and
  - **E (Evolving)** programs
- In circa 1980, Meir M. Lehman proposed an SPE classification scheme to explain the ways in which programs vary in their evolutionary characteristics.

Lehman observed a key difference between:

- Software developed to meet a fixed set of requirements, and
- Software developed to solve a real world problem which changes with time.
- The observation leads to the identification of types **S (Specified)**, **P (Problem)**, and **E (Evolving)** programs.

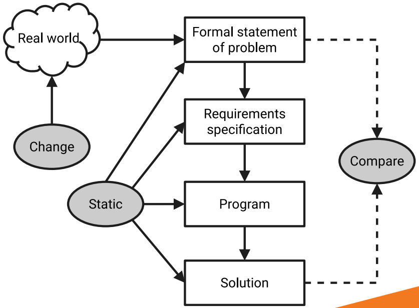
S-Type Programs

Characteristics:

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

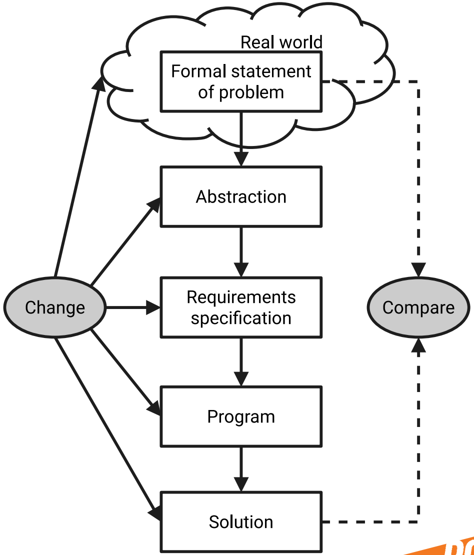
Examples of **S-type programs**:

1. Calculation of the lowest common multiple of two integers
2. Perform matrix addition, multiplication, and inversion



P-Type Programs

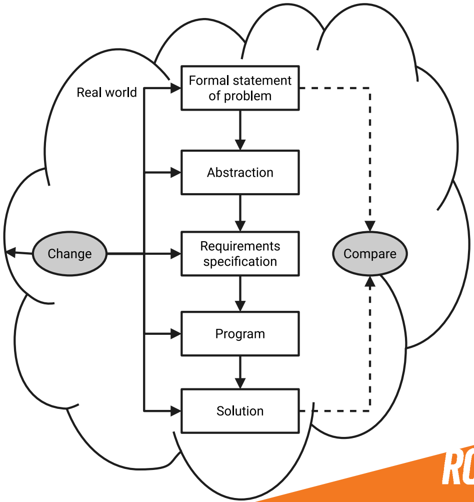
- Based on a practical abstraction of the problem, instead of relying on a completely defined specification.
- Example: A program that plays chess.
- The **P-type program** resulting form the changes cannot be considered a new solution to a new problem. Rather, it is a modification of the old solution to better fit the existing problem.
- In addition, the real world may change, hence the problem changes



E-Type Program

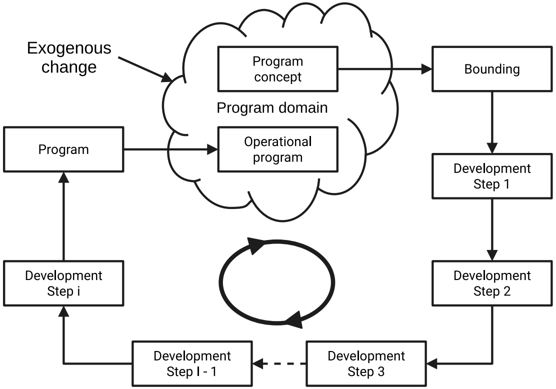
A program is one that is embedded in the real world and it changes as the world does.

- These programs mechanize a human or society activity, make simplifying assumptions, and interface with the external world by requiring or providing services.
- The acceptance of an E-type program entirely depends upon the stakeholders' opinion and judgment of the solution



E-Type Program

- The first characteristic of an E-type program is that the outcome of executing the program is not definitely predictable
- the second characteristic is that program execution changes its operational domain, and the evolution process is viewed as a feedback system.



Laws of Software Evolution

- Lehman formulated a set of observations that he called laws of evolution.
- The laws themselves have evolved from three in 1974 to eight by 1997
- 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 solves a real world problem and have human users."

Laws of Software Evolution

- Lehman's laws were not meant to be used in a mathematical sense, as, say, Newton's laws are used in physics.
- The term "laws" was used because the observed phenomena were beyond the influence of managers and developers.
- The laws were an attempt to study the nature of software evolutions and the evolutionary trajectory likely taken by software.

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 or 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

Laws of Software Evolution

Names of the laws	Brief descriptions
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.

Laws of Software Evolution

First Law Continuing Change: E-type programs must be continually adapted, else they become progressively less satisfactory

- 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.
- 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 change requests will emerge.
- When the updated and modified program is reintroduced into the operational domain, it continues to satisfy user needs for a while.
- Next, more changes occur in the operation environment, additional user needs are identified, and additional change requests are made.
- As a result, the evolution process moves into a vicious cycle.

Laws of Software Evolution

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

- 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.
- These changes lead to a decline in the product quality, unless additional work is performed to arrest the decline.
- The only way to avoid this from happening is to invest in preventive maintenance.
- In preventive maintenance, one spends time to improve the structure of the software without adding to its functionality.

Laws of Software Evolution

Third Law 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.

- This law 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.
- The various groups within the large organization apply constraining information controls and reinforcing information controls influenced by past and present performance indicators.
- 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 are normally distributed as a result of a huge number of largely independent implementation and managerial decisions.

Laws of Software Evolution

Fourth Law Conservation of Organizational Stability: The average effective global activity rate in an evolving E-type program is invariant over the product's lifetime.

- This law suggests that most large software projects work in a "stationary" state, which means that changes in resources or staffing have small effects on long-term evolution of the software.
- To a certain extent management certainly do control resource allocation and planning of activities. However, as suggested by the third law, program evolution is essentially independent of management decisions.
- Activities during the lifecycle of a system are not exclusively decided by management, but by wide spectrum of controls and feedback inputs.

Laws of Software Evolution

Fifth Law Conservation fo Familiarity: The average content of successive releases is constant during the life-cycle of an evolving E-type program.

- The law suggest 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.
- Conservation familiarity implies that maintenance engineers need to have the same high level of understanding of a new release even if more functionalities have been added to it.

Laws of Software Evolution

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

- It is important to distinguish this law from the first law which focuses on “Continuing Change.”
- 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.

Laws of Software Evolution

Seventh Law 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.

- This law directly follows from the first and sixth laws.
- 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 of the program in terms of interactions between its components increases, and the program structure deteriorates.
- The term for this increase in complexity over time is called **entropy**.
- The average rate at which software **entropy** increases is about 1-3 percent per calendar year.

Laws of Software Evolution

Seventh Law 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.

- There is a significant decline in stakeholder satisfaction because of growing entropy, declining performance, increasing number of faults, and mismatch of operational domains.
- The above factors also cause a decline in software quality from the user's perspective.
- The decline of software quality over time is related to the growth in entropy associated with software product **aging or code decay**.
- 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

Software Aging

There are two types of **aging** in software lifecycles:

- **software process execution aging:** It manifests in degradation in performance or transient failures in continuously running the software system,
- **software product aging:** It manifests in degradation of quality of software code and documentation due to frequent changes.

Software Aging

The symptoms of Aging in software are:

- **Pollution:** Pollution means that there are many modules or components in a system which are not used in the delivery of the business functions of the system.
- **Embedded knowledge:** Embedded knowledge is the knowledge about the application domain that has been spread throughout the program such that the knowledge cannot be precisely gathered from the documentation.
- **Poor lexicon:** Poor lexicon means that the component identifiers have little lexical meaning or are incompatible with the commonly understood meaning of the components that they identify.
- **Coupling:** Coupling means that the programs and their components are linked by an elaborate network of control flows and data flows.

Code Decay

The code is said to have decayed if it is very difficult to change it, as reflected by the following three key responses:

1. the cost of the change, which is effective only on the personnel cost for the developers who implement it
2. the calendar or clock time to make the changes
3. the quality of the changed software

It is important to note that code decay is antithesis of evolution in the sense that while the evolution process is intended to make the code better, changes are generally degenerative thereby leading to code decay.

Laws of Software Evolution

Eighth Law Feedback System: The evolution processes in E-type programs constitute multi-agent, multi-level, multi-loop feedback systems.

The eighth law is based on the observation that evolution process of the E-type software constitutes a multi-level, multi-loop, multi-agent feedback system:

❶ multi-loop means that it is an iterative process

❷ multi-level refers to the fact that it occurs in more than one aspect of the software and its documentation

❸ 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.

Empirical Studies

- In circa 1976, Belady and Lehman studied 20 releases of the OS/360 operating system.
- The results of their study led them to postulate five laws of software evolution:
  - **Continuing change**
  - **Increasing complexity**
  - **Self regulation**
  - **Conservation of organizational stability**
  - **Conservation of familiarity**
- Yuen, a collaborator of Lehman, notes that the last three of the aforementioned laws are more based upon those of human organizations involved in the maintenance process rather than the properties of the software itself.

Empirical Studies

- In project FEAST (Feedback, Evolution, And Software Technology), Lehman and his colleagues studied evolution of releases from four CSS systems:
  1. Two operating systems (IBM OS/360 and ICL VME OS)
  2. One financial system (Logica's FW banking transaction system)
  3. A real-time telecommunication system (Lucent Technologies)
- The studies suggest that during the maintenance process a system tracks a growth curve that can be approximated either as linear or inverse-square.
- The inverse-square model represents the growth phenomena as an inverse square of the continuing effort

Idaho State University
Computer Science

Empirical Studies

- Those trends increase the confidence of validity of the following six laws:
  - I Continuing change
  - II Increasing complexity
  - III Self regulation
  - V Conservation of familiarity
  - VI Continuing growth
  - VII Feedback system
- Confidence in the seventh law “Declining quality” is based on the theoretical analysis, whereas the fourth law “Conservation of organizational stability” is neither supported nor falsified based on the metric presented

ROAR

Practical Implications of the Laws

- Lehman suggested more than 50 rules based on the eight laws.
- Those 50+ rules are put into three broad categories:
  1. Assumptions Management
  2. Evolution Management
  3. Release Management

Evolution of FOSS Systems

- The FOSS movement is attributed to Richard M. Stallman, who started the GNU (O.S.) project
- FOSS – also referred to as FLOSS (Free/Libre/Open Source Software)
- FOSS is a class of software that is both free software and open source
- FOSS have lots of new characteristics. Eric S. Raymond concisely documented the FOSS approach in an article entitle “The Cathedral and the Bazaar”

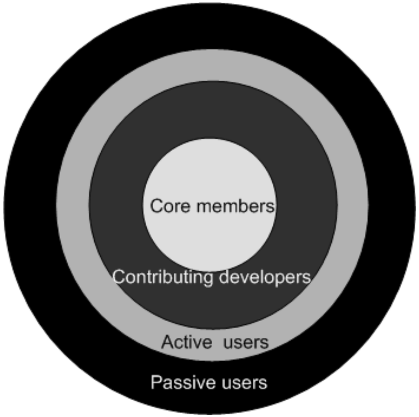
FOSS is made available to the general public with either relaxed or non-existent intellectual property restrictions.

The **free** emphasizes the freedom to modify and redistribute under the terms of the license while **open** emphasizes the accessibility to the source code

Evolution of FOSS Systems

- There are huge differences between the evolutions of FOSS based software and CSS based software in terms of:
  - Team structure, Process, Releases, and Global factors
- Example FOSS:
  - Linux: started by Linus Torvalds

Onion Model of FOSS Dev Structure



Evolution of FOSS Systems

- In circa 1988, Pirzada pointed out the differences between the evolution of the Unix OS and systems studied by Lehman
  - Pirzada argued that the differences in academic and industrial software development could lead to differences in the evolutionary pattern

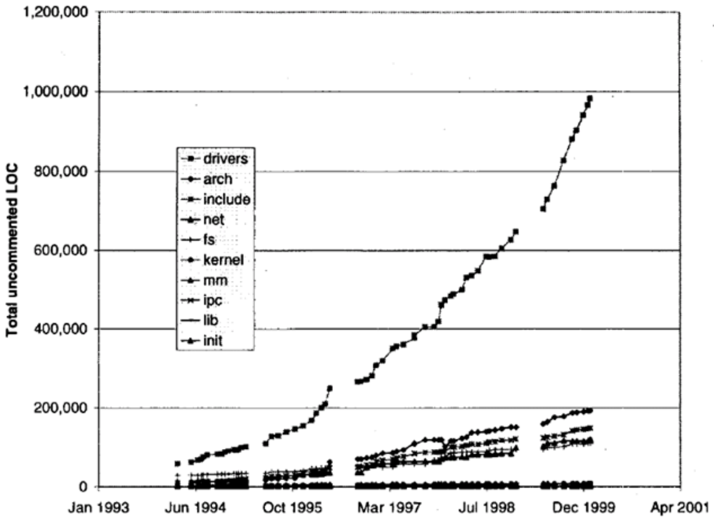
- In circa 2000, empirical study of FOSS was conducted by Godfrey and Tu
  - They found that the grow trends from 1994-1999 for the evolution of FOSS Linux OS to be super-linear, that is greater than linear

- Robles and his collaborator concluded that Lehman's laws, 3, 4, and 5 are not fitted to large scale FOSS systems such as Linux

Idaho State University
Computer Science

Evolution of FOSS Systems

Growth of major Subsystems of the Linux OS



ROAR

Maintenance of COTS Systems

- A COTS component is defined as:
  - A unit of composition with contractually specified and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.
- The use of COTS components is increasing in modern software development because of the following reasons:
  1. there is significant gain in productivity due to reusing commercial components
  2. the time to market the product decreases
  3. the product quality is expected to be high, assuming that the COTS components have been well tested
  4. there is efficient use of human resources due to the fact that development personnel will be freed up for other tasks

Maintenance of COTS Systems

- The black-box nature of COTS components prevents system integrators from modifying the components to better meet user’s requirements
- The integrators have no visibility of the details of the components, their evolutions, or their maintenance
- The only source code being written and modified by the integrators is what is needed for integrating the COTS based systems
- This includes code for:
  1. Tailoring and Wrapping the individual components
  2. Glue code required to put the components together

Maintenance of COTS Systems

Wrapper: It is a piece of code that one builds to isolate the underlying components from other components of the system

Glue: A glue component provides the functionality to combine different components

Tailoring: Components tailoring refers to the ability to enhance the functionality of a component

- Tailoring is done by adding some elements to a component to enrich it with a functionality not provided by the vendor
- Tailoring does not involve modifying the source code of the component

Why Maintenance of CBS is Difficult?

- Frozen functionality
- Incompatibility of upgrades
- Trojan horses
- Unreliable COTS components
- Defective middleware

Maintenance Activities for CBSs

- Vigder and Kark have surveyed several organizations maintaining systems with a significant portion of COTS elements
- In their study, they identified the following cost-drivers:
  - Component reconfiguration
  - Testing and debugging
  - Monitoring of systems
  - Enhancing functionality for users
  - Configuration management

Design Properties of CBSs

- The architecture of a CBS has significant impact on its maintainability
- The main areas influencing CBS maintainability are as follows:
  1. choice of the components
  2. architecture and design used to perform integration on the components
- **Component selection:** The following attributes of components effect the evolution maintenance of CBSs:
  1. Openness of components
  2. Tailorability of components
  3. Available support community

Idaho State University
Computer Science

Design Properties of CBSs

The following design attributes of a maintainable CBS have been identified by Vigder and Kark:

- Encapsulated component collaborations
- Controlled component interfaces
- Controlled component dependencies
- Minimal component coupling
- Consistent failure handling
- High level of visibility
- Minimal build and deployment effort

ROAR

Idaho State University
Computer Science

For Next Time

- Review EVO Chapter 2.3 - 2.5
- Read EVO Chapter 3.1 - 3.7
- Watch the Lecture 04
- 4423
  - Start Homework 01
  - Start forming Project Teams
  - Complete Quiz 01

Are there any questions?