SWE-417 SOFTWARE REENGINEERING

Software Engineering Department

Sir Syed University of Engineering & Technology

Week No. 6

CONTENTS

- Taxonomy Of Software Maintenance And Evolution
  - SPE Taxonomy
  - Fundamental concepts of software aging, hazard & mishap
- Reuse-Oriented Models

SPE TAXONOMY

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

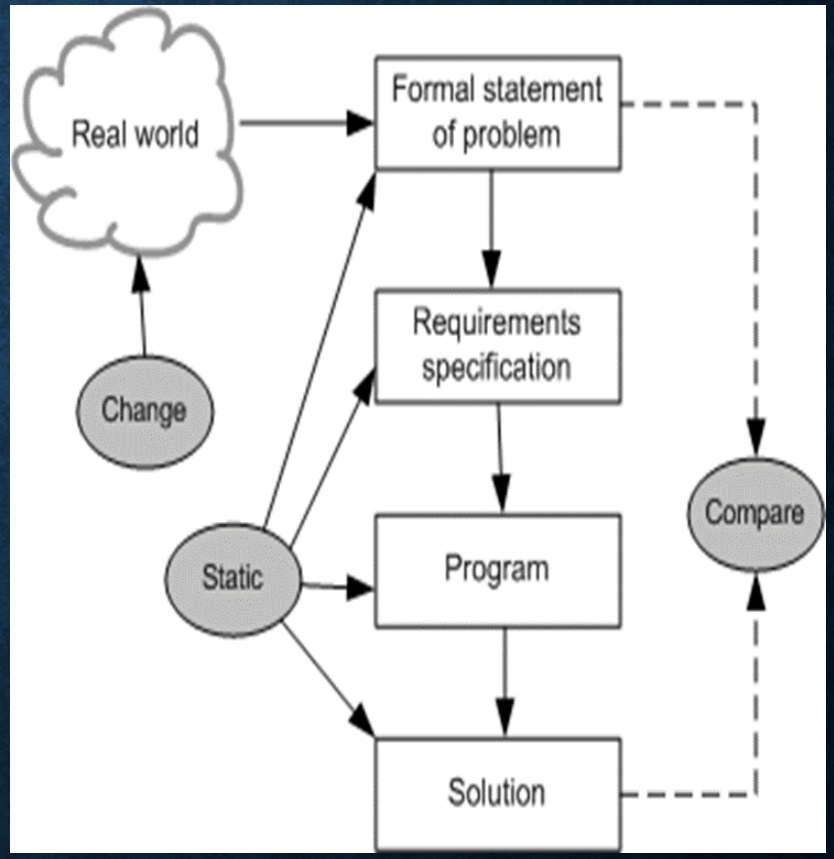
SPE TAXONOMY

- S-type (Specified) programs have the following characteristics:
  - All the non-functional 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.

Examples of **S-type programs**:

(i) Calculation of the lowest common multiple of two integers.

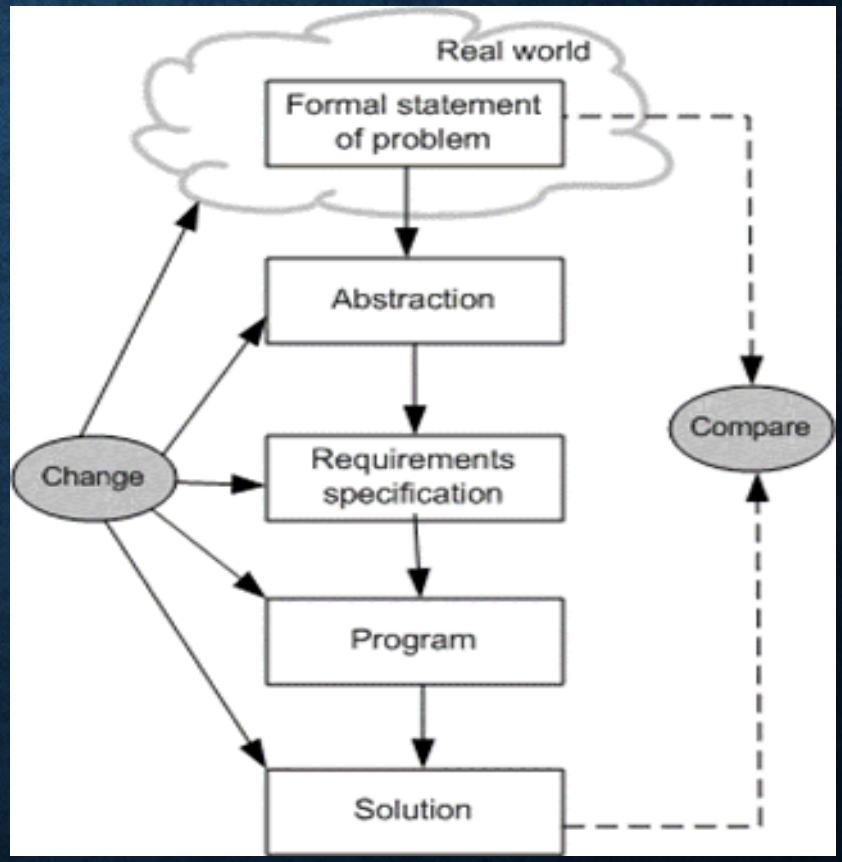
(ii) Perform matrix addition, multiplication, and inversion.



S-type programs

SPE TAXONOMY

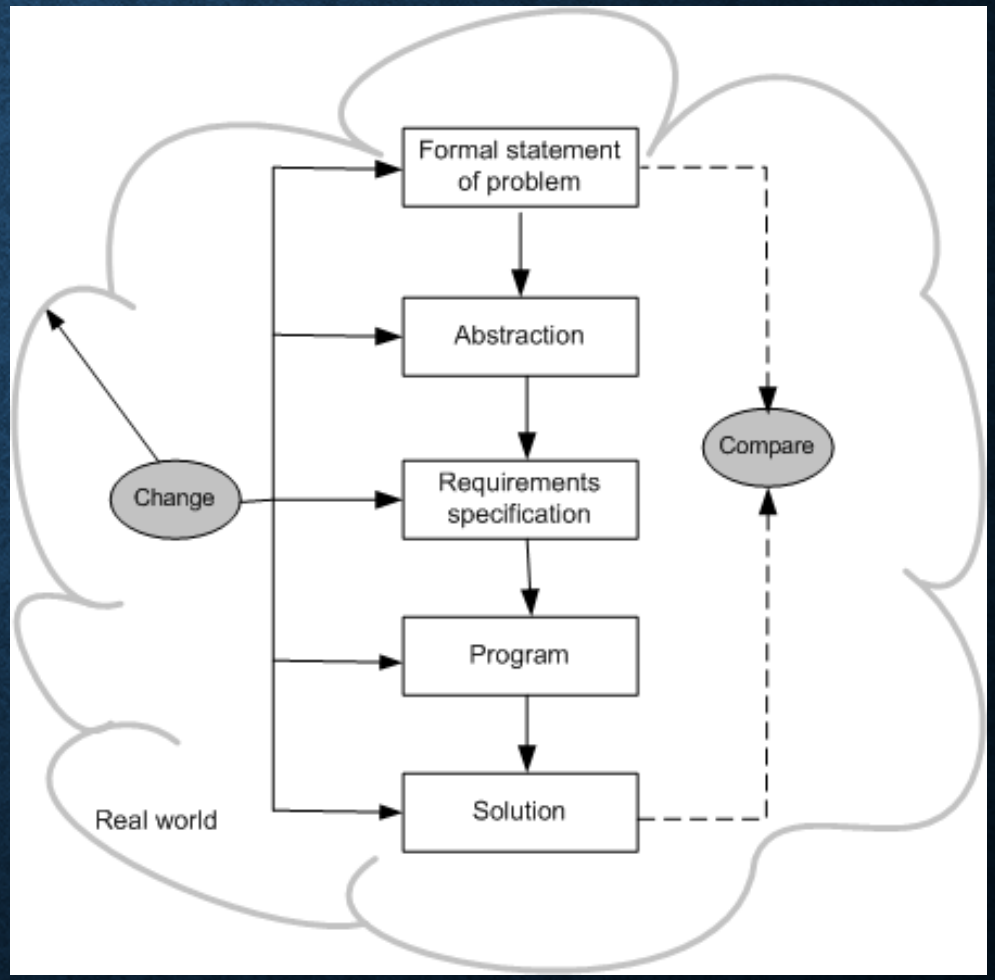
- P-type (Problem) program is based on a practical abstraction of the problem, instead of relying on a completely defined specification.
- Example: A program That play chess.
- The P-type program resulting from the changes cannot be considered a new solution to a new problem. Rather, it is a modification of the old solution to better fit the existing problem.
- In addition, the real world may change, hence the problem changes.



P-type programs

SPE TAXONOMY

- An E-type (Evolving/Embedded) 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.
- Example: Banking Software System, Air Traffic Control Systems etc.



E-type programs

SOFTWARE FAULT CLASSIFICATION

- Software faults, also known as software defects or bugs, are classified in several ways to aid in identifying, analyzing, and resolving the issues effectively. Here's an overview of common classifications:

1. By Type of Fault
   - Syntax Errors
   - Semantic Errors
   - Logic Errors
   - Runtime Errors
   - Interface Errors

SOFTWARE FAULT CLASSIFICATION

2. By Severity
   - Critical Faults
   - Major Faults & Minor Faults
3. By Lifecycle Phase
   - Requirement Faults
   - Design Faults
   - Implementation Faults
   - Testing Faults
   - .Maintenance Faults

SOFTWARE FAULT CLASSIFICATION

4. By Source
   - Algorithmic Faults
   - Resource Management Faults
   - Configuration Faults
   - Human Errors
5. By Detectability
   - Detected
   - Undetected

Importance of Fault Classification: Classifying software faults helps teams prioritize fixes, improve quality control, and refine testing and development processes.

SOFTWARE FAULT CLASSIFICATION

Some extended Software Fault Classification:

- Bohrbugs
- Mandelbugs
- Aging related Bugs

- Bohrbugs: Software bugs that are reproducible, easily found and (often) fixed during the testing and debugging phase
- Mandelbugs: Software bugs that are hard to find and fix; (often) remain in the software during the operational phase. These bugs may never be fixed, but if the operation is retried or the system is rebooted, the bugs may not manifest themselves as failures.
- Aging related Bugs: Another cause software failures is resource exhaustion, e.g., memory leakage, swap space fragmentation. Software appears to "Age" due to resource exhaustion

FUNDAMENTAL CONCEPTS OF SOFTWARE AGING

- **Software aging** is a phenomenon that causes system performance degradation and eventual failures.
- A general characteristic of this phenomenon is the fact that, as the run-time of the system or process increases, its failure rate also increases.
- Software aging is the tendency for software to fail or cause a system failure after running continuously for a certain time, or because of ongoing changes in systems surrounding the software.
- Failure can take the form of incorrect service (e.g., erroneous outcomes), no service (e.g., halt and/or crash of the system), or partial failure (e.g., gradual increase in response time).

FUNDAMENTAL CONCEPTS OF SOFTWARE AGING

- Aging effects can also be classified into **volatile** and **nonvolatile** effects.
- They are considered volatile if they are removed by re-initialization of the system or process affected, for example via a system reboot. In contrast, non-volatile aging effects still exist after reinitializing of the system/process.
- Aging effects in a system can only be detected while the system is running, by monitoring **aging indicators**. Aging indicators are markers for aging detection, like antigens are markers to detect cancer disease.

FUNDAMENTAL CONCEPTS OF SOFTWARE AGING: AGING INDICATORS

- Aging Indicators: To detect and monitor software aging, various aging indicators are used. These indicators can include system performance metrics (e.g., response time, throughput), resource consumption (e.g., CPU usage, memory usage), error rates, system logs, and system health monitoring data.
- Tracking these indicators helps identify signs of aging and potential areas for improvement.

Tools for Aging Indicators:

- Performance Monitoring Tools: New Relic, AppDynamics
- Resource Monitoring Tools: Nagios, Zabbix
- Profiling and Performance Analysis Tools: Java VisualVM, Visual Studio Profiler (for .NET applications)

KEY CONCEPTS OF SOFTWARE AGING

- Resource Exhaustion:
  - ✔ Over time, software may consume increasing amounts of system resources (e.g., memory, disk space, file handles) due to issues like memory leaks, where memory is allocated but never released. This can result in system slowdowns, crashes, or unresponsiveness.
- Memory Leaks and Fragmentation:
  - ✔ Memory leaks occur when a program does not release allocated memory after it's no longer needed. Memory fragmentation, on the other hand, happens when free memory becomes scattered in small blocks, leading to inefficient memory utilization and potential crashes.

KEY CONCEPTS OF SOFTWARE AGING

- Data Corruption:
  - Continuous use of software may lead to accumulated data corruption or "bit rot," where data becomes erroneous over time. This can degrade software functionality or cause unpredictable behavior.
- Environmental Drift:
  - As the underlying hardware, operating system, and external libraries evolve, the software may no longer operate optimally or may even become incompatible.
- Increasing Code Complexity:
  - Over time, codebases often grow and become more complex due to new features, patches, and changes. This can introduce new bugs, slow down performance, and make the software more challenging to maintain.

STRATEGIES TO PREVENT SOFTWARE AGING

- **Regular System Rejuvenation**

"Software rejuvenation: Proactive fault management technique aimed at postponing/ preventing crash failures and/or performance degradation".

- ✔ Rebooting: Periodically restarting the software or system can clear up memory leaks, refresh resource states, and mitigate some aging effects.
- ✔ Check-pointing and Recovery: Implementing a checkpoint system where the application state is saved periodically allows for quick recovery in case of a failure, helping to maintain continuous availability.

STRATEGIES TO PREVENT SOFTWARE AGING

- **Memory Leak Detection and Management**
  - Automated Leak Detection Tools: Tools like Valgrind, Purify, and AddressSanitizer can be used to detect memory leaks during development and testing.
  - Garbage Collection: For languages that support it (e.g., Java, C#), garbage collection can manage memory more effectively and help minimize memory leaks.

Note:

- Software aging is an inevitable challenge but can be managed effectively through proactive measures. Regular maintenance, continuous monitoring, rejuvenation techniques, and keeping the system modular and updated are all effective ways to mitigate aging and extend software life.

EXAMPLE OF SOFTWARE AGING

- Scenario: Memory Leaks
  Description: Over time, a software application may experience memory leaks, where memory allocated for certain operations is not properly released. This can cause the application to consume excessive memory resources, leading to performance degradation or even crashes.
- Solution: Regularly perform memory profiling and debugging to identify and fix memory leaks. Use tools like memory analyzers to detect and resolve memory leaks in the code. Additionally, follow best practices for memory management, such as deallocating resources properly and optimizing memory usage.

FUNDAMENTAL CONCEPTS OF HAZARD & MISHAP

- Hazard and mishap are related terms commonly used in the context of safety and risk management.
- A hazard refers to any potential source or situation that has the potential to cause harm, injury, damage, or adverse effects to people, property, or the environment.
- A mishap, on the other hand, refers to an unintended event, accident, or incident that causes harm, damage, or unintended consequences. It represents the actual occurrence or manifestation of an undesired outcome.
- Hazards are the potential risks or dangers that exist in a situation, while mishaps are the actual incidents or accidents that result from those hazards.

EXAMPLE OF HAZARD & MISHAP

For each of the following situations, explain whether it is a hazard or a mishap:

Scenario: Water in a swimming pool becomes electrified.

Solution: This situation can be considered a hazard. The presence of electricity in the water poses a potential risk of electric shock to individuals who come into contact with it. It represents a potential danger or source of harm, highlighting the hazard of electric shock in the swimming pool.

EXAMPLE OF HAZARD & MISHAP

- Scenario: E-commerce Checkout Failure
- Solution: This situation can be considered a Mishap. Because during a peak shopping period, an e-commerce website experiences an issue where the checkout process fails to complete transactions for a significant number of customers. This results in frustrated customers, lost sales, and damage to the company's reputation.

REUSE-ORIENTED MODELS

- One obtains a new version of an old system by modifying one or several components of the old system and possibly adding new components. As a consequence, the new system is likely to reuse many components of the old system.
- A new version of the system can be created after the maintenance activities are implemented on some of the old system's components. Based on this concept, three process models for maintenance have been proposed by Basili:
  - Quick fix model
  - Iterative enhancement model
  - Full reuse model

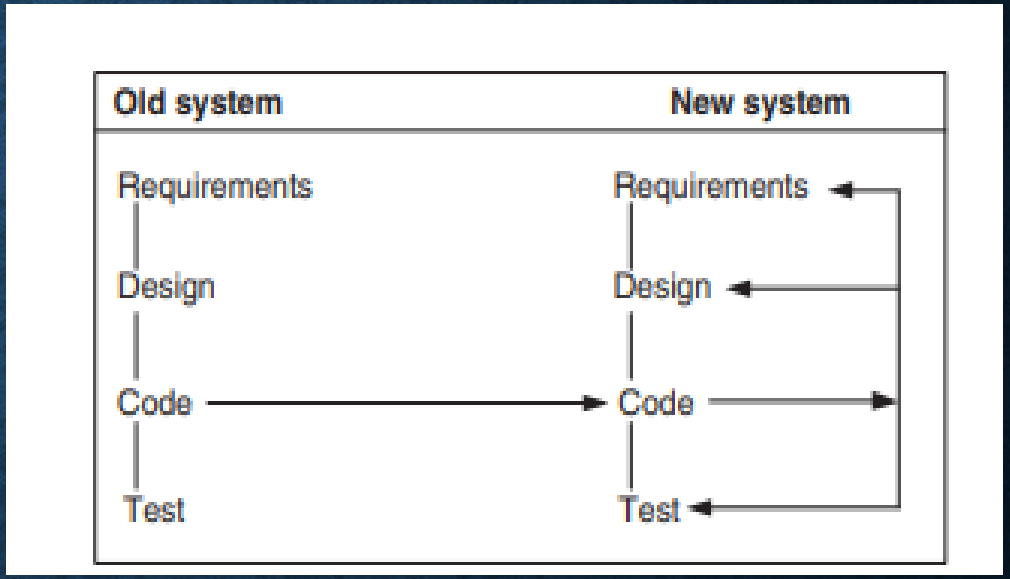
QUICK FIX MODEL

In this model, as illustrated in Figure,

(i) source code is modified to fix the problem;

(ii) necessary changes are made to the relevant documents; and

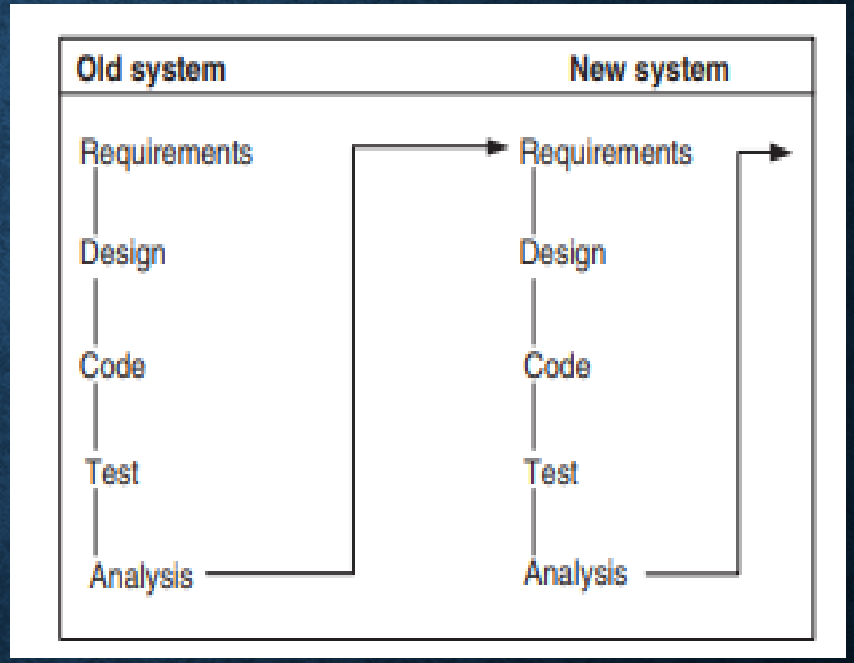
(iii) the new code is recompiled to produce a new version.



ITERATIVE ENHANCEMENT MODEL

The iterative enhancement model, explained in Figure, shows how changes flow from the very top-level documents to the lowest-level documents.

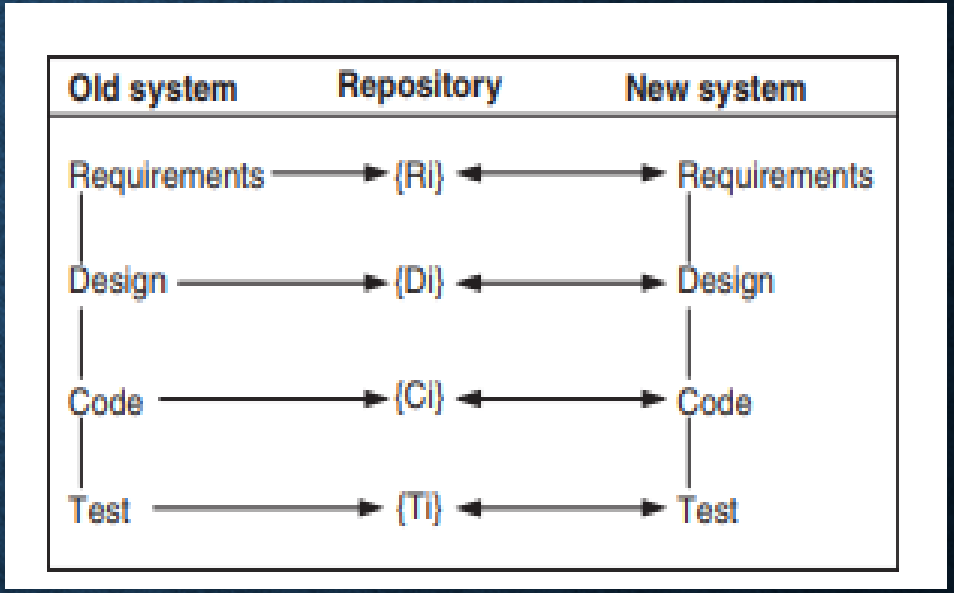
The model works as follows:
(i) It begins with the existing system's artifacts, namely, requirements, design, code, test, and analysis documents.
(ii) It revises the highest-level documents affected by the changes and propagates the changes down through the lower-level documents.
(iii) The model allows maintainers to redesign the system, based on the analysis of the existing system.



FULL REUSE MODEL

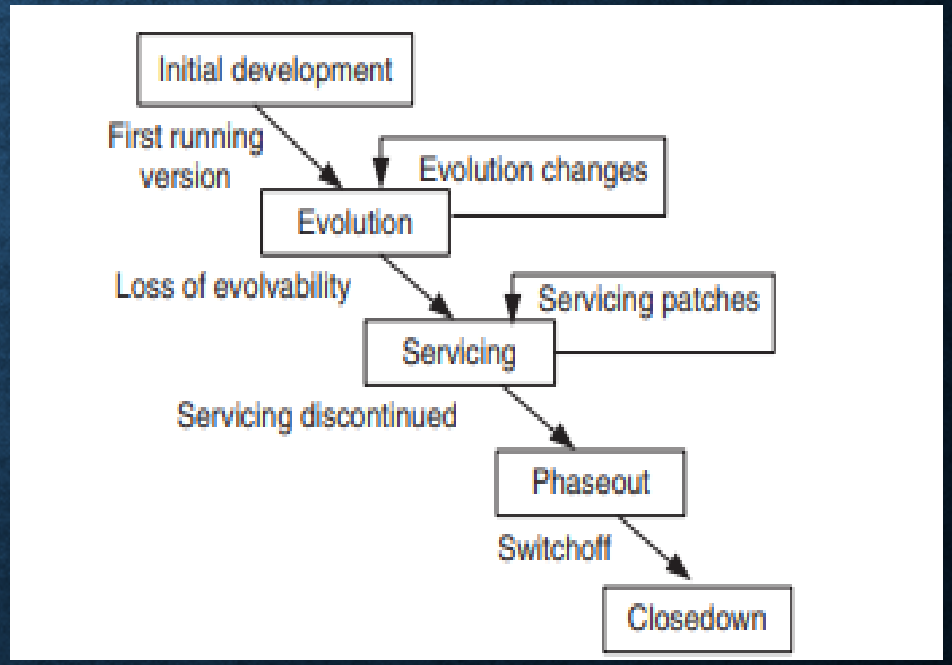
The model illustrated in Figure shows maintenance as a special case of reuse-based software development.

Full reuse comprises two major steps:
(i) perform requirement analysis and design of the new system; and
(ii) use the appropriate artifacts, such as requirements, design, code, and test from any earlier versions of the old system.

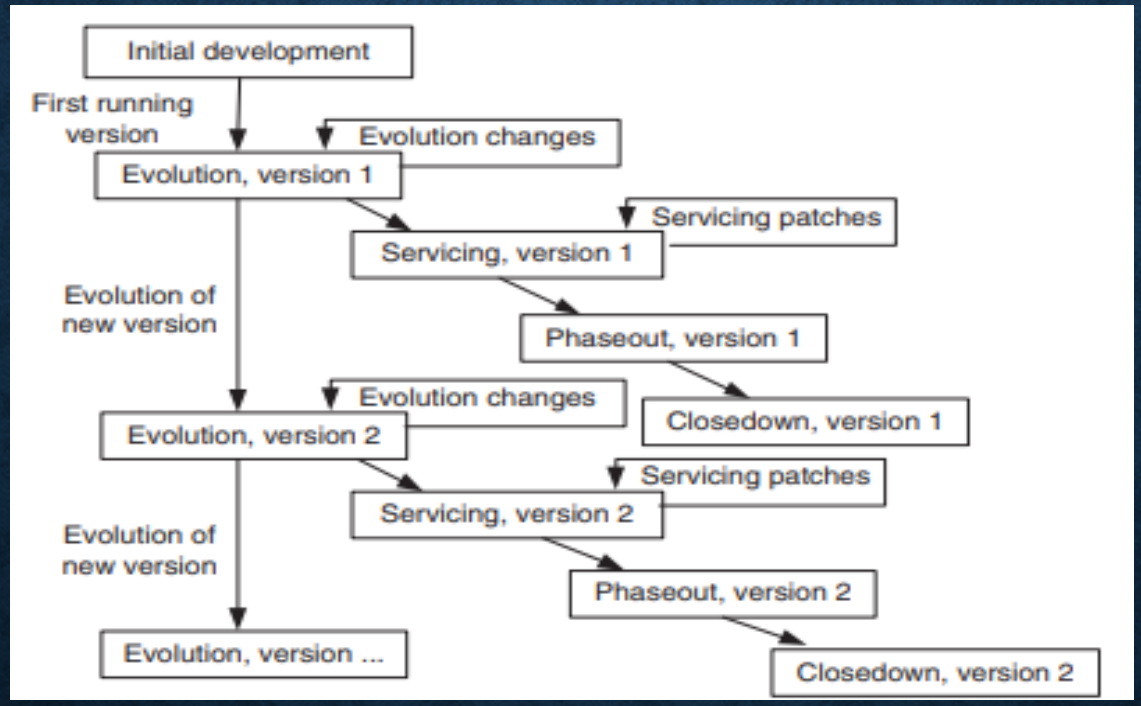


THE STAGED MODEL FOR CLOSED SOURCE SOFTWARE

1. Initial development: Develop the first functioning version of the software.
2. Evolution. The developers improve the functionalities and capabilities of the software to meet the needs and expectations of the customer.
3. Servicing: The developers only fix minor and emergency defects, and no major functionality is included.
4. Phaseout: In this phase, no more servicing is undertaken, while the vendors seek to generate revenue as long as possible.
5. Closedown. The software is withdrawn from the market, and customers are directed to migrate to a replacement



THE VERSIONED STAGED MODEL



The versioned staged model for the CSS life cycle

SUMMARY

- SPE Taxonomy provides a structured approach to organizing software engineering processes, from development to quality assurance and risk management.
- Software Aging involves gradual performance degradation due to resource exhaustion, data corruption, environmental changes, and increased code complexity.
- Hazards are conditions that could cause system issues, while mishaps are actual events of harm or failure, often arising from unaddressed hazards.
- Three process models for maintenance have been proposed by Basili has been discussed: Quick fix model, Iterative enhancement model & Full reuse model