

# SOEN 6431 Software Maintenance and Program Comprehension



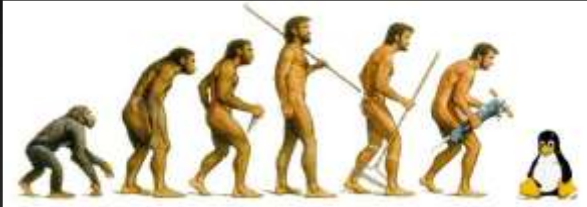
**GINA CODY**  
SCHOOL OF ENGINEERING  
AND COMPUTER SCIENCE

**Juergen Rilling, PhD**  
Professor  
Department of Computer Science  
and Software Engineering

MAILING ADDRESS  
1455 DE MAISONNEUVE BLVD. W., EV-3.211  
MONTREAL, QUEBEC, CANADA H3G 1M8  
**CONCORDIA.CA**

514-848-2424 ext. 3016  
juergen.rilling@concordia.ca  
concordia.ca/ginacody  
STREET ADDRESS  
1515 Ste. Catherine St. W., EV-3.211

## Week 1 – Introduction Software Evolution



# Evolution and Maintenance

"Evolution is what happens while you're busy making other plans."

*Usually, we consider **evolution to begin once the first version has been delivered:***

*-Maintenance is the planned set of tasks to effect changes.*

*Evolution is what actually happens to the software.*

# Terminology

## Maintenance/Evolution

- Software Maintenance – When changing an existing system after the initial release.
- The maintenance phase – the time from first release to retirement of a system, often many years.
- Almost all industrial SW development is “maintenance”, systems are seldom created from scratch
- The term “Software Evolution” is often used instead

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.”

# Importance of Software Evolution

---

Organizations have made huge investments in their software systems – they are **critical business assets**.

---

To **maintain the value** of these assets to the business, they must be changed and updated.

---

The **majority of the software budget** in large companies is devoted to evolving existing software rather than developing new software.

---

Studies indicate that up **to 75% of all software professionals** are involved in some form of maintenance/ evolvability activity. This number is increasing...

# 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 solves a real-world problem and have human users.

# Software Evolution: Laws of Lehman

---

***Continuing change*** (1<sup>st</sup>) – A system will become progressively less satisfying to its user over time, unless it is continually adapted to meet new needs.

---

***Increasing complexity*** (2<sup>nd</sup>) – A system will become progressively more complex, unless work is done to explicitly reduce the complexity.

---

***Self-regulation*** (3<sup>rd</sup>) – 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*** (4<sup>th</sup>) – 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.

# Software Evolution: Laws of Lehman

---

***Conservation of familiarity*** (5<sup>th</sup>) – The amount of new content in each successive release of a system tends to stay constant or decrease over time.

---

***Continuing growth*** (6<sup>th</sup>) – The amount of functionality in a system will increase over time, in order to please its users.

---

***Declining quality*** (7<sup>th</sup>) – A system will be perceived as losing quality over time, unless its design is carefully maintained and adapted to new operational constraints.

---

***Feedback system*** (8<sup>th</sup>) – Successfully evolving a software system requires recognition that the development process is a multi-loop, multi-agent, multi-level feedback system.

# Software Evolution: Open-Source Systems (FOSS<sup>\*1</sup>) System

- Self-regulation (3<sup>rd</sup>),
- Conservation of organizational stability (4<sup>th</sup>)
- Conservation of familiarity

---

In 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 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 system such as Linux.

---

<sup>\*1</sup>**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."



## Applicability of Lehman's laws

**Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.**

- Confirmed in early 2000's by work by Lehman on the FEAST project.

**It is not clear how they should be modified for:**

- Systems that incorporate a significant number of COTS components
- Small organisations
- Medium sized systems.

# Terminology: Reengineering

Jacobson and Lindstorm defined following formula:

Reengineering = Reverse engineering +  $\Delta$  + 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  $\Delta$  captures alteration that is change of the system.

# Re-engineering/ Reverse Engineering

---

**Re-engineering** = Reverse Engineering (+ actions/intentions) + Forward Engineering

---

**Reverse engineering** = extracting more abstract representations (e.g. extract UML from code)

---

**Forward engineering** = the opposite, develop more detailed design (or code) from higher level documents

---

**Actions/intentions** = e.g. re-structuring, documentation, verification, performed on a higher-level representation of the system (a model), resulting from the reverse engineering.

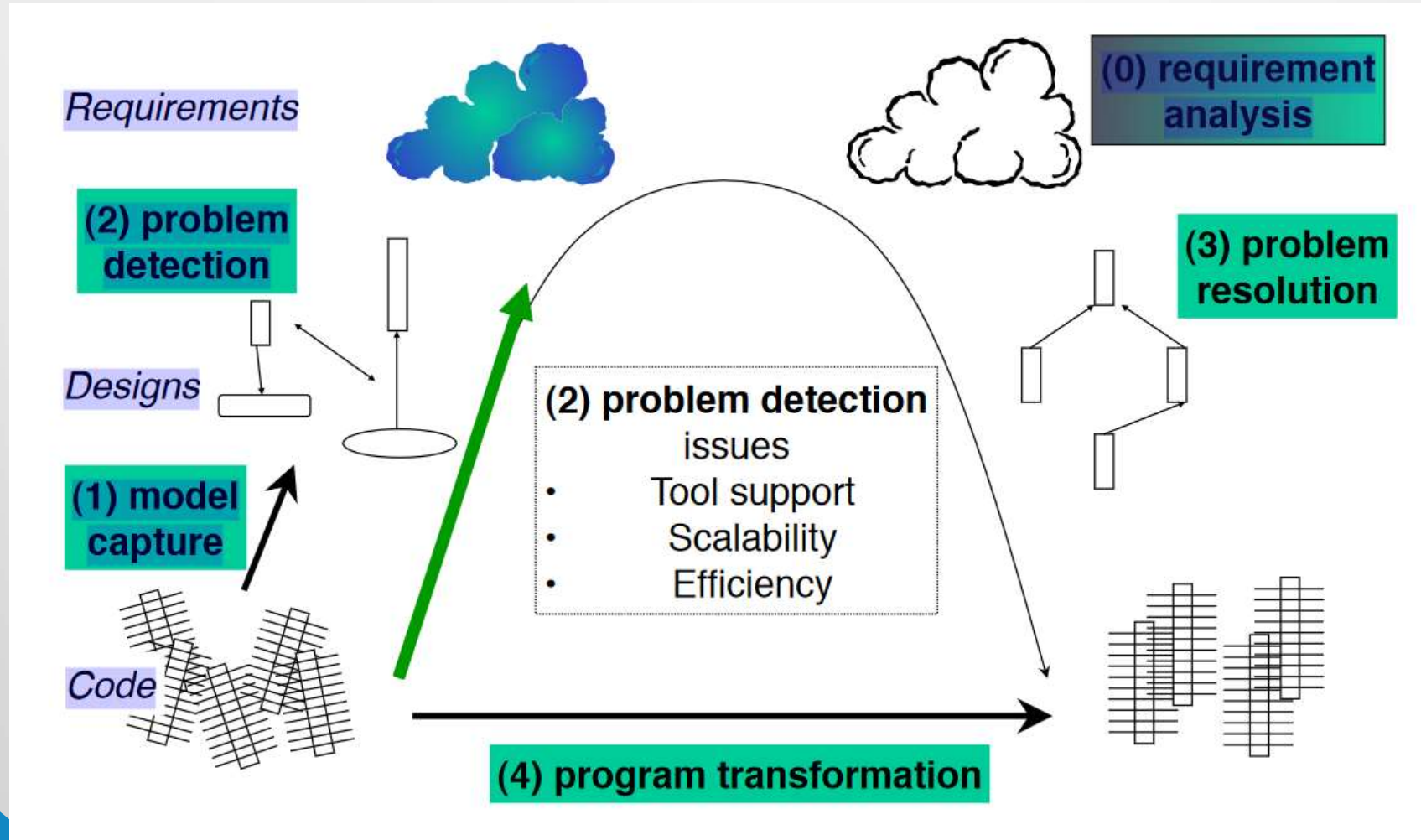
## Terminology: Reengineering

**Reengineering** is 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.”

# Re-engineering Lifecycle



# Legacy System

- *A sum of money, or a specified article, given to another by will; anything handed down by an ancestor or predecessor. — Oxford English Dictionary*
- A **legacy system** is a piece of software that:
  - you have *inherited* **and** *is valuable* to you.
- Typical **problems** with legacy systems:
  - original developers *not available*
  - *outdated* development methods used
  - extensive patches and *modifications* have been made
  - *missing* or outdated documentation

=> So further evolution and development may be prohibitively expensive !

# Terminology: How to manage 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.



# What is your opinion?

- Can we compare: Aging of software with aging of people?
  - Does software age?
  - What are characteristics of aging?
  - Are they applicable in some form to both software and people?







## Opinion #1

*"It does **not** make sense to talk about software aging!"*



# Software “Aging” – Opinion #1

*“It does **not** make sense to talk about software aging!”*

- ☐ *Software is a mathematical product; mathematics does not decay with time.*
- ☐ *if a theorem was correct 200 years ago, it will be correct tomorrow.*
- ☐ *If a program is correct today, it will be correct 100 years from now.*
- ☐ *If a program is wrong 100 years from now, it must have been wrong when it was written.*

***All of the above statements are true, but are they really relevant?***



## Software “Aging” – Opinion #2

*“It does **not** make sense to talk about software aging!”*

# Opinion #2

## Software Does Age

*Software aging is gaining in significance because:*

- ☐ *of the growing economic importance of software,*
- ☐ *software is the “**capital**” of many high-tech firms. The authors and owners of new software products often look at aging software with disdain.*
- ☐ *“If only the software had been designed using today’s languages and techniques ...”*
- ☐ *Like a young jogger scoffing at an 86 year old man (ex-champion swimmer) and saying that he should have exercised more in his youth!*

*There are **two types** of causes for software aging:*

## Software Aging - Causes

**Lack of Movement:** *Aging caused by the failure of the product's owners to modify it to meet changing needs.*

- Unless software is frequently updated, its user's will become dissatisfied and change to a new product.
- Excellent software developed in the 60's would work perfectly well today, but nobody would use it.
- That software has aged even though nobody has touched it. Actually, it has aged because nobody bothered to touch it.

**Ignorant Surgery:** *Aging caused as a result of changes that are made.*

*This "one-two punch" can lead to rapid decline in the value of a software product.*

- One must upgrade software to prevent aging.
- Changing software can cause aging too.
- Changes are made by people who do not understand the software.>Hence, software structure degrades.
- After many such changes nobody understands the software:
  - the original designers no longer understand the modified software,
  - those who made the modification still do not understand the software.
- Changes take longer and introduce new bugs.
- Inconsistent and inaccurate documentation makes changing the software harder to do.

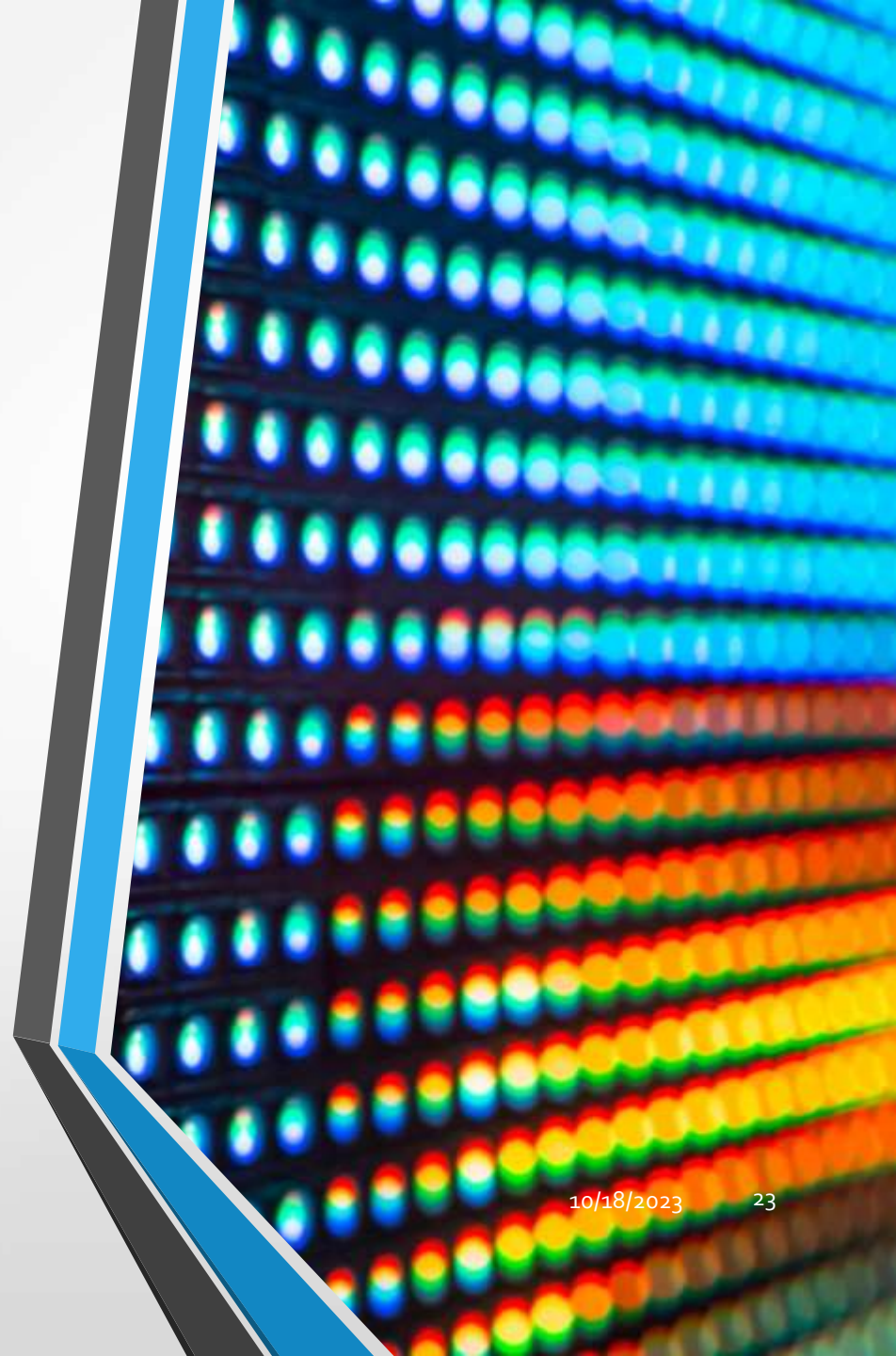
# Software Aging

“Programs, like people, get old. We can’t prevent aging, but we can understand its causes, take steps to limit its effects, temporarily reverse some of the damage it has caused, and prepare for the day when the software is no longer viable. ... (We must) lose our preoccupation with the first release and focus on the long-term health of our products.”

Parnas, D.L. Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on Volume , Issue , 16-21 May 1994 Page(s):279 - 287



# Video



# Types of Software Maintenance



## Two Types of Maintenance Activities – Bug Fixing

# Bug Fixing

- **Corrective Maintenance**
  - Identify and remove defects
  - Correct actual errors

- ***Preventive Maintenance***
  - *Identify and detect latent faults*
  - *Systems with safety concerns*

- ***Emergency Maintenance***
  - *Unscheduled corrective maintenance*  
*(Risks due to reduced testing)*

# Two Types of Maintenance Activities – Development/Migration

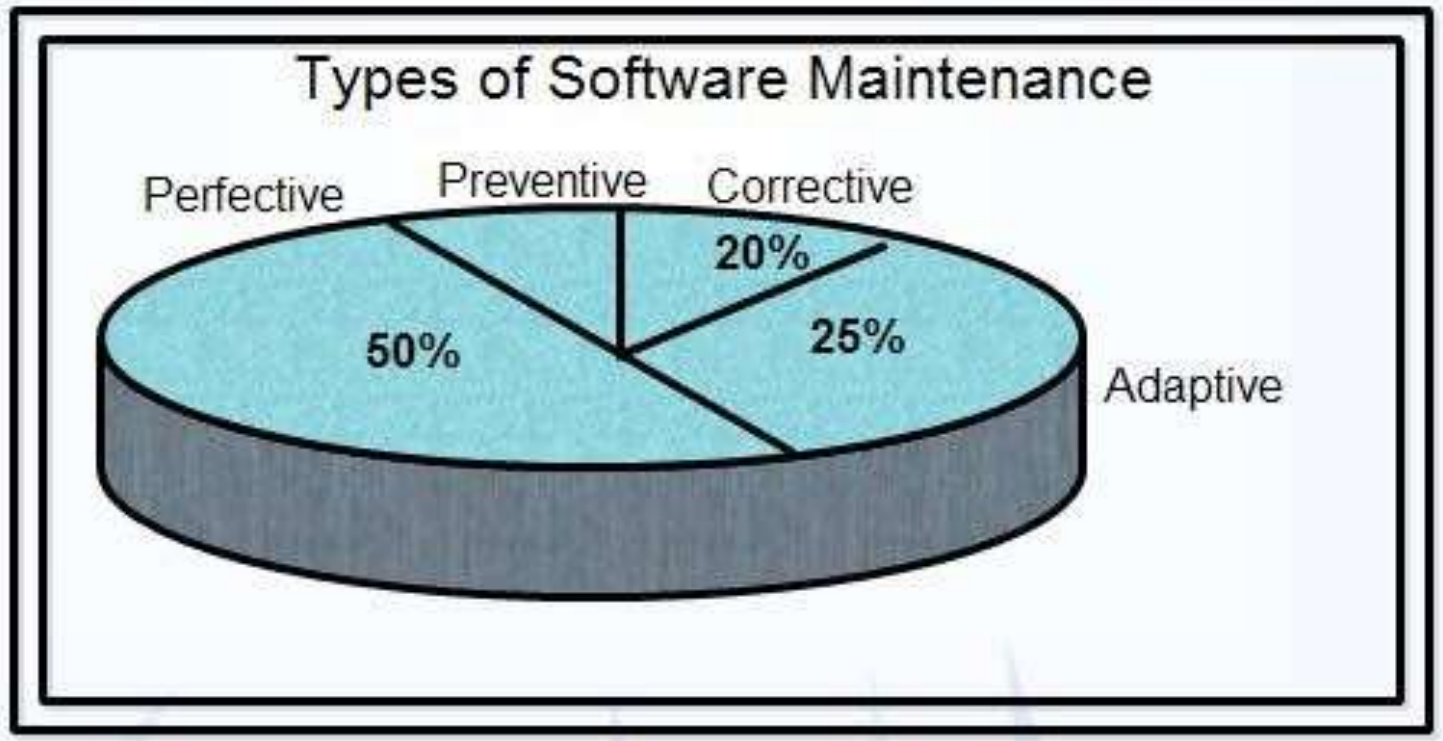
## Post-Delivery

- Perfective Maintenance
  - Improve performance, dependability, maintainability
  - Improve Usability, add missing functionality
  - Update documentation

- *Adaptive Maintenance*
  - *Adapt to a new/upgraded environment (e.g., hardware, operating system, middleware)*
  - *Incorporate new capability*

## Development/Migration

# Cost associated with various maintenance activities



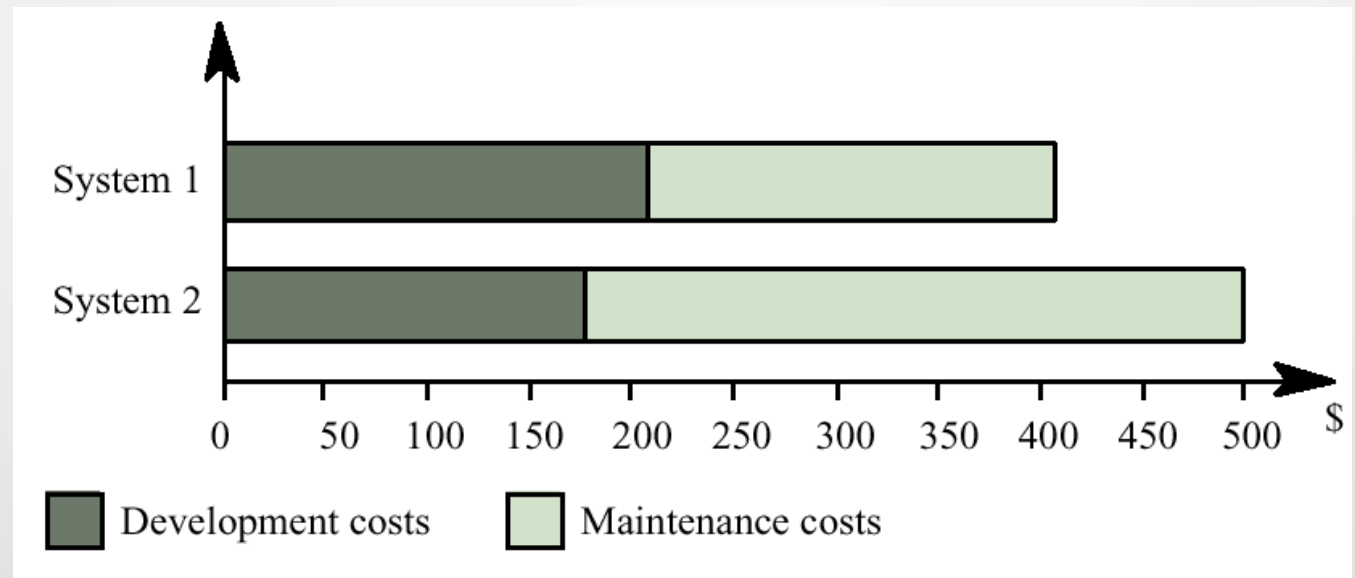
# Maintenance cost

Why maintenance costs are higher than development costs?

## Factors:

- **Team stability:** development teams break up after delivery
- **Contractual responsibility:** different teams or organizations have the responsibility for maintenance
- **Staff skills:** more experienced software engineers tend to avoid maintenance
- **Program age and structure:** not structured in the first place, the program copes poorly with changes and its structure degrades

# Development versus maintenance costs



## Worldwide IT Spending since 2010, including costs for software modernization

Other sources indicate that US \$35 trillion is a significant underestimate because labor costs are not totally accounted for.

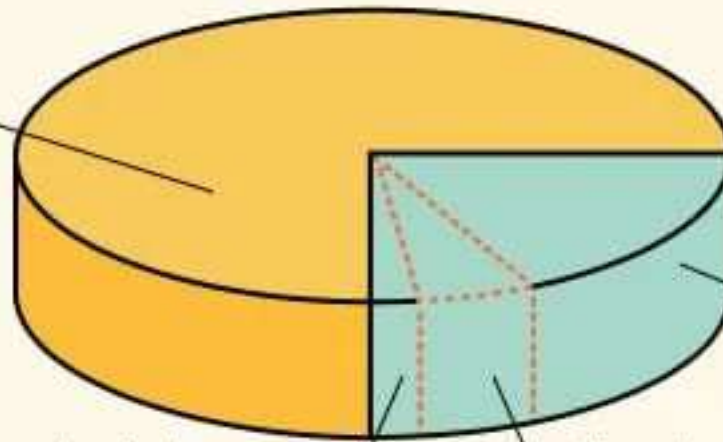
**TOTAL: US \$35 TRILLION**

Operations and maintenance:  
**\$26.25 trillion**

Development,  
modernization and  
enhancement:  
**\$8.75 trillion**

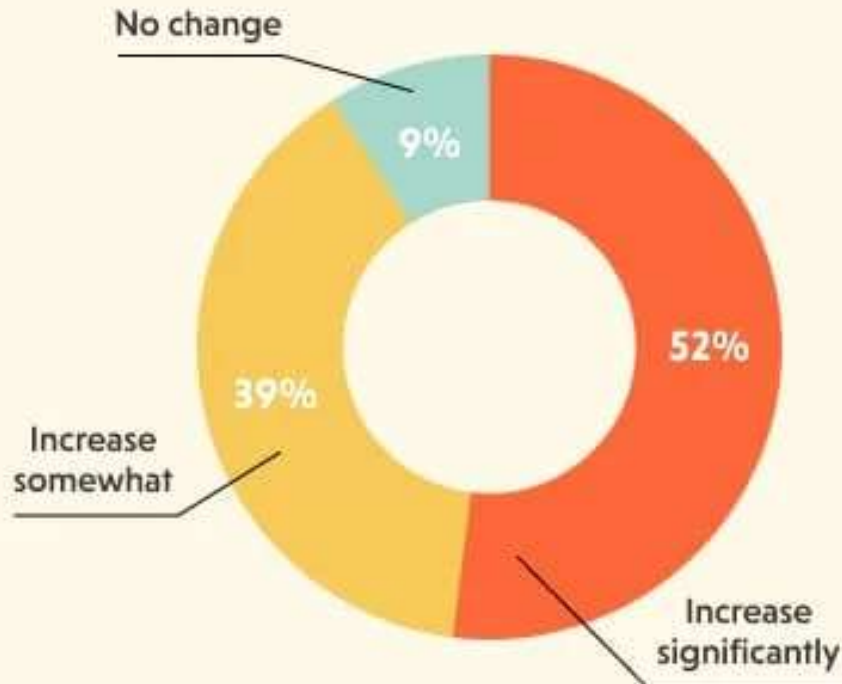
Failed modernization  
efforts: **\$720 billion**

IT modernization:  
**\$2.5 trillion**



## Hidden costs of maintaining legacy software

### Expected Change in Digital Transformation Pace



- The burden on the IT department
- Maintaining or upgrading the database
- Ensuring security for data
- Frequent downtimes
- Poor customer experience
- Missed business opportunities

Source: Flexera 2020 CIO Priorities Report Survey



ADD THIS FEATURE  
TO THE SOFTWARE.



Dilbert.com @ScottAdamsSays

**GAAA!!!** WHY DIDN'T  
YOU ASK FOR THIS  
WEEKS AGO WHEN IT  
WOULD HAVE BEEN  
EASY???



7-20-17 © 2017 Scott Adams, Inc./Dist. by Andrews McMeel

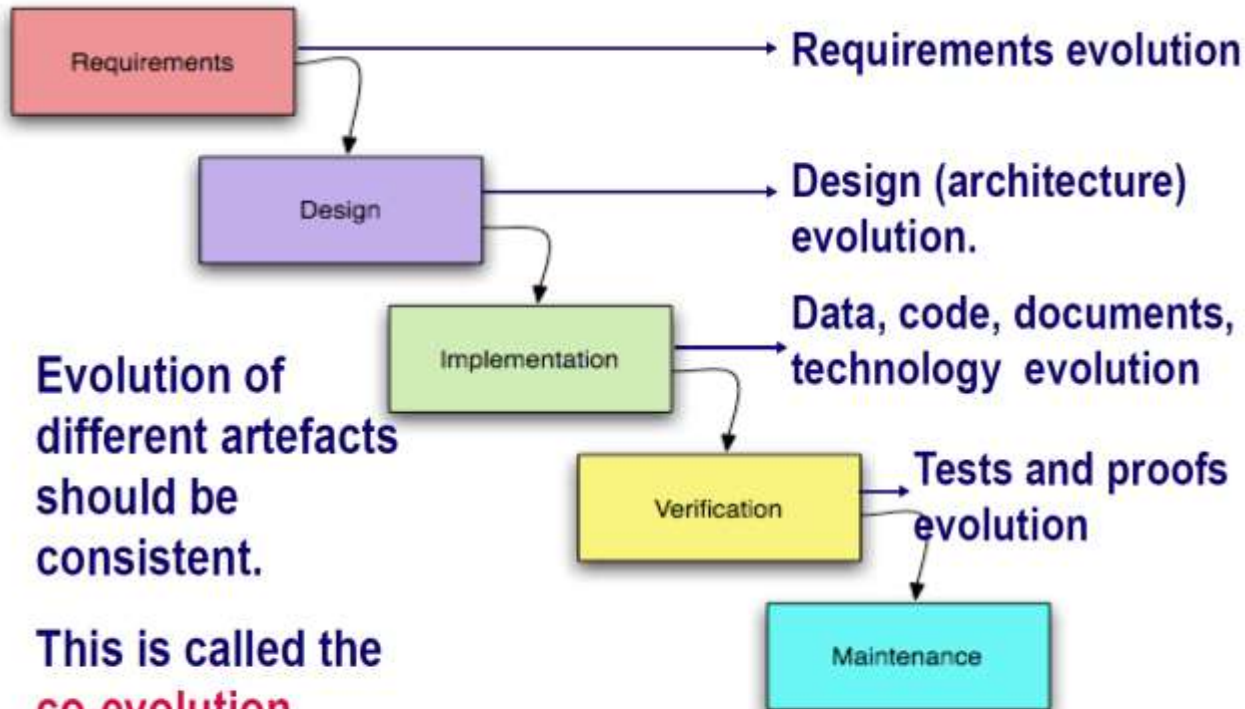
THIS IS NOTHING.  
WAIT UNTIL YOU SEE  
THE FEATURE I ASK  
FOR NEXT WEEK.







# Software Evolution - What is evolving ?



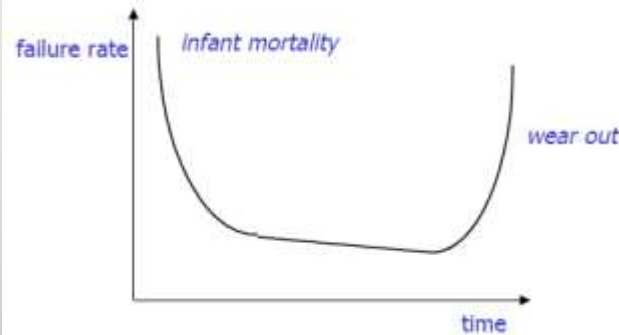
Evolution of different artefacts should be consistent.

This is called the **co-evolution problem**.

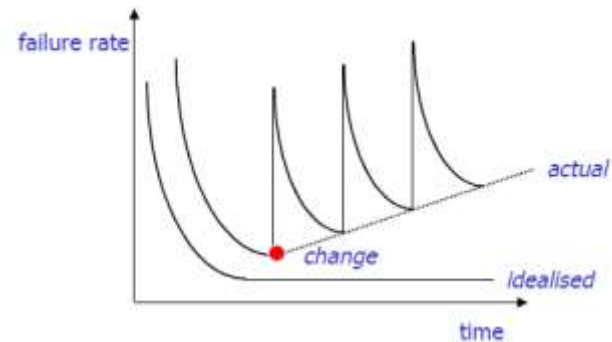
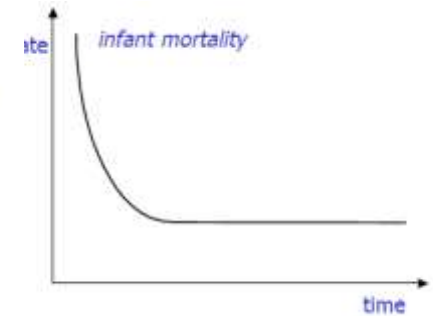
# What is evolving?

# Comparing Failures over time

Hardware:



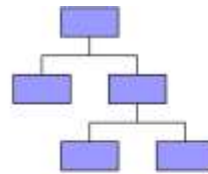
Software (idealized):



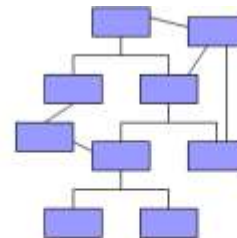
Actual software failures

# Premise

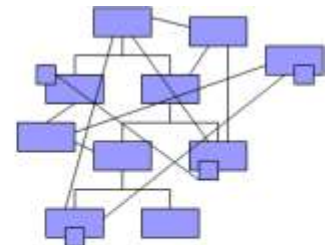
"All known compound objects decay and become more complex with the passage of time. Software is no exception."



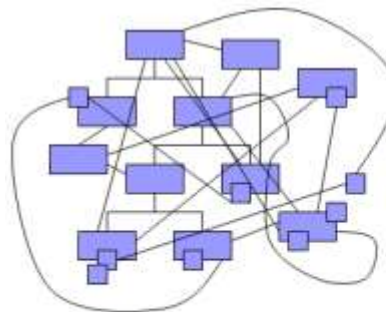
Cost of change:  $C$



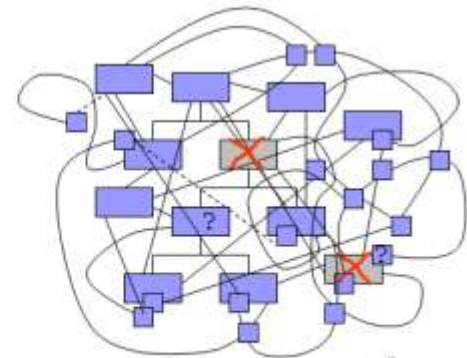
Cost of change:  $C + n$



Cost of change:  $C \times n$



Cost of change:  $C^n$



Cost of change:  $C^{n^n}$

## Inability To Keep Up

As software ages, it grows bigger.

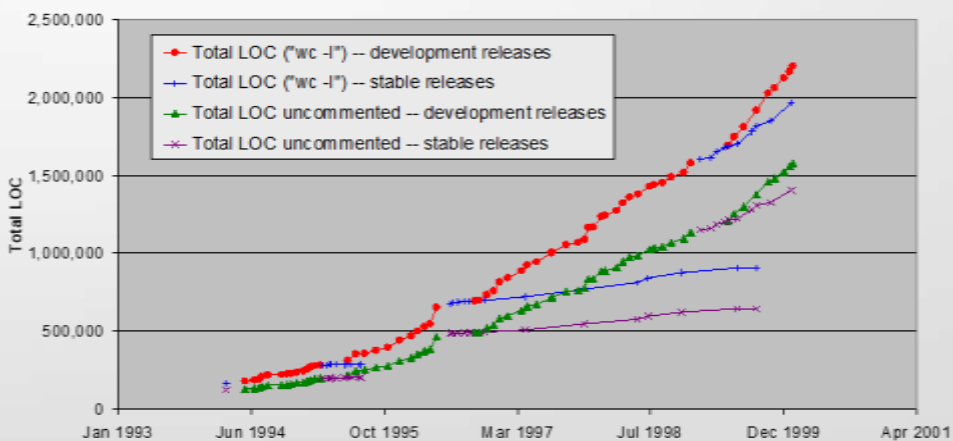
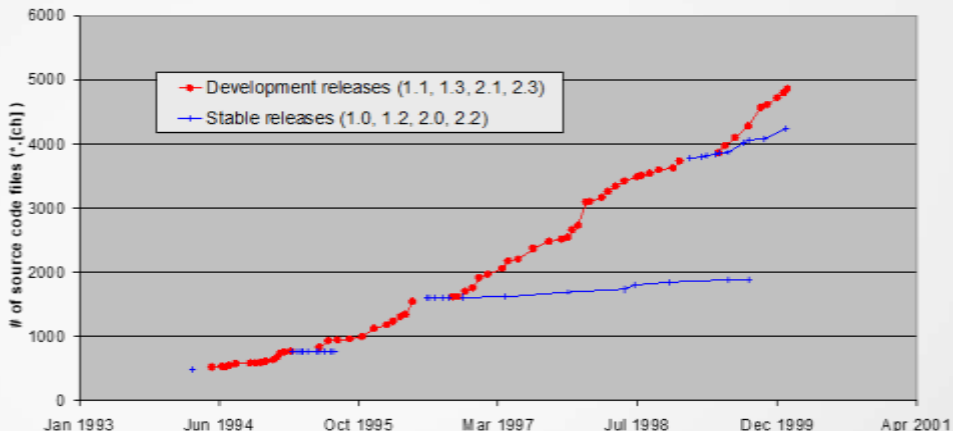
“Weight gain” is a result of the fact that the easiest way to add a feature is to add new code.

Changes become more difficult as the size of the software increases because:

- There is more code to change,
- it is more difficult to find the routines that must be changed.

**Result:** Customers switch to a “younger” product to get the new features.

## Linux kernel



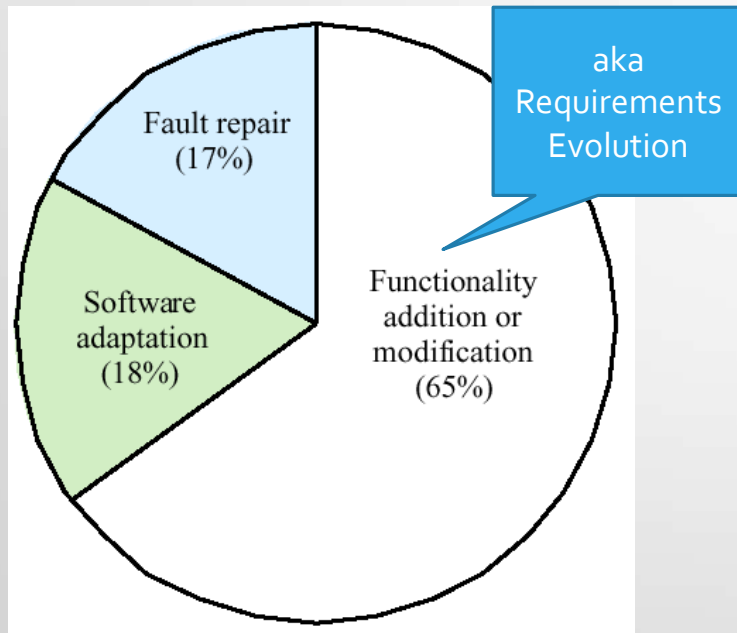
Program	Time frame (years)	Releases	First release			Last release		
			Version	Date	Size (LOC)	Version	Date	Size (LOC)
Samba	15	89	1.5.14	12/08/1993	5,514	3.3.1	02/24/2009	1,045,928
Sendmail	15	57	8.6.4	10/31/1993	25,912	8.14.4a	01/13/2009	87,842
Bind	9	168	9.0.0b1	02/04/2000	169,306	9.6.1b1	03/12/2009	321,689
OpenSSH	9	78	1.0pre2	10/27/1999	12,819	5.2p1	02/22/2009	52,284
SQLite	8	172	1.0	08/17/2000	17,273	3.6.11	02/18/2009	65,108
Vsftpd	8	60	0.0.9	01/28/2001	6,774	2.1.0	01/21/2009	15,711
Quagga	5	29	0.96	08/12/2003	41,623	0.99.11	09/05/2008	47,511

Guowu Xie, Jianbo Chen and Iulian Neamtii

[Towards a Better Understanding of Software Evolution: An Empirical Study on Open-Source Software](#)

ICSM 2009

# Requirements Evolution



- **Corrective Maintenance**

- Identify and remove defects
- Correct actual errors

- **Adaptive Maintenance**

- Adapt to a new/upgraded environment (e.g., hardware, operating system, middleware)
- Incorporate new capability

- **Preventive Maintenance**

- Identify and detect latent faults
- Systems with safety concerns

- **Perfective Maintenance**

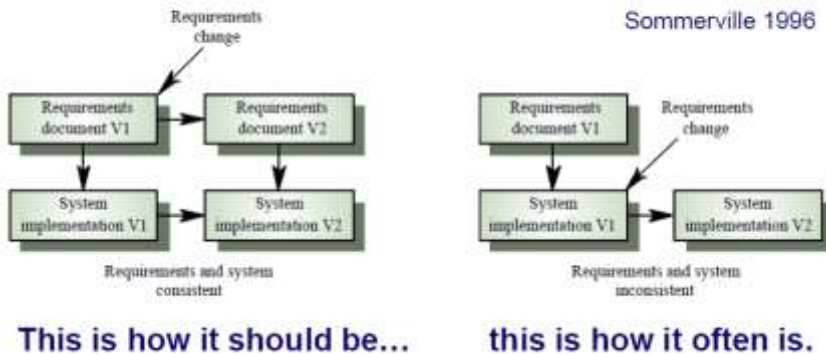
- Improve performance, dependability, maintainability
- Update documentation

- **Emergency Maintenance**

- *Unscheduled corrective maintenance*  
(Risks due to reduced testing)

# Requirements Evolution

- Requirements are “just a piece of paper”...
- Tend to be forgotten during the evolution

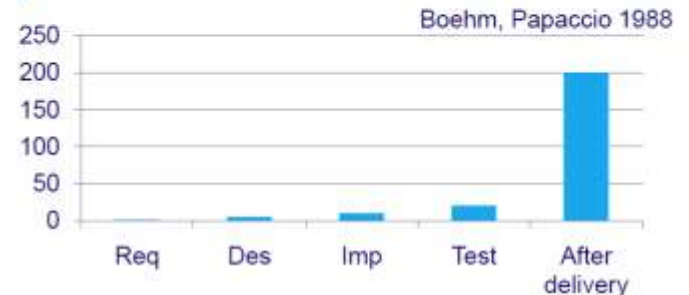


- Errors in requirements are:

- **common:**

- 25% of all the errors (Jones '91)
- 1 error per function point (~ 80 LOC Java; Jones '95)

- **expensive**





# Why requirements are so difficult

- When is evolution difficult?
- Per requirement:
  - “Bad requirements”: vague, subjective, weak, underspecified, overtly complex, unreadable...
  - Volatile requirements
    - Related to dependencies between the requirements
- Per requirements document:
  - Missing requirements
  - Inconsistent requirements



Problem	Indicators (examples)
Vagueness	clear, significant, useful, adequate, good, bad
Subjectivity	similar, as ... as possible, taking ... into account
Optionality	possibly, if needed, if appropriate, eventually
Weakness	could, might
Underspecification	(write/read) access, (data/control) flow, “TBD”
Multiplicity	and, or
Implicitity	Anaphora (it, these, previous, above)



- Industrial approach: guidelines, checklists, templates
- Manual verification
  - “Each requirement should state consequences of losses of availability and breaches of security” (European eGovernment program)
  - “Each requirement is measurable” (SMART)
- Assessment
  - is subjective
  - requires training and experience

# Volatile requirements ?

- “More stable requirements should not depend on less stable ones”
- A affects B (B depends on A) if changing A might require changing B.



- At requirements engineering time:
  - Try to find a more stable alternative
  - Put special attention to traceability (backwards – rationale, forwards – design, implementation, tests)
  - Anticipate and record responses for future changes
- At design time:
  - Encapsulate volatile requirements in separate modules

# Types of dependencies

- **Temporal**

- Satisfaction of A should precede/follow satisfaction of B
- “IMSETY shall require users to be logged in before they can use any of the system’s functionality.”

- **Satisfiability**

- Satisfaction of A implies satisfaction of B
- More general than “generalization/refinement”
- But also
  - “The system shall interface with an MCS for communication with satellites.”
  - “IMSETY shall log all communications with the MCSes.”

- **Use**

- A explicitly refers to B
- “IMSETY shall adhere to Table 2.1 for user rights”

- **Generalization/refinement**

- A is a more general case of B
- “Observers shall not be authorized to manipulate experiments.”

Users

# Project



# Evolution Life Cycle

10/18/2023

SOEN 6431 45

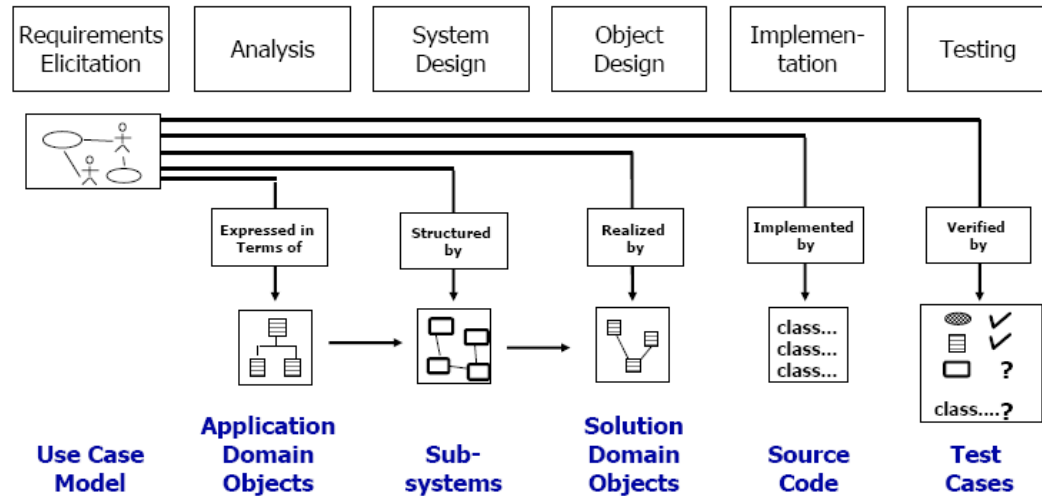


# General Idea

Software Development we do have different Software Life Cycle Models, e.g.,

- ☐ Waterfall
- ☐ Spiral Model
- ☐ Prototyping
- ☐ Iterative
- ☐ Open source

These models describing the sequence of activities (e.g., requirements analysis, implementation, testing) to be performed during the development



Traditional Software Development Life Cycle and its models.




# Software maintenance has unique characteristics:

**Constraints of an existing system:** Maintenance is performed on an operational system. Therefore, all modifications must be compatible with the constraints of the existing architecture, design, and code.

**Shorter time frame:** A maintenance activity may span from a few hours to a few months, whereas software development may span one or more years.

**Available test data:** In software development, test cases are designed from scratch, whereas software maintenance can select a subset of these test cases and execute them as regression tests.

Software maintenance should have its own Software Maintenance Life Cycle (SMLC) model as it involves many unique activities.

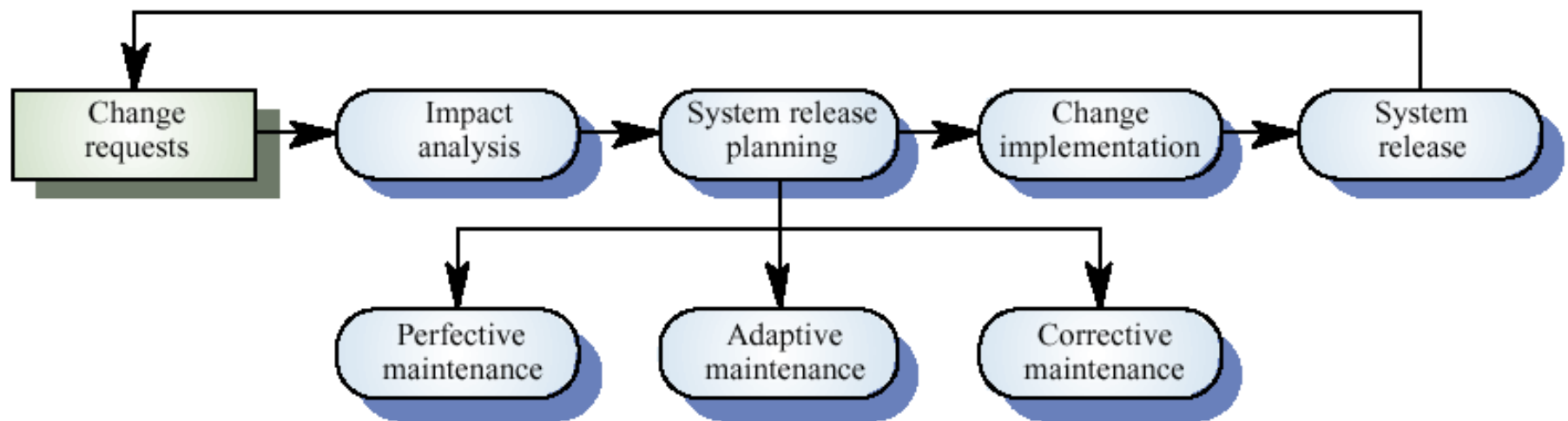


## General Idea



# A Closer look at a Maintenance Process

# The maintenance process



# Software Maintenance Process 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:

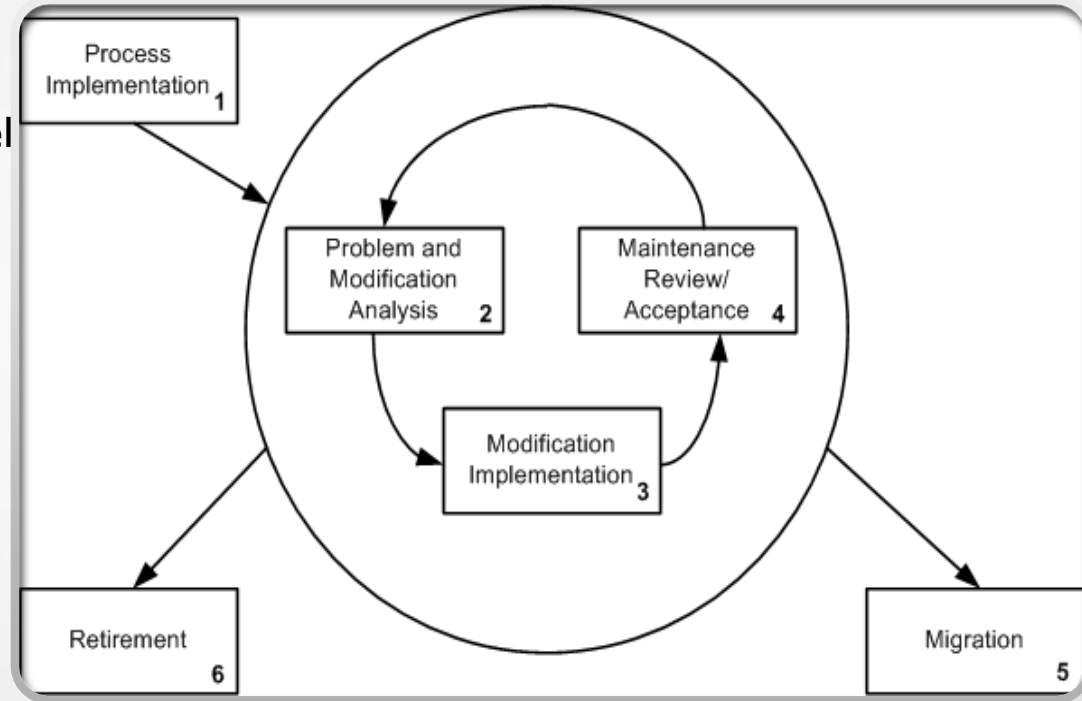
- process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration and retirement.

Each of these maintenance activity is made up of tasks. Each task describes a specific action with inputs and outputs.

# ISO/IEC 14764 Maintenance Process

The maintenance process comprises the following high-level activities:

- 1. Process Implementation.
- 2. Problem and Modification Analysis.
- 3. Modification Implementation.
- 4. Maintenance Review and Acceptance.
- 5. Migration.
- 6. Retirement.



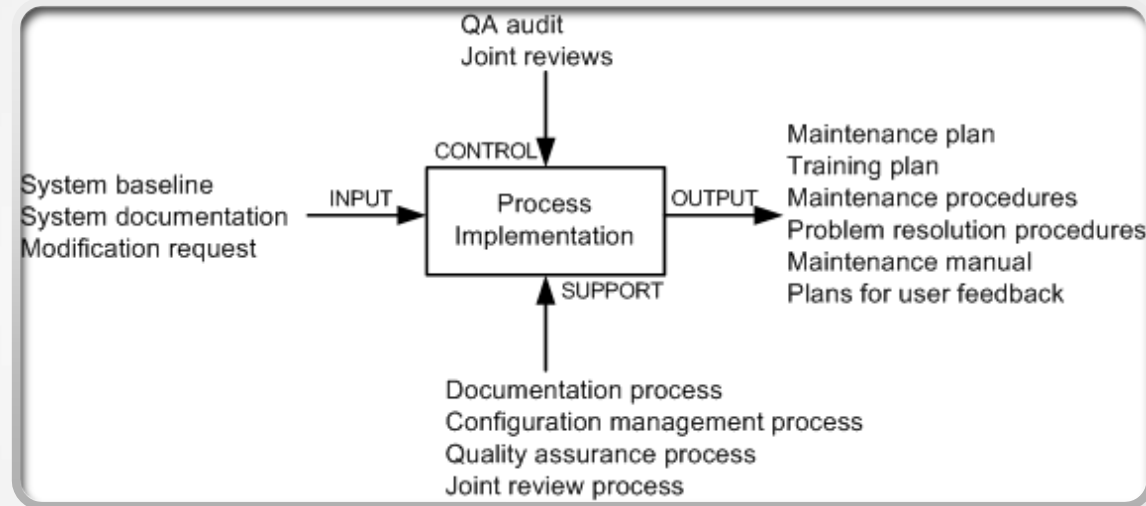
[ISO/IEC 14764 iterative maintenance process ©IEEE, 2004](#)

# ISO/IEC 14764 Maintenance Process

## Example of a process step: Process Implementation

The process implementation activity consists of three major tasks:

- Maintenance plan.
- Modification requests (MR/CR).
- Configuration management.



**Process implementation activity**

# ISO/IEC 14764 Maintenance Process

**TABLE 3.1 Template of a Maintenance Plan**

---

**1. Introduction**

*This section outlines the goals, purpose, and general scope of the maintenance effort. Also, deviations from the standard are identified.*

**2. References**

*The documents that impose constraints on the maintenance effort are identified in this section.*

*In addition, other documents supporting maintenance activities are identified.*

**3. Definitions**

*All terms required to understand the maintenance plan are defined in this section.*

*If some terms are already defined in other documents, then references are provided to those documents.*

**4. Software Maintenance Overview**

*This section briefly describes the following aspects of the maintenance process:*

4.1 Organization

4.2 Scheduling Priorities

4.3 Resource Summary

4.4 Responsibilities

4.5 Tools, Techniques, and Methods

**5. Software Maintenance Process**

*This section describes the actions to be executed in each phase of the maintenance process. Each action is described in the form of input, output, process, and control.*

5.1 Problem Identification/Classification and Prioritization

5.2 Analysis

5.3 Design

5.4 Implementation

5.5 System Testing

5.6 Acceptance Testing

5.7 Delivery

**6. Software Maintenance Reporting Requirements**

*This section briefly describes the process for gathering information and disseminating it to members of the maintenance organization.*

**7. Software Maintenance Administrative Requirements**

*Describes the standard practices and rules for anomaly resolution and reporting.*

7.1 Anomaly Resolution and Reporting

7.2 Deviation Policy

7.3 Control Procedures

7.4 Standards, Practices, and Conventions

7.5 Performance Tracking

7.6 Quality Control of Plan

**8. Software Maintenance Documentation Requirements**

*Describes the procedures to be followed in recording and presenting the outputs of the maintenance process.*

---

Source: From Reference 25. © 1998 IEEE.

**TABLE 3.5 Review and Approval Task Steps**

---

**Review Task Steps**

1. Track the MRs from requirement specification to coding.
2. Ensure that the code is testable.
3. Ensure that the code conforms to coding standards.
4. Ensure that only the required software components were changed.
5. Ensure that the new code is correctly integrated with the system.
6. Ensure that documentations are accurately updated.
7. CM personnel build software items for testing.
8. Perform testing by an independent test organization.
9. Perform system test on a fully integrated system.
10. Develop test report.

**Approval Task Steps**

1. Obtain quality assurance approval.
  2. Verify that the process has been followed.
  3. CM prepares the delivery package.
  4. Conduct functional and physical configuration audit.
  6. Notify operators.
  7. Perform installation and training at the operator's facility.
-



# Life cycle models for software maintenance



# Reuse Oriented Models

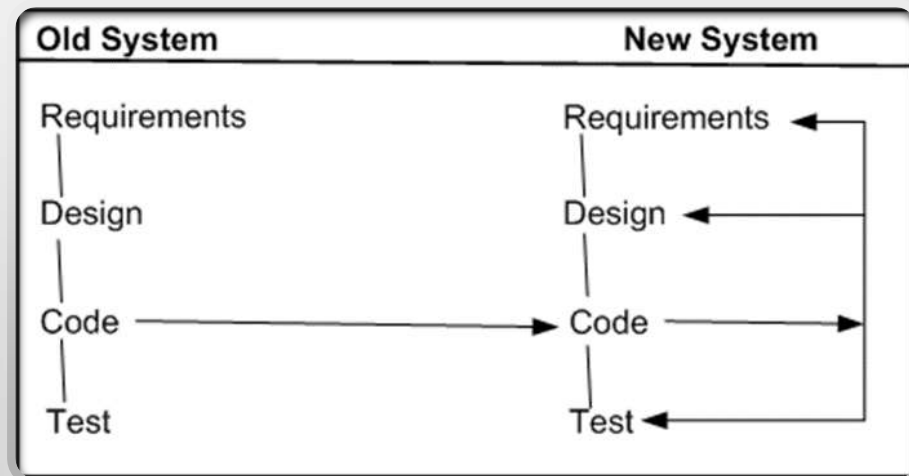
- One obtains a new version of an old system by modifying one or several components of the old system and possibly adding new components.
- Based on this concept, **three process models for maintenance** have been proposed by Basili:
  - **Quick fix model:** In this model, necessary changes are quickly made to the code and then to the accompanying documentation.
  - **Iterative enhancement model:** In this model, first changes are made to the highest level documents. Eventually, changes are propagated down to the code level.
  - **Full reuse model:** In this model, a new system is built from components of the old system and others available in the repository.

# Reuse Oriented Models

## For Quick Fix Model

- source code is modified to fix the problem;
- necessary changes are made to the relevant documents; and
- the new code is recompiled to produce a new version.

Changes to the source code are often made with no prior investigation such as analysis of impact of the changes, ripple effects of the changes, and regression testing.



# Reuse Oriented Models

---

## Iterative Vs. Incremental

---

The terms **iteration** and **increment** are liberally used when discussing iterative and incremental development.

---

However, they are not synonyms in the field of software engineering.

---

**Iteration** implies that a process is basically cyclic, thereby meaning that the activities of the process are repeatedly executed in a structured manner.

---

**Iterative** development is based on scheduling strategies in which time is set aside to improve and revise parts of the system under development.

---

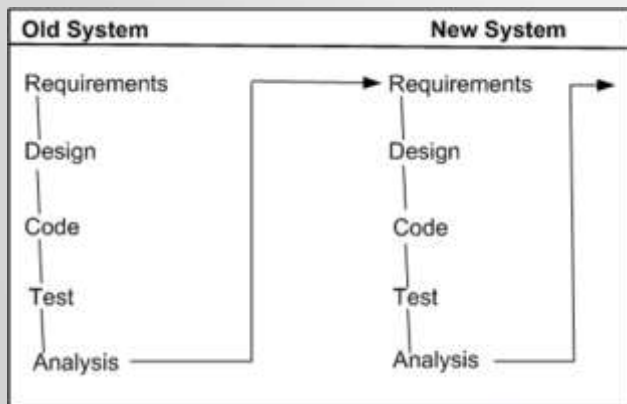
**Incremental** development is based on staging and scheduling strategies in which parts of the system are developed at different times and/or paces, and integrated as they are completed.



# Reuse Oriented Models

The **iterative enhancement model** shows how changes flow from the very top level documents to the lowest-level documents.

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

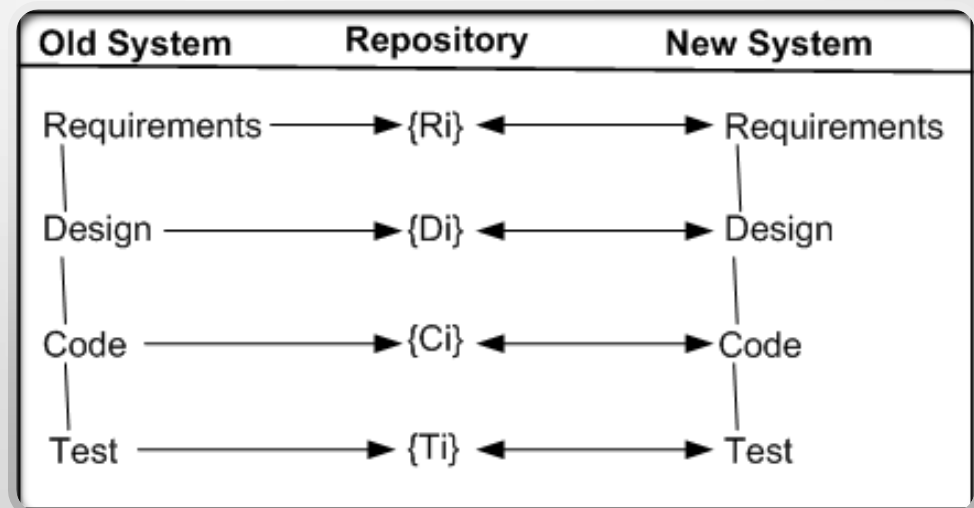


**The iterative enhancement model ©IEEE, 1990**

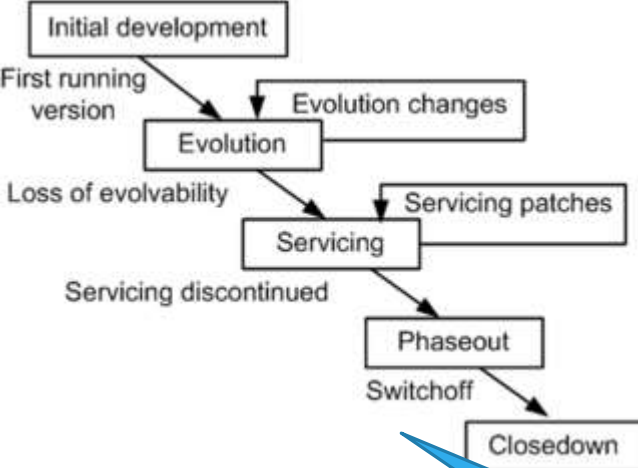
# Reuse Oriented Models

The main assumption in **full reuse model** is the availability of a repository of artifacts describing the earlier versions of the systems.

- In the **full reuse model**, reuse is explicit, and the following activities are performed:
  - identify the components of the old system that are candidates for reuse
  - understand the identified system components.
  - modify the old system components to support the new requirements.
  - integrate the modified components to form the newly developed system.



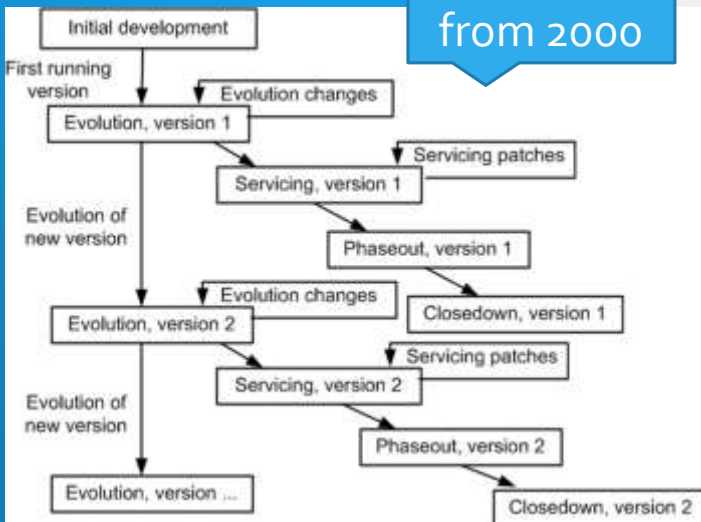
The full reuse model ©IEEE, 1990



# The Staged Model for Software Systems

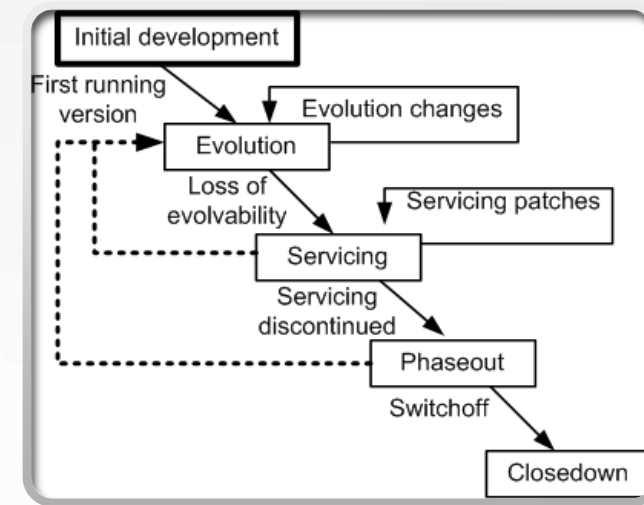
original version from 1990

Updated version from 2000



- Rajlich and Bennett have presented a descriptive model of software evolution called the **staged model of maintenance and evolution**.
- Its primary objective is at improving understanding of how long-lived software evolves, rather than aiding in its management.
- Their model divides the lifespan of a typical system into four stages:
  - **Initial development** – The first delivered version is produced. Knowledge about the system is fresh and constantly changing. In fact, change is the rule rather than the exception. Eventually, an architecture emerge and stabilizes.
  - **Evolution** – Simple changes are easily performed, and more major changes are possible too, albeit the cost and risk are now greater than in the previous stage. Knowledge about the system is still good, although many of the original developers will have moved on. For many systems, most of its lifespan is spent in this phase.
  - **Servicing** – The system is no longer a key asset for the developers, who concentrate mainly on maintenance tasks rather than architectural or functional change. Knowledge about the system has lessened and the effects of change have become harder to predict. The costs and risks of change have increased significantly.
  - **Phase out** – It has been decided to replace or eliminate the system entirely, either because the costs of maintaining the old system have become prohibitive or because there is a newer solution waiting in the pipeline. An exit strategy is devised and implemented, often involving techniques such as wrapping and data migration. Ultimately, the system is shut down.

# The Staged Model for FLOSS



The staged model for FLOSS system ©ACM, 2007

- Three major differences are identified between CSS systems and FLOSS systems.
  - The rectangle with the label “Initial development” has been visually highlighted because it can **be the only initial development stage** in the evolution of FLOSS systems. In other words, it does not have any evolution track for FLOSS system.
  - With some systems that were analyzed, after a transition from Evolution to Servicing, a new period of evolution was observed. This possibility is depicted below as a broken arc from the Servicing stage to the Evolution stage.
  - In general, the active developers of FLOSS systems get frequently replaced by new developers. Therefore, the dashed line in Figure exhibits this possibility of a transition from **Phaseout stage to Evolution stage**.