Software Maintenance (I)

Software decay

Frequent failures
Overly complex structure
Running in emulation mode
Very large components
Excessive resource requirements
Deficient documentation
High personnel turnover
Different technologies in one system

Software evolution

Organizations have 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 <u>evolving</u> existing software rather than <u>developing</u> new software.

Studies indicate that up to **75%** of all software professionals are involved in some form of maintenance/evolution activity.

Software evolution – variety of facets

- Data evolution
  - Migrating to a new database schema.
  - Verifying that the information in the existing databases is preserved.
- Runtime evolution
  - How to modify a system without stopping it.
  - Encompasses runtime reconfiguration, dynamic adaptation, dynamic upgrading.
- Language evolution
  - Dealing with changes in the programming language definition.
  - multi-language systems.
  - Designing languages to make them more robust to evolution.

Evolution of software artifacts

- Requirements evolution
  - Managing requirements changes.
- Architecture evolution
  - Reengineering the architectures of legacy systems.
  - Migration to distributed architectures, e.g., service-oriented architectures.
  - Maintenance issues with modern architectures.
- Design evolution
  - Evolution of design models.
- Test case evolution
  - Adding and modifying test cases to verify that the system behavior was changed as intended.
- Traceability management
  - How to assure the consistency of different artifacts.

Software change

- Software change is inevitable:
  - New requirements emerge when the software is used;
  - The business environment changes;
  - Errors must be eliminated;
  - New computers and equipment is added to the system;
  - The performance /reliability of the system has to be improved.

A key problem for organizations is implementing and managing change to their existing software systems.

Types of changes

- **Repair** software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- **Adapt** software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- **Add** to or **modify** the system's functionality
  - Modifying the system to satisfy new requirements.
- **Improve** the program structure and system performance
  - Rewriting all or parts of the system to make it more efficient and maintainable.

Software evolution and software maintenance

- No standard definitions.
- Broad definition of evolution: Generally, software evolution refers to the study and management of the process of making changes to software over time.
  - In this definition, software evolution comprises:
    - Development activities
    - **Maintenance activities**
    - Reengineering activities
- Narrow definition of evolution: Sometimes, software evolution is used to refer to the activity of adding new functionality to existing software.
- Maintenance refers to the activity of modifying software after it has been put to use in order to maintain its usefulness.

Types of changes

“Maintenance”

“Evolution”

“Reengineering”

- **Repair** software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- **Adapt** software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- **Add to or modify** the system’s functionality
  - Modifying the system to satisfy new requirements.
- **Improve** the program structure and system performance
  - Rewriting all or parts of the system to make it more efficient and maintainable.

History

- 1960s – 1970s
  - Inclusion of maintenance in waterfall lifecycle after delivery of the software product.
  - Perception that post-delivery activities only consisted of faults fixes and minor adjustments.
  - Did not account for the need to add functionality due to new and changed requirements.
- 1970s
  - Lehman postulated the initial laws of program evolution.
  - Stressed the need for continuous evolution due to changes in the software’s operational environment.
- Late 1970s – 1980s
  - Initial process models that handled change requests.
- 1990s
  - General acceptance of software evolution.
  - Development of new process models that accounted for evolution activities: evolutionary development, spiral model, agile software development.

Laws of program evolution (Lehman)

Continuing change
Any software that reflects some external reality undergoes continual change or becomes progressively less useful. The change continues until it is judged more cost effective to replace the system entirely

Increasing complexity
As the software evolves, its complexity increases unless steps are taken to control it

Fundamental law of program evolution
Software evolution is self-regulating with statistically determinable trends and invariants

Laws of Program evolution (Lehman)

Conservation of organizational stability
During the active life of a software system, the work output of a development project is roughly constant (regardless of resources) [adding more staff does not improve output]

Conservation of familiarity
During the active life of a program the amount of change in successive releases is roughly constant

Laws of Program evolution

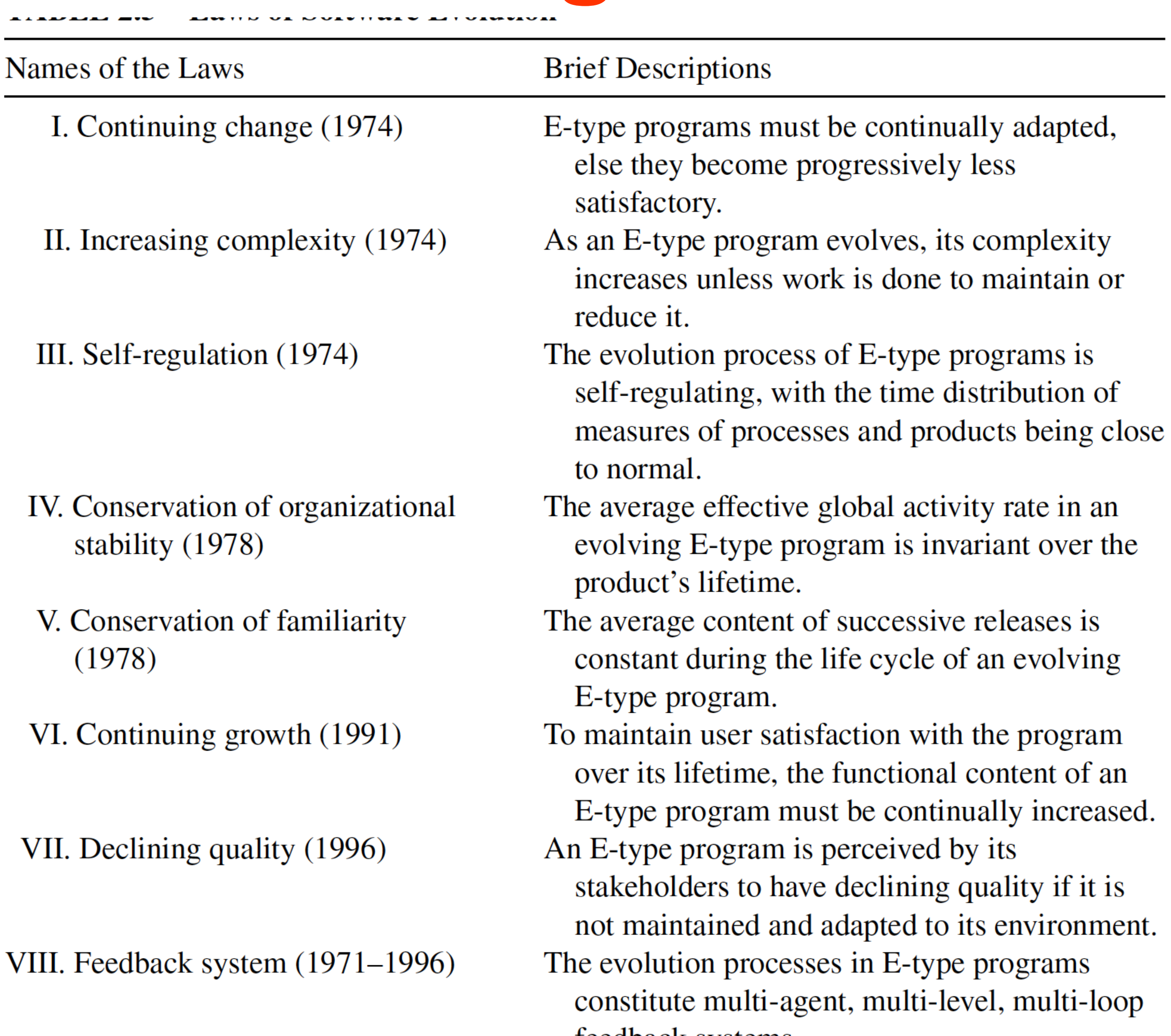


TABLE 2.5 Laws of Software Evolution

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.

Program (System) Types

S-type (Specifiable)

P-type (Problem-solving)

E-type (Embedded)

Program (System) Types

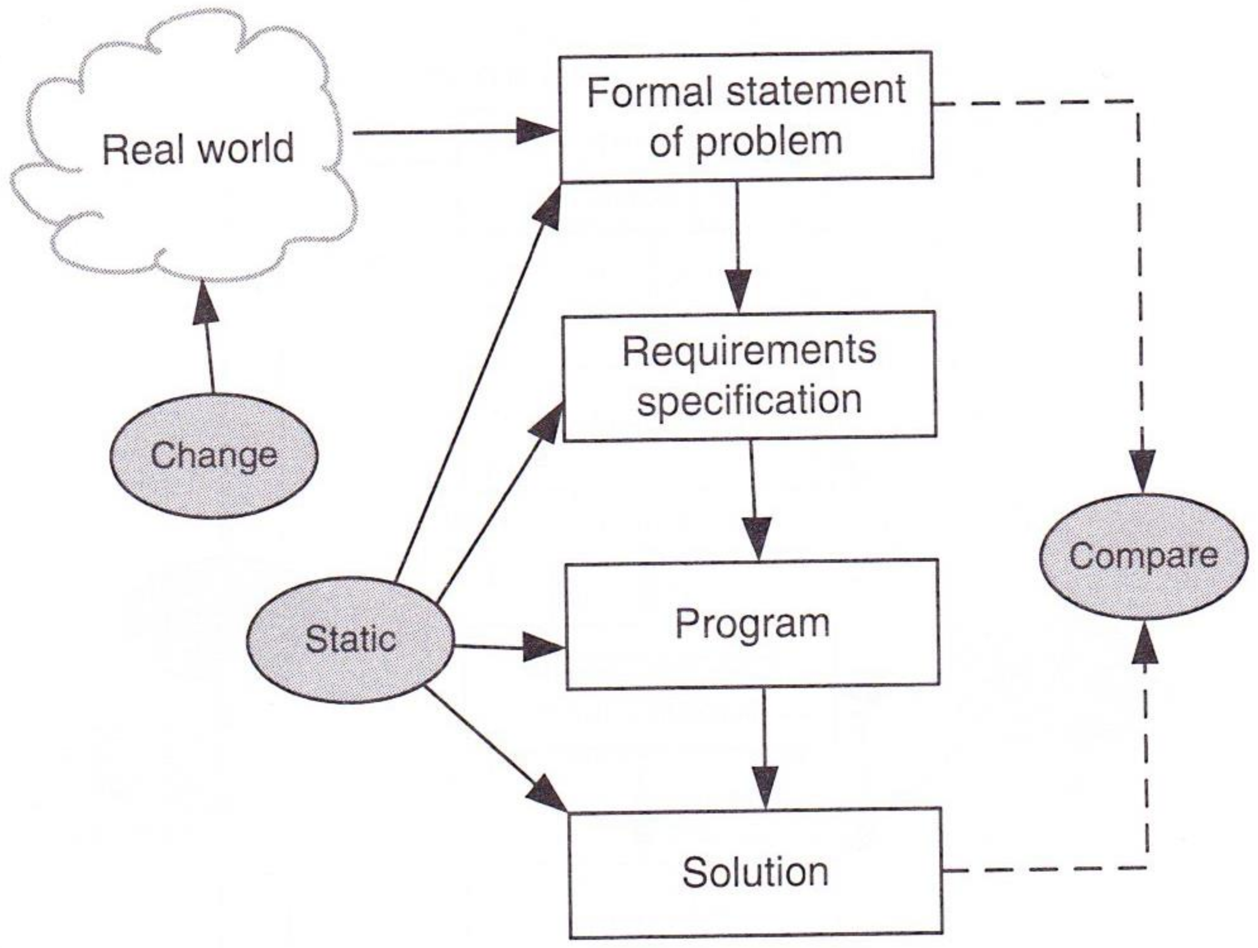
S-type (Specifiable)

Problem can be stated formally and completely.
The problem is completely defined and there is a well- defined solution. We are concerned with the correctness of the implementation of the solution.

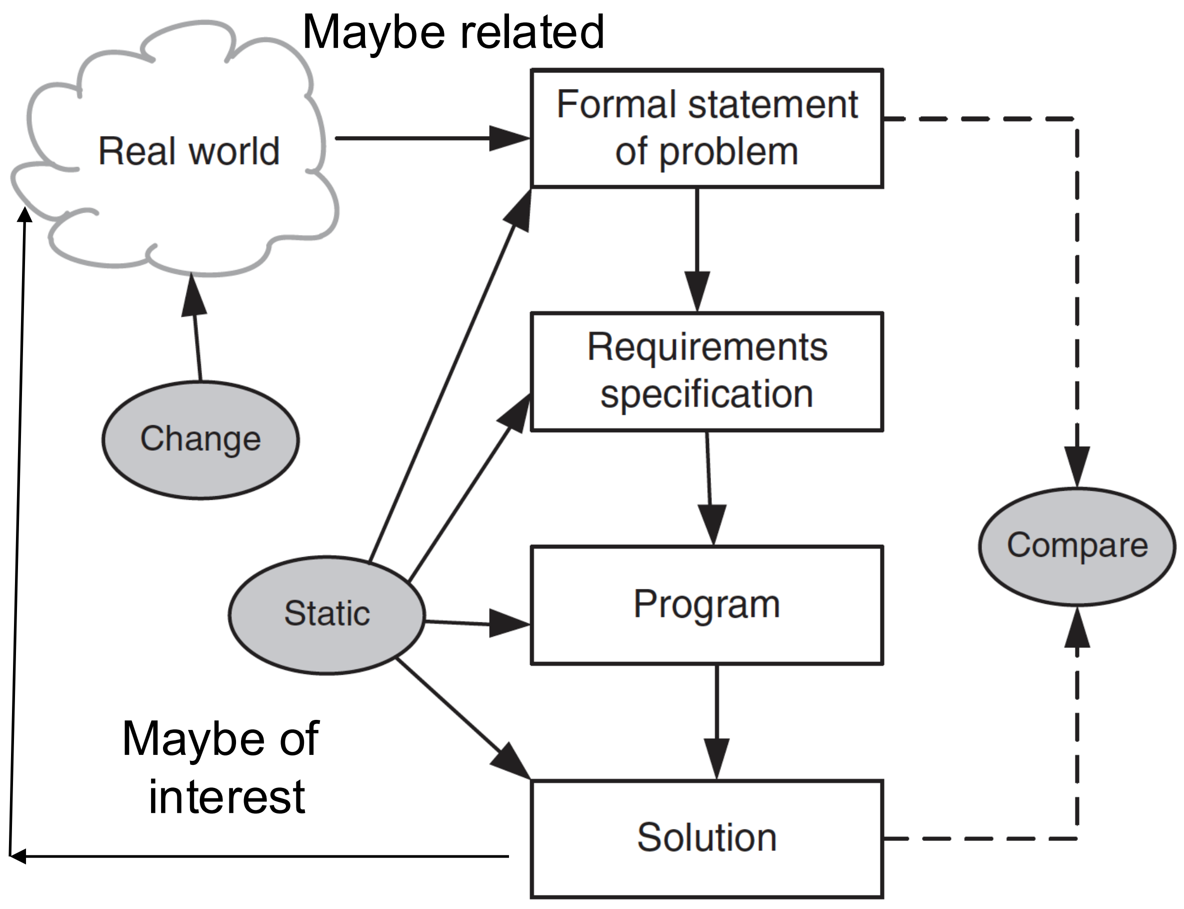
Acceptance:
is the program correct according to its specification?

The software does not evolve
any change in the specification defines a new problem, hence a new program

S-Type


Real world
Formal statement of problem
Requirements specification
Change
Compare
Static
Program
Solution

S-Type


Maybe related
Real world
Formal statement of problem
Requirements specification
Program
Solution
Change
Static
Compare
Maybe of interest

Program (System) Types

P-type (Problem-Solving, Practical abstraction)

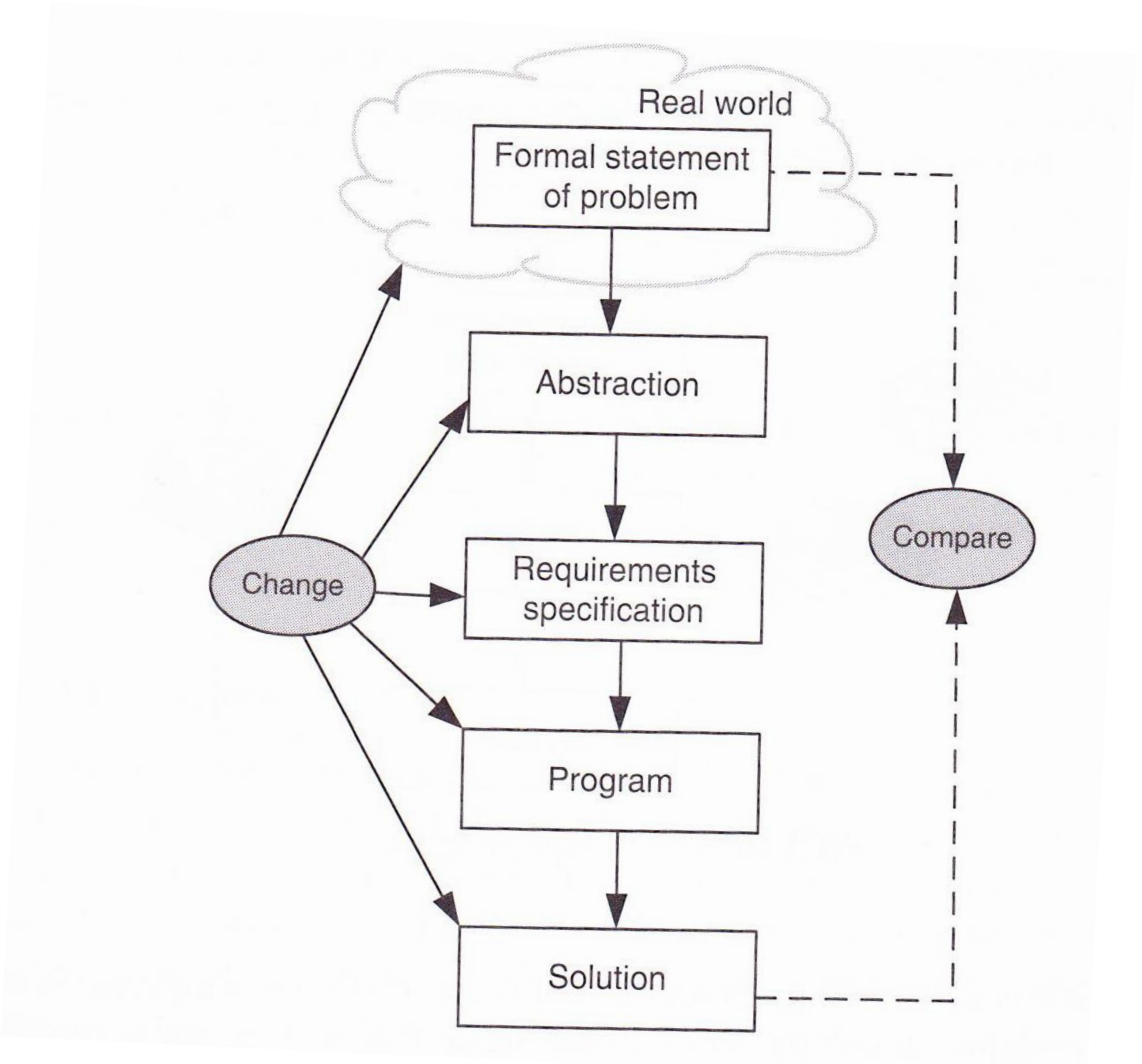
Imprecise (approximate) statement of a real-world problem
Example: chess playing

Acceptance:
is the program an acceptable solution to the problem?

The software is likely to evolve continuously
Solution is never perfect and can be improved
Real-world changes and hence the problem changes

Examples: weather prediction systems, stock market analysis tools…

P- Type


Real world
Formal statement of problem
Abstraction
Requirements specification
Program
Solution
Change
Compare

Program (System) Types

E-type (Embedded, Evolutionary)

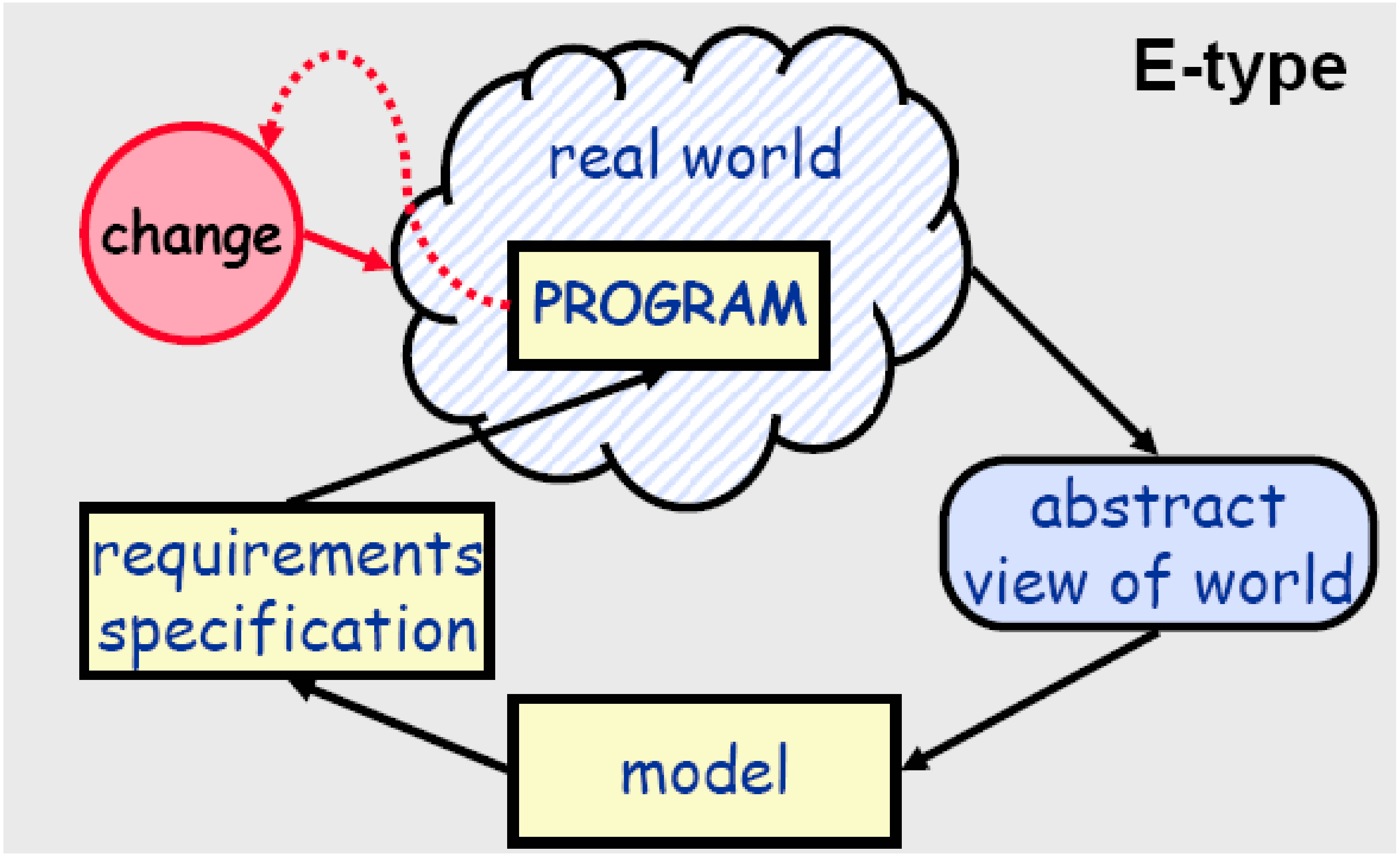
A system becomes part of the world that it models

Acceptance:
Depends entirely on opinions and judgments

The software is inherently evolutionary
Changes in the software and the world affect each other

Examples: e-commerce applications, social media platforms, enterprise resource planning, banking, healthcare, content management systems, operating systems….

E- Type


E-type
real world
change
PROGRAM
abstract view of world
requirements specification
model

Software Maintenance: definitions

… the performance of those activities required to keep a software system operational and responsive after it is accepted and placed into production

Maintenance covers the life of a software system from the time it is installed until it is phased out

Modification of a software product after delivery to correct faults, to improve performance or other attributes, or to adapt the product to a modified environment [IEEE 1219 1993]

… software product undergoes modification to code and associated documentation due to a problem or the need for improvement.
The objective is to modify existing software product while preserving its integrity [ISO/IEC 12207 1995]

Software Maintainability

The ease with which software can be maintained, enhanced, adapted, or corrected to satisfy specified requirements

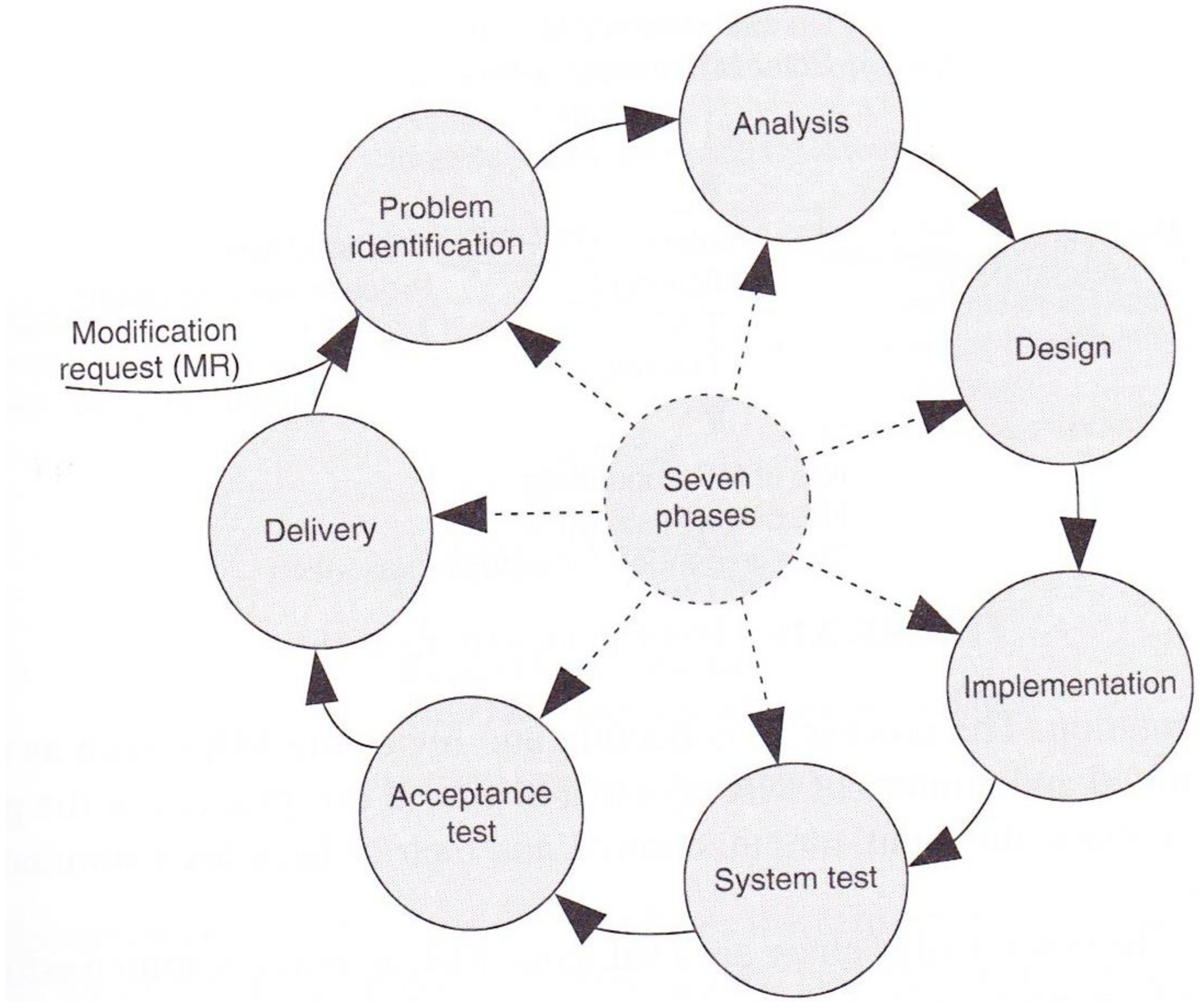
(1) The ease with which a software system or component can be modified to correct faults, improve performance or other attributes, or adapt to a changed environment, or
(2) The ease with which a hardware system or component can be retained in, restored to, a state in which it can perform its required functions

IEEE Computer Society [ANSI/IEEE 610.12.1990]

A set of attributes that bear on the effort needed to make specified modifications

[ISO/IEC 8402 1986]

Software Maintenance Process (IEEE)



Software Maintenance - Types

Corrective maintenance:

- Fixing faults that cause the system to fail

Adaptive maintenance:

- Making changes in existing software to accommodate a changing environment

Software Maintenance- Types

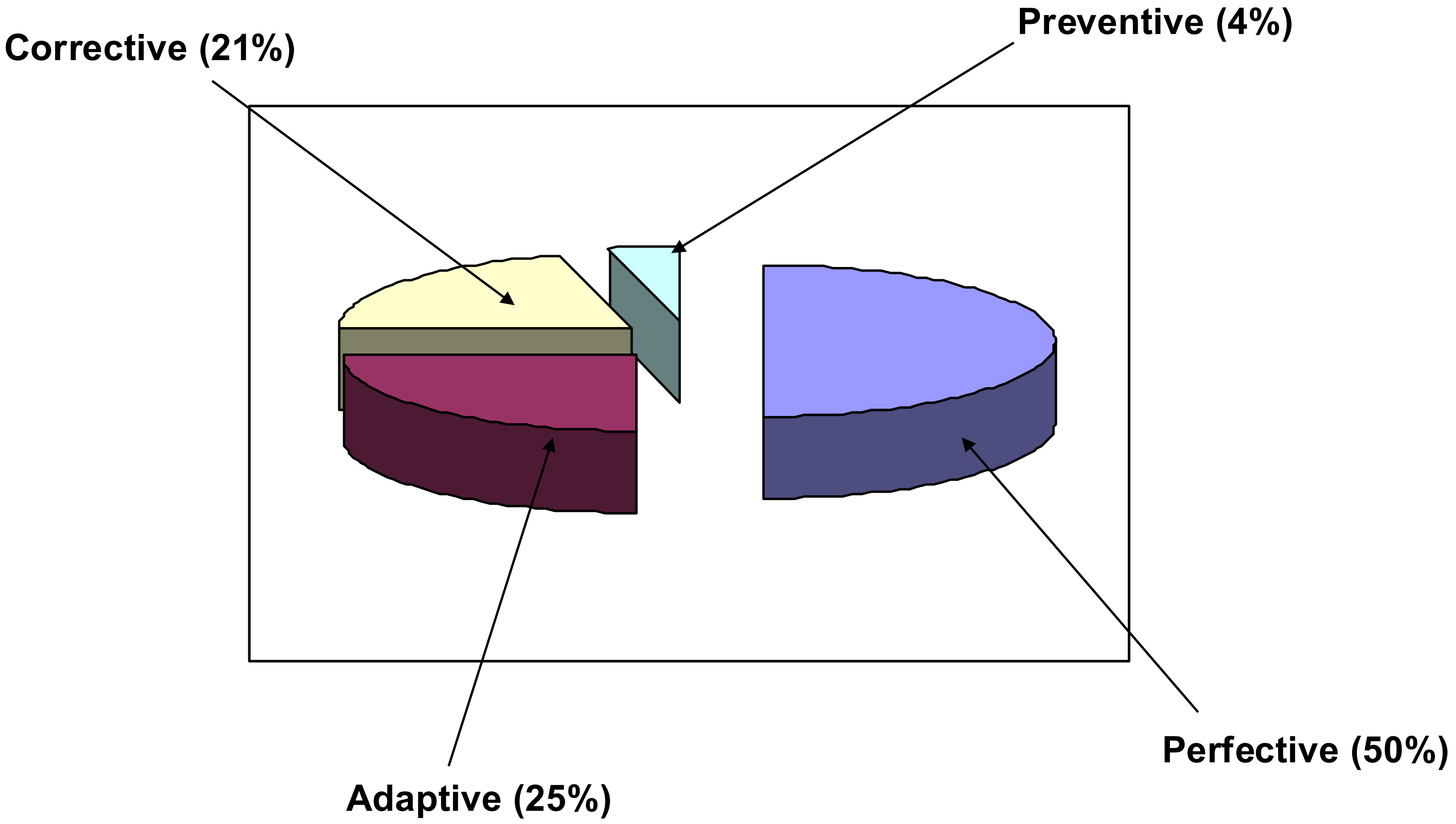
Perfective maintenance

- Making improvements to the existing system without effecting end-user functionality
  - To make it easier to extend and add new features in the future
  - Also called *re-engineering*
  - *Refactoring* is a one type of perfective maintenance

Preventive maintenance

- Preventing failures by fixing defects in advance of failures
- A kind of perfective maintenance
- Key examples: Y2K and Daylight Savings Time adjustments

Types of Maintenance- stats



Approaches to Maintenance (1)

Maintenance philosophies:

Throw it over-the-wall – someone else is responsible for maintenance

investment in knowledge and experience is lost
maintenance becomes a challenge of reverse engineering

Mission orientation
Development team makes a long term commitment to maintaining the software

Approaches to Maintenance (2) Basili's maintenance process models

Quick-fix model
•Changes made at the code level, as easily as possible
•Rapid degradation of the structure of the software

Iterative enhancement model
•Changes made based on an analysis of the existing system
•Attempts to control complexity and maintain good design

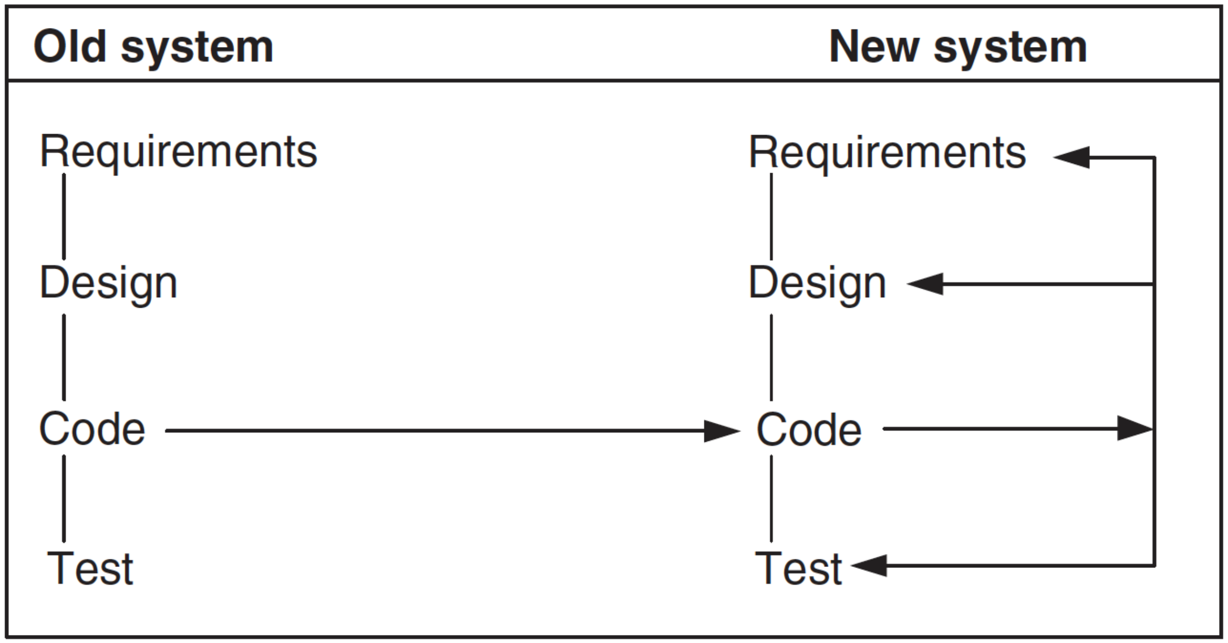
Full reuse model
•Starts with requirements for the new system, reusing as much as possible
•Needs a mature reuse culture to be successful

Quick-fix models

Quick-fix model

•Changes made at the code level, as easily as possible

•Rapid degradation of the structure of the software

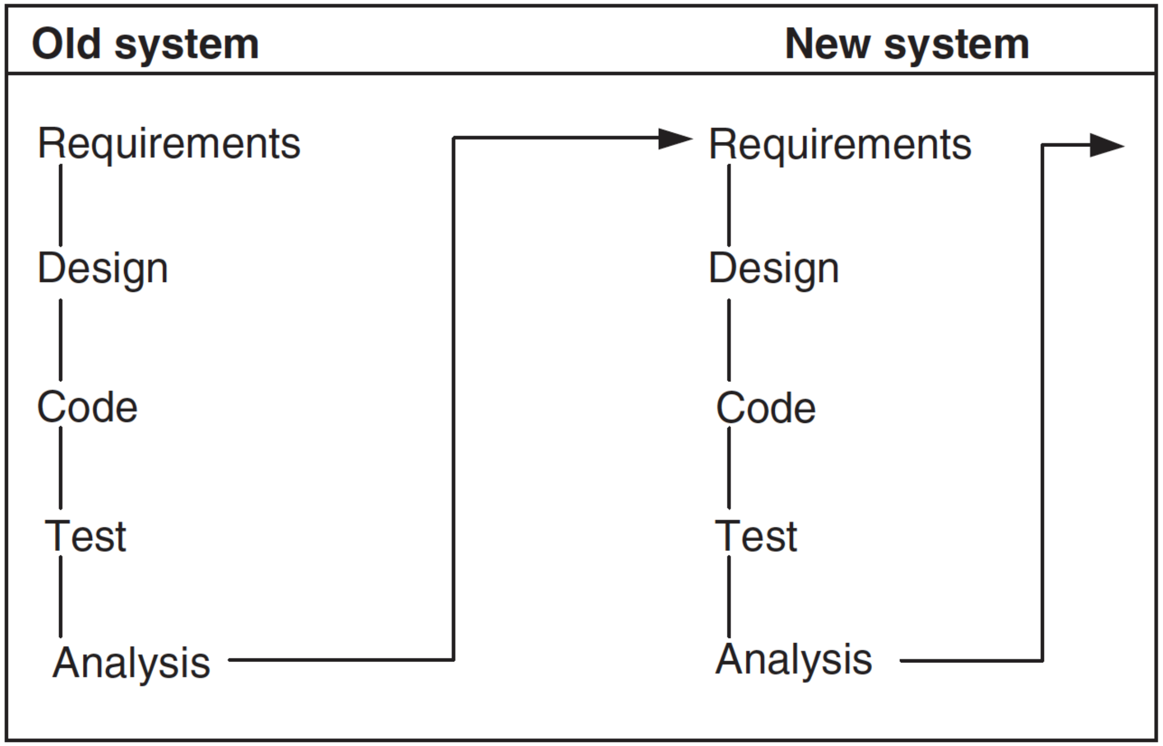


Iterative enhancement model

Iterative enhancement model

•Changes made based on an analysis of the existing system

•Attempts to control complexity and maintain good design

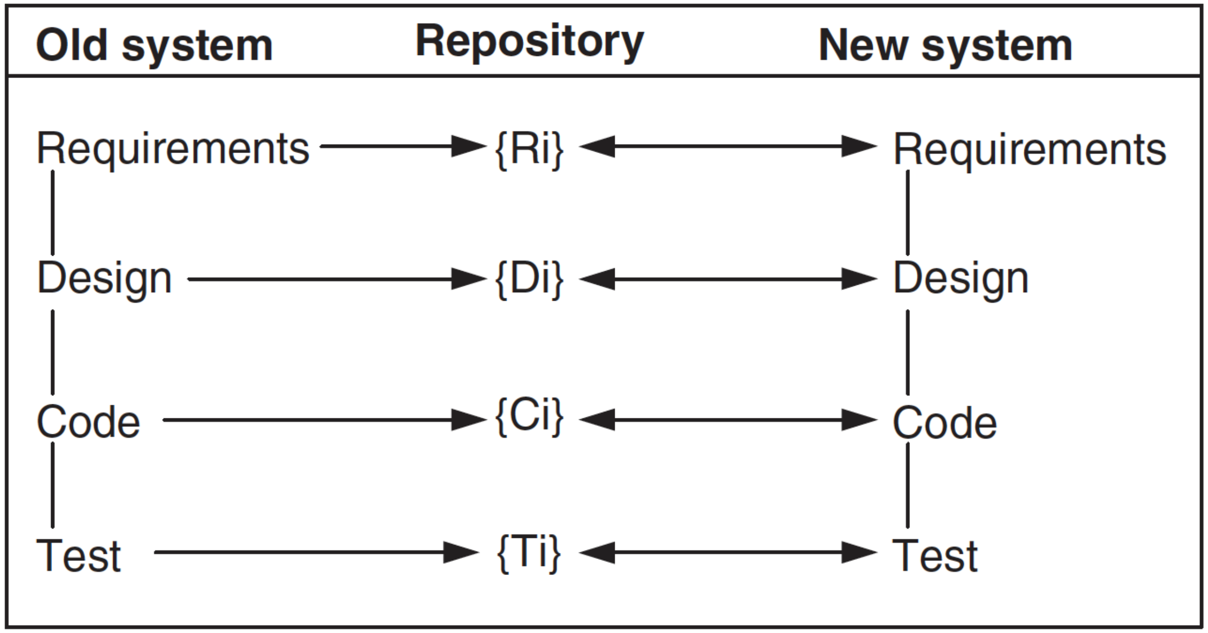


Full reuse model

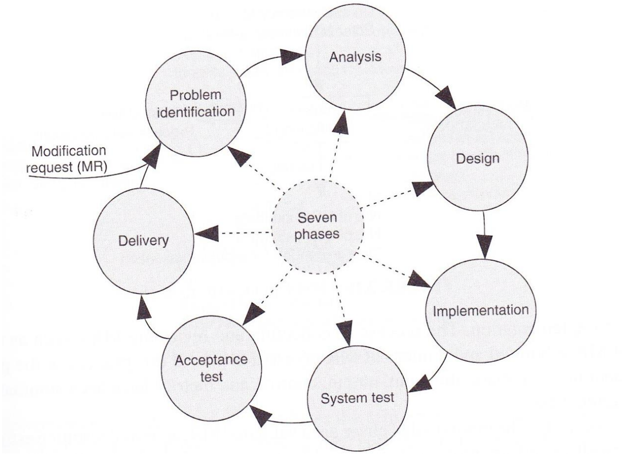
Full reuse model

•Starts with requirements for the new system, reusing as much as possible

•Needs a mature reuse culture to be successful



Software Maintenance- Steps (1)



- Understand the existing system
  - Study whatever form of documentation exists about the system to be modified
    - Often the only reliable source of information is the source code
  - Use tools to recover the high-level design models of the system

Software Maintenance - Steps (2)

- Define the maintenance objectives
  - Set the requirements
- Analysis
  - Evaluate alternatives for handling the modification
    - Estimate the costs and benefits of the alternative modifications
    - Perform *impact analysis*
      - Determine the effect of the change on the rest of the system

Software Maintenance-Steps (3)

- Design, implement, and test the changes

- Revalidate
  - Running *regression tests* to make sure that the unchanged code still works and is not adversely affected by the new changes

Software Maintenance-Steps (4)

- Train
  - Inform users of the changes
- Convert and release
  - Generate and release/install a new version with the modified parts
  - May involve *migrating* database schema changes and data at the same time

Program Comprehension

- Program comprehension:
  - The discipline concerned with studying the way software engineers understand programs
- Objective of those studying program comprehension:
  - design tools that will facilitate the understanding of large programs

Program Comprehension Strategies (1)

- The bottom-up model:
  - Comprehension starts with the source code and abstracting from it to reach the overall comprehension of the system
  - Steps:
    - Read the source code
    - Mentally group together low-level programming details (*chunks*) to build higher-level abstractions
    - Repeat until a high-level understanding of the program is formed

Program Comprehension Strategies (2)

- The top down model:
  - Comprehension <u>starts with a general idea, or hypothesis</u>, about how the system works
    - Often obtained from a very quick look at what components exist
  - Steps
    - First formulate hypotheses about the system functionality
    - Verify whether these hypotheses are valid or not
    - Create other hypotheses, forming a hierarchy of hypotheses
    - Continue until the low-level hypotheses are matched to the source code and proven to be valid or not

Program Comprehension Strategies (3)

- The Integrated Model:
  - Combines the top down and bottom up approaches
  - Empirical results show that maintainers tend to switch among the different comprehension strategies depending on
    - The code under investigation
    - Their expertise with the system

Reverse Engineering

- The process of analyzing a subject system
  - to identify the system's components and their interrelationships
  - and to create representations of the system, in another form, at a higher level of abstraction"

Chikofsky and Cross

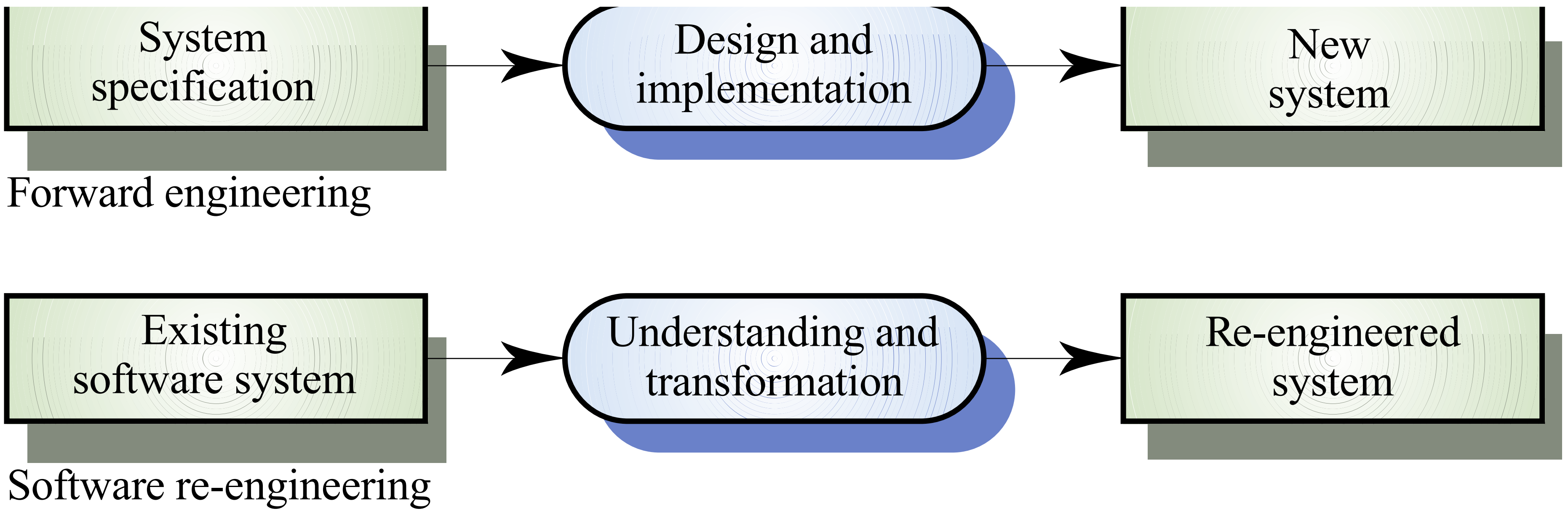
System Reengineering

- Re-structuring or re-writing part or all of a legacy system without changing its functionality

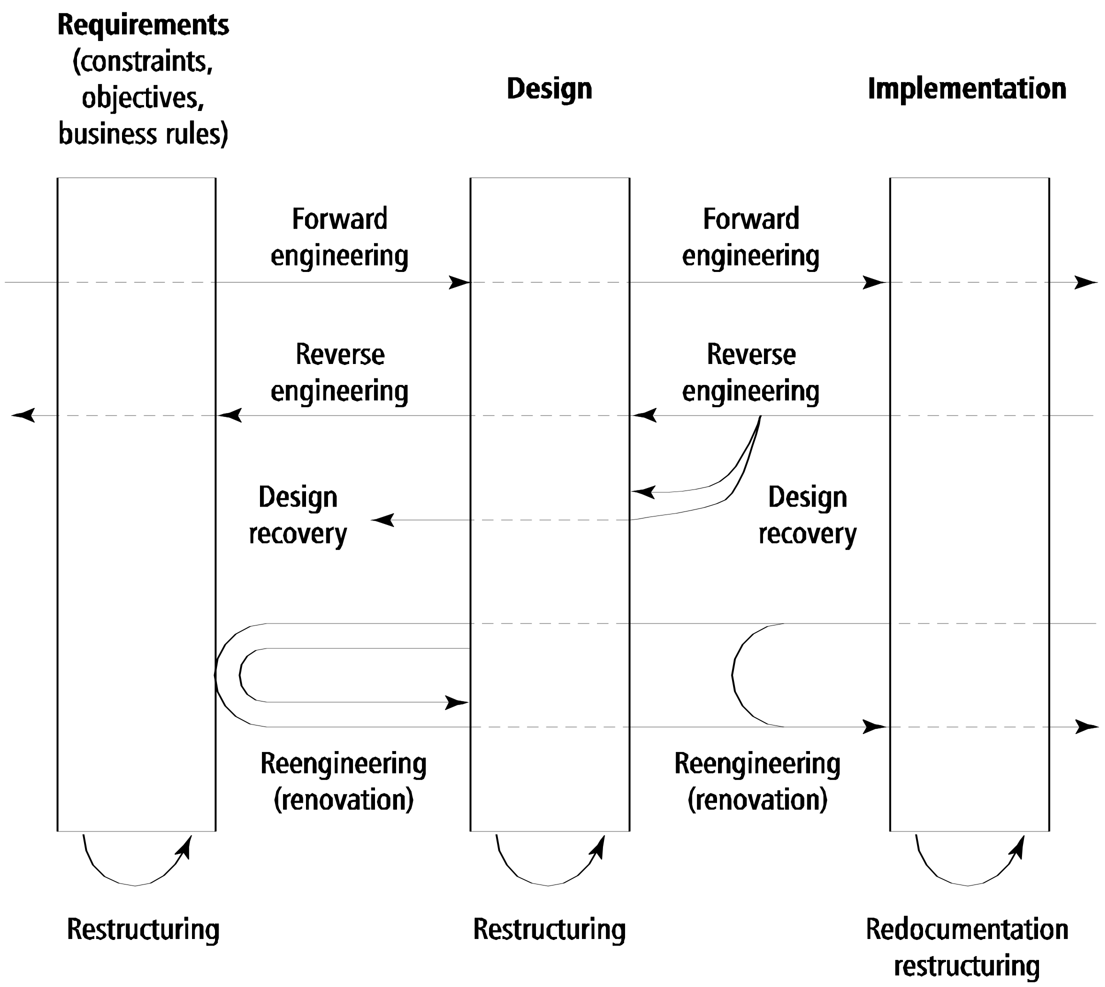
- Applicable where some but not all sub-systems of a larger system require frequent maintenance

- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented

Forward Engineering and Re-engineering



Reverse Engineering



Main levels of reverse engineering

- Binary reverse engineering
  - Take a binary executable
    - Recover source code you can then modify
  - Useful for companies that have lost their source code
  - Used extensively by hackers
  - Can be used legally, e.g. to enable your system to interface to existing system
  - Illegal in some contexts
- Source code reverse engineering
  - Take source code
    - Recover high level design information
  - By far the most widely performed type of reverse engineering
  - Binary reverse engineers also generally do this too

Reverse Engineering Objectives (1)

- Cope with complexity:
  - Have a better understanding of voluminous and complex systems
  - Extract relevant information and leave out low-level details
- Generate alternative views:
  - Enable the designers to analyze the system from different angles

Reverse Engineering Objectives (2)

- Recover lost information:
  - Changes made to the system are often undocumented;
    - This enlarges the gap between the design and the implementation

Reverse Engineering Objectives (3)

- Detect side effects:
  - Detect problems due to the effect a change may have on the system before it results in failure
- Synthesize higher-level abstractions

Reverse Engineering Objectives (4)

- Facilitate reuse
  - Detect candidate system components that can be reused

Advantages of reengineering

- Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

Refactoring

refactoring: the process of restructuring existing code without changing its external behavior.

primary goal:

improve code readability,

simplify maintenance, and

enhance the internal structure.

Refactoring & bad smells (1)

Single Responsibility Principle

Duplicated Code

Long Method

Large Class

Long Parameter List

Divergent Change: a class frequently needs to change in different ways for different reasons.

Shotgun Surgery: a single change requires modifying multiple classes or methods.

Refactoring & bad smells (2)

Data Clumps: the same group of variables appears in multiple places, likely should be positioned together. Refactoring by creating a class to encapsulate these variables.

Primitive Obsession Using primitive types for things that could be represented as objects

.

Switch Statements: Long switch or if-else chains can indicate that a class needs polymorphism.

Refactoring

Define functions for repetitive computing

any instead of loops

List comprehension

Replace *if* statement with *if* expression

Move *assignments* closer to their usage¶

Merge nested *if* statements

```python
numbers = [1, 2, 3, 4, 5]
result = []
for number in numbers:
    if number % 2 == 0:
        result.append(number)
```

```python
numbers = [1, 2, 3, 4, 5]
result = [number for number in numbers if number % 2 == 0]
```

```python
numbers = [-1, -2, -4, 0, 3, -7]
has_positives = False
for n in numbers:
    if n > 0:
        has_positives = True
        break

# -> refactor
has_positives = any(n > 0 for n in numbers)
```

```python
if condition:
    x = 1
else:
    x = 2

# -> refactor
x = 1 if condition else 2
```

```python
if a:
    if b:
        pass

# -> refactor
if a and b:
    pass
```