Chapter 11

Maintaining the System

Shari L. Pfleeger
Joann M. Atlee

4th Edition

Contents

11.1 The Changing System
11.2 The Nature of Maintenance
11.3 Maintenance Problems
11.4 Measuring Maintenance Characteristics
11.5 Maintenance Techniques and Tools
11.6 Software Rejuvenation
11.7 Information System Example
11.8 Real Time Example
11.9 What this Chapter Means for You

Chapter 11 Objectives

• System evolution
• Legacy systems
• Impact analysis
• Software rejuvenation

11.1 The Changing System

Maintenance: any work done to change the system after it is in operation

• Software does not degrade or require periodic maintenance
• However, software is continually evolving
  • Maintenance process can be difficult

Lehman’s System Types

• S-system: formally/mathematically defined, derivable from a specification
  • Matrix manipulation
• P-system: requirements based on approximate solution to a problem, but real-world remains stable
  • Chess program
• E-system: embedded in the real world and changes as the world does
  • Software to predict how economy functions (but economy is not completely understood)

11.1 The Changing System

S-System

Problem solved is related to the real world
11.1 The Changing System

P-System

- The solution produces information that is compared with the problem

![Diagram showing the comparison between the real world and the problem]

11.1 The Changing System

E-System

- It is an integral part of the world it models
- The changeability depends on its real-world context

![Diagram showing the changeability of the system]

11.1 The Changing System

Changes During the System Life Cycle

- S-system: un-changed
- P-system: incremental change
  - An approximate solution
  - Changes as discrepancies and omissions are identified
- E-system: constant change

11.1 The Changing System

Examples of Change During Software Development

<table>
<thead>
<tr>
<th>Activity from which initial change results</th>
<th>Artifacts requiring consequent change</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement analysis</td>
<td>Requirement specification</td>
</tr>
<tr>
<td>System design</td>
<td>Architectural design specification</td>
</tr>
<tr>
<td>Program design</td>
<td>Technical design specification</td>
</tr>
<tr>
<td>Program implementation</td>
<td>Program code</td>
</tr>
<tr>
<td>Program documentation</td>
<td>Program documentation</td>
</tr>
<tr>
<td>Unit testing</td>
<td>Test plans</td>
</tr>
<tr>
<td>System testing</td>
<td>Test plans</td>
</tr>
<tr>
<td>System testing</td>
<td>Test scripts</td>
</tr>
<tr>
<td>System delivery</td>
<td>User documentation</td>
</tr>
<tr>
<td></td>
<td>Operator documentation</td>
</tr>
<tr>
<td></td>
<td>System guide</td>
</tr>
<tr>
<td></td>
<td>Programmer's guide</td>
</tr>
<tr>
<td></td>
<td>Training classes</td>
</tr>
</tbody>
</table>

11.1 The Changing System

The System Life Span

- Will we need maintenance phase?
  - Even if best practices are followed, still need maintenance (because of E and P systems)
- Development time vs. maintenance time
  - Recent surveys: 20% vs 80%
- How much change can we expect?
  - System evolution vs. system decline: better to discard and build a new?
    - Cost/reliability/adaptability to change unacceptable?
    - Laws of software evolution

11.1 The Changing System

Development Time Vs. Maintenance Time

- Parikh and Zveigintzov (1983)
  - Development time: 2 years
  - Maintenance time: 5 to 6 years
- Fjedstad and Hamlen (1979)
  - 39% of effort in development
  - 61% of effort in maintenance
- 80–20 rule
  - 20% of effort in development
  - 80% of effort in maintenance
11.1 The Changing System
System Evolution vs. Decline

- Is the cost of maintenance too high?
- Is the system reliability unacceptable?
- Can the system no longer adapt to further change, and within a reasonable amount of time?
- Is system performance still beyond prescribed constraints?
- Are system functions of limited usefulness?
- Can other systems do the same job better, faster or cheaper?
- Is the cost of maintaining the hardware great enough to justify replacing it with cheaper, newer hardware?

11.1 The Changing System
Laws of Software Evolution

- Continuing change: leads to less utility
- Increasing complexity: structure deteriorates
- Fundamental law of program evolution: program obeys statistically-determined trends and invariants
- Conservation of organizational stability: global activity rate is invariant
- Conservation of familiarity: release content (changes) is statistically invariant

Sidebar 11.1 Bell Atlantic (Verizon) Replaces Three Systems with One Evolving One

- Sales Service Negotiation System (SSNS)
  - Replaced three legacy systems
  - The goals of the system changed from order-taking to needs-based sales
  - Replaced archaic commands with plain English
  - Originally written in C and C++, the system was modified with Java

11.2 The Nature of Maintenance
Types of Maintenance

- Corrective: maintaining control over day-to-day functions
- Adaptive: maintaining control over system modifications
- Perfective: perfecting existing functions
- Preventive: preventing system performance from degrading to unacceptable levels

Who Performs Maintenance

- Separate maintenance team
  - May be more objective
  - May find it easier to distinguish how a system should work from how it does work
- Part of development team
  - Will build the system in a way that makes maintenance easier
  - May feel over confident, and ignore the documentation to help maintenance effort

Maintenance Team Responsibilities

- Understanding the system
- Locating information in system documentation
- Keeping system documentation up-to-date
- Extending existing functions to accommodate new or changing requirements
- Adding new functions to the system
- Finding the source of system failures or problems
- Locating and correcting faults
- Answering questions about the way the system works
- Restructuring design and code components
- Rewriting design and code components
- Deleting design and code components that are no longer useful
- Managing changes to the system as they are made
11.2 The Nature of Maintenance
Use of Maintenance Time

- Graphical representation of distribution of maintenance effort (Lientz and Swanson)

![Graphical representation](image)

11.3 Maintenance Problems
The Need to Compromise

- Balancing need for change with the need for keeping the system available to users
  - Principles of SE compete with expediency and cost
- Fixing problem with quick but inelegant solution, or more involved but elegant way
  - Solving problem involves only the immediate correction of a fault
- Depend on the type of maintenance

11.3 Maintenance Problems
Factors Affecting Maintenance Approach

- The type of failures
- The failure’s critically or severity
- The difficulty of the needed changes
- The scope of the needed changes
- The complexity of the components being changed
- The number of physical locations at which the changes must be made

11.3 Maintenance Problems
Sidebar 11.2 The Benefits and Drawbacks of Maintaining OO System

- Benefits
  - Maintenance changes to a single object class may not affect the rest of the program
  - Maintainers can reuse objects easily
- Drawbacks
  - OO techniques may make programs more difficult to understand
  - Multiple parts can make it difficult to understand overall system behavior
  - Inheritance can make dependencies difficult to trace
  - Dynamic binding makes it impossible to determine which of several methods will be executed
  - By hiding the details of data structure, program function is often distributed across several classes

11.3 Maintenance Problems
Factors Affecting Maintenance Effort

- Application type
- System novelty
- Turnover and maintenance staff ability
- System life span
- Dependence on a changing environment
- Hardware characteristics
- Design quality
- Code quality
- Documentation quality
- Testing quality
Maintainability can be viewed in two ways:

- External view of the software: users, person performing maintenance
- Internal view of the software: measuring before delivery
11.4 Measuring Maintenance Characteristics

Internal Attributes Affecting Maintainability

- Cyclomatic number (McCabe, 1976)
  - The structural complexity of the source code
  - #linearly independent paths
  - Based on graph theoretic concept

Example for Calculating Cyclomatic Number

Consider the following code

```cpp
Scoreboard::drawscore(int n) {
    while(numdigits-- > 0) {
        score[numdigits]->erase();
    }
    // build new score in loop, each time update position
    numdigits = 0;
    // if score is 0, just display "0"
    if (n == 0) {
        delete score[numdigits];
        score[numdigits] = new Displayable(digits[0]);
        score[numdigits]->move(Point((700-numdigits*18),40));
        score[numdigits]->draw();
        numdigits++;
    }
    while (n) {
        int rem = n % 10;
        delete score[numdigits];
        score[numdigits] = new Displayable(digits[rem]);
        score[numdigits]->move(Point(700-numdigits*18),40));
        score[numdigits]->draw();
        n /= 10;
        numdigits++;
    }
}
```

#Linearly independent paths = e - n + 2
- e: edges, n: nodes

Other Measures: readability

- Fog index: textual products, readability affects maintainability
  - \( F = 0.4 \times \frac{\text{number of words}}{\text{number of sentences}} + \frac{\text{percentage of words of three or more syllables}}{\text{sentences}} \)
- De Young and Kampen readability
  - \( R = 0.295a - 0.499b + 0.13c \)
  - a: the average normalized length of variable
  - b: number of lines containing statements
  - c: McCabe’s cyclomatic number

Sidebar 11.4 Models of Fault Behavior

- Hatton and Hopkins (1989) studied the NAG Fortran scientific subroutine library
  - Smaller components contained proportionately more faults than larger ones
- They note similar evidence
  - at Siemens
  - Ada code at Unisys
  - Fortran products at NASA Goddard

Sidebar 11.5 Maintenance Measures at Hewlett-Packard

- Used maintainability index
  - Index was calibrated with a large number of metrics
  - A tailored polynomial index was calculated using extended cyclomatic number, lines of code, number of comments, and an effort measure
  - The polynomial was applied to 714 components containing 236,000 lines of C code developed by third party
11.5 Maintenance Techniques and Tools

- Configuration management
  - Configuration control board
  - Change control
- Impact analysis
- Automated maintenance tools

11.5 Maintenance Techniques and Tools

Configuration Control Process

- Problem discovered by or change requested by user/customer/developer, and recorded
- Change reported to the configuration control board
- CCB discusses problem: determines nature of change, who should pay
- CCB discusses source of problem, scope of change, time to fix; they assign severity/priority and analyst to fix
- Analyst makes change on test copy
- Analyst works with librarian to control installation of change
- Analyst files change report

11.5 Maintenance Techniques and Tools

Change Control Issues

- Synchronization: When was the change made?
- Identification: Who made the change?
- Naming: What components of the system were changed?
- Authentication: Was the change made correctly?
- Authorization: Who authorized that the change be made?
- Routing: Who was notified of the change?
- Cancellation: Who can cancel the request for change?
- Delegation: Who is responsible for the change?
- Valuation: What is the priority of the change?

11.5 Maintenance Techniques and Tools

Impact Analysis

- The evaluation of many risks associated with the change, including estimates of effects on resources, effort, and schedule
- Helps control maintenance cost

11.5 Maintenance Techniques and Tools

Software Maintenance Activities

- Graph illustrates the activities performed when a change is requested

11.5 Maintenance Techniques and Tools

Measuring Impact of Change

- Workproduct: any development artifact whose change is significant
- Horizontal traceability: relationships of components across collections of workproducts
- Vertical traceability: relationships among parts of a workproduct
11.5 Maintenance Techniques and Tools

**Horizontal Traceability**
- The graphical relationships and traceability links among related workproducts

![Horizontal Traceability Diagram](image)

**Sidebar 11.6 Applying Traceability to Real-World System**
- Five kinds of traceability
  - object-to-object
  - association-to-association
  - use case-to-use case
  - use case-to-object
  - two-dimensional object-to-object
- How tracing is performed
  - Using explicit links
  - Using textual references to different documents
  - Using names and concepts that are the same and similar
  - Using knowledge and domain knowledge

![Sidebar 11.6 Diagram](image)

**11.6 Software Rejuvenation**
- Redocumentation: static analysis adds more information
- Restructuring: transform to improve code structure
- Reverse engineering: recreate design and specification information from the code
- Reengineering: reverse engineer and then make changes to specification and design to complete the logical model; then generate new system from revised specification and design

![Sidebar 11.7 Panvalet](image)
11.6 Software Rejuvenation

**Taxonomy**

- Graph illustrates the relationship among the four types of rejuvenation

---

**Redocumentation**

- Begins by submitting the code to an analysis tool
- Output may include:
  - component calling relationships
  - data–interface tables
  - data–dictionary information
  - data flow tables or diagrams
  - control flow tables or diagrams
  - pseudocode
  - test paths
  - component and variable cross-references

---

**Redocumentation Process**

- Redocumentation process

---

**Restructuring Activities**

- Interpreting the source code and representing it internally
- Simplifying the internal representation
- Regenerating structured code

---

**Restructuring Activities (continued)**

- Graph illustrates the three major activities involved in restructuring: (1) static analysis (2) simplification of the representations (3) refined representation used to generate a structured version

---

**Reverse Engineering**

- Attempting to recover engineering information based on software specification and design methods
- Obstacles remain before reverse engineering can be used universally
  - Real time system problem
  - Extremely complex system
11.6 Software Rejuvenation

Reverse Engineering Process

- Graph depicts the reverse-engineering process

11.6 Software Rejuvenation

Reengineering

- An extension of reverse engineering
  - produces new software code without changing the overall system function
- Reengineering steps
  - The system is reverse-engineered
  - The software system is corrected or completed
  - The new system is generated

11.6 Software Rejuvenation

Reengineering Process

- Graph illustrates the steps in reengineering process

11.6 Software Rejuvenation

Sidebar 11.8 Reengineering Effort

- The U.S National Institute of Standard and Technology (NIST) studied the results of reengineering 13,131 lines of COBOL source statements using automatic translation
  - Entire effort took 35 person-months
- Boehm points out that original COCOMO model estimated 152 person months for reengineering the same type of system, clearly unacceptable level of accuracy
  - COCOMO II has been revised to include a factor for automatic translation

11.7 Information System Example

Piccadilly System

- The software can not be an S-system
  - the problem may change dramatically
- The software can not be a P-system
  - P-system requires a stable abstraction, while Piccadilly changes constantly
- The software must be E-system
  - The system is an integral part of the world it models

11.8 Real-Time Example

Ariane–5

- Developers focused on mitigating random failure
  - The inertial reference system failed because of a design fault, not a result of a random failure
- Needs to change the failure strategy and implement a series of preventive enhancements
11.9 What this Chapter Means for You

- The more a system is linked to the real world, the more likely it will change and the more difficult it will be to maintain
- Maintainers have many jobs in addition to software developers
- Measuring maintainability is difficult
- Impact analysis builds and tracks links among the requirements, design, code, and test cases
- Software rejuvenation involves redocumenting, restructuring, reverse engineering, and reengineering