Software Maintenance

Chapter 14

Software Maintenance

- Your system is developed…
  - It is deployed to customers…
    - What next?
- Maintenance
  - Categories of maintenance tasks
  - Major causes of problems
  - Reverse engineering
  - Management of maintenance activities

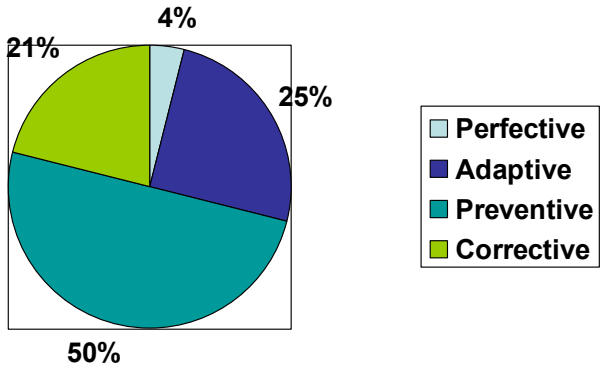
IEEE Definition

- Maintenance is
  - The process of modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment.
  - More than fixing bugs!
- Estimates:
  - More than 100 billion lines of code in production in the world
  - As much as 80% is unstructured, patched, and badly documented
  - Try a monster.com search on “COBOL”

Maintenance Activities

- Corrective
  - Repair of faults that are discovered
- Adaptive
  - Modify software to changes in the environment, e.g. new hardware, new OS.
  - No change in functionality.
- Perfective
  - Accommodate new or changed user requirements.
  - Changes in functionality.
- Preventive
  - Activities aimed at increasing maintainability, e.g. documentation, code arrangement.

% Effort



- Maintenance is also 50% of the total lifecycle costs
- Hasn't changed over the last twenty years – evolution inevitable!

Addressing Maintenance

- Higher quality code, better test procedures, better documentation, adherence to standards and conventions
- Design for change
- Prototyping and fine-tuning to user needs can help reduce perfective maintenance
- Write less code
  - Reuse

Major Causes of Maintenance

- Anecdote by David Parnas on re-engineering software for fighter planes
  - Plane has two altimeters
  - Onboard software tries to read either meter and display the result
  - Code, can be deciphered with a little effort:

```
IF not-read1(V1) GOTO DEF1;
display(V1);
GOTO C;
DEF1: IF not-read2(V2) GOTO DEF2;
display(V2);
GOTO C;
DEF2: display(3000);
C:
```

Better, Structured Version

```
If read-meter1(V1) then
        display(V1)
else if read-meter2(V2) then
        display(V2)
else
        display(3000);
endif
```

- But why the number 3000?
- Magic Number

Parnas Anecdote

- 3000:
  - Programmer asked fighter pilots what average flying altitude was, used that in case neither altimeter was readable
- Re-engineering
  - Plane flies high or low, not at average very often, wanted to re-write the code to display a warning instead, e.g. “PULL UP”
  - Denied since pilots were trained to react to the default message, even put in the manual: “If altimeter reads 3000 for more than a second, pull up”

Parnas Anecdote

- Illustrates major causes of maintenance problems
  - Unstructured Code
  - Maintenance programmer has insufficient knowledge of the system or application domain
  - Documentation absent, out of date, or insufficient
- One other:
  - Maintenance has a bad image

Laws of Software Evolution

- The laws of software evolution also force maintenance to occur
- Law of increasing complexity
  - A program that is changed becomes less and less structured and thus becomes more complex. One has to invest extra effort in order to avoid the increase in entropy.
- Law of continuing change
  - A system that is being used undergoes continuing change, until it is judged more cost-effective to restructure the system or replace it by a completely new version

Scant Knowledge Available

- Maintenance programmers generally lack detailed knowledge about the system or application domain
  - Problem in general, but worse for maintenance
  - Often scarce sources available to reference
  - Usually requires going to the source code to figure out how the system works
  - Experienced programmers have learned to distrust documentation, usually insufficient and out of date
    - When a quick-fix is done, the documentation is often not updated to reflect changes

Limited Understanding

- Pfleeger:
  - 47% of software maintenance effort devoted to understanding the software
    - E.g. if there are n modules and we change k of them, for each changed module we need to know possible interactions with the other n-1 modules
  - >50% of effort can be attributed to lack of user understanding
    - Incomplete or mistaken reports of errors and enhancements

Common knowledge problems

- A design rationale is often missing
  - Why 3000?
  - Programmers tend to document how the code works, not the rationale behind the decisions for the code
  - Maintenance programmers must reconstruct the decisions and may do so incorrectly
- Maintenance Programming in Opportunistic Mode
  - Maintenance programmers abstract a structure out of the code based on stereotypical solutions to problems
  - If these assumptions are incorrect, the programmer may encounter further problems

Negativity

- Maintenance sometimes considered second-rate job
  - Goes to inexperienced, one gets promoted to development
  - Generally lower salary
  - Affects morale, try to change jobs, high turnover
  - But maintenance actually requires programmers with the most experience
    - Less documentation, often more time pressures, bulk of the lifecycle

Reverse Engineering

- The process of analyzing a system to
  - Identify the system's components and interrelationships
  - Create representations of the system in another form or at a higher level of abstraction
- Akin to the reconstruction of a lost blueprint
- Redocumentation
  - Derive a semantically equivalent description at the same level of abstraction, e.g. change formatting, coding standards, flowcharts
- Design Recovery
  - Derive a semantically equivalent description at a higher level of abstraction, e.g. derive UML diagram from source code
  - Some tools available to help do this, e.g. Rational Rose

Restructuring vs. Reengineering

- Restructuring
  - Transformation of a system from one representation to another at the same level of abstraction
  - Functionality of system does not change
  - Revamping: UI is modernized, spaghetti-like code organized into objects
- Reengineering or Renovation
  - Real changes made to the system in terms of functions
  - Often followed by a traditional forward engineering requirements phase

Maintenance Mindset

- Maintenance programmers study the original program code one and a hafl times as long as its documentation
- Maintenance programmers spend as much time reading the code as they do implementing a change
- Result of the source code being the only truly reliable source of information
- How does the programmer study the source?

Maintenance Mindset

- Programming plan
  - A program fragment that corresponds to a stereotypical action
  - E.g. loop to sum a series of numbers, process all elements in an array
- Beacon
  - A key feature that indicates the presence of a particular structure or operation
  - E.g.
    - temp = x[i];
    - x[i]=x[j];
    - x[j]=temp;
- If the beacons or plan inherent in the code don't correspond to the actual design, the maintenance programmer is in for a tough time

Maintenance Strategies

- As-Needed
  - To maintain a feature, the programmer goes directly to the code for that feature, hypotheses formulated on the basis of local information
- Systematic
  - An overall understanding of the system is formed by a systematic top-down study of the program text.
  - Gives better insight into causal relationships between program components than As-Needed, can better avoid ripple effects

Code Example

- What does the following do?

```
boolean A[][];
…

for i:=1 to n do
        for j:=1 to n do
                if A[j][i] then
                        for k:=1 to n do
                                if A[i][k] thenA[j][k]:=true
                        endfor
                endif
        endfor
endfor
```

Second Example

- Mostly incomprehensible code fragment

```
Procedure A(var x:w)
        begin
                b(y,n1);
                b(x,n2);
                m(w[x]);
                y:=x;
                r(p[x])
        end;
```

```
Procedure A(var nw: window)
        begin
          border(current_win, HIGHLIGHT);
          border(nw,HIGHLIGHT);
          move_cursor(w[nw]);
          current_win:=nw;
          resume(process[nw]);
        end;
```

Maintenance Organization

- Two approaches to maintenance
- Throw it over the wall approach
  - A new team is responsible for maintenance
  - Advantages
    - Clear accountability to separate cost and effort for maintenance from new development investment
    - Intermittent and unpredictable demands on maintenance make it hard for people to do both
    - Separation motivates development team to clean up the code before handoff
    - Team can become more specialized on maintenance, service-orientation as opposed to product-orientation, can increase maintenance productivity
  - Disadvantages
    - Investment in knowledge and experience is lost
    - Coordination efforts take time
    - Maintenance becomes a reverse engineering challenge
    - De-motivation due to status differences
    - Possible duplication of communication to users

Maintenance Organization

- Mission orientation
  - Development team make a long term commitment to maintaining the software
  - Reverse advantages/disadvantages of a separate organization for maintenance
- Which to use?
  - Most express slight preference for separate organization units, with careful procedures to mitigate the disadvantages

Service Perspective to Software Maintenance

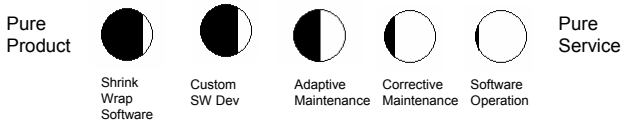
- Survey of aspects of software quality customers consider most important:
  - Service responsiveness
  - Service capacity
  - Product reliability
  - Service efficiency
  - Product functionality
- Maintenance often can be seen as providing a service to end users as opposed to delivering a product
- Should judge maintenance quality differently from development

Services vs. Products

- Services are
  - Intangible
  - Depends on factors difficult to control
    - Ability of customers to articulate needs
    - Willingness of personnel to satisfy needs
    - Level of demand for service
  - Produced and consumed simultaneously
    - Centralization and mass production difficult
  - Perishable
- Product vs. Service not clear cut

Software Product/Service

- Continuum for software development and maintenance



Gap Model

- Used to illustrate differences between perceived service delivery and expected services
- Gap 1
  - Expected service perceived by the service provider differs from the service expected by the customer
  - Often caused by service provider focusing on something different than the customer wants, e.g. on features instead of maximum availability
  - Customer service expectations should be translated into clear service agreements

Gap Model

- Gap 2
  - Service specification differs from the expected service as perceived by the service provider.
  - E.g. customer expects a quick restart of system in the event of a crash, but service provider is focused on analyzing the reasons
  - Would hopefully be caught in a review of service requirements
- Gap 3
  - Actual service delivery differs from specified services.
  - Often caused by deficiencies in human resource policies, failure to match demand and supply, customers not filling their role.
  - E.g. not tracking bugs adequately, insufficient staff, customers bypassing help desk to programmers

Gap Model

- Gap 4
  - Communication about the service does not match the actual service delivery.
  - Ineffective management of customer expectations, promising too much, or ineffective horizontal communications.
  - E.g. customer not updated on reported bugs
  - Can address with bug tracking, helpdesk tools, proper managerial mindset of a customer/service orientation

IEEE 1219

- Process for controlling maintenance changes
  - Identify and classify change requests
    - Each CR is given an tracking ID, classified into a maintenance category
    - Analyzed to see if it will be accepted, rejected, further evaluation
    - Cost estimate
    - Prioritized
  - Analysis of change requests
    - Decision made which CR's to address based on cost, schedule, etc.
  - Implement the change
    - Design, implementation, testing of the change with corresponding new documentation

Reality: Quick-Fix Model

- As changes are needed, you take the source code, make the changes, and recompile to get a new version
  - Quick and cheap now, but rapidly degrades the structure of the software as patches are made upon patches
  - Should at least update docs and higher-level designs after code fixed, but often this is left out, and only done as time permits
  - Should only be done for emergency fixes
- Non emergency situations
  - Planned releases with new versions, change log

Other Models

- Iterative enhancement model
  - Changes made based on an analysis of the existing system
  - Start with highest level document affected by changes, propagate the change down through the remaining documents and code
  - 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

Quality Issues

- Software quality affects maintenance effort
- Should use measurement techniques previously discussed to ensure good quality
- Observed trends:
  - Studies have found correlations to complexity metrics like Cyclomatic complexity to maintenance efforts
  - If certain modules require frequent change or much effort to realize change, a redesign should be given serious consideration

Redesign

- When to redesign and reengineer the entire system?
- The more of the following you have, the greater the potential for redesign
  - Frequent system failures
  - Code over seven years old
  - Overly-complex program structure and logic flow
  - Code written for previous generation hardware
  - Very large modules
  - Excessive resource requirements
  - Hard-coded parameters
  - Difficult keeping maintenance personnel
  - Deficient documentation
  - Missing or incomplete design specs

Summary

- Maintenance activities made up of
  - Corrective
  - Adaptive
  - Perfective
  - Preventive
- Most work spent in Perfective maintenance
- Software evolution makes maintenance inescapable
- Problems can be mitigated by avoiding poor documentation, unstructured code, insufficient knowledge about the system or design for the maintenance programmers
- Maintenance requires a service mentality