Software Maintenance (Chapter 14)

Mark van den Brand
Tom Verhoeff

TU/e Technische Universiteit Eindhoven University of Technology

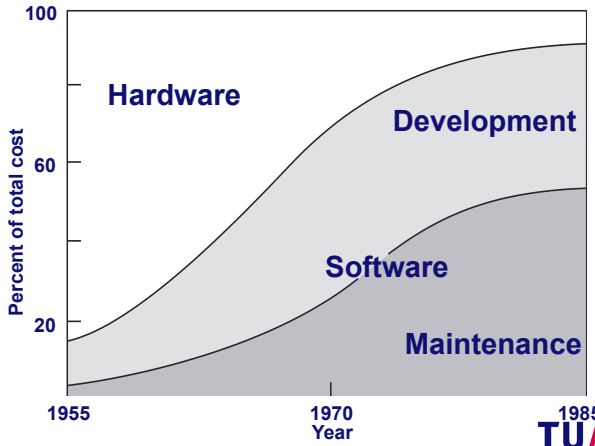
Where innovation starts

Software Maintenance

- Main issues:
  - why maintenance is such an issue
  - reverse engineering and its limitations
  - how to organize maintenance

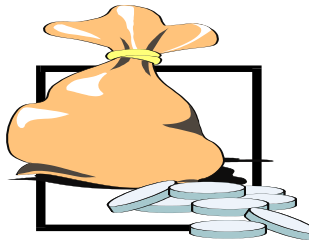
TU/e Technische Universiteit Eindhoven University of Technology

Relative distribution of software/ hardware costs



TU/e Technische Universiteit Eindhoven University of Technology

Point to ponder #1



- Why does software maintenance cost so much?

TU/e Technische Universiteit Eindhoven University of Technology

Software Maintenance, definition

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

Maintenance is thus concerned with:

- correcting errors found after the software has been delivered
- adapting the software to changing requirements, changing environments, ...

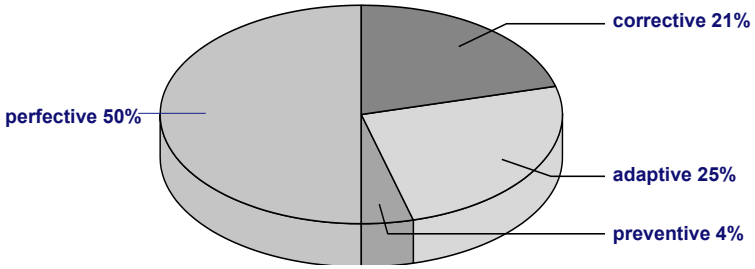
Key to maintenance is in development

- Higher quality ⇒ less (corrective) maintenance
- Anticipating changes ⇒ less (adaptive and perfective) maintenance
- Better tuning to user needs ⇒ less (perfective) maintenance
- Less code ⇒ less maintenance

Kinds of maintenance activities

- *corrective maintenance*: correcting errors
- *adaptive maintenance*: adapting to changes in the environment (both hardware and software)
- *perfective maintenance*: adapting to changing user requirements
- *preventive maintenance*: increasing the system's maintainability

Distribution of maintenance activities



Growth of maintenance problem

- 1975: ~75,000 people in maintenance (17%)
- 1990: 800,000 (47%)
- 2005: 2,500,000 (76%)
- 2015: ??

(Numbers from Jones (2006))

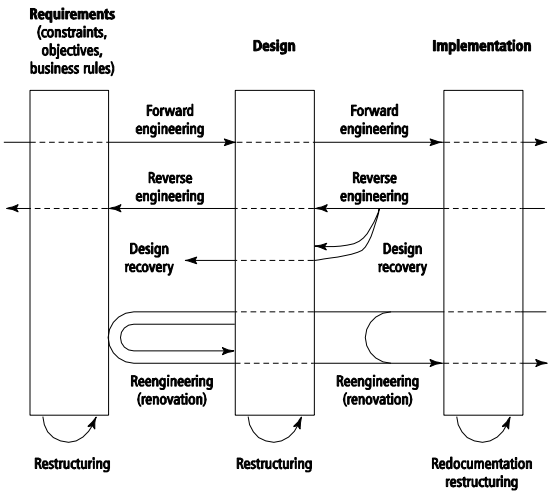
Shift in type of maintenance over time

- Introductory stage: emphasis on user support
- Growth stage: emphasis on correcting faults
- Maturity: emphasis on enhancements
- Decline: emphasis on technology changes

Major causes of maintenance problems

- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation

Reverse engineering



Reverse engineering

- Does not involve any adaptation of the system
- Akin to reconstruction of a blueprint
- *Design recovery*: result is at higher level of abstraction
- *Redocumentation*: result is at same level of abstraction

Restructuring

- Functionality does *not* change
- From one representation to another, at the same level of abstraction, such as:
  - From spaghetti code to structured code
  - *Refactoring* after a design step in agile approaches
  - Black box restructuring: add a wrapper
  - With platform change: *migration*

Reengineering (renovation)

- Functionality *does* change
- Then reverse engineering step is followed by a forward engineering step in which the changes are made

Refactoring in case of bad smells

- Long method
- Large class
- Primitive obsession
- Data clumps
- Switch statements
- Lazy class
- Duplicate code
- Feature envy
- Inappropriate intimacy
- …

Categories of bad smells

- Bloaters: something has grown too large
- Object-oriented abusers: OO not fully exploited
- Change preventers: hinder further evolution
- Dispensables: can be removed
- Encapsulators: deal with data communication
- Couplers: coupling too high

Program comprehension

- Role of programming plans, beacons
- As-needed strategy vs systematic strategy
- Use of outside knowledge (domain knowledge, naming conventions, etc.)

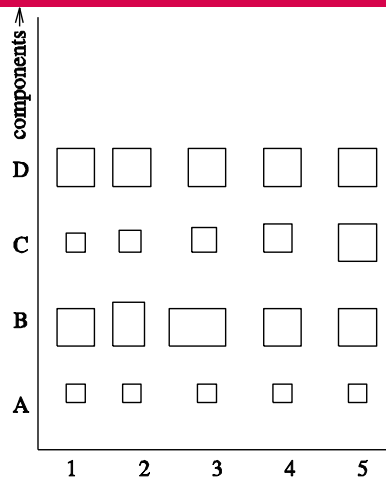
Software maintenance tools

- Tools to ease perceptual processes (reformatters)
- Tools to gain insight in static structure
- Tools to gain insight in dynamic behavior
- Tools that inspect version history

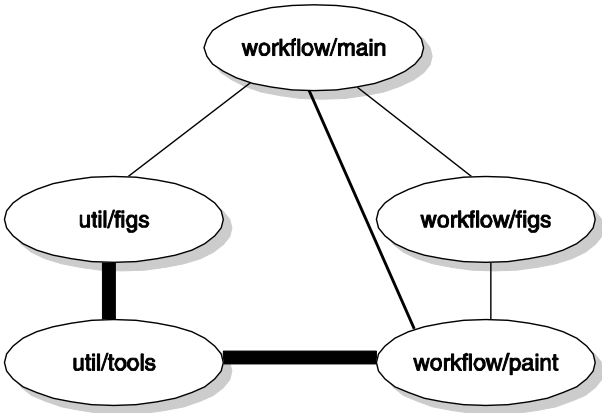
Analyzing software evolution data

- Version-centered analysis: study differences between successive versions
- History-centered analysis: study evolution from a certain viewpoint (e.g. how often components are changed together)

Example version-centered analysis



Example history-centered analysis



Organization of maintenance

- W-type: by work type (analysis vs programming)
- A-type: by application domain
- L-type: by life-cycle type (development vs maintenance)
- L-type found most often

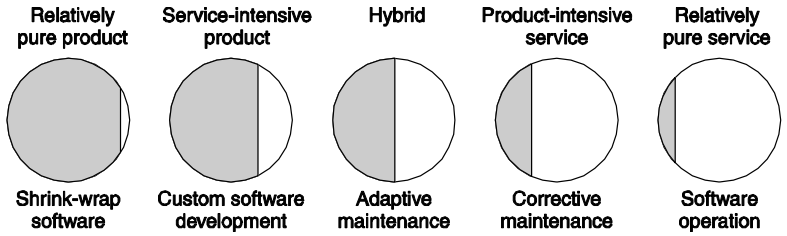
Advantages of L-type departmentalization

- Clear accountability
- Development progress not hindered by unexpected maintenance requests
- Better acceptance test by maintenance department
- Higher QoS by maintenance department
- Higher productivity

Disadvantages of L-type departmentalization

- Demotivation of maintenance personnel because of status differences
- Loss of system knowledge during system transfer
- Coordination costs
- Increased acceptance costs
- Duplication of communication channels with users

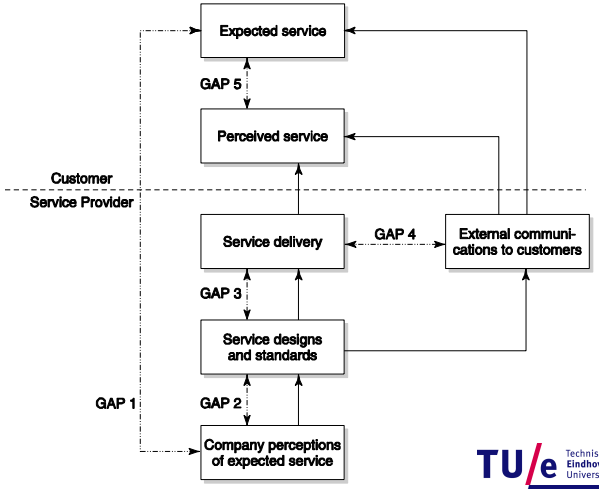
Product-service continuum and maintenance



Service gaps

1. Expected service as perceived by provider differs from service expected by customer
2. Service specification differs from expected service as perceived by provider
3. Service delivery differs from specified services
4. Communication does not match service delivery

Gap model of service quality



Maintenance control

- **Configuration control:**
  - Identify, classify change requests
  - Analyze change requests
  - Implement changes
- **Fits in with *iterative enhancement model* of maintenance (first analyze, then change)**
- **As opposed to *quick-fix model* (first patch, then update design and documentation, if time permits)**

Indicators of system 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**

SUMMARY

- **Most of maintenance is (*inevitable*) evolution**
- **Maintenance problems:**
  - **Unstructured code**
  - **Insufficient knowledge about system and domain**
  - **Insufficient documentation**
  - **Bad image of maintenance department**
- **Lehman's 3rd law: a system that is used, *will* change**