Software Engineering

Natallia Kokash
email: nkokash@liacs.nl

Leiden Institute of Advanced Computer Science

Software maintenance

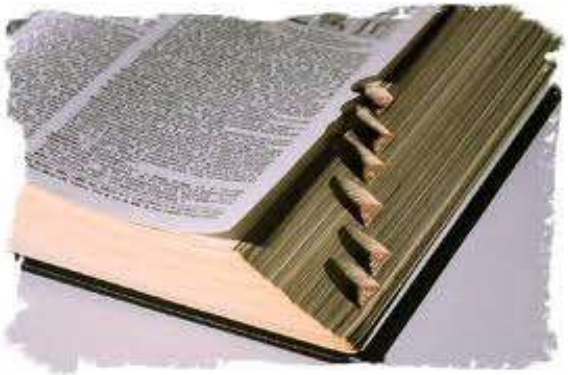
- Definition
- Software maintenance issues
- Maintenance and evolution
- Reverse engineering (refactoring, bad smells)
- Program comprehension
- Maintenance tools
- Organization and control of maintenance

Need for software maintenance

- 100 billion lines of code
- 80% of it is unstructured, patched and badly documented

- A multinational bank:
  - 100+ offices
  - 10+ mainframes at a central site
  - 1000+ workstations
  - 24 * 7 availability
  - 100+ application systems
  - 500K+ LOC
  - Obsolete languages (Fortran, COBOL)
  - Huge databases

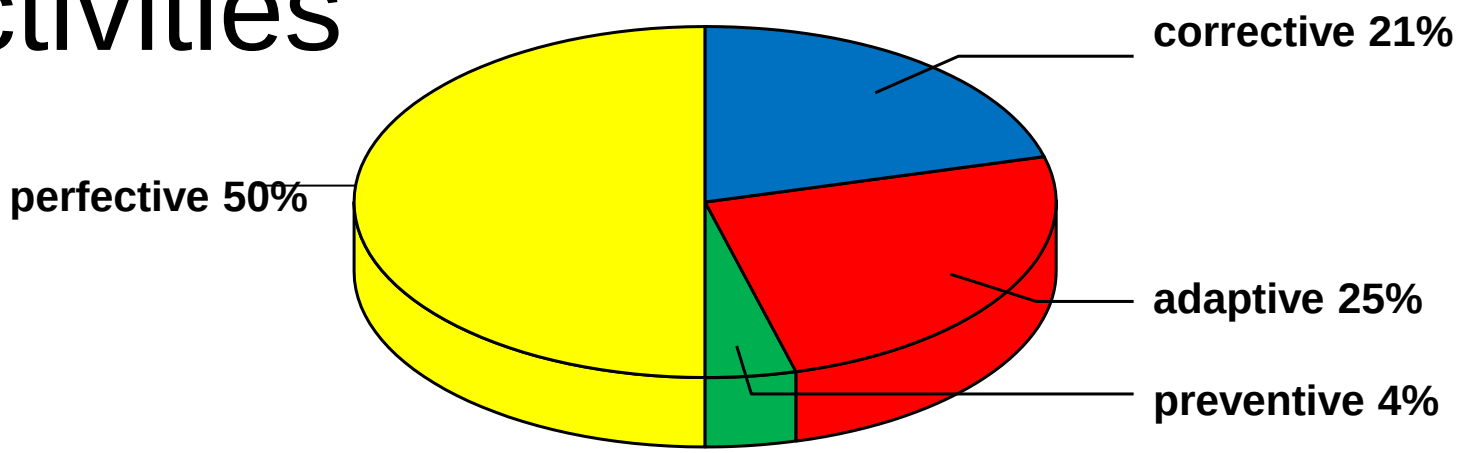
Definition



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

Distribution of maintenance activities



- *Corrective maintenance*: correcting discovered errors
- *Preventive maintenance*: correcting latent errors
- *Adaptive maintenance*: adapting to changes in the environment
- *Perfective maintenance*: adapting to changing user requirements

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))

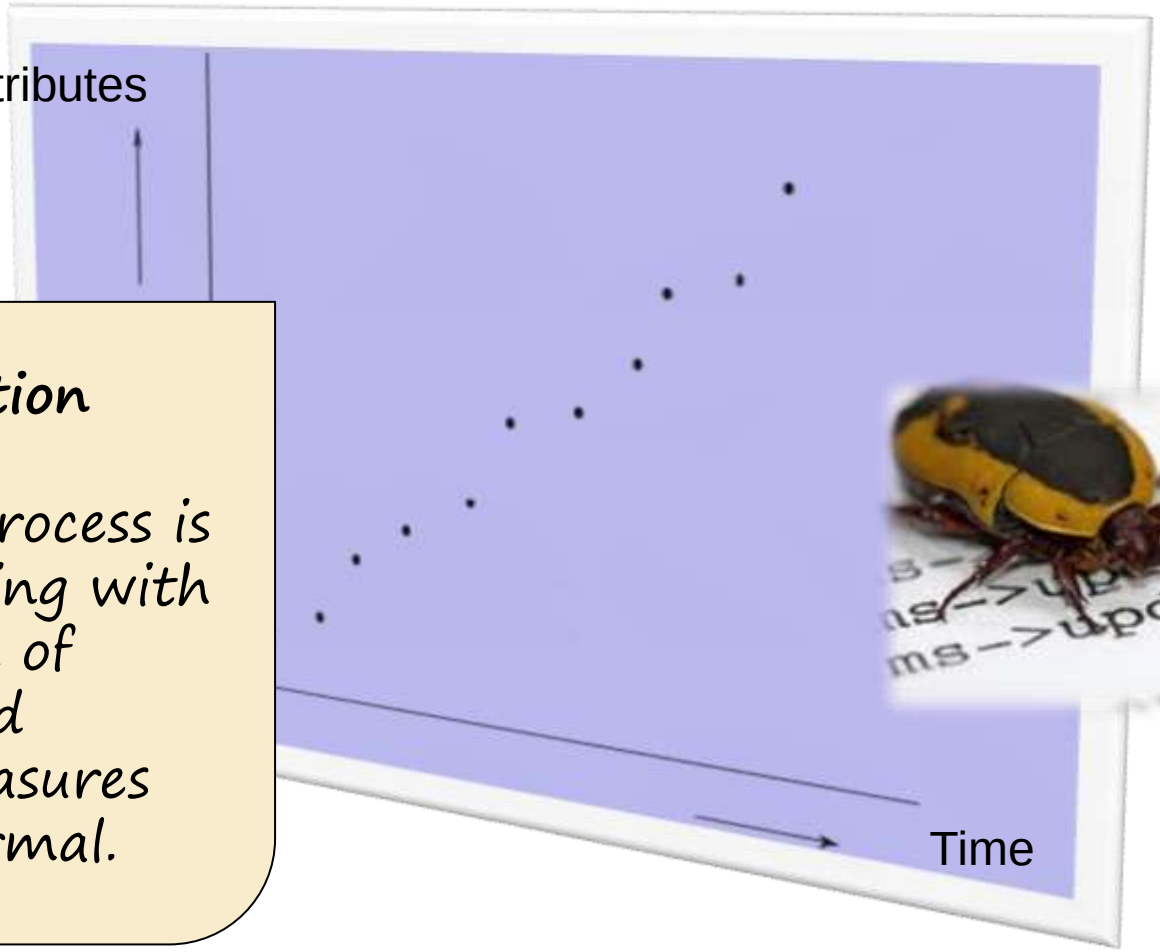
Maintenance or Evolution

- **Some observations:**
  - Systems are not built from scratch
  - There is time pressure on maintenance
  - Software is embedded in the real world and become part of it, thereby changing it.
  - This leads to a feedback system where the program and its environment evolve in concert.

Laws of software evolution
(Lehman 1974)

- **Continuing Change**
  - Programs must be continually adapted or they become progressively less satisfactory.
- **Increasing Complexity**
  - As a program evolves its complexity increases unless work is done to maintain or reduce it.
- **Self Regulation**
  - Evolution process is self regulating with distribution of product and process measures close to normal.

Illustration of the third law



Self Regulation

Evolution process is self regulating with distribution of product and process measures close to normal.

Laws of software evolution

(Lehman 1974)

- **Conservation of Organizational Stability** (invariant work rate)
  - The average effective global activity rate in an evolving system is invariant over product lifetime
- **Conservation of Familiarity** (incremental growth)
  - As a system evolves all associated with it (e.g., developers) must maintain mastery of its content and behavior to achieve satisfactory evolution. Excessive growth diminishes that mastery.

Laws of software evolution

(… and later)

- **Continuing Growth**
  - The functional content of systems must be continually increased to maintain user satisfaction over their lifetime.
- **Declining Quality**
  - The quality of systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.
- **Feedback System**
  - Evolution processes constitute multi-level, multi-loop, multi-agent feedback systems and must be treated as such.

Fighter plane control system

```
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:
```

Major causes of maintenance problems

- Unstructured code
- Insufficient domain knowledge
- Insufficient documentation

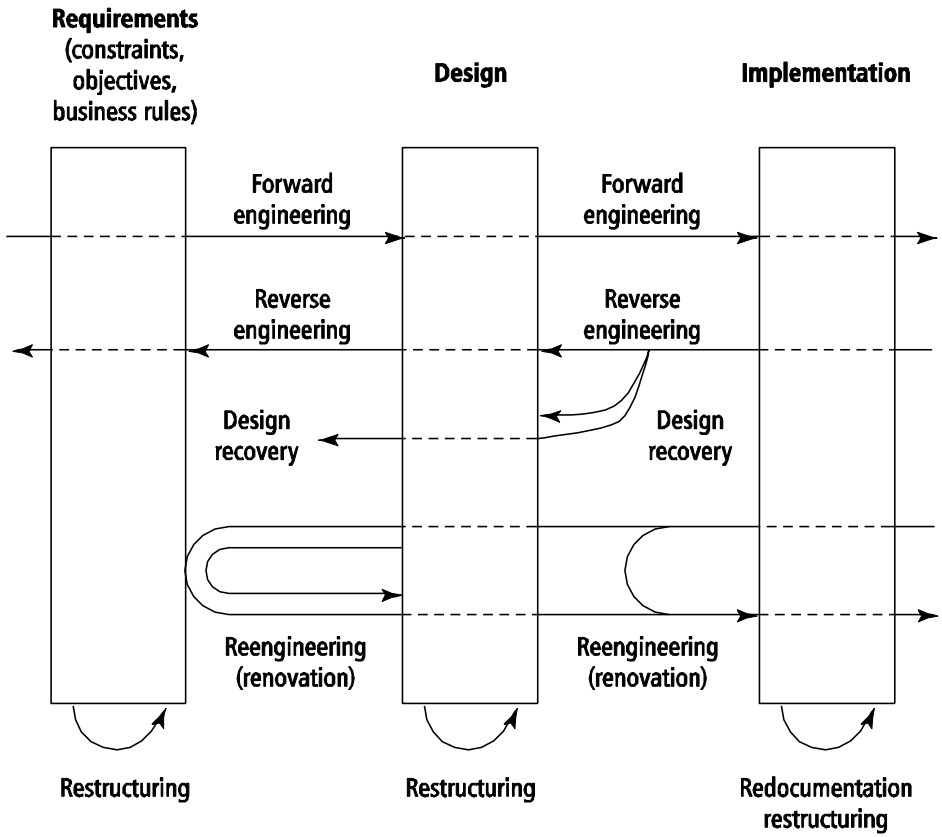
Key to maintenance is in development

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

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

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 (1)

- Long method
- Large class
- Primitive obsession
- Long parameter list
- Data clumps
- Switch statements
- Temporary field
- Refused bequest
- Alternative classes with different interfaces
- Parallel inheritance hierarchies

Refactoring in case of bad smells (2)

- Lazy class
- Data class
- Duplicate code
- Speculative generality
- Message chains
- Middle man
- Feature envy
- Inappropriate intimacy
- Divergent change
- Shotgun surgery
- Incomplete library class

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

Categories of bad smells

Category	Bad smell
Bloaters	Long method, Large class, Primitive obsession, Long parameter list, Data clumps
OO abusers	Switch statements, Temporary field, Refused bequest, Alternative classes with different interfaces, Parallel inheritance hierarchies
Change preventers	Divergent change, Shortgun surgery
Dispensables	Lazy class, Data class, Duplicate code, Speculative generality
Encapsulators	Message chains, Middle man
Couplers	Feature envy, Inappropriate intimacy
Others	Incomplete library class, comments

What does this code do?

```
for (i=1; i<n; i++){
   for (j=1; j<n; j++){
        if (A[i,j]){
               for (k=1; k<n; k++) {
                     if (A[i,k]) A[j,k]=true;
               }
        }
   }
}
```

- Warshall's algorithm to compute a transitive closure of a relation (graph)
- What is "transitive closure"? "Relation"?

Program comprehension

- 50% of time
- Programming plans
  - fragments that correspond to stereotypical actions
- Beacons
  - key features that represents the presence of a particular structure or operation
  - e.g., **swap** operation indicates **sort**
- 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
  - See lecture 2 (Configuration management)

Bug/defect tracking systems

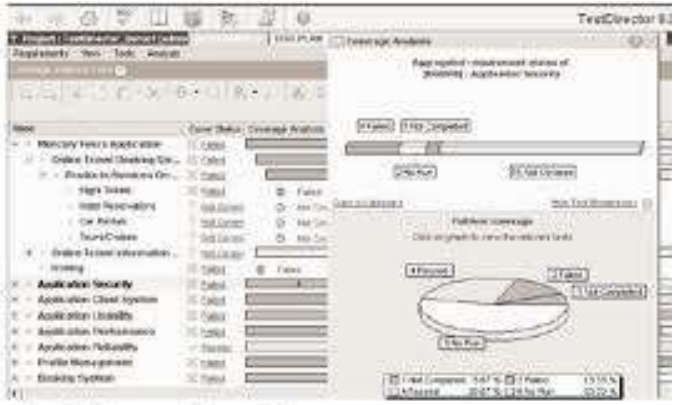
- **Bugzilla** by Mozilla foundation
- **Test Director** by Mercury Interactive
- **Silk Radar** by Segue Software
- **SQA Manager** by Rational software
- **QA director** by Compuware
- **HP Quality Center**
- **IBM Rational Quality Manager**
  - Information presented through custom-designed dashboards
- **Micro Focus SilkPerformer**
  - Performs load tests

Clone finding tools

- **Black Duck Suite** - software analyzing suite
- **CCFinder** (C/C++, Java, COBOL, Fortran, etc.)
- **Checkstyle** (Java)
- **CloneAnalyzer** (C/C++ and Java / Eclipse plug-in)
- **Clone Digger** (Python and Java)
- **CloneDR** - (Ada, C, C++, C#, Java, COBOL, Fortran, Python, VB.net, VB6, PHP4/5, PLSQL, SQL2011, XML, many others)
- **Copy/Paste Detector (CPD) from PMD** (Java, JSP, C, C++, Fortran, PHP)
- **ConQAT** (ABAP, ADA, Cobol, C/C++, C#, Java, PL/I, PL/SQL, Python, Text, Transact SQL, Visual Basic, XML)
- **JPlag** (Java, C#, C, C++, Scheme, natural language text)
- **Pattern Miner (CP Miner)**
- **Simian (**Similarity Analyzer**)** software
- **Google CodePro Analytix** - (Java / Eclipse plug-in)

Maintenance management tools

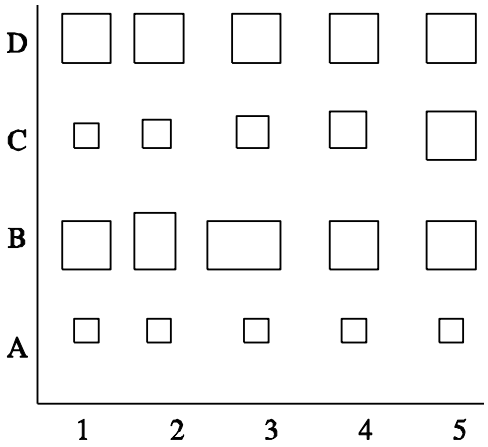
- **ProTeus III Expert CMMS**
  - mid-size maintenance management program for one to four users
  - schedule preventative maintenance
  - generate automatic work orders
  - document equipment maintenance history
  - track assets and inventory
  - track personnel
  - create purchase orders
  - generate reports

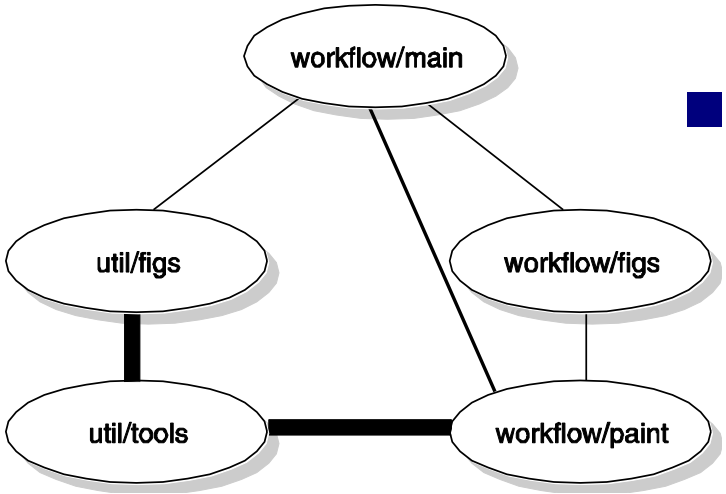


TestDirector's Requirements Manager links test cases to testing requirements, ensuring traceability.

- **Resharper**
  - Code inspection, refactoring, navigation, analysis
- **NDepend**
  - Tool to manage .NET code
  - **Software quality** can be measured using *code metrics*, visualized using *graphs* and *treemaps,* and enforced using *standard* and *custom rules*
  - Also **JDepend** for Java
  - See lecture 5 (Software quality)

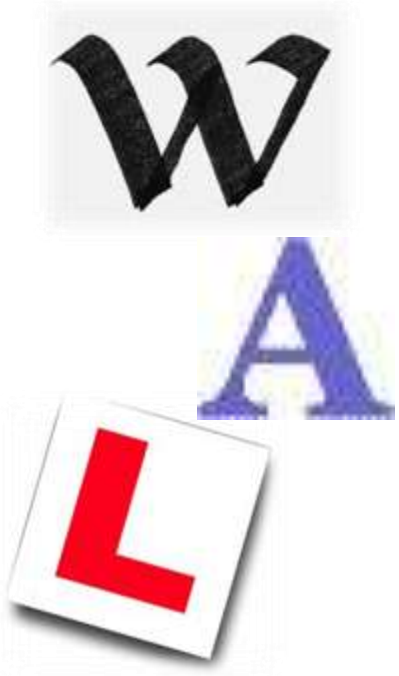
Analyzing software evolution data





- 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)

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

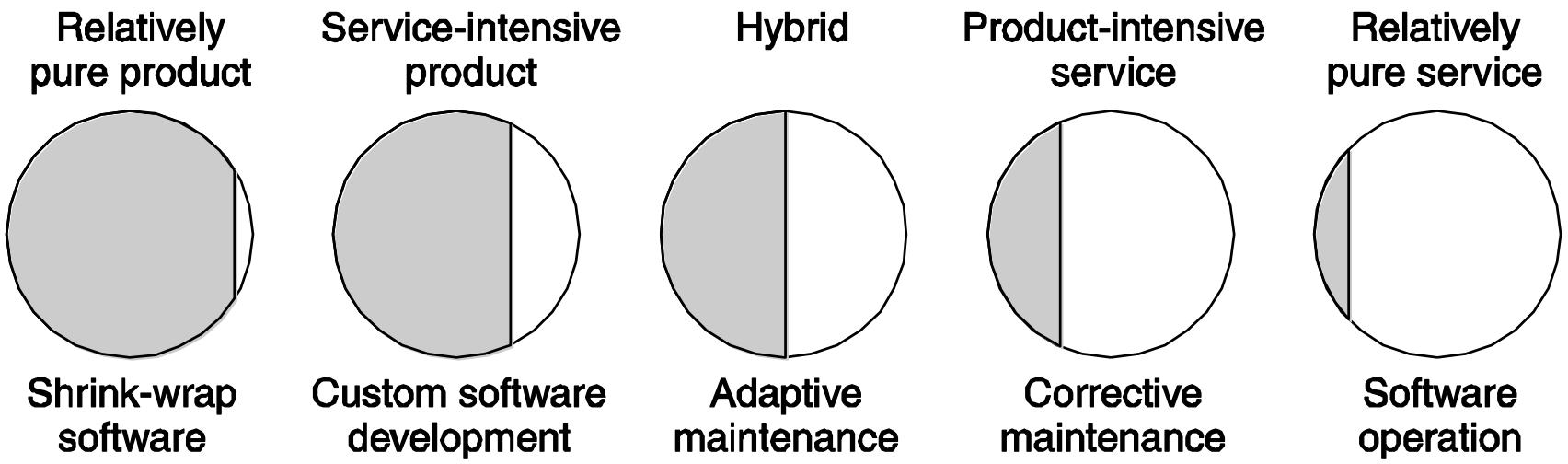
Advantages of L-type departmentalization

- 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

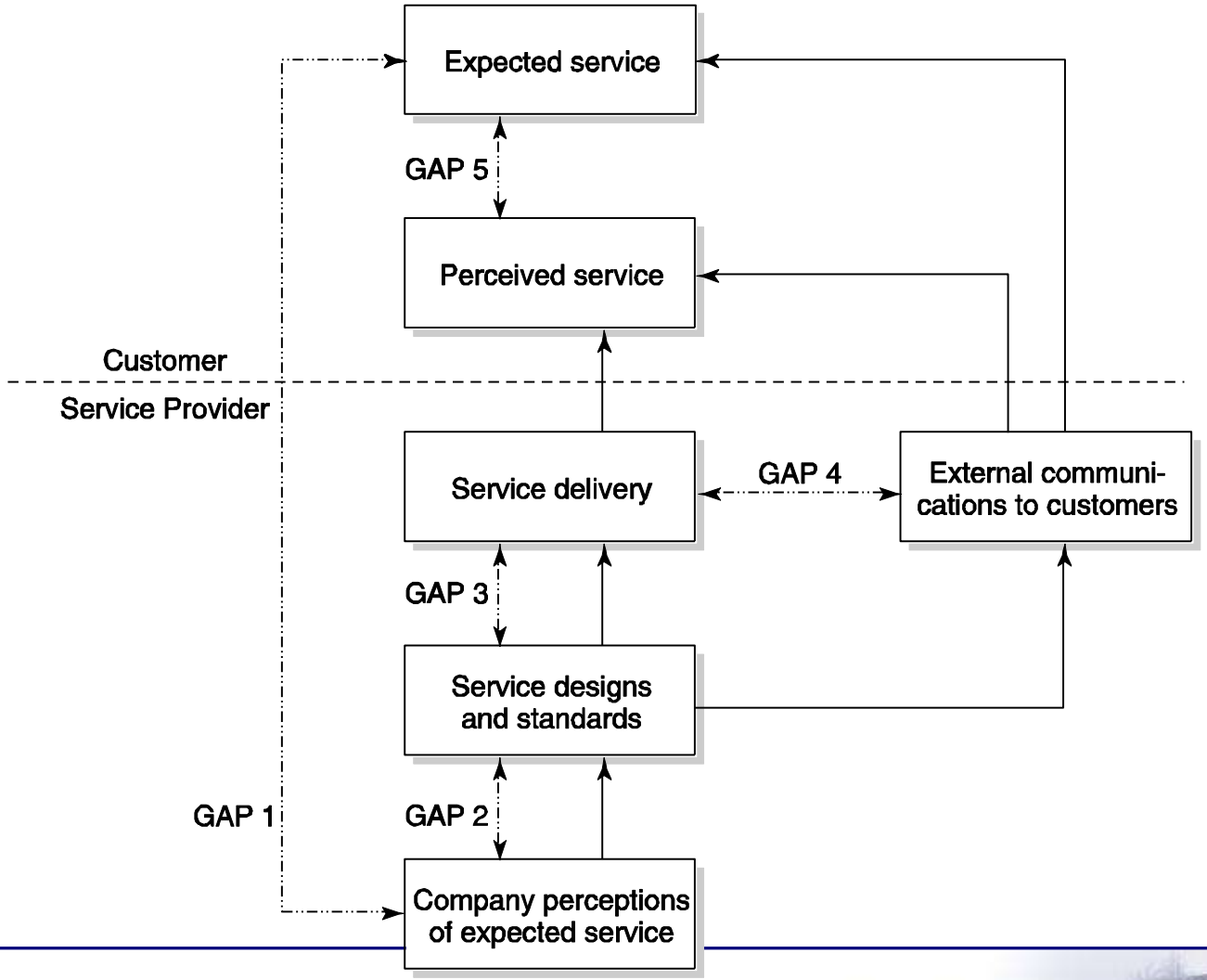
Product-service continuum and maintenance



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)

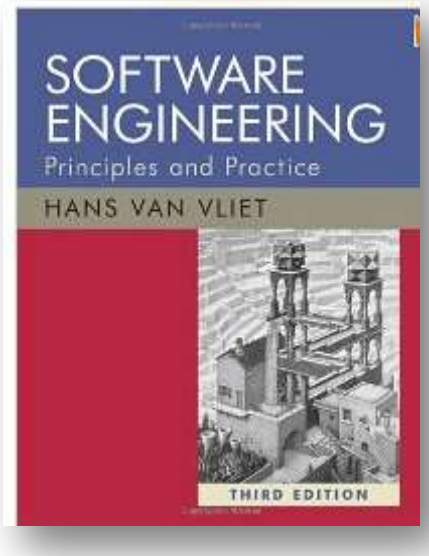
Indicators of system decay

- 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

- Read chapter 14



- 11. Software Architecture
- 12. Software Design
- 13. Software Testing
- 14. Software Maintenance
- 17. Software Reusability
- 18. Component-Based Software Engineering
- 19. Service Orientation
- 20. Global Software Development