

UNIT 7

Software Maintenance

(3 Hours) — Software Engineering (CACS253)

7.1 Evolving Nature of Software

Software development does not end when the system is delivered. Software **evolves** because business needs, technology, laws and user expectations keep changing. **Software maintenance** is the general process of changing a system after it has been delivered. Lehman's laws describe how large programs behave over time:

Lehman's Law	Meaning
Continuing change	A program in use must continually be adapted or it becomes progressively less useful.
Increasing complexity	As a program evolves, its structure degrades and complexity grows unless work is done to reduce it.
Large program evolution	Program attributes such as size and error rate are approximately invariant for each release.
Organisational stability	The rate of development is approximately constant regardless of resources devoted.
Conservation of familiarity	Each release adds roughly the same amount of change, since users and developers must stay familiar with the system.
Declining quality	Quality appears to decline unless the system is rigorously maintained and adapted to its environment.

Note: Software does not wear out, but its structure deteriorates as it is modified. Maintenance is unavoidable and often lasts many years, sometimes decades.

7.2 Different Types of Maintenance

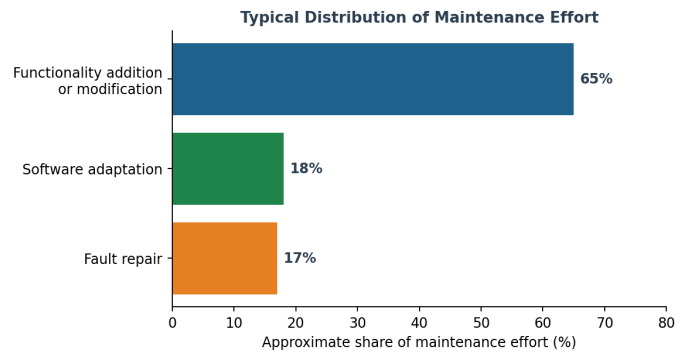


Figure 7.1: Approximate distribution of maintenance effort

Type	Description	Example
Fault repair (corrective)	Fixing coding errors (cheap), design errors (more expensive) and requirements errors (most expensive – may require extensive redesign).	Correcting a wrong interest calculation reported by users
Software adaptation (adaptive)	Changing the software to work in a new or changed environment – hardware, operating system, platform, regulations.	Porting an app to a new OS version; updating tax rates
Functionality addition or modification (perfective)	Extending the system to satisfy new or changed requirements and improving performance or maintainability.	Adding a new report; improving response time
Preventive	Restructuring software to prevent future problems and to improve maintainability.	Refactoring code, updating documentation

In practice these types overlap. Studies show that most maintenance effort goes into adding or modifying functionality, not fixing faults. Maintenance costs are typically **2 to 4 times** the development cost for long-lived systems, because it is usually more expensive to add functionality after a system is in operation than to implement the same functionality during development.

Reasons why maintenance is expensive:

- **Team stability** – the original development team may have disbanded and new staff do not understand the system.
- **Poor development practice** – contracts separate development from maintenance so there is no incentive to write maintainable software.
- **Staff skills** – maintenance staff are often inexperienced and unfamiliar with the domain.
- **Program age and structure** – as programs age their structure degrades, and they become harder to understand and change; documentation is missing or outdated.

7.3 Maintenance Prediction

Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs. It helps managers decide how much to invest in changes and whether to accept a change request, re-engineer or replace.

- **Change acceptance** depends on the maintainability of components affected by the change.
- **Implementing changes** degrades the system and reduces its maintainability.
- **Maintenance costs** depend on the number of changes, and the cost of each change depends on maintainability.

Prediction Question	Indicator (Metric)
What parts of the system are most likely to be affected by changes?	Number and complexity of system interfaces; number of volatile system requirements
What parts will be the most expensive to maintain?	Complexity of system components (cyclomatic complexity); processes that use them
What will be the lifetime maintenance cost?	Number of change requests; number of outstanding requests; average time to implement a request
How many change requests can be expected?	Number of requests for corrective maintenance; average time for impact analysis

Process metrics used to assess maintainability: number of requests for corrective maintenance, average time required for impact analysis, average time taken to implement a change request and number of outstanding change requests. If any or all of these are increasing, it may indicate a decline in maintainability.

7.4 Re-engineering

Software re-engineering is restructuring or rewriting part or all of a legacy system without changing its functionality. It is applicable where some but not all sub-systems of a larger system require frequent maintenance. It involves adding effort to make the system easier to maintain; the system may be restructured and re-documented.

Activity	Description
Source code translation	Convert code to a newer version of the language or a different language.
Reverse engineering	Analyse the program to understand and recover its design and specification (documentation is regenerated).
Program structure improvement	Restructure control logic to make it easier to read and understand.
Program modularisation	Related parts of the program are grouped together and redundancy is removed.

Activity	Description
Data re-engineering	Clean up and restructure data and databases to match the improved program.

Re-engineering	New Software Development
Reduced risk – critical software already exists	High risk – specification and development problems
Lower cost – typically a fraction of new development cost	Higher cost
Functionality preserved	Functionality must be re-specified
Limited to improvements possible without changing behaviour	Full freedom of new architecture and technology

*Note: Re-engineering improves structure and understandability; **refactoring** does this continuously in small steps during development, while re-engineering is done after a system has been in operation for some time.*

7.5 Configuration Management (CM)

Software systems are constantly changing during development and use. **Configuration management** is the development and use of standards and procedures for managing an evolving software system. It controls the costs and effort involved in making changes to a system and ensures that everyone works with the correct version of each item.

Importance of CM

- Different versions of the system (for different hardware, customers or fixes) must be tracked.
- Multiple developers work on the same code: CM prevents overwriting and conflicts.
- Allows rolling back to a previous stable version when a change fails.
- Provides traceability between requirements, design, code and tests.
- Supports consistent builds and releases, and controlled change of baselines.

Configuration Items

A **configuration item (CI)** is any work product placed under configuration control – anything that can change and must be tracked. A **baseline** is a formally reviewed and agreed version of a set of configuration items that serves as the basis for further development and can be changed only through a formal change-control procedure.

Category	Configuration Items
Documents	SRS, design documents, project plan, user manual, test plan
Source items	Source code files, scripts, build files, libraries
Data	Test data, database schemas, configuration files
Executables	Object code, installers, release packages

Versioning

Version management keeps track of the different versions of software components. A **version** is an instance of a system that differs in some way from other instances. A **release** is a version that has been distributed to customers. A common numbering scheme is **major.minor.patch**.

Version Numbering – Example

2.4.1 → Major version 2 (large changes/new architecture), minor 4 (new features, backward compatible), patch 1 (bug fixes).

Version 1.0 → 1.1 (small enhancement) → 1.1.1 (bug fix) → 2.0 (major redesign).

- **Repository** stores all versions; developers *check out* to work and *check in* (commit) changes.
- **Branching and merging** allows parallel development streams (e.g., a release branch and a development branch).
- Tools: Git, SVN, Mercurial; CI servers such as Jenkins.

CM Activities

Activity	Description
Configuration identification	Identify configuration items and give each a unique name/ID.
Change control	Evaluate, approve or reject change requests via a Change Control Board (CCB), then implement and record.
Version control	Manage versions and branches of configuration items.
Configuration status accounting	Record and report the state and history of all items and changes.
Configuration audit	Verify that the configuration items match requirements and that baselines are consistent.
Release management	Build, package and distribute releases.

Change management process: change request submitted → impact analysis (cost, effort, risk) → change control board approves or rejects → change implemented → validated and tested → new version released and records updated.

Key Takeaways

- ✓ Software must evolve; without maintenance it becomes less useful (Lehman's laws).
- ✓ Maintenance types: **fault repair, software adaptation, functionality addition/modification** (largest share), plus preventive.
- ✓ Maintenance often costs more than development; predict it with complexity and change-request metrics.
- ✓ Re-engineering restructures and re-documents a legacy system without changing functionality.
- ✓ CM manages versions, baselines and change control; configuration items include code, documents and data.
- ✓ Version numbering major.minor.patch; use repositories, check-in/check-out and change control boards.

— *End of Unit 7* —