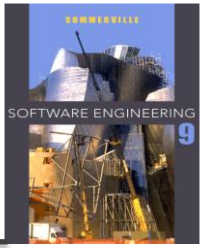
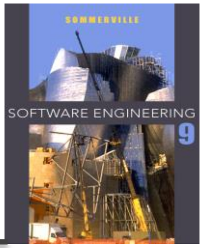


Chapter 9 – Software Evolution

Lecture 1

Topics covered

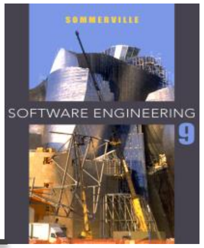


✧ Evolution processes
- Change processes for software systems

✧ Program evolution dynamics
- Understanding software evolution

✧ Software maintenance
- Making changes to operational software systems

Software change

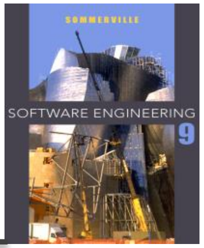


✧ Software change is inevitable

- New requirements emerge when the software is used;
- The business environment changes;
- Errors must be repaired;
- New computers and equipment is added to the system;
- The performance or reliability of the system may have to be improved.

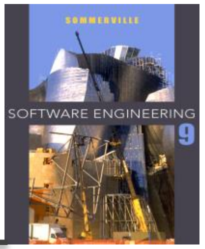
✧ A key problem for all organizations is implementing and managing change to their existing software systems.

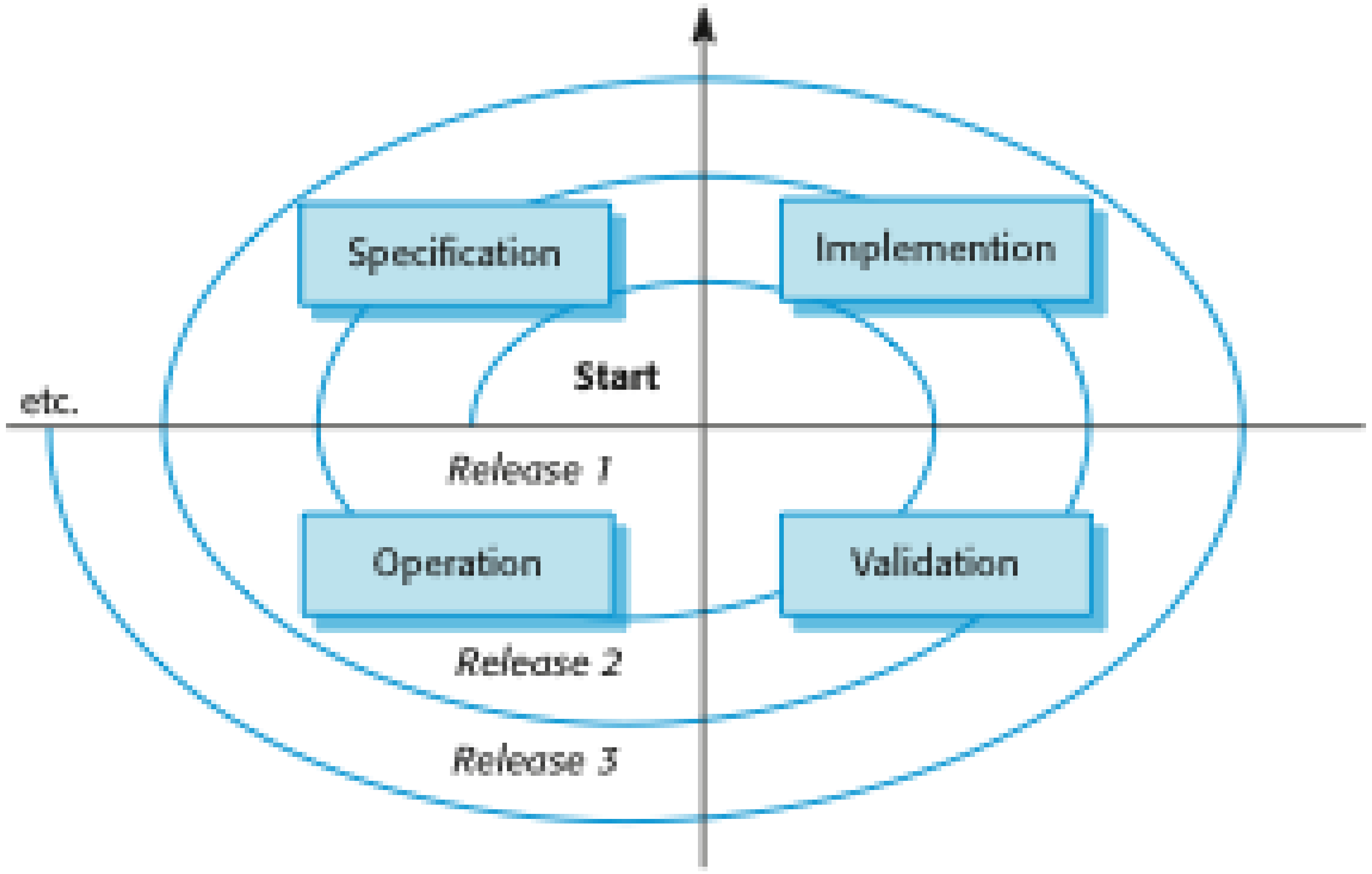
Importance of evolution

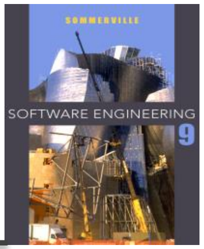


- Organisations have huge investments in their software systems - they are critical business assets.
- To maintain the value of these assets to the business, they must be changed and updated.
- The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software.

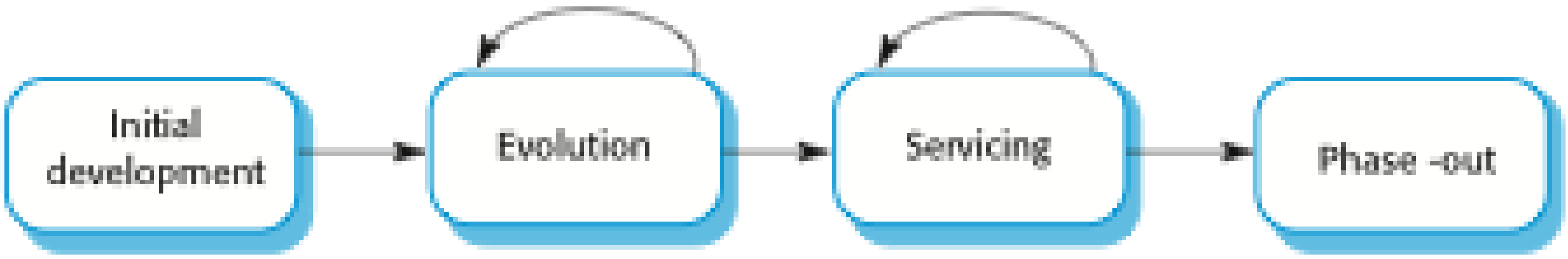
A spiral model of development and evolution



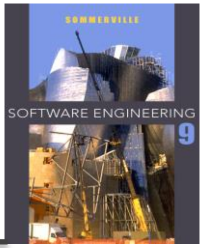




Evolution and servicing

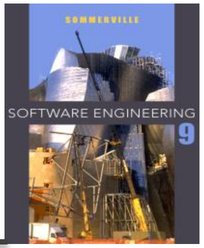


Evolution and servicing



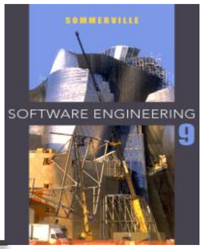
- Evolution
  - The stage in a software system's life cycle where it is in operational use and is evolving as new requirements are proposed and implemented in the system.
- Servicing
  - At this stage, the software remains useful but the only changes made are those required to keep it operational i.e. bug fixes and changes to reflect changes in the software's environment. No new functionality is added.
- Phase-out
  - The software may still be used but no further changes are made to it.

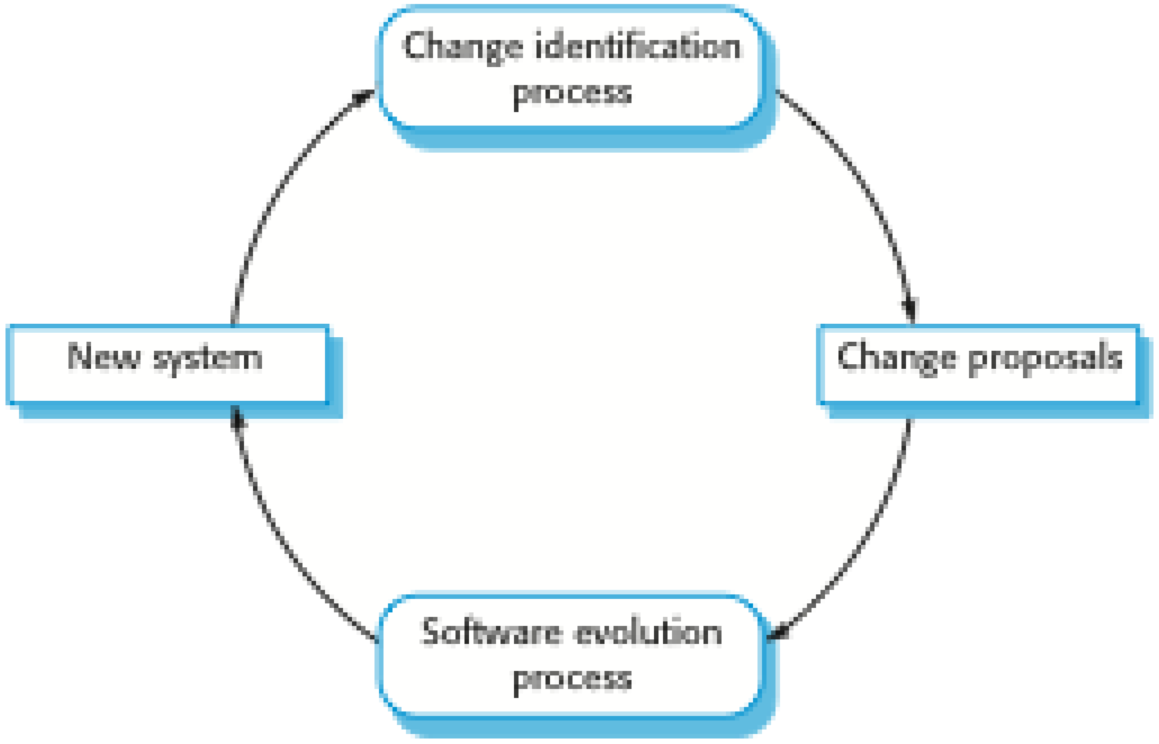
Evolution processes



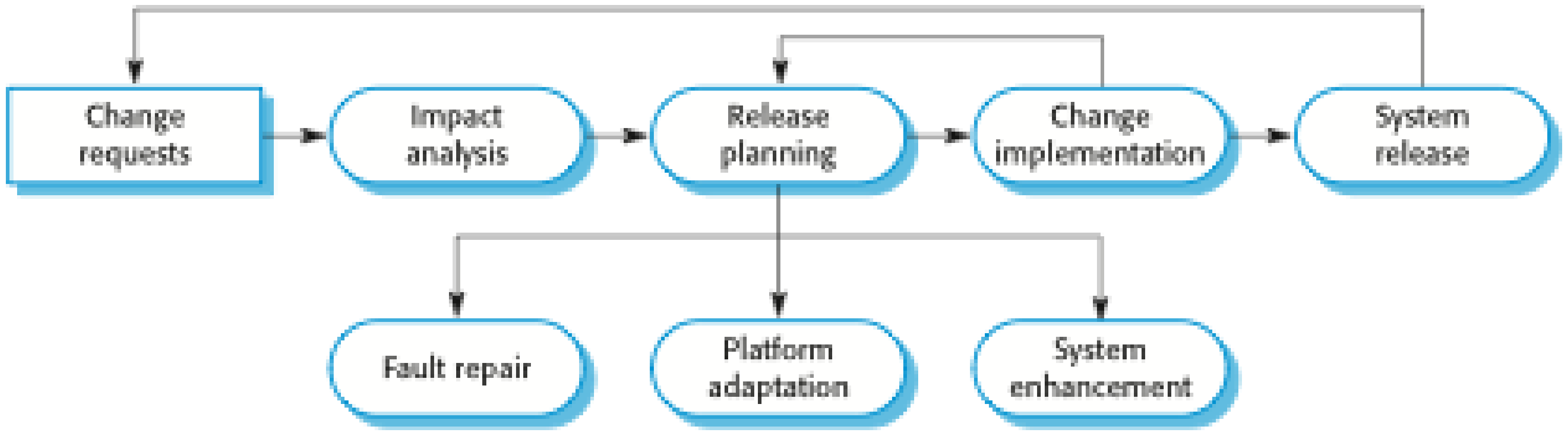
- ✧ Software evolution processes depend on
  - The type of software being maintained;
  - The development processes used;
  - The skills and experience of the people involved.
- ✧ Proposals for change are the driver for system evolution.
  - Should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated.
- ✧ Change identification and evolution continues throughout the system lifetime.

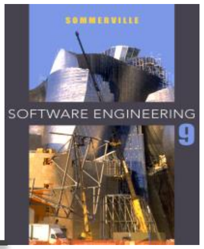
Change identification and evolution processes



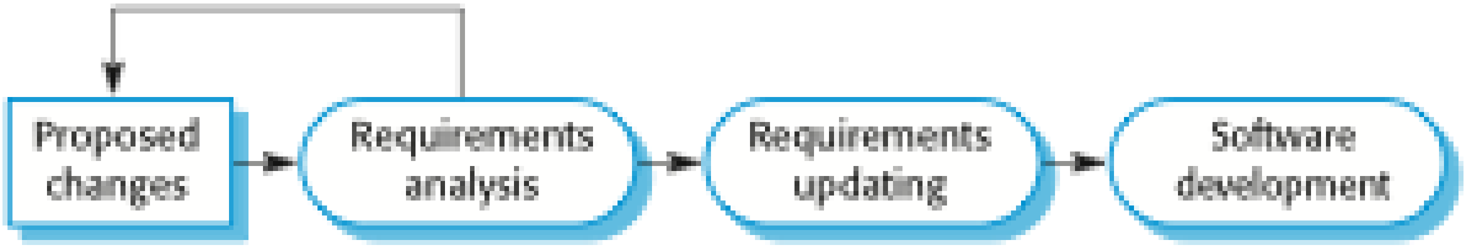


The software evolution process

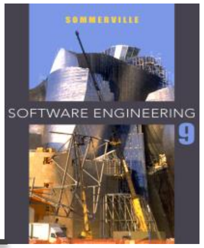




Change implementation



Change implementation

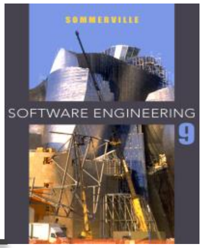


- Iteration of the development process where the revisions to the system are designed, implemented and tested.

- A critical difference is that the first stage of change implementation may involve program understanding, especially if the original system developers are not responsible for the change implementation.

- During the program understanding phase, you have to understand how the program is structured, how it delivers functionality and how the proposed change might affect the program.

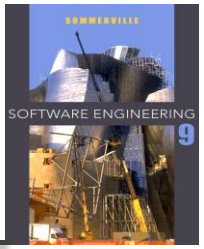
Urgent change requests

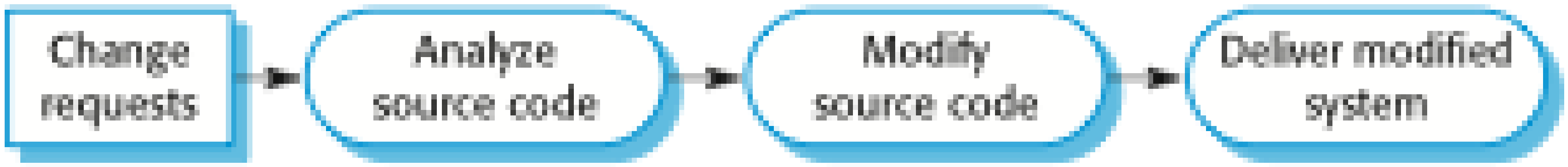


✧ Urgent changes may have to be implemented without going through all stages of the software engineering process

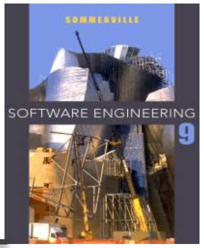
- If a serious system fault has to be repaired to allow normal operation to continue;
- If changes to the system's environment (e.g. an OS upgrade) have unexpected effects;
- If there are business changes that require a very rapid response (e.g. the release of a competing product).

The emergency repair process



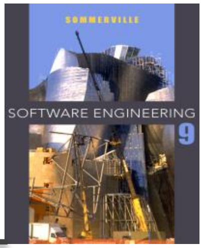


Agile methods and evolution



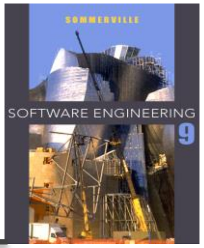
- ✧ Agile methods are based on incremental development so the transition from development to evolution is a seamless one.
  - Evolution is simply a continuation of the development process based on frequent system releases.
- ✧ Automated regression testing is particularly valuable when changes are made to a system.
- ✧ Changes may be expressed as additional user stories.

Handover problems



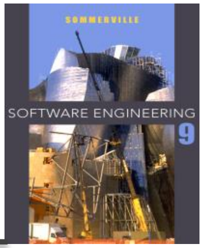
- ✧ Where the development team have used an agile approach but the evolution team is unfamiliar with agile methods and prefer a plan-based approach.
  - The evolution team may expect detailed documentation to support evolution and this is not produced in agile processes.
- ✧ Where a plan-based approach has been used for development but the evolution team prefer to use agile methods.
  - The evolution team may have to start from scratch developing automated tests and the code in the system may not have been refactored and simplified as is expected in agile development.

Program evolution dynamics



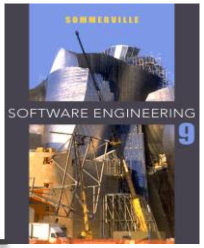
- ✧ *Program evolution dynamics* is the study of the processes of system change.
- ✧ After several major empirical studies, Lehman and Belady proposed that there were a number of ‘laws’ which applied to all systems as they evolved.
- ✧ There are sensible observations rather than laws. They are applicable to large systems developed by large organisations.
  - It is not clear if these are applicable to other types of software system.

Change is inevitable



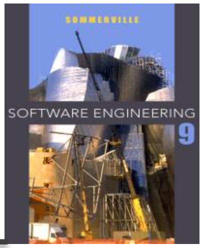
- ✧ The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- ✧ Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- ✧ Systems MUST be changed if they are to remain useful in an environment.

Lehman’s laws

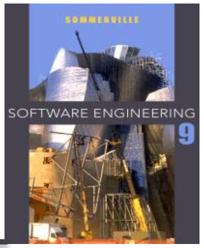


Law	Description
Continuing change	A program that is used in a real-world environment must necessarily change, or else become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases, and the number of reported errors is approximately invariant for each system release.
Organizational stability	Over a program’s lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.

Lehman’s laws



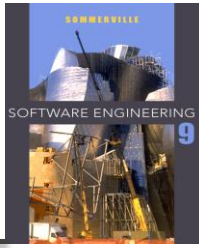
Law	Description
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will decline unless they are modified to reflect changes in their operational environment.



Chapter 9 – Software Evolution

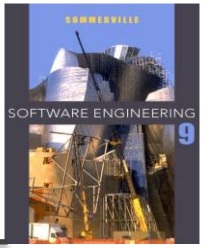
Lecture 2

Software maintenance



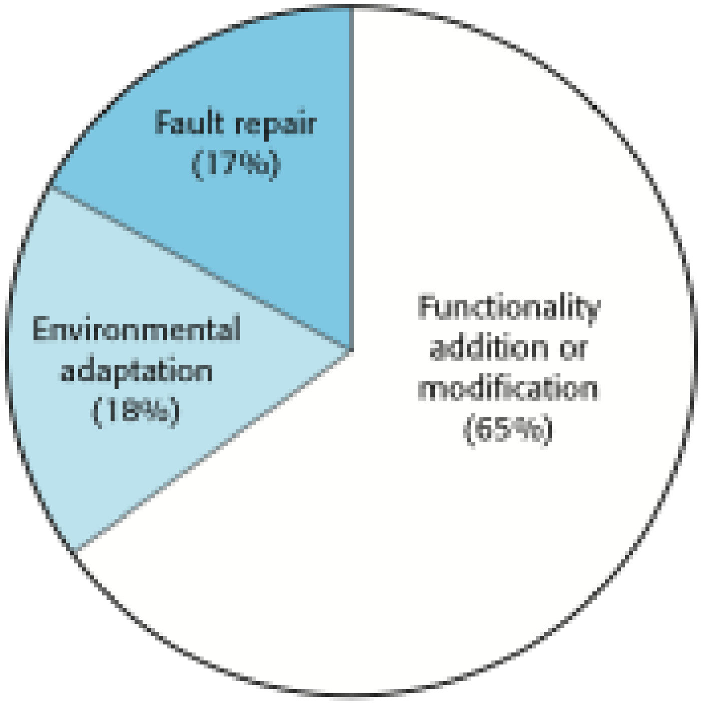
- ✧ Modifying a program after it has been put into use.
- ✧ The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions.
- ✧ Maintenance does not normally involve major changes to the system's architecture.
- ✧ Changes are implemented by modifying existing components and adding new components to the system.

Types of maintenance

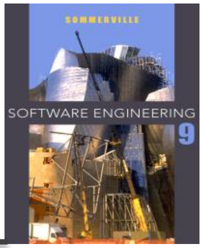


- ✧ Maintenance to repair software faults
  - Changing a system to correct deficiencies in the way meets its requirements.
- ✧ Maintenance to adapt software to a different operating environment
  - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- ✧ Maintenance to add to or modify the system’s functionality
  - Modifying the system to satisfy new requirements.

Figure 9.8 Maintenance effort distribution

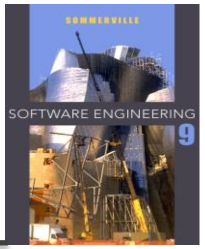


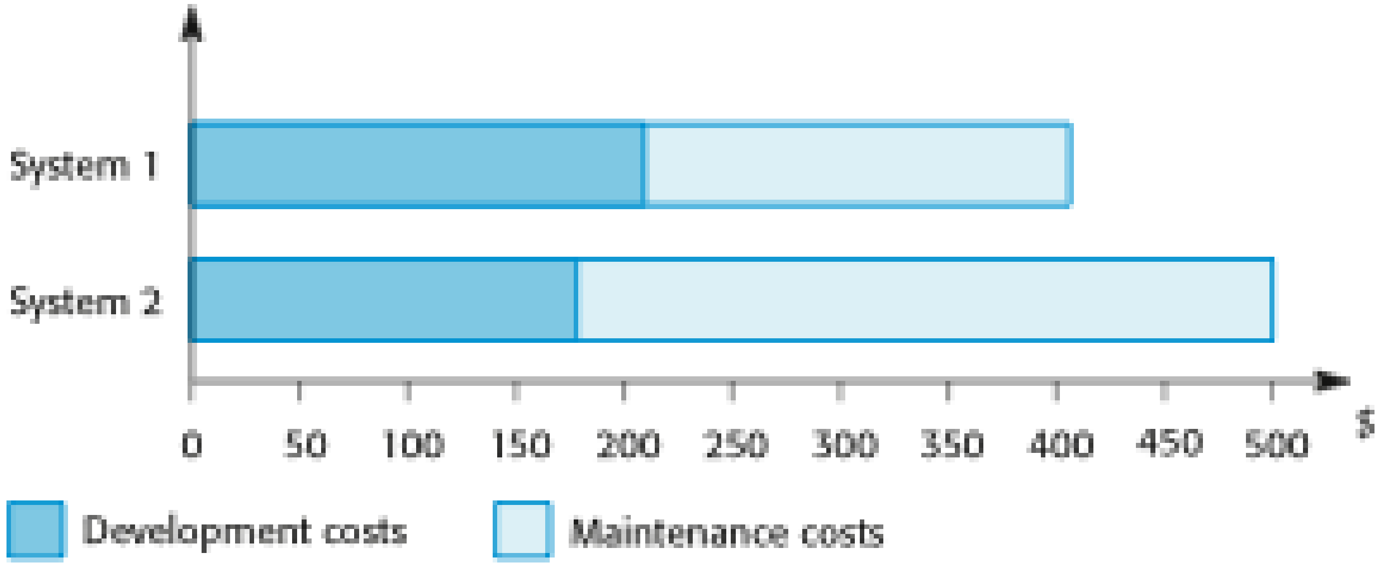
Maintenance costs



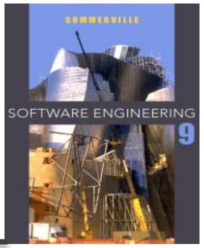
- ✧ Usually greater than development costs (2* to 100* depending on the application).
- ✧ Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult.
- ✧ Ageing software can have high support costs (e.g. old languages, compilers etc.).

Figure 9.9 Development and maintenance costs



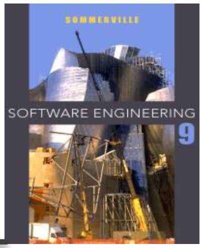


Maintenance cost factors



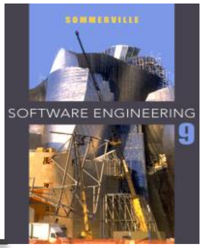
- ✧ Team stability
  - ▪ Maintenance costs are reduced if the same staff are involved with them for some time.
- ✧ Contractual responsibility
  - ▪ The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- ✧ Staff skills
  - ▪ Maintenance staff are often inexperienced and have limited domain knowledge.
- ✧ Program age and structure
  - ▪ As programs age, their structure is degraded and they become harder to understand and change.

Maintenance prediction

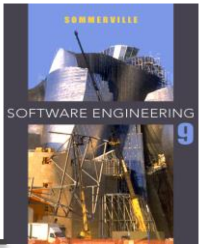


- ✧ Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
  - Change acceptance depends on the maintainability of the components affected by the change;
  - Implementing changes degrades the system and reduces its maintainability;
  - Maintenance costs depend on the number of changes and costs of change depend on maintainability.

Change prediction



- ✧ Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- ✧ Tightly coupled systems require changes whenever the environment is changed.

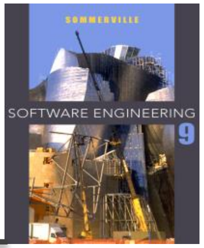


Complexity metrics

✧ Complexity depends on

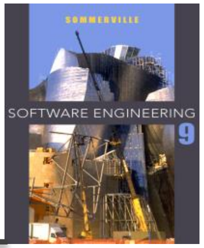
- Complexity of control structures;
- Complexity of data structures;
- Object, method (procedure) and module size.

System re-engineering



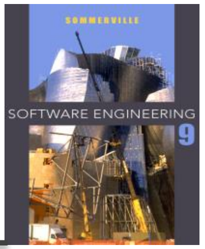
- Re-structuring or re-writing part or all of a system without changing its functionality.
- Applicable where some but not all sub-systems of a larger system require frequent maintenance.
- Re-engineering involves adding effort to make them easier to maintain. The system may be re-structured and re-documented.

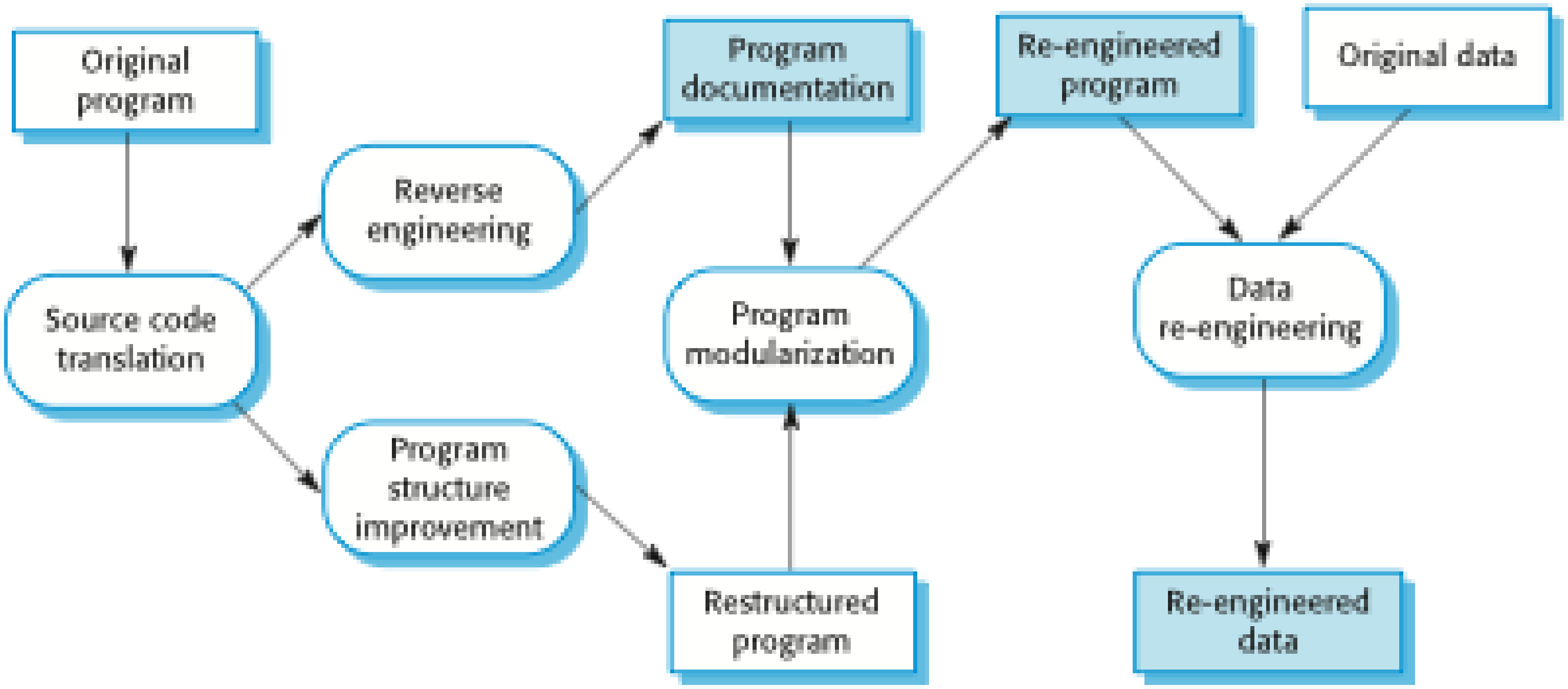
Advantages of reengineering



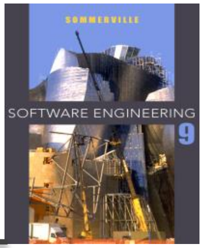
- ✧ Reduced risk
  - There is a high risk in new software development. There may be development problems, staffing problems and specification problems.
- ✧ Reduced cost
  - The cost of re-engineering is often significantly less than the costs of developing new software.

The reengineering process



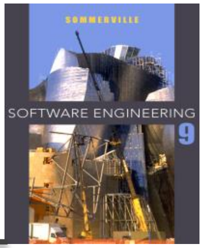


Reengineering process activities



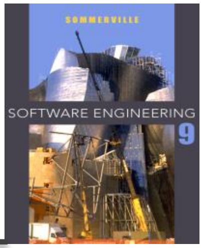
- ✧ Source code translation
  - Convert code to a new language.
- ✧ Reverse engineering
  - Analyse the program to understand it;
- ✧ Program structure improvement
  - Restructure automatically for understandability;
- ✧ Program modularisation
  - Reorganise the program structure;
- ✧ Data reengineering
  - Clean-up and restructure system data.

Reengineering cost factors



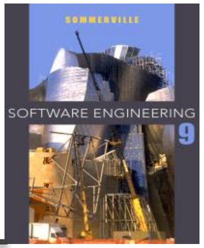
- ✧ The quality of the software to be reengineered.
- ✧ The tool support available for reengineering.
- ✧ The extent of the data conversion which is required.
- ✧ The availability of expert staff for reengineering.
  - ▪ This can be a problem with old systems based on technology that is no longer widely used.

Preventative maintenance by refactoring



- Refactoring is the process of making improvements to a program to slow down degradation through change.
- You can think of refactoring as ‘preventative maintenance’ that reduces the problems of future change.
- Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand.
- When you refactor a program, you should not add functionality but rather concentrate on program improvement.

‘Bad smells’ in program code

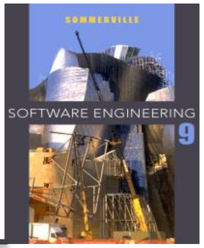


✧ Duplicate code
- The same or very similar code may be included at different places in a program. This can be removed and implemented as a single method or function that is called as required.

✧ Long methods
- If a method is too long, it should be redesigned as a number of shorter methods.

✧ Switch (case) statements
- These often involve duplication, where the switch depends on the type of a value. The switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.

‘Bad smells’ in program code



- Data clumping
  - Data clumps occur when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- Speculative generality
  - This occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.