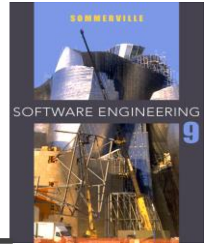
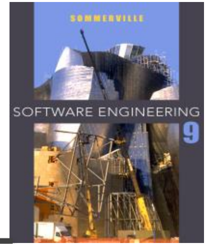


Chapter 9 – Software Evolution

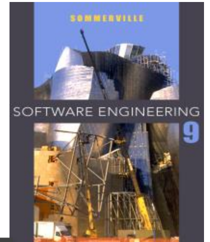
Lecture 1

Topics covered



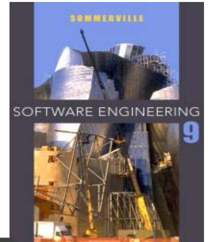
- Evolution processes
  - Change processes for software systems
- Program evolution dynamics
  - Understanding software evolution
- Software maintenance
  - Making changes to operational software systems
- Legacy system management
  - Making decisions about software change

Software change

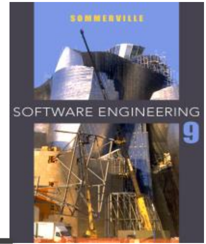


- Software **change is inevitable**
  - **New requirements** emerge when the software is used;
  - The business **environment changes** (**changes to other systems** in a software system's environment);
  - **Errors** must be repaired;
  - **New computers and equipment** is added to the system;
  - The **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

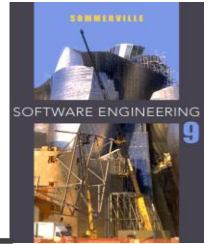


- Organizations have **huge investments** in their software systems - they are critical business assets.
- **To maintain the value** of these assets to the business, they must be changed and updated.
- The majority of the **software budget** in large companies is **devoted to changing and evolving existing software** rather than developing new software.
- Based on an informal industry poll, Erlikh (2000) suggests that **85–90%** of organizational software costs are evolution costs.
- Other surveys suggest that about **two-thirds** of software costs are evolution costs.



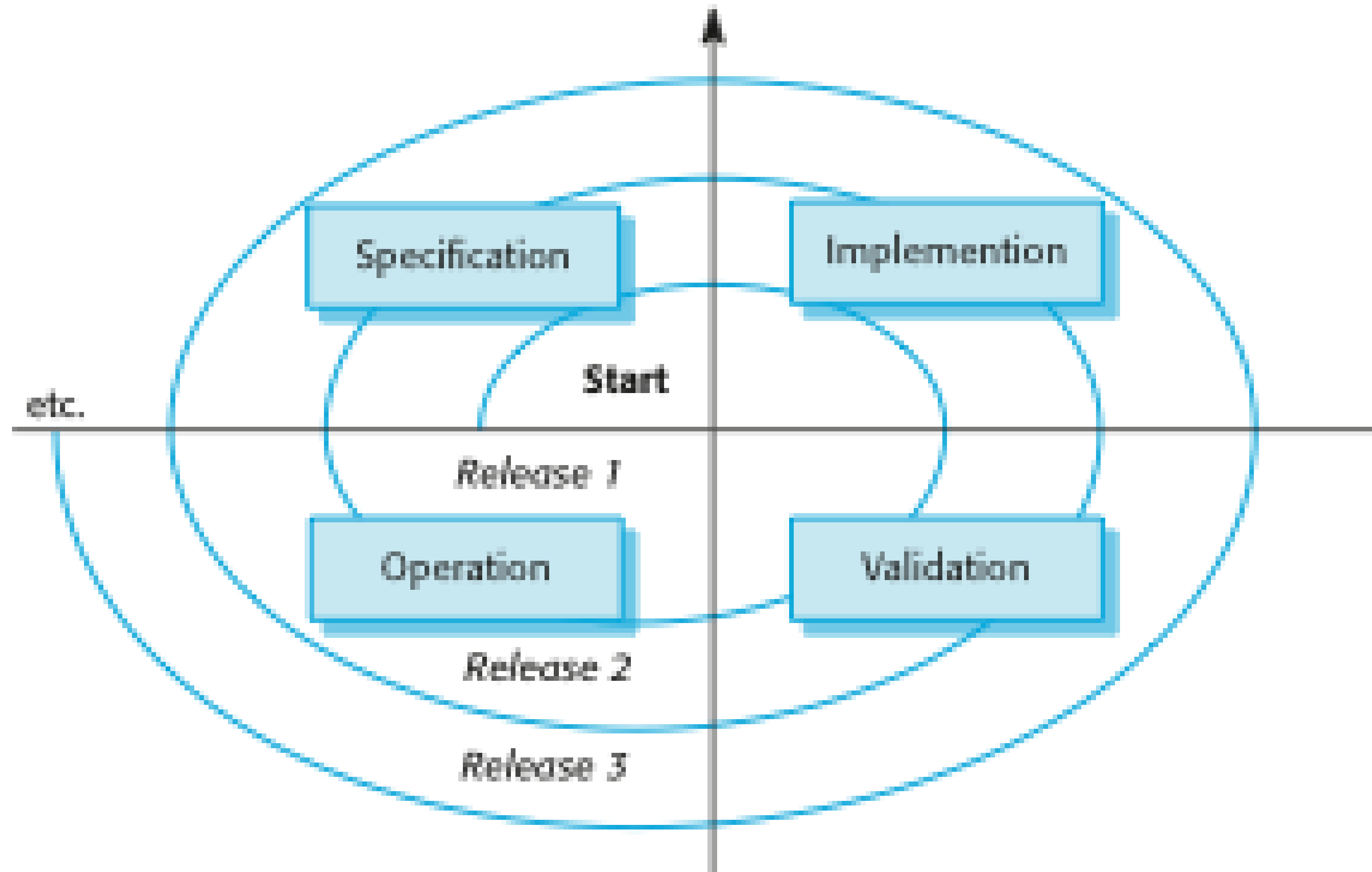
- Obviously, the requirements of the installed systems change as the business and its environment change.
- Therefore, **new releases** of the systems, incorporating changes, and updates, are usually created **at regular intervals.**
- You should, therefore, **think of software engineering as a spiral process** with requirements, design, implementation, and testing going on throughout the lifetime of the system.
- **See Figure below.**

Software Engineering as a Spiral Process

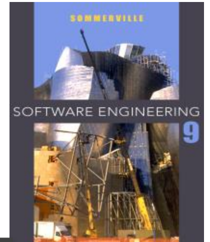


- You start by creating release 1 of the system.
- Once delivered, changes are proposed and the development of release 2 starts almost immediately.
- In fact, the **need for evolution** may become obvious even before the system is deployed so that later releases of the software maybe under development **before the current version has been released**.

A spiral model of development and evolution

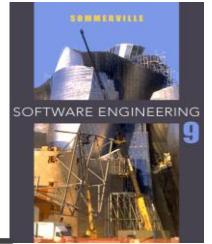


soft-ware maintenance



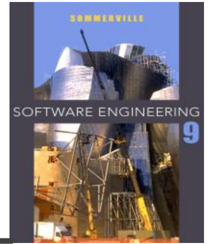
- A **different development team** or even organization might be in charge of the evolution.
- In this case, there are **likely to be discontinuities in the spiral process**.
- Requirements and design documents **may not be passed from one company to another**.
- When the transition from development to **evolution is not seamless**, the process of changing the software after delivery is often **called 'soft-ware maintenance**'.
- Maintenance **involves extra process activities**, such as program **understanding**, in addition to the normal activities of software development.

Evolution and Servicing



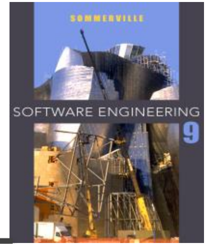
- Rajlich and Bennett (2000) proposed an alternative view of the software evolution life cycle.
- Figure below.
- In this model, they **distinguish between evolution and servicing**.
- **Evolution** is the phase in which **significant changes** to the software architecture and functionality may be made.
- **During servicing**, the **only changes that are made are relatively small**, **essential changes**.

A Transition Point



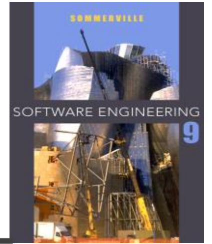
- As the software is modified, its **structure tends to degrade** and **changes become more and more expensive**.
- This often happens after a few years of use **when other environmental changes,** such as hardware and operating systems, are also often required.
- **At some stage** in the life cycle, **the software reaches a transition point** where significant changes, implementing new requirements, **become less and less cost effective**.

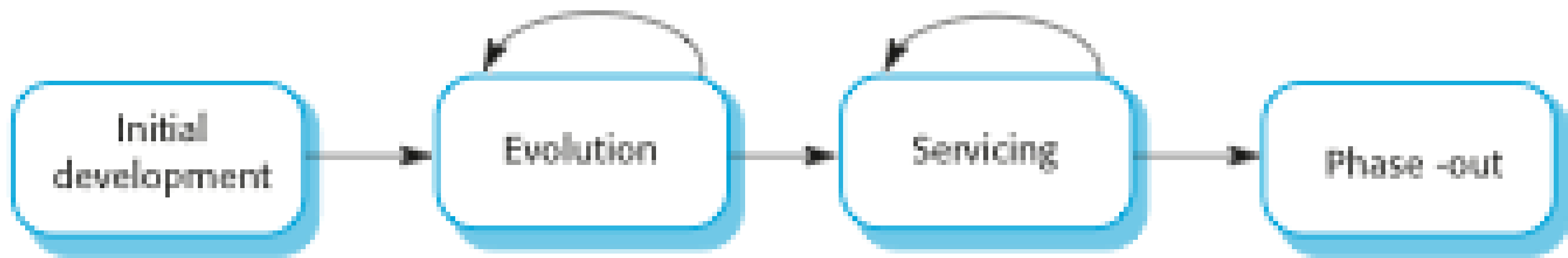
The Phase-Out Stage



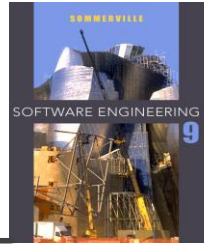
- In **the final stage**, phase-out, the software may still be **used but no further changes** are being implemented.
- **Users have to work around any problems** that they discover.

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.

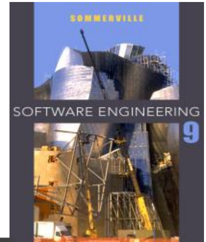
✧ Servicing

- 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.

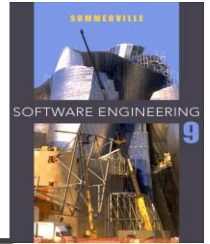
✧ **Proposals for change** are **the driver** for system evolution.

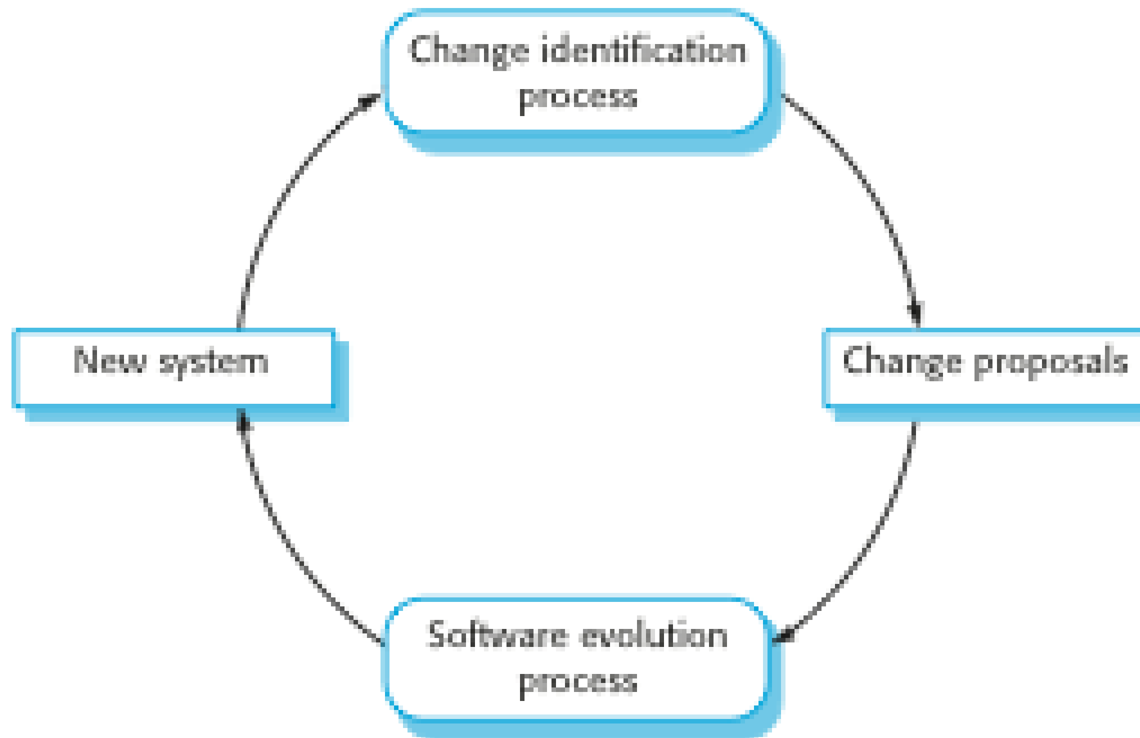
- Should be linked with **components that are aff**ected by the change, thus **allowing the cost and impact of the change to be estimated.**

✧ **Change identification and evolution** continues throughout the system lifetime.

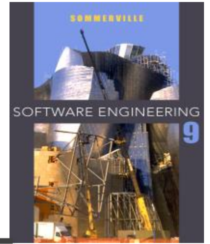
✧ The **processes of change** identification and system evolution are **cyclic and continue** throughout the lifetime of a system. **See the Figure below**

Change identification and evolution processes



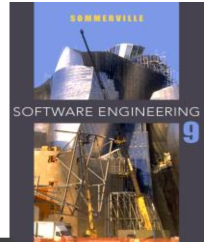


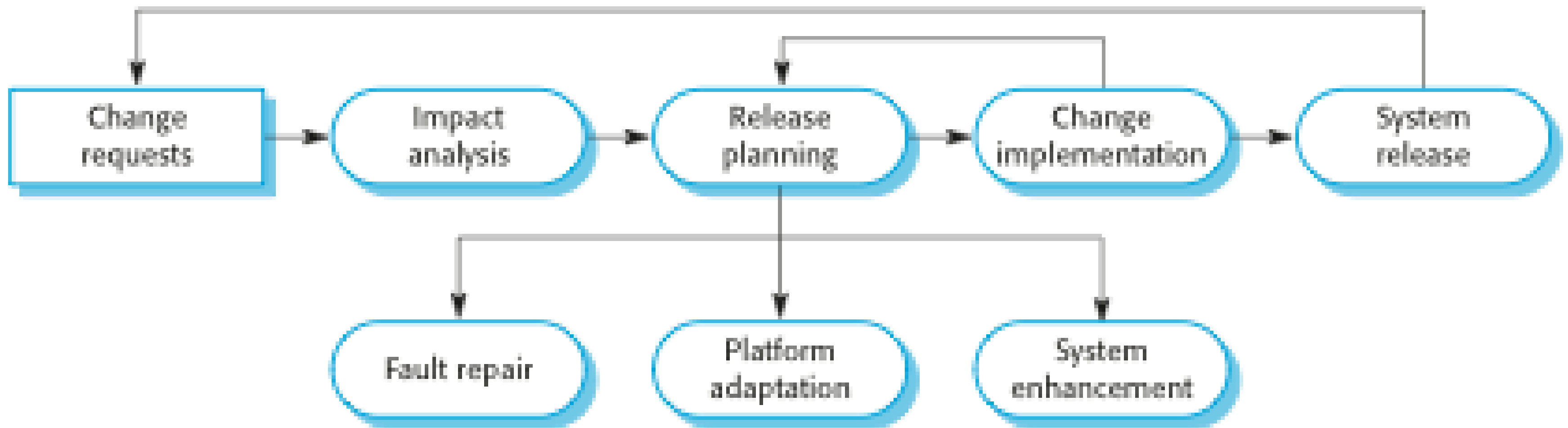
Change Management



- Change proposals **should be linked to the components** of the system that have to be modified to implement these proposals.
- This **allows the cost and the impact** of the change to be assessed.
- This is part of the general process of **change management**, which also **should ensure that the correct versions of components are included** in each system release.
- Figure below

The software evolution process

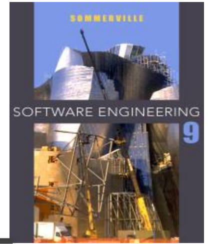


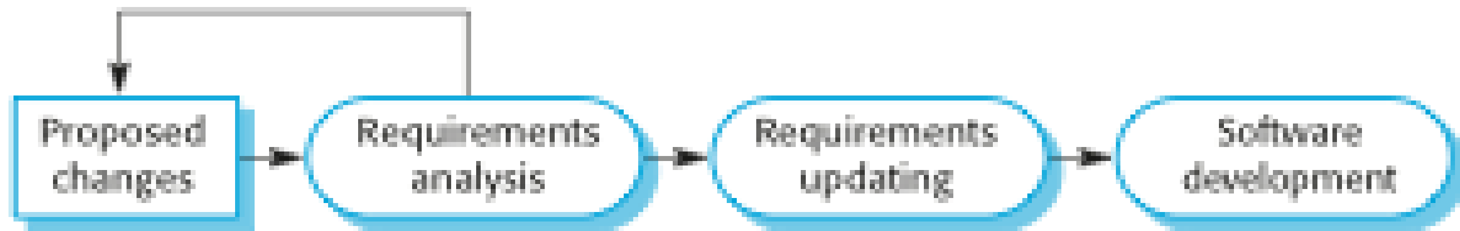


SOFTWARE ENGINEERING 9

- The **process includes** the fundamental activities of **change analysis, release planning, system implementation**, and releasing a system to customers.
- The **cost and impact of these changes are assessed** to see how much of the system is affected by the change and how much it might cost to implement the change.
- **If the proposed changes are accepted**, a new release of the system is planned.
- During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered.
- A decision is then made on **which changes to implement in the next version** of the system.
- The **changes are implemented and validated**, and a new version of the system is released.

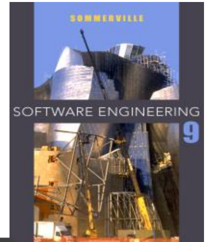
Change implementation





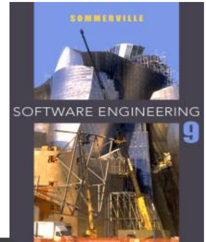
Ideally, the change implementation stage of this process **should modify the system specification, design, and implementation** to reflect the changes to the system

Change implementation



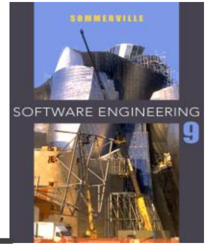
- ✧ **Iteration of the development process** where the revisions to the system are designed, implemented and tested.
- ✧ A critical difference is **that the first stage of change** implementation **may involve program understanding**, especially if the original system developers are not responsible for 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**.

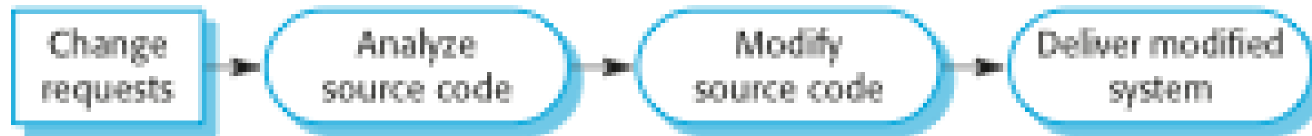
Urgent change requests requiring an emergency fix to the program



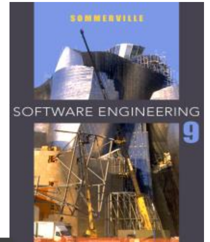
- ✧ Urgent changes **may have to be implemented without going through all stages** of the software engineering process
  - If a **serious system fault** has to be repaired to allow normal operation to continue;
  - If **changes to the system's environment** (e.g. an OS upgrade) have **unexpected effects**;
  - If there are **business changes that require a very rapid res**ponse (e.g. the release of a competing product).
- ✧ However, **the danger is** that the requirements, the software design, and the code **become inconsistent**
- ✧ This **accelerates the process of software ageing** so that **future changes** become progressively **more difficult and maintenance costs increase.**

The emergency repair process



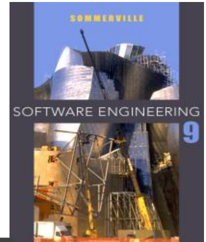


Agile methods and evolution



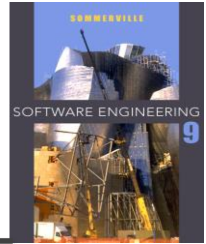
- ✧ Agile methods and processes **may be used for program evolution** as well as program development.
- ✧ Agile methods are **based on incremental development** so **the transition** from development to evolution **is a seamless** one.
  - Evolution is simply **a continuation of the development** process 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



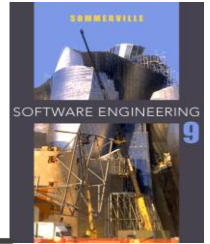
- Where the development team have used an agile approach but the **evolution team** is unfamiliar with agile methods and **prefer a plan-based approach**.
  - **The evolution team may expect detailed documentation** to support evolution and this is not produced in agile processes.
- Where a plan-based 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



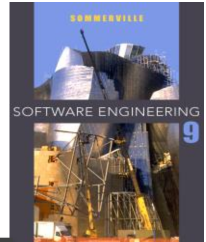
- *Program evolution dynamics* is the study of the processes of system change.
- After several major empirical studies, Lehman and Belady proposed that there were a **number of 'laws'** which applied to all systems as they evolved.
- There are **sensible observations rather than laws**.
- They are **applicable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

Change is inevitable



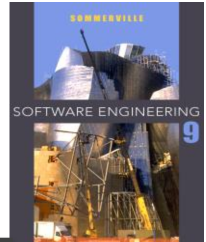
- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!
- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment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

Lehman’s laws



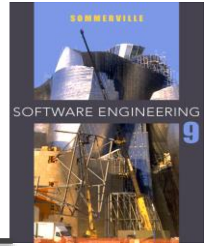
Law	Description
Continuing change **or change is inevitable**	A program that is used in a real-world environment must necessarily change, **or else become progressively less useful** in that environment.
Increasing complexity (as **program structures degrades**)	As an evolving program changes, its structure tends to become more complex. **Extra resources must be devoted** to preserving and simplifying the structure. You **spend time improving** the software structure **without adding to its functionality.**
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. This law confirms that large software development teams are often **unproductive because communication overheads** dominate the work of the team.

Lehman's laws



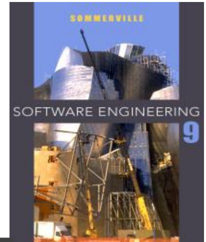
Law	Description
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant. **Adding new functionality to a system inevitably introduces new system faults**. The **more functionality added** in each release, **the more faults there will be**.
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.
Feedback system	Evolution processes incorporate multiagent, multiloop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

The Third Law: a consequence of structural factors and organizational factors



- The third law is, perhaps, the most interesting and the most contentious of Lehman's laws.
- This determines the gross trends of the system maintenance process and **limits the number of possible system changes**.
- The **structural factors** that affect the third law come from the complexity of large systems. As you change and extend a program, its structure tends to degrade.
- This degradation, if unchecked, makes it more and more difficult to make further changes to the pro-gram.
- Making **small changes reduces** the extent of structural degradation and so **lessens the risks of causing serious** system dependability problems.
- If you try and make **large changes**, there is a high probability that these **will introduce new faults**.

The organizational factors that affect the third law



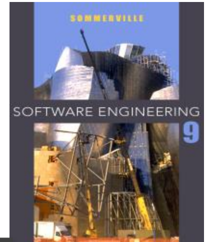
- ✧ Large systems are usually produced by large organizations.
- ✧ These companies have **internal bureaucracies** that set the **change budget** for each system and **control the decision-making process**.
- ✧ Companies have to **make decisions on the risks** and **value of the changes and the costs involved**.
- ✧ Such **decisions take time to make and**, sometimes, it takes **longer to decide** on the changes to be made than change implementation.
- ✧ The **speed of the organization's decision-making** processes therefore **governs the rate of change** of the system.

- Lehman’s observations seem generally sensible.
- **They should be taken into account** when planning the maintenance process.
- It may be that business considerations require them to be **ignored** at any one time.
- For example, **for marketing reasons**, it may be necessary to make several major system changes in a single release.
- The likely consequences of this are that **one or more releases devoted to error repair are likely to be required**.
- You often see this in **personal computer software** when **a major new release** of an application is often quickly **followed by a bug repair update.**

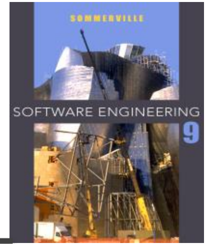
Applicability of Lehman's laws

- Lehman's laws seem to be generally applicable to large, tailored systems developed by large organisations.
  - Confirmed in early 2000's by work by Lehman on the FEAST project.
- It is not clear how they should be modified for
  - Shrink-wrapped software products;
  - Systems that incorporate a significant number of COTS components;
  - Small organisations;
  - Medium sized systems.

Key points



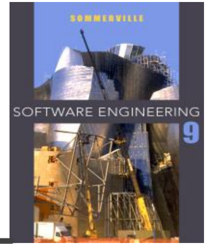
- Software development and evolution can be thought of as an integrated, iterative process that can be represented using a spiral model.
- For custom systems, the costs of software maintenance usually exceed the software development costs.
- The process of software evolution is driven by requests for changes and includes change impact analysis, release planning and change implementation.
- Lehman's laws, such as the notion that change is continuous, describe a number of insights derived from long-term studies of system evolution.



Chapter 9 – Software Evolution

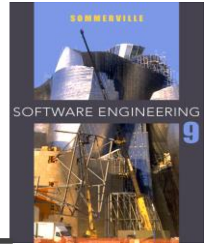
Lecture 2

Software maintenance



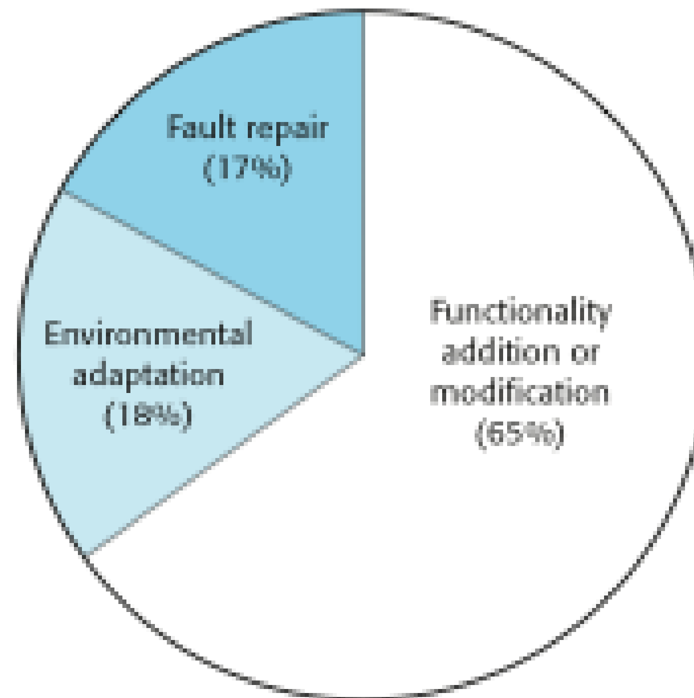
- ✧ Modifying a program **after it has been put into use**.
- ✧ The term is mostly used for **changing custom software**. **Generic software products are said to evolve** to create new versions.
- ✧ Maintenance **does not normally involve major changes** to the system's architecture.
- ✧ Changes are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

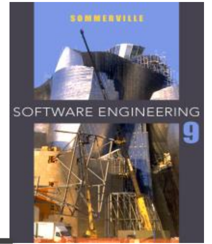


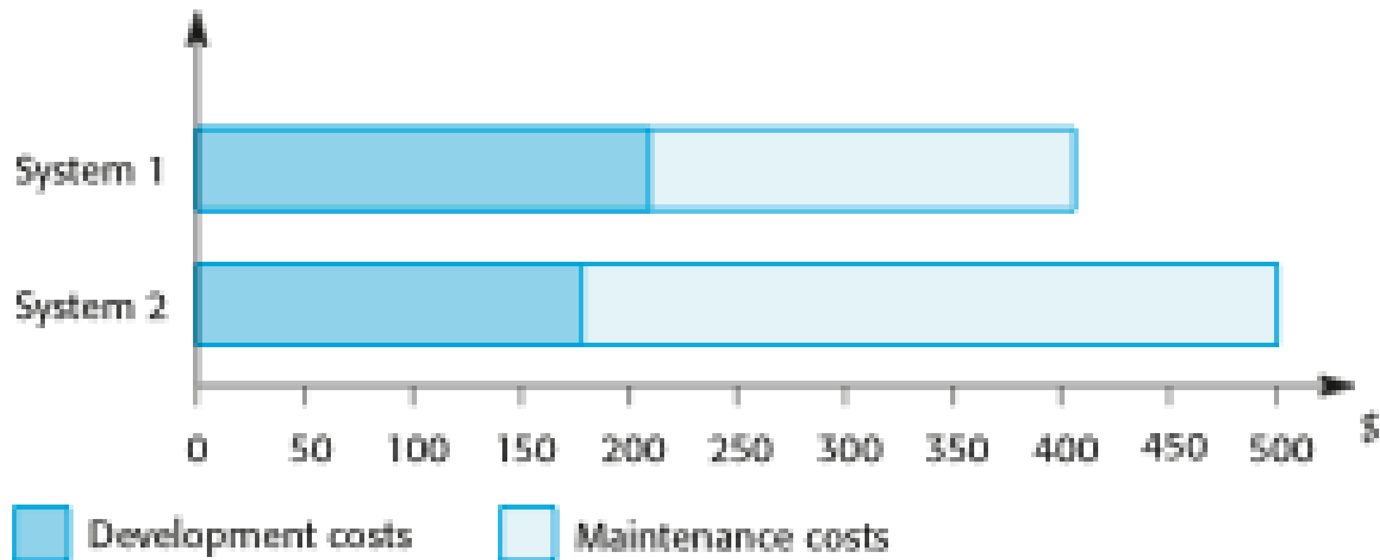
Maintenance costs

- ✧ Usually **greater than development costs** (2* to 100* depending on the application).
- ✧ Affected by both **technical and non-technical factors**.
- ✧ **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.).

- The following Figure shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system.
- Because of the potential reduction in costs of understanding, analysis, and testing, there is a significant multiplier effect when the system is developed for maintainability.
- For System 1, extra development costs of $25,000 are invested in making the system more maintainable.
- This results in a savings of $100,000 in maintenance costs over the lifetime of the system.

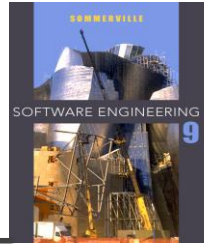
Figure 9.9 Development and maintenance costs





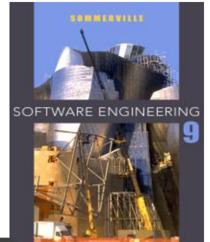
- I t is usually **more expensive to add functionality** after a system is in operation than it is to implement the same functionality during development.
- The reasons for this are:
- Team stability
  - Maintenance costs are reduced **if the same staff are involved** with them for some time.
- Contractual responsibility
  - The developers of a system may have no contractual responsibility for maintenance so there is **no incentive to design for future change**.

Maintenance cost factors



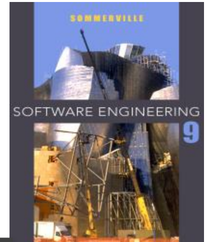
- Staff skills
  - Maintenance **staff are often inexperienced** and have limited domain knowledge.
  - Maintenance has a poor image among software engineers.
  - It is seen as a less-skilled process than system development and is often allocated to the most junior staff.
  - Furthermore, old systems maybe written in obsolete programming languages.
  - The maintenance staff may not have much experience of development in these languages and must learn these languages to maintain the system.
- Program age and structure
  - As programs age, their structure is degraded and they become harder to understand and change.

Maintenance prediction



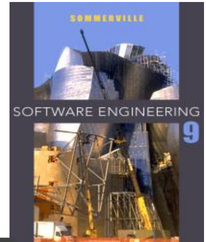
- Managers hate surprises, especially if these result in unexpectedly high costs.
- You should therefore try to predict what system changes might be proposed and what parts of the system are likely to be the most difficult to maintain.
- You should also try to estimate the overall maintenance costs for a system in a given time period.
- The following Figure shows these predictions and associated questions.

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.

Maintenance prediction



What parts of the system are most likely to be affected by change requests?

What parts of the system will be the most expensive to maintain?

Predicting maintainability

Predicting system changes

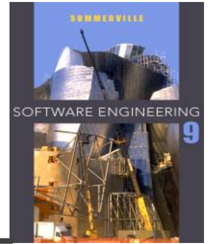
Predicting maintenance costs

What will be the lifetime maintenance costs of this system?

How many change requests can be expected?

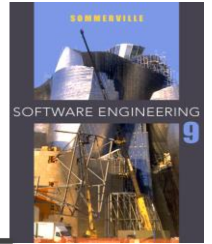
What will be the costs of maintaining this system over the next year?

Change prediction



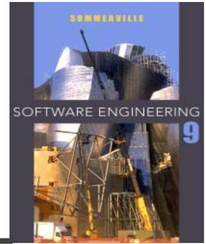
- ✧ Predicting the number of changes requires and understanding of the relationships between a system and its environment.
- ✧ Tightly coupled systems require changes whenever the environment is changed.
- ✧ Factors influencing this relationship are
  - Number and complexity of system interfaces;
  - Number of inherently volatile system requirements;
  - The business processes where the system is used.

Complexity metrics



- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
  - Complexity of control structures;
  - Complexity of data structures;
  - Object, method (procedure) and module size.

Process metrics

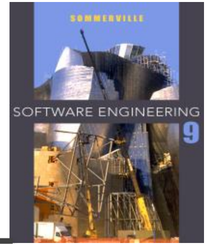


- ✧ To make legacy software systems easier to maintain, you can reengineer these systems to improve their structure and understandability.
- ✧ Process metrics may be used to assess maintainability
  - Number of requests for corrective maintenance;
  - Average time required for impact analysis;
  - Average time taken to implement a change request;
  - Number of outstanding change requests.
- ✧ If any or all of these is increasing, this may indicate a decline in maintainability.

✧ Reengineering may involve

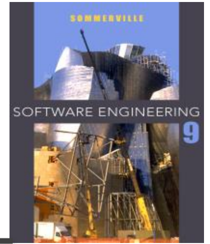
- Re-documenting the system,
- refactoring the system architecture,
- translating programs to a modern programming language,
- and modifying and
- updating the structure and values of the system’s data.

System re-engineering



- ✧ Re-structuring or re-writing part or all of a legacy 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



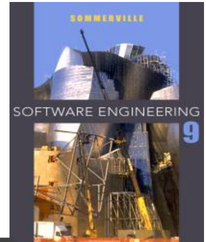
✧ Reduced risk

- 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



Original program

Source code translation

Reverse engineering

Program structure improvement

Program documentation

Program modularization

Restructured program

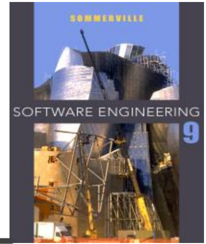
Re-engineered program

Original data

Data re-engineering

Re-engineered data

Reengineering process activities



✧ Source code translation
  - Convert code to a new language.

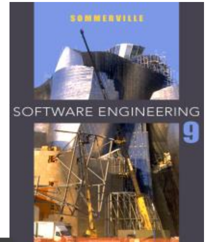
✧ Reverse engineering
  - Analyse the program to understand it;

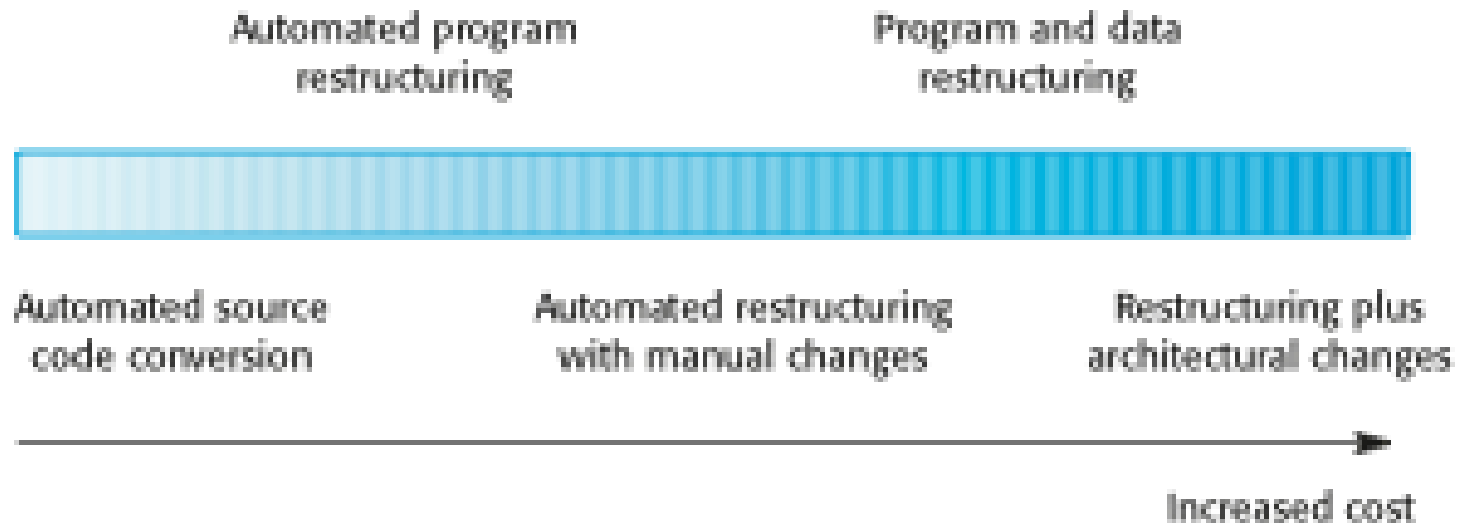
✧ Program structure improvement
  - Restructure automatically for understandability;

✧ Program modularisation
  - Reorganise the program structure;

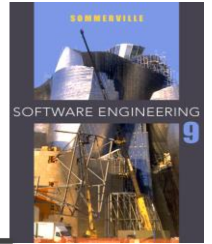
✧ Data reengineering
  - Clean-up and restructure system data.

Figure 9.12 Reengineering approaches



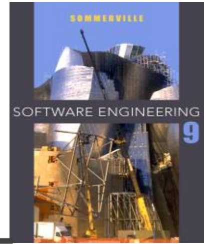


Reengineering cost factors



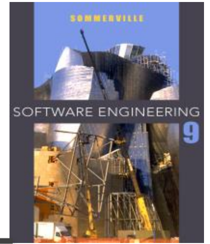
- The **quality of the software** to be reengineered.
- The **tool support** available for reengineering.
- The **extent of the data conversion** which is required.
- The **availability of expert staff** for reengineering.
  - This can be **a problem with old systems** based on technology that is no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

Preventative maintenance by refactoring



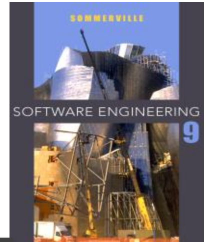
- ✧ Refactoring is the process of **making improvements to a program to slow down degradation** through change.
- ✧ You can think of refactoring as **'preventative maintenance'** that reduces the problems of future change.
- ✧ Refactoring involves
  - ✧ modifying a program to **improve its structure**,
  - ✧ **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.**

Refactoring and reengineering



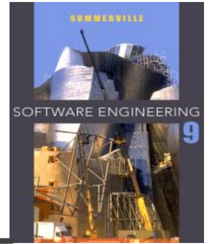
- Re-engineering **takes place after a system has been maintained** for some time **and maintenance costs are increasing**.
  - You **use automated tools** to process and r**e-engineer a legacy system** to create a new system that is more maintainable.
- Refactoring is a continuous process of improvement throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

‘Bad smells’ in program code



- Examples of bad smells that can be improved through refactoring include:
- 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



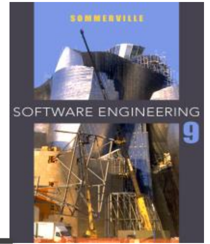
✧ Data clumping

- 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.

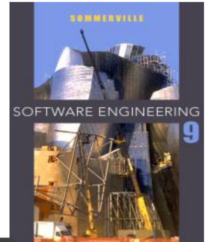
Legacy system management

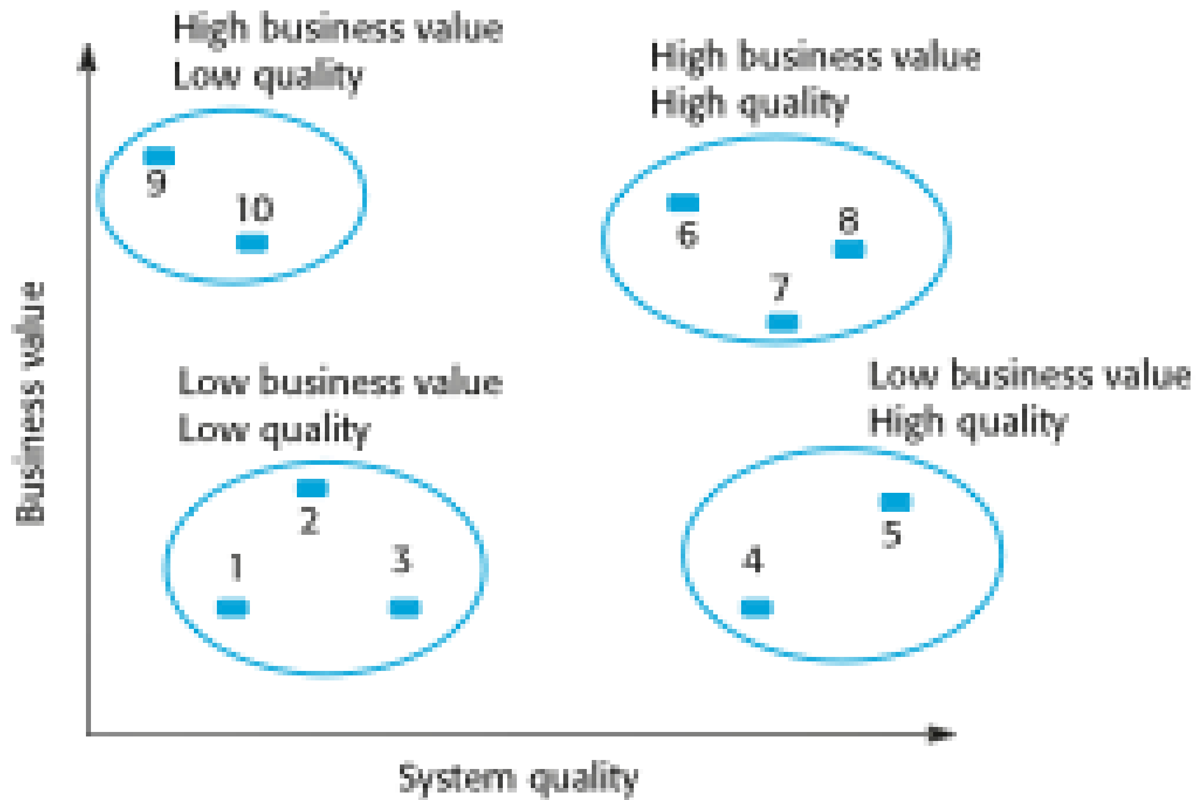


- Organisations that rely on legacy systems must choose a strategy for evolving these systems
  - Scrap the system completely and modify business processes so that it is no longer required;
  - Continue maintaining the system;
  - Transform the system by re-engineering to improve its maintainability;
  - Replace the system with a new system.
- The strategy chosen should depend on the system quality and its business value.

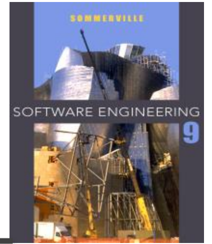
- For example, assume that an organization has 10 legacy systems.
- You should assess the quality and the business value of each of these systems.
- You may then **create a chart showing relative business value and system quality.**
- This is shown in Figure 9.13.
- From Figure 9.13, you can see that there **are four clusters of systems**:

Figure 9.13 An example of a legacy system assessment



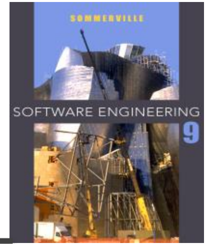


Legacy system categories



- ✧ Low quality, low business value
  - These systems should be scrapped.
- ✧ Low-quality, high-business value
  - These make an important business contribution but are expensive to maintain. Should be re-engineered or replaced if a suitable system is available.
- ✧ High-quality, low-business value
  - Replace with COTS, scrap completely or maintain.
- ✧ High-quality, high business value
  - Continue in operation using normal system maintenance.

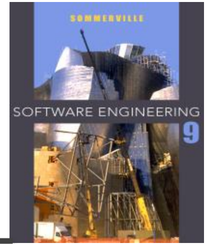
Business value assessment



✧ To assess the business value of a system,
- you have to identify system stakeholders, such as
- end-users of the system and their managers, and
- ask a series of questions about the system.

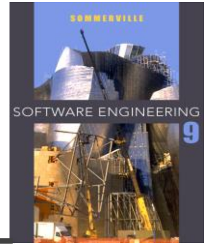
✧ There are four basic issues that you have to discuss:

Business value assessment



- Assessment should take different viewpoints into account
  - System end-users;
  - Business customers;
  - Line managers;
  - IT managers;
  - Senior managers.
- Interview different stakeholders and collate results.

Issues in business value assessment



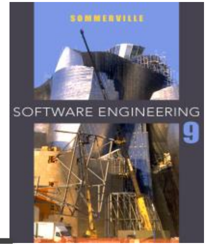
✧ The use of the system
- If systems are only used occasionally or by a small number of people, they may have a low business value.

✧ The business processes that are supported
- A system may have a low business value if it forces the use of inefficient business processes.

✧ System dependability
- If a system is not dependable and the problems directly affect business customers, the system has a low business value.

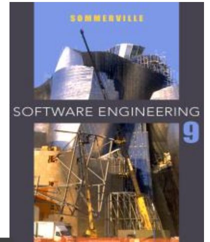
✧ The system outputs
- If the business depends on system outputs, then the system has a high business value.

System quality assessment



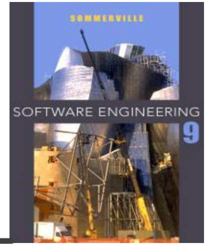
- Business process assessment
  - How well does the business process support the current goals of the business?
- Environment assessment
  - How effective is the system's environment and how expensive is it to maintain?
- Application assessment
  - What is the quality of the application software system?

Business process assessment



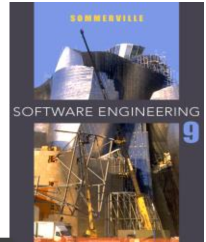
- ✧ Use a viewpoint-oriented approach and seek answers from system stakeholders
  - Is there a defined process model and is it followed?
  - Do different parts of the organisation use different processes for the same function?
  - How has the process been adapted?
  - What are the relationships with other business processes and are these necessary?
  - Is the process effectively supported by the legacy application software?
- ✧ Example - a travel ordering system may have a low business value because of the widespread use of web-based ordering.

Factors used in environment assessment



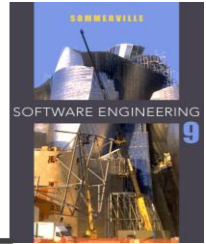
Factor	Questions
Supplier stability	Is the supplier still in existence? Is the supplier financially stable and likely to continue in existence? If the supplier is no longer in business, does someone else maintain the systems?
Failure rate	Does the hardware have a high rate of reported failures? Does the support software crash and force system restarts?
Age	How old is the hardware and software? The older the hardware and support software, the more obsolete it will be. It may still function correctly but there could be significant economic and business benefits to moving to a more modern system.
Performance	Is the performance of the system adequate? Do performance problems have a significant effect on system users?

Factors used in environment assessment



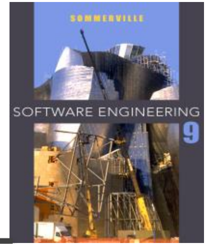
Factor	Questions
Support requirements	What local support is required by the hardware and software? If there are high costs associated with this support, it may be worth considering system replacement.
Maintenance costs	What are the costs of hardware maintenance and support software licences? Older hardware may have higher maintenance costs than modern systems. Support software may have high annual licensing costs.
Interoperability	Are there problems interfacing the system to other systems? Can compilers, for example, be used with current versions of the operating system? Is hardware emulation required?

Factors used in application assessment



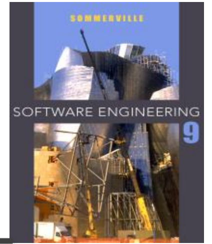
Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up to date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?

Factors used in application assessment



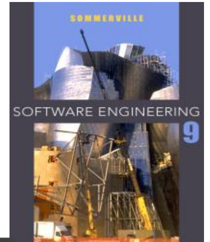
Factor	Questions
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

System measurement



- ✧ You may collect quantitative data to make an assessment of the quality of the application system
  - The number of system change requests;
  - The number of different user interfaces used by the system;
  - The volume of data used by the system.

Key points



- ✧ There are 3 types of software maintenance, namely bug fixing, modifying software to work in a new environment, and implementing new or changed requirements.
- ✧ Software re-engineering is concerned with re-structuring and re-documenting software to make it easier to understand and change.
- ✧ Refactoring, making program changes that preserve functionality, is a form of preventative maintenance.
- ✧ The business value of a legacy system and the quality of the application should be assessed to help decide if a system should be replaced, transformed or maintained.