MODULE 3

CHAPTER 2: SOFTWARE EVOLUTION

Once a system has been deployed, it inevitably has to change if it is to remain useful. There are many reasons why software change is inevitable

- As business changes and changes to user expectations generate new requirements for the existing software
- Parts of the software may have to be modified to correct errors that are found in operation,
- To adapt changes to its hardware and software platform
- To improve its performance or other non-functional characteristics

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

- 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.

A spiral model of development and evolution represents how a software system evolves through a sequence of multiple releases. (Figure 1)

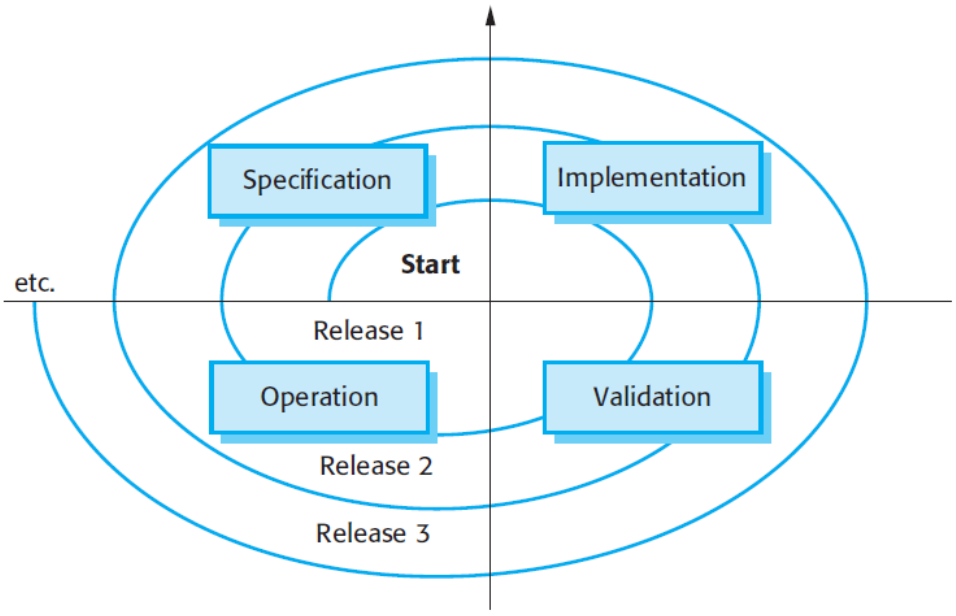

Figure 1: A spiral model of development and evolution

Figure 2 shows the alternative view of the software evolution life cycle. In this model, they distinguish between 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.

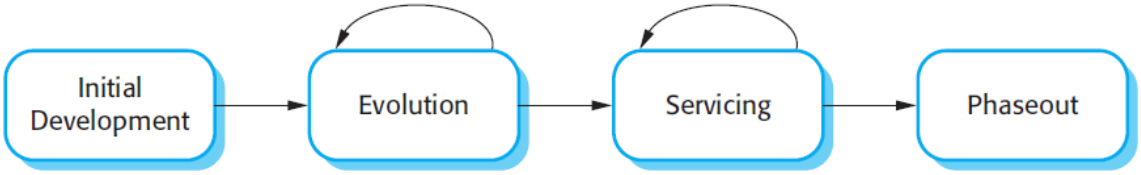

Figure 2: Evolution and servicing

2.1 Evolution processes

- Software evolution processes vary depending on the type of software being maintained, the development processes used in an organization and the skills of the people involved.
- System change proposals are the driver for system evolution in all organizations. Change proposals may come from existing requirements that have not been implemented in the released system, requests for new requirements, bug reports from system stakeholders, and new ideas for software improvement from the system development team.
- The processes of change identification and system evolution are cyclic and continue throughout the lifetime of a system as shows in Figure 3.

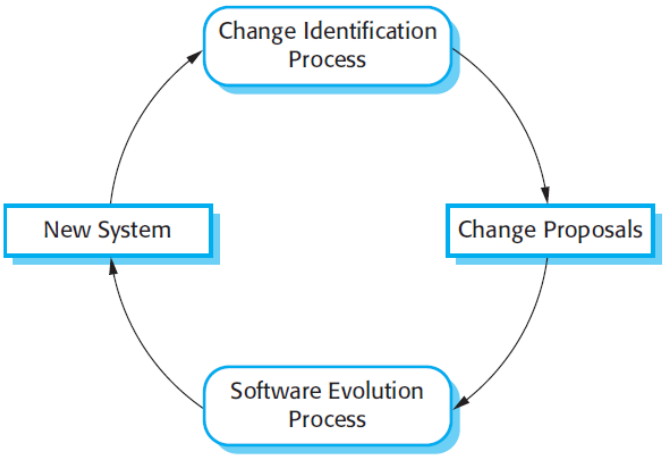

Figure 3: Change identification and evolution processes

The software evolution process

- Figure 4 shows an overview of the evolution process. The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.
- If the proposed changes are accepted, a new release of the system is planned.
- During release planning, all proposed changes such as 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.
- The process then iterates with a new set of changes proposed for the next release.

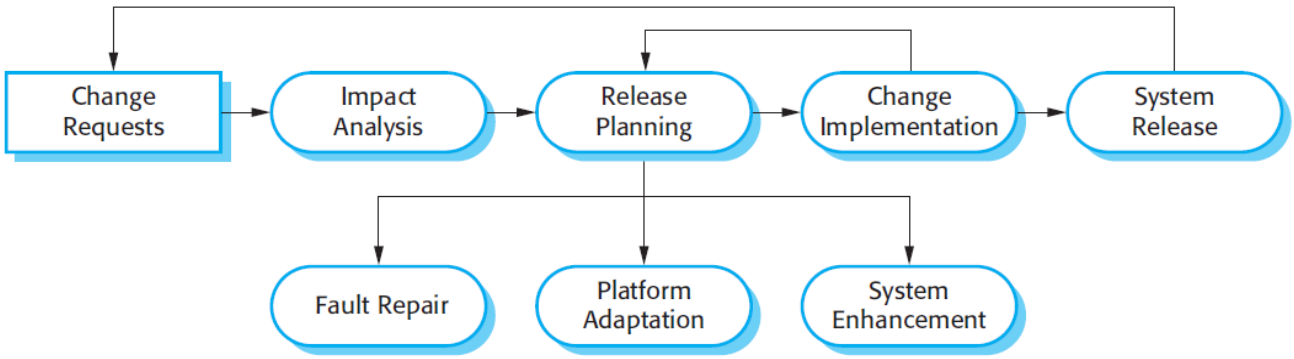

Figure 4: The software evolution process

- The **Change implementation** stage should modify the system specification, design, and implementation to reflect the changes to the system which is shown in Figure 5.
- New requirements that reflect the system changes are proposed, analyzed, and validated. System components are redesigned and implemented and the system is retested.

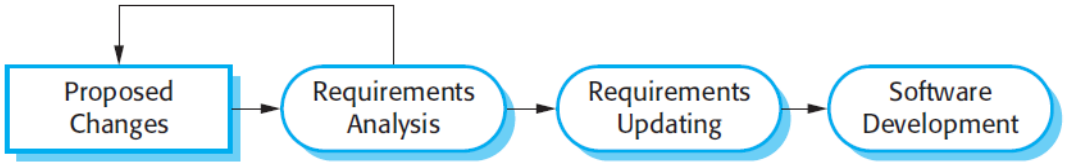

Figure 5: Change Implementation

Change requests arise for three reasons:

1. If a serious system fault occurs that has to be repaired.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation.
3. If there are unanticipated changes to the business running the system

- To make the **change quickly** means that you may not be able to follow the formal change analysis process. Rather than modify the requirements and design, make an emergency fix to the program to solve the immediate problem (Figure 6).

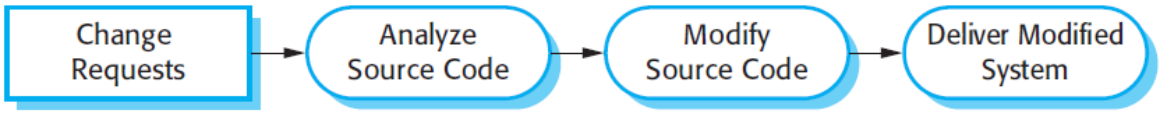

Figure 6: The emergency repair process

2.2 Program Evolution Dynamics

- Program evolution dynamics is the study of system change.
- After several major empirical studies, Lehman and Belady proposed that there were a number of 'laws' which apply to all systems as they evolved.They are applicable to large systems developed by large organizations.

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.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
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.

Figure: Lehman's laws

- The first law states that system maintenance is an inevitable process.
- The second law states that, as a system is changed, its structure is degraded.
- The third law states that the system size, time between releases and system errors are invariant for each system release
- Lehman's fourth law suggests that most large programming projects work in a 'saturated' state. That is, a change to resources or staffing has invisible effects on the long-term evolution of the system.
- Lehman's fifth law is concerned with the change increments in each system release. Adding new functionality to a system inevitably introduces new system faults. The more functionality added in each release, the more faults there will be.
- The sixth and seventh laws are similar and essentially say that users of software will become increasingly unhappy with it unless it is maintained and new functionality is added to it.

2.3 Software Maintenance

- Software maintenance focuses on **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.

There are three different types of software maintenance

1. **Fault repairs:** Coding errors are usually relatively cheap to correct. Design errors are more expensive as they may involve rewriting several program components. Requirements errors are the most expensive to repair because of the extensive system redesign which may be necessary.
2. **Environmental adaptation**: When the system's environment such as the hardware, the platform operating system, or other support software changes, this types of maintenance is required. The application system must be modified to adapt it to cope with these environmental changes.
3. **Functionality addition**: This type of maintenance is necessary when the system requirements change in response to organizational or business change.

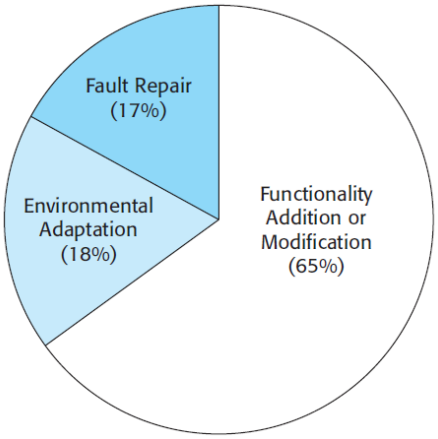

Figure 8: Maintenance effort distribution

Figure 9 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.

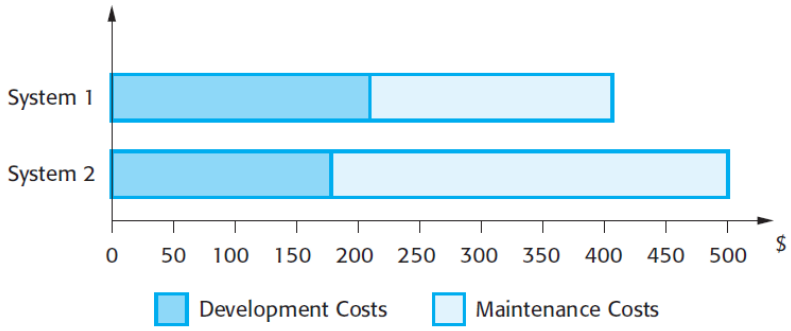

Figure 9: Development and maintenance costs

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. This assumes that a percentage increase in development costs results in a comparable percentage decrease in overall system costs.

The reasons for more expensive Maintenance cost include factors such as:

1. **Team stability:** After a system has been delivered, the development team to be broken up and for people to work on new projects. The new team or the individuals responsible for system maintenance do not understand the system or the background to system design decisions. They need to spend time understanding the existing system before implementing changes to it.
2. **Poor development practice:** The contract to maintain a system is usually separate from the system development contract. Lack of team stability, if there is no incentive for a development team to write maintainable software and if a development team cut corners to save effort during development then software is more difficult to change in the future.

3. **Staff skills:** Maintenance staff are relatively inexperienced and unfamiliar with the application domain.
4. **Program age and structure:** as programs age, their structure is degraded and they become harder to understand and change.

2.3.1 Maintenance prediction

Maintenance prediction is concerned with what system changes might be proposed and what parts of the system are likely to be the most difficult to maintain and to estimate the overall maintenance costs for a system in a given time period.

Figure 10 shows these predictions and associated questions.

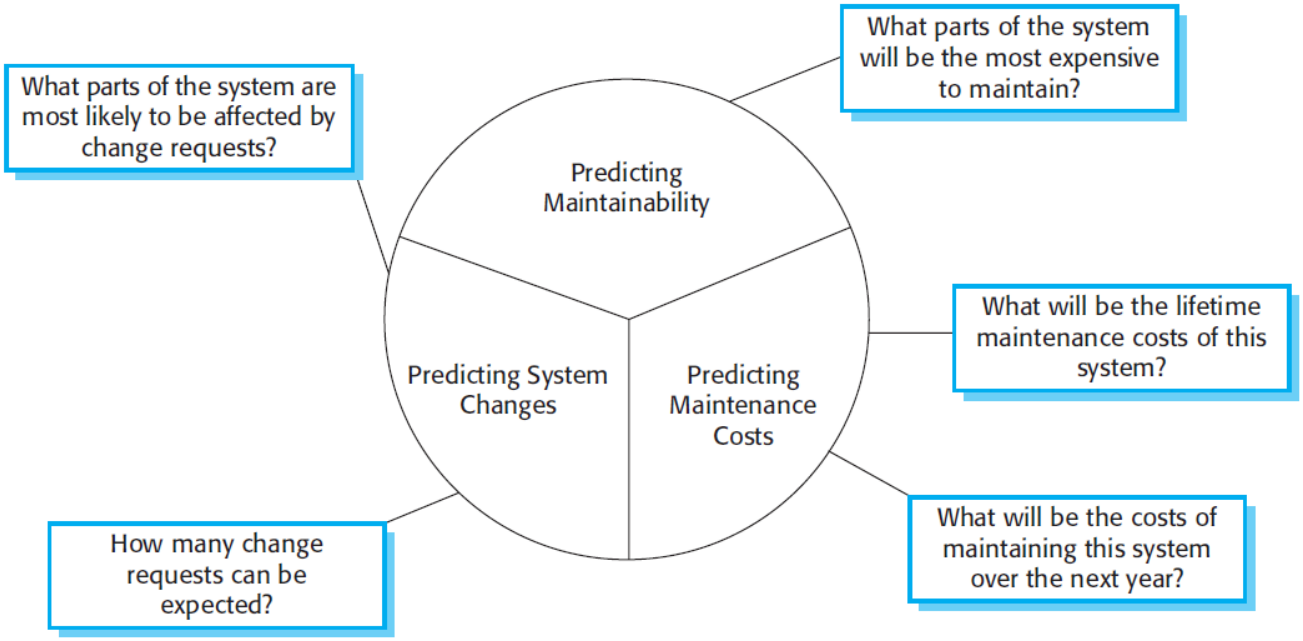

Figure 10: Maintenance prediction

To evaluate the relationships between a system and its environment, assess the following:

1. The number and complexity of system interfaces:
2. The number of inherently volatile system requirements:
3. The business processes in which the system is used

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:

1. Complexity of control structures
2. Complexity of data structures
3. Object, method (procedure) and module size.

Process metrics may be used to assess maintainability are as follows:

1. **Number of requests for corrective maintenance:** An increase in the number of bug and failure reports may indicate that more errors are being introduced into the program than are being repaired during the maintenance process.
2. **Average time required for impact analysis:** This reflects the number of program components that are affected by the change request. If this time increases, it implies more and more components are affected and maintainability is decreasing.
3. **Average time taken to implement a change request:** This is the amount of time need to modify the system and its documentation. An increase in the time needed to implement a change may indicate a decline in maintainability.
4. **Number of outstanding change requests:** An increase in this number over time may imply a decline in maintainability.

2.3.2 Software Reengineering

Reengineering involves re-documenting the system, refactoring the system architecture, translating programs to a modern programming language, modifying and updating the structure, and values of the system's data.

There are two important benefits from reengineering rather than replacement.

1. **Reduced risk:** there is a high risk in new software development. There may be development problems, staffing problems and specification problems.
2. **Reduced cost**: the cost of reengineering is significantly less than the costs of developing new software.

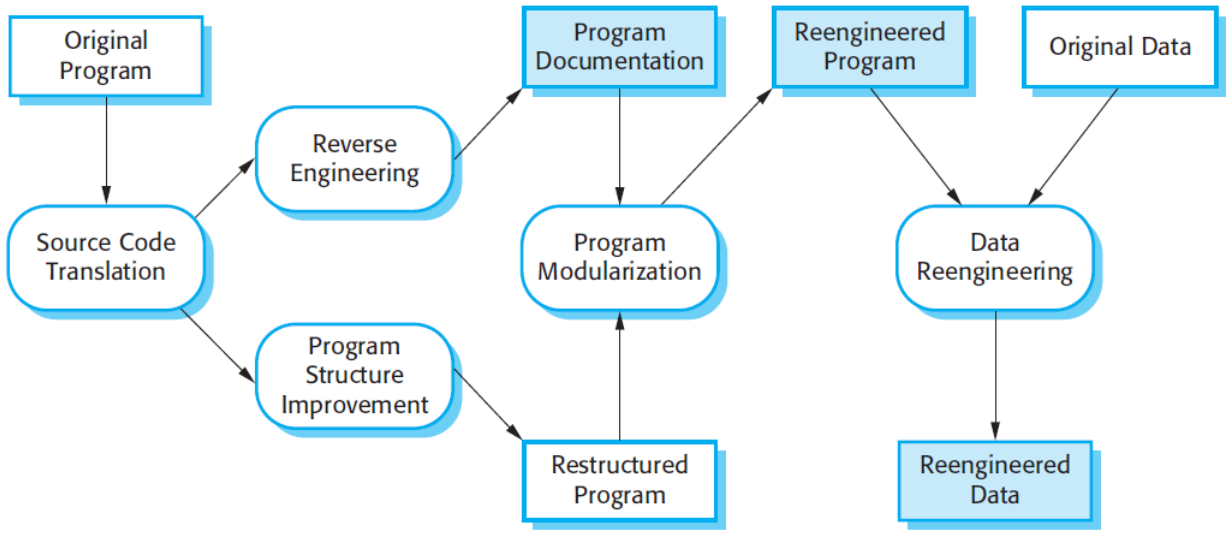

Figure 11: The reengineering process

Figure11 is a general model of the reengineering process. The input to the process is a legacy program and the output is an improved and restructured version of the same program.

The activities in reengineering process are as follows:

1. **Source code translation:** Using a translation tool, the program is converted from an old programming language to a more modern version of the same language or to a different language.
2. **Reverse engineering:** The program is analyzed and information extracted from it. This helps to document its organization and functionality.
3. **Program structure improvement:** The control structure of the program is analyzed and modified to make it easier to read and understand. This can be partially automated but some manual intervention is usually required.
4. **Program modularization:** Related parts of the program are grouped together and, where appropriate, redundancy is removed and architectural refactoring may be involved. This is a manual process.
5. **Data reengineering:** The data processed by the program is changed to reflect program changes and clean up the data.

The costs of reengineering depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in Figure 12.

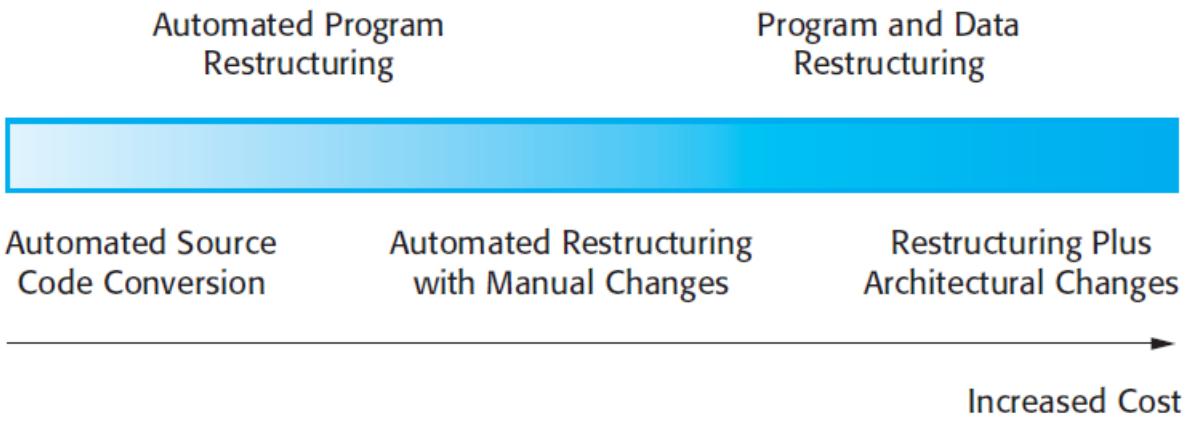

Figure 12: Reengineering approaches

2.3.3 Preventative maintenance by refactoring

- Refactoring is the process of making improvements to a program to slow down degradation through change. Refactoring could be assumed 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.
- Reengineering takes place after a system has been maintained for some time and maintenance costs are increasing. Automated tools can be used to process and reengineer 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' of code are stereotypical situations in which the code of a program can be improved through refactoring:

1. **Duplicate code:** The same of 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.
2. **Long methods**: If a method is too long, it should be redesigned as a number of shorter methods.
3. **Switch (case) statements**: These often involve duplication, where the switch depends on the type of some value. The switch statements may be scattered around a program. In object-oriented languages, polymorphism can be used to achieve the same thing.
4. **Data clumping**: Data clumps occur when the same group of data items reoccur in several places in a program. These can be replaced with an object encapsulating all of the data.
5. **Speculative generality**: This occurs when developers include generality in a program in case it is required in future. This can often simply be removed.

9.4 Legacy system management

Legacy system is an old method, technology, computer system, application program or an outdated computer system.
Organizations that rely on legacy systems a must make realistic assessment of their legacy systems and then deciding on the most appropriate strategy.

1. **Scrap the system completely:** This option should be chosen when the system is not making an effective contribution to business processes.
2. **Leave the system unchanged and continue with regular maintenance:** This option should be chosen when the system is still required but is fairly stable and the system users make relatively few change requests
3. **Reengineer the system to improve its maintainability:** This option should be chosen when the system quality has been degraded by change and where a new change to the system is still being proposed.
4. **Replace all or part of the system with a new system:** This option should be chosen when old system cannot continue in operation or where off-the-shelf systems would allow the new system to be developed at a reasonable cost.

For example, assume that an organization has 10 legacy systems. Assessing should be done based on the quality and the business value of each of these systems.

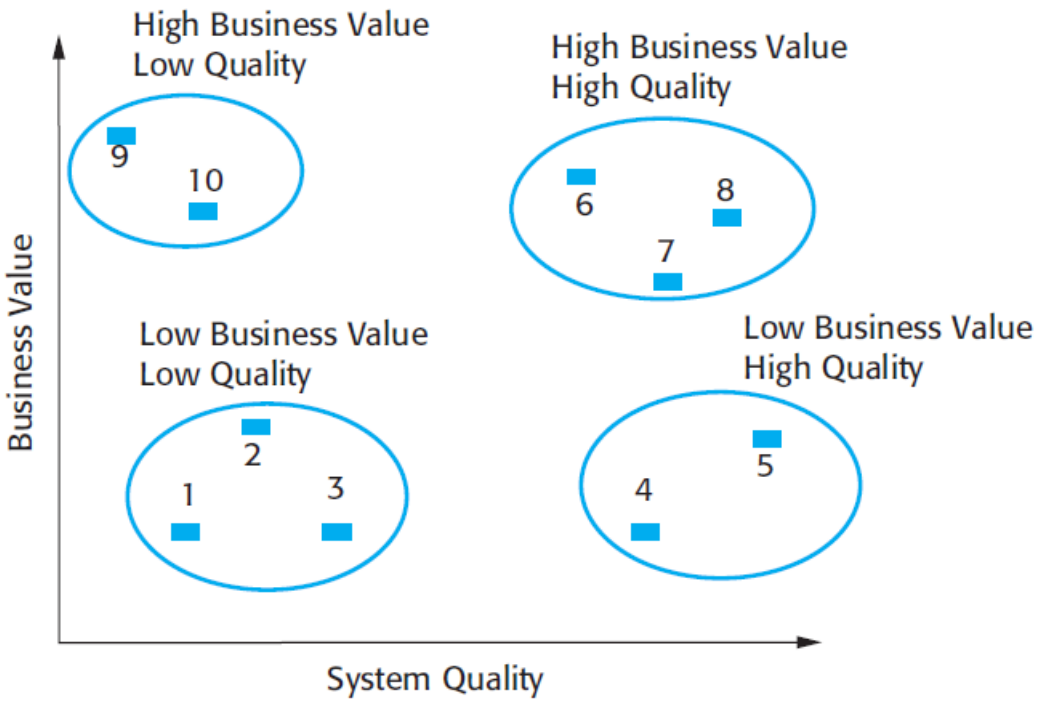

Figure 13: An example of a legacy system assessment

In Figure 13, there are four clusters of systems:

1. **Low quality, low business value:** These systems should be scrapped.
2. **Low quality, high business value:** These systems are making an important business contribution so they cannot be scrapped. These systems should be reengineered or may be replaced, if a suitable system is available.
3. **High quality, low business value:** It is not worth replacing these systems so normal system maintenance may be continued if expensive changes are not required and the system hardware remains in use. If expensive changes become necessary, the software should be scrapped.
4. **High quality, high business value:** These systems have to be kept in operation

To assess a software system from a technical perspective, consider both the application system itself and the environment in which the system operates.

Consider factors during the environment assessment are shown in Figure 14.

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?
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?

Figure 14: Factors used in environment assessment

To assess the technical quality of an application system, assess a range of factors as shown in Figure 15.

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?
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?

Figure 15: Factors used in application assessment

Data collected may be useful in quality assessment are:

1. **The number of system change requests:** System changes usually corrupt the system structure and make further changes more difficult.
2. **The number of user interfaces:** The more interfaces, the more likely that there will be inconsistencies and redundancies in these interfaces.
3. **The volume of data used by the system:** The higher the volume of data, more data inconsistencies that reduce the system quality.