Module 5

Software Evolution

Introduction

Software development does not stop when a system is delivered but continuesthroughout the lifetimeof the system. After a system has been deployed, it inevitablyhas to change if it is to remain useful.

Causes of system change

- Business changes and changes to user expectationsgenerate new requirements for the existing software.
- Parts of the softwaremay have to be modified to correct errors that are found in operation
- To adapt itforchanges to its hardware and software platform
- To improve its performance orother non- functional characteristics.

An alternative view of the software evolutionlife cycle, as shown in the following figure

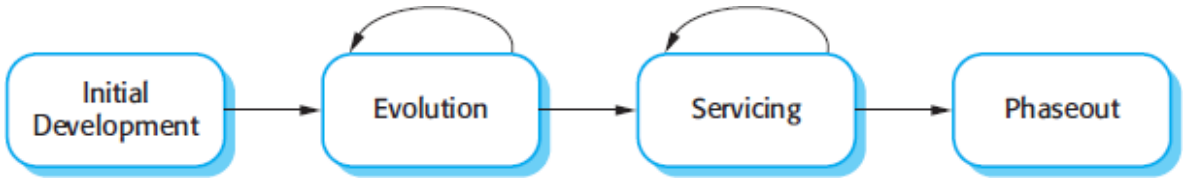

Fig: Evolution and Servicing

- This model is depicted as having four phases – initial development, evolution, servicing and phaseout.
- Evolution is the phase in which significant changes to the softwarearchitecture and functionality may be made.
- During evolution, the software is used successfully and there is a constant streamof proposedrequirements changes.
- At some stage in the life cycle, thesoftware reaches a transition point where significant changes,implementing newrequirements, become less and less cost effective.
- At that stage, the software moves from evolution to servicing.
- During servicing, the only changes thatare made are relatively small, essential changes. However, the software is still useful.
- In the final stage, phase-out, the software may still be used but nofurther changes are beingimplemented.

Evolution processes

Change identification and evolution process

System change proposals causes system evolution in all organizations.

Change proposals may come from

- existing requirements that have not been implementedin the released system
- requests for new requirements
- bug reports from system stakeholders
- new ideas for software improvement from the system development team

The processes ofchange identification and system evolution are cyclic andcontinue throughout the lifetime ofa system. This process is as shown in the following figure

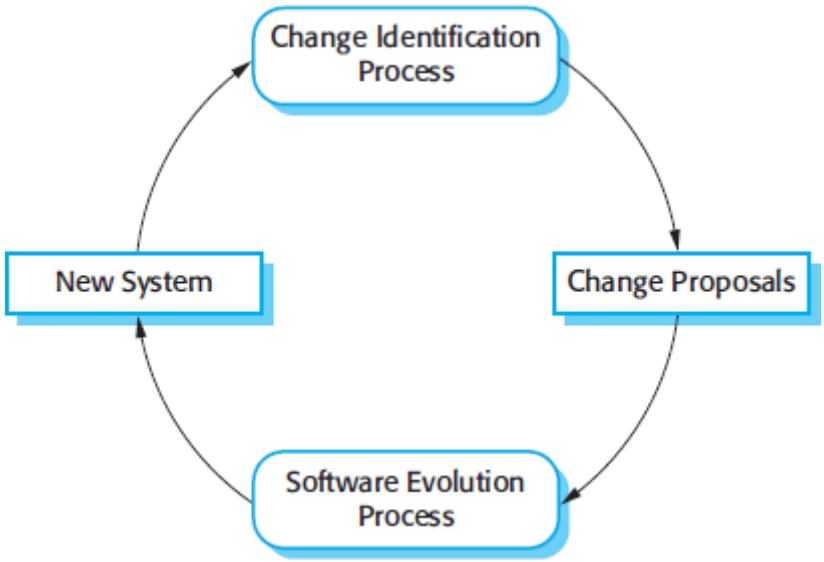

Fig: Change identification and evolution process

The software evolution process

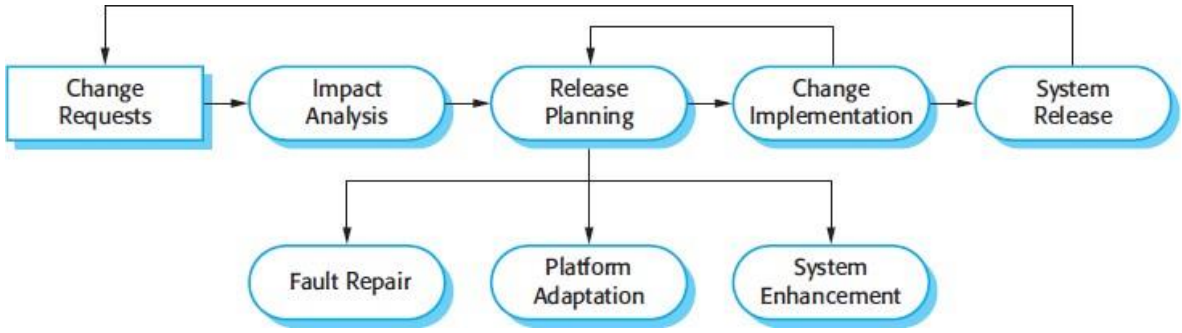

Fig: The software evolution process

- An overview of the evolution process is as shown in the above figure
- The process includes the fundamental activities of *change analysis, release planning,systemimplementation*, and *releasing a system* to customers.
- In the impact analysis stage, cost and impact of proposed changes are assessed to see
  - how much of the system is affected by the change,
  - how much it might cost toimplement the change.
- If the proposed changes are accepted,a new release of the system isplanned.

- During release planning, all proposed changes - fault repair, adaptation, and newfunctionality - 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 processthen iterates with a new set of changes proposed for the next release.

The change implementation process

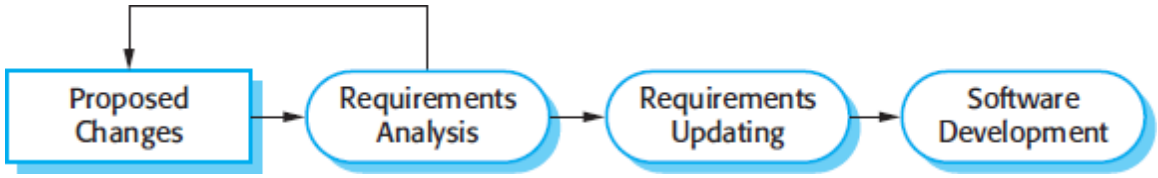

Fig: Change Implementation

- Change implementation is an iteration of the development process, where the revisions to the system are designed, implemented, and tested.
- The change implementation process is as shown in the above figure.
- New requirements that reflect the system changes are proposed, analysed, and validated.
- System components are redesigned and implemented and the system is retested.
- If appropriate, prototyping of the proposed changes may be carried out as part of the change analysis process.

The emergency repair process

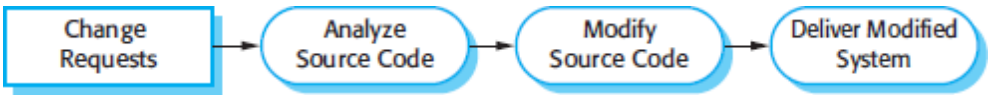

Fig: The emergency repair process

- Change requests sometimes relate to system problems that have to be tackled urgently.
- These urgent changes can arise for three reasons:
  1. If a serious system fault occurs that has to be repaired to allow normal operation to continue.
  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, such as the emergence of new competitors or the introduction of new legislation that affects the system.
- The emergency repair process is required to quickly fix the above problems.
- The source code is analyzed and modified directly, rather than modifying the requirements and design
- The disadvantages of emergency repair process are as follows
  - the requirements, the software design, and the code become inconsistent
  - the process of software aging is accelerated since a quick workable solution is chosen rather than the best possible solution for quick fix
  - future changes become more difficult and maintenance costs increase

Program evolution dynamics

- Program evolution dynamics is the study of system change.
- Lehman's laws' concerning system change are as shown below

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.

Continuing change

- The first law states that system maintenance is an inevitable process.
- As the system's environment changes, new requirements emerge and the system must be modified.

Increasing complexity

- The second law states that, as a system is changed, its structure is degraded.
- To avoid this, invest in preventative maintenance.
- Time is spent improving the software structure without adding to its functionality.
- This means additional costs, more than those of implementing required system changes.

Large program evolution

- It suggests that large systems have a dynamic of their own
- This law is a consequence of *structural factors* that influence and constrain system change, and *organizational factors* that affect the evolution process.

- **Structural factors:**

- These factors come from complexity of large systems.
- As you change and extend a program, its structure tends to degrade.

- Making large changes to a program may introduce new faults and then inhibit further program changes.
  - **Organisational factors:**
- These are produced by large organizations.
- Companies have to make decisions on the risks and value of the changes and the costs involved. Such decisions take time to make.
- The speed of the organization's decision-making processes therefore governs the rate of change of the system.

Organizational stability

- In most large programming projects a change to resources or staffing has imperceptible (slight) effects on the long-term evolution of the system.

Conservation of familiarity

- Adding new functionality to a system inevitably introduces new system faults.
- The more functionality added in each release, the more faults there will be.
- Relatively little new functionality should be included in this release.
- This law suggests that you should not budget for large functionality increments in each release without taking into account the need for fault repair.

Continuing growth

- The functionality offered by systems has to continually increase user satisfaction.
- The users of software will become increasingly unhappy with it unless it is maintained and new functionality is added to it.

Declining quality

- The quality of systems will decline unless they are modified to reflect changes in their operational environment.

Feedback system

- Evolution processes must incorporate feedback systems to achieve significant product improvement.

Software maintenance

- It is the general process of changing a system after it has been delivered.
- There are three different types of software maintenance:
  - Fault repairs
  - Environmental adaptations
  - Functionality addition

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.

Environmental adaptation

- This type of maintenance is required when some aspect of the system's environment such as the hardware, the platform operating system, or other support software changes.
- The application system must be modified to adapt it to cope with these environmental changes.

Functionality addition

- This type of maintenance is necessary when the system requirements change in response to organizational or business change.

Maintenance effort distribution

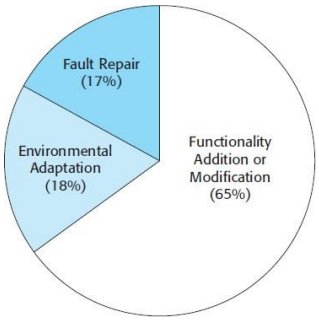

Fig: Maintenance effort distribution

- Software maintenance takes up a higher proportion of IT budgets than new development.
- Also, most of the maintenance budget and effort is spent on implementing new requirements than on fixing bugs. This is shown in the above figure

Development and maintenance costs

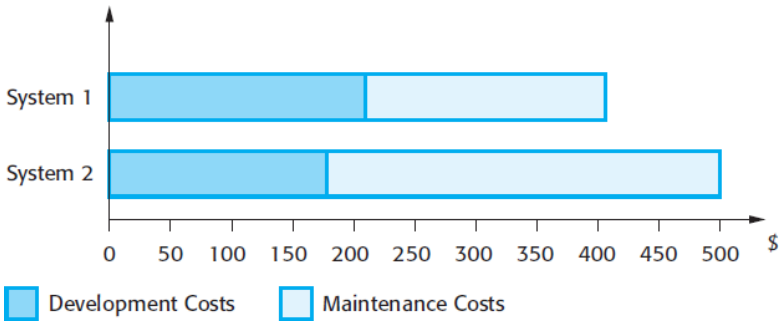

Fig: Development and maintenance costs

- The above figure shows that overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system.
- In system 1, more development cost has resulted in lesser overall lifetime costs when compared to system 2.
- It 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:

1. Team stability

- The new team or the individuals responsible for system maintenance are usually not the same as the people involved in development
- They 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.
- There is no incentive for a development team to write maintainable software.
- The development team may not write maintainable software to save effort.
- This means that the software is more difficult to change in the future.

3. Staff skills

- Maintenance is seen as a less-skilled process than system development.
- It is often allocated to the most junior staff.
- Also, old systems may be 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.

4. Program age and structure

- As changes are made to programs, their structure tends to degrade.
- As programs age, they become harder to understand and change.
- System documentation may be lost or inconsistent.
- Old systems may not have been subject to stringent configuration management so time is often wasted finding the right versions of system components to change.

Maintenance prediction

- It is important try to predict what system changes might be proposed and what parts of the system are likely to be the most difficult to maintain.
- Also estimating the overall maintenance costs for a system in a given time period is important.

- The following figure shows these predictions and associated questions

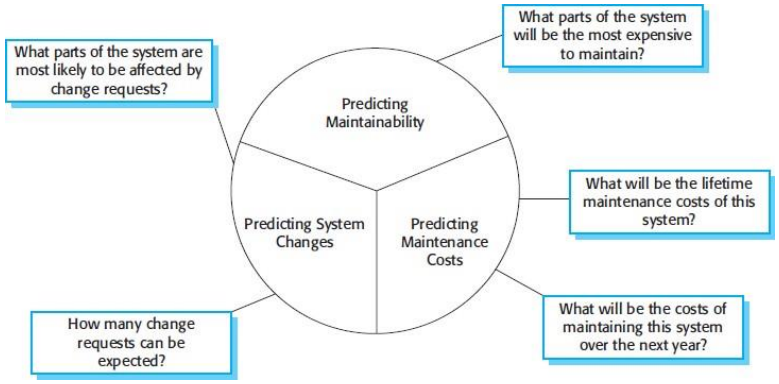

Fig: Maintenance prediction

- The number of change requests for a system requires an understanding of the relationship between the system and its external environment.

- Therefore, to evaluate the relationships between a system and its environment the following assessment should be made

1. **The number and complexity of system interfaces** The larger the number of interfaces and the more complex these interfaces, the more likely it is that interface changes will be required as new requirements are proposed.
2. **The number of inherently volatile system requirements** The requirements that reflect organizational policies and procedures are likely to be more volatile than requirements that are based on stable domain characteristics.
3. **The business processes in which the system is used** As business processes evolve, they generate system change requests. The more business processes that use a system, the more the demands for system change.

- After a system has been put into service, the process data may be used to help predict maintainability.
- The process metrics that can be used for assessing 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. This may indicate a decline in maintainability.
2. **Average time required for impact analysis** 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 needed 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.

Software reengineering

- Reengineering is done to improve the structure and understandability of legacy software systems
- Reengineering makes legacy software systems easier to maintain
- Reengineering may involve redocumenting 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.
- The functionality of the software is not changed due to reengineering

Benefits of reengineering

Reduced risk Reengineering reduces the high risk in redeveloping business-critical software. Errors may be made in the system specification or there may be development problems. Delays in introducing the new software may mean that business is lost and extra costs are incurred.

Reduced cost The cost of reengineering may be significantly less than the cost of developing new software.

The reengineering process

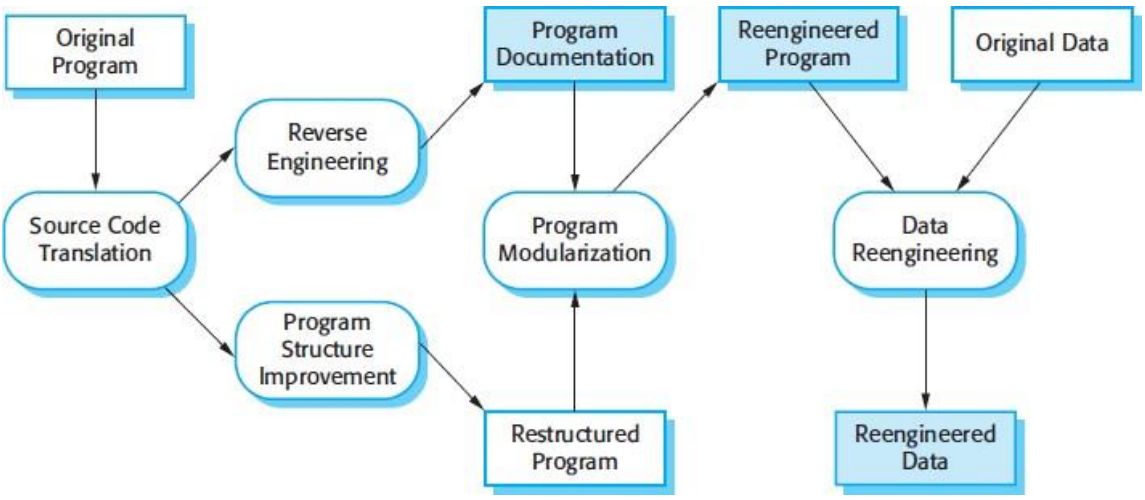

Fig: The reengineering process

The activities involved 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.
   - This process is usually completely automated.

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.
   - Where appropriate, redundancy is removed.
   - This is a manual process.

5. **Data reengineering**
   - The data processed by the program is changed to reflect program changes.
   - This may mean redefining database schemas, converting existing databases to the new structure, clean up the data, finding and correcting mistakes, removing duplicate records, etc.
   - Tools are available to support data reengineering.

Reengineering approaches

- The costs of reengineering depend on the extent of the work that is carried out.
- The following figure shows a spectrum of possible approaches to reengineering

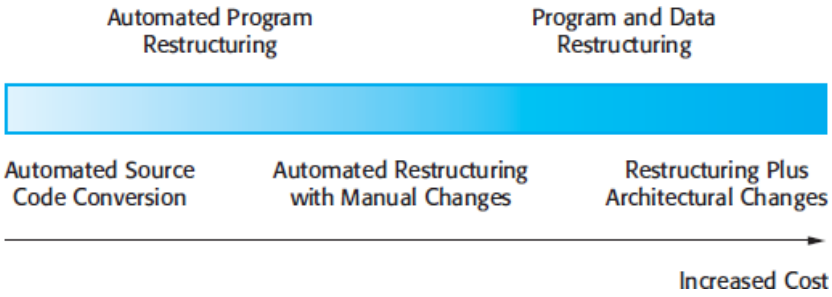

Fig: **Reengineering approaches**

- Costs increase from left to right.
- Source code translation is the cheapest option.
- Reengineering as part of architectural migration is the most expensive.

Disadvantages of reengineering

- There are limits to how much you can improve a system by reengineering.
- It isn't possible to convert a system written using a functional approach to an object-oriented system.
- Major architectural changes of the system data management cannot be carried out automatically.
- The reengineered system will probably not be as maintainable as a new system developed using modern software engineering methods.

<u>Preventative maintenance by refactoring</u>

- Refactoring is the process of making improvements to a program to slow down degradation through change.
- It means modifying a program to improve its structure, to reduce its complexity, or to make it easier to understand.
- Refactoring a program is not adding new functionality.
- Refactoring is considered as *'preventative maintenance'* that reduces the problems of future change.

Difference between reengineering and refactoring

- Reengineering takes place after a system has been maintained for some time and maintenance costs are increasing.
- 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.

- There are situations (bad smells) in which the code of a program can be improved or refactored. They are as follows

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, you can often use polymorphism to achieve the same thing.
4. **Data clumping** Data clumps occur when the same group of data items (fields in classes, parameters in methods) reoccur in several places in a program. These can often 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.

Legacy system management

- Organizations have a limited budget for maintaining and upgrading legacy systems.
- They have to decide how to get the best return on their investment.
- This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems.

- There are four strategic options:

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 factors, such as new hardware, mean that the old system cannot continue in operation or where off-the-shelf systems would allow the new system to be developed at a reasonable cost.

Legacy system assessment

Legacy systems can be assessed from two perspectives

- **Business perspective** is to decide whether or not the business really needs the system.
- **Technical perspective** is to assess the quality of the application software and the system's support software and hardware.

There are four clusters of systems

1. **Low quality, low business value** Keeping these systems in operation will be expensive and the rate of the return to the business will be fairly small. These systems should be *scrapped.*
2. Low quality, high business value These systems are making an important business contribution so they cannot be scrapped. However, their low quality means that it is expensive to maintain them. These systems should be *reengineered* to improve their quality. They may be replaced, if a suitable off-the-shelf system is available.

***3.* High quality, low business value** These are systems that don't contribute much to the business but which may not be very expensive to maintain. 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. However, their high quality means that you don't have to invest in transformation or system replacement. *Normal system maintenance* should be continued.

Business perspective

The four basic issues that have to be discussed with system stakeholders to assess business value of the system are as follows

1. 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. However, there may be occasional but important use of systems. For example, in a university, a student registration system may only be used at the beginning of each academic year. However, it is an essential system with a high business value.

2. The business processes that are supported When a system is introduced, business processes are designed to exploit the system's capabilities. However, as the environment changes, the original business processes may become obsolete. Therefore, a system may have a low business value because it forces the use of inefficient business processes.

3. The system dependability If a system is not dependable and the problems directly affect the business customers or mean that people in the business are diverted from other tasks to solve these problems, the system has a low business value.

4. The system outputs If the business depends on the system outputs, then the system has a high business value. Conversely, if these outputs can be easily generated in some other way or if the system produces outputs that are rarely used, then its business value may be low.

Technical perspective

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

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?

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?
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?

Data can be collected to assess the quality of the system. The data that can be collected are

1. The number of system change requests System changes usually corrupt the system structure and make further changes more difficult. The higher this value, the lower the quality of the system.

2. The number of user interfaces The more interfaces, the more likely that there will be inconsistencies and redundancies in these interfaces, hence reducing system quality.