Module-4 Software Testing

Testing is intended to show that a program does what it is **intended to do** and to **discover program defects** before it is put into use. When you test software, you execute a program using artificial data (**test cases**). You check the results of the test run for errors, anomalies or information about the programs non-functional attributes.

Verification	Validation
Are we building the product right, The software should **conform to its specification.**	Are we building the right product". The software **should do** what the user really requires
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	**Validation** is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
Following activities are involved in **Verification**: Reviews, Meetings and Inspections.	Following activities are involved in **Validation**: Testing like black box testing, white box testing, gray box testing etc
Execution of code is not comes under **Verification**.	Execution of code is comes under **Validation**.
Cost of errors caught in **Verification** is less than errors found in Validation	Cost of errors caught in **Validation** is more than errors found in Verification.

- **Defect Testing**

- To discover faults or defects in the software where its behavior is incorrect or not in conformance with its specification
- A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

- **An input-output model of program testing**

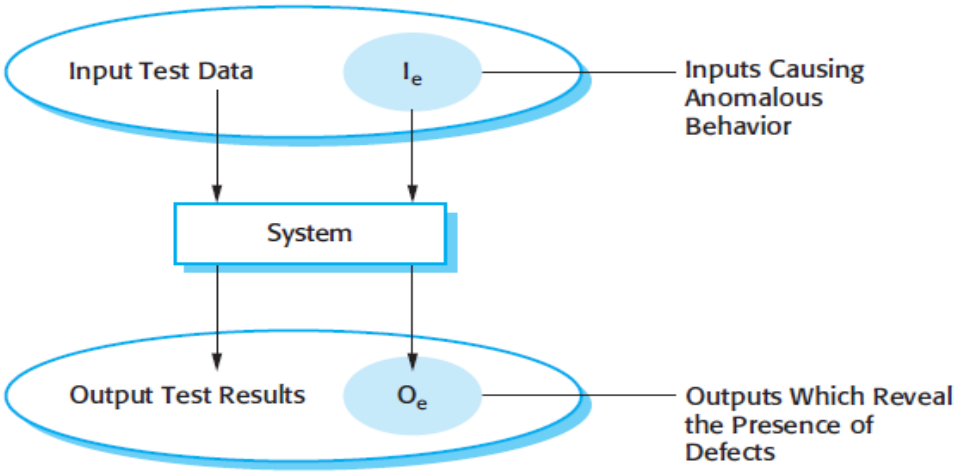

Fig 1 : Input-Output Model Of Program Testing

- **Inspections and testing**
  - Software inspections Concerned with **analysis** of the static system representation to discover problems *(*static verification)
    - May be supplement by tool-based document and code analysis.
  - Software testing Concerned with **exercising and observing product behaviour** (dynamic verification)
    - The system is executed with test data and its operational behaviour is observed.

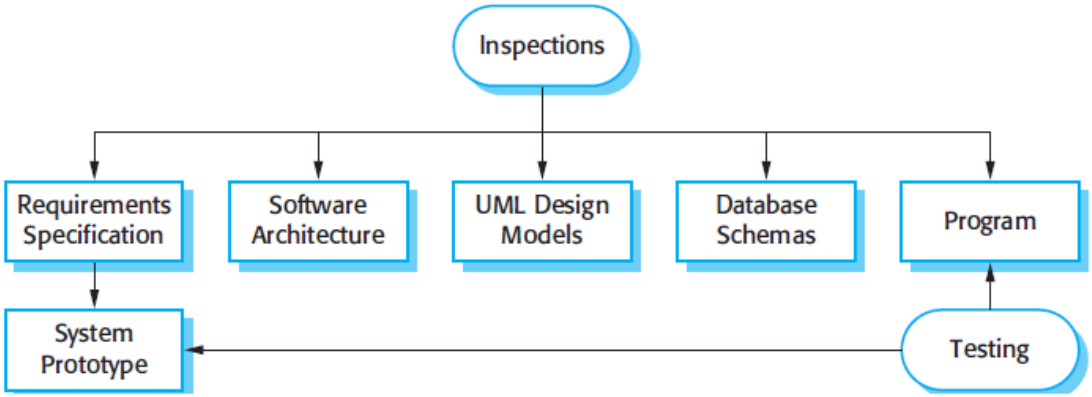

Fig 2: Inspection Method

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.

- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

➢ **Advantages of inspections**

- Incomplete versions of a system can be inspected **without additional costs**. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available.
- As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability and maintainability.

1.1 Stages of Testing Process (Activities)

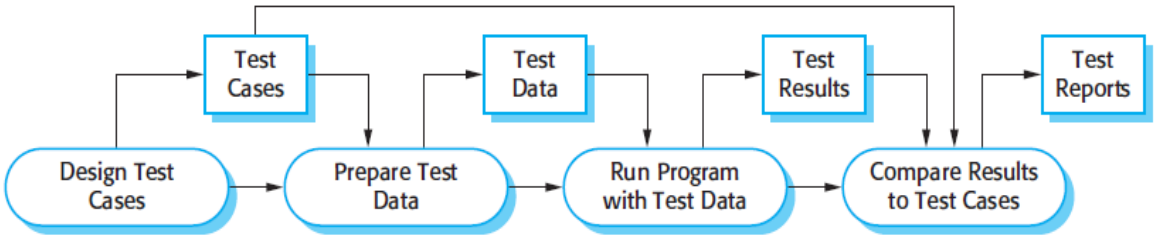

Fig 3: Stages Of Software Process (Activities)

1.2 Testing Types

1. **Development testing**, where the system is tested during development to discover bugs and defects.
2. **Release testing**, where a separate testing team test a complete version of the system before it is released to users.
3. **User testing,** where users or potential users of a system test the system in their own environment.

1.3 Development testing

- Development testing includes all testing activities that are carried out by the team developing that particular software or system. **Which includes 3 stages,**

1. **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
2. **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.

3. **System testing,** where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

1.3.1 Unit testing

- Unit testing is the process of **testing individual components** in isolation.
- It is a defect testing process.
- Units may include,
  - **Individual functions** or methods within an object
  - **Object classes** with several attributes and methods
  - **Composite components** with defined interfaces used to access their functionality.
- Complete test coverage of a class involves
  - **Testing all operations** associated with an object
  - **Setting and interrogating** all object attributes
  - **Exercising the object** in all possible states.
- **Problem in unit testing** : **Inheritance** makes it more difficult to design object class tests as the information to be tested is not localised

➢ **Example: The weather station object interface**

Weather station is class here, in that identifier is method with attributes and associated test cases are,

Reportweather->reportstatus->powersave->remotecontrol->reconfigure->restart->shutdown

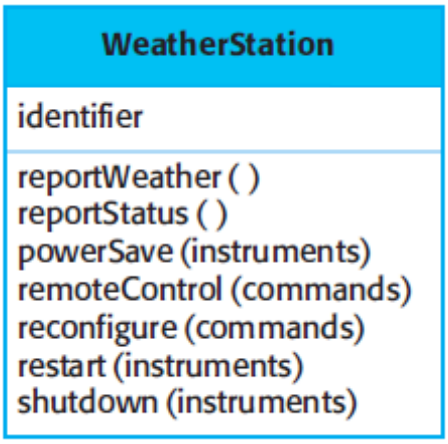

Fig 3: weather station

➢ **Unit test cases:**

Effective unit test case means,

- The **defect in the components** must be revealed by the test cases.
- The component under test must show the results as expected in various test cases.

➢ **Testing Strategies**

1. **Partition testing:** where you identify groups of inputs that have common characteristics and should be processed in the same way.
    - You should choose tests from within each of these groups.
2. **Guideline-based testing:** where you use testing guidelines to choose test cases.
    - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

➢ **<u>Partition testing</u>**

◇ Input data and output results often fall into different classes where all members of a class are related.

◇ Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.

◇ Test cases should be chosen from each partition.

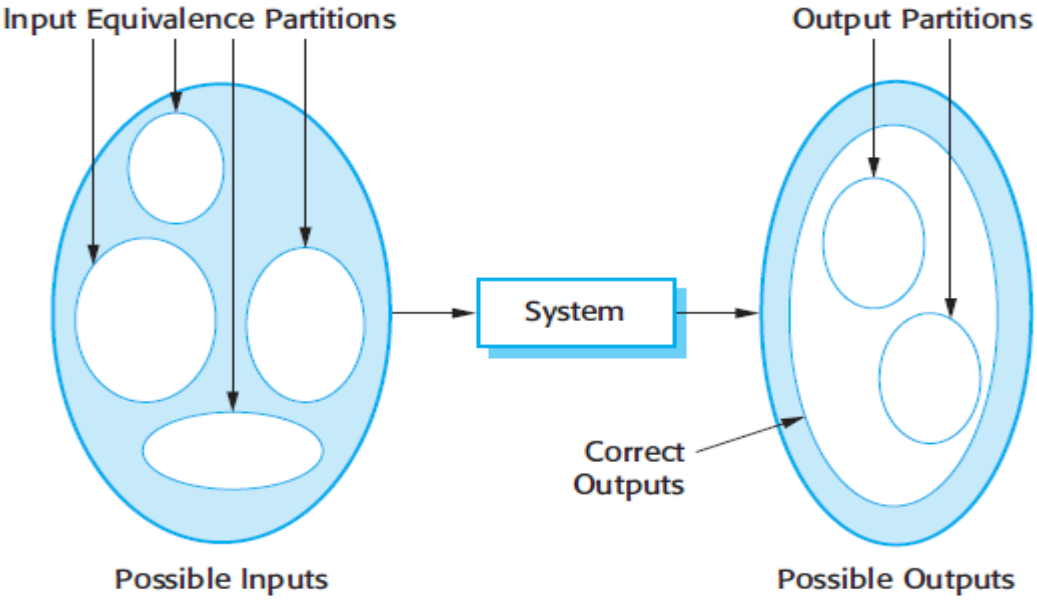

Fig 4: Equivalence partitioning

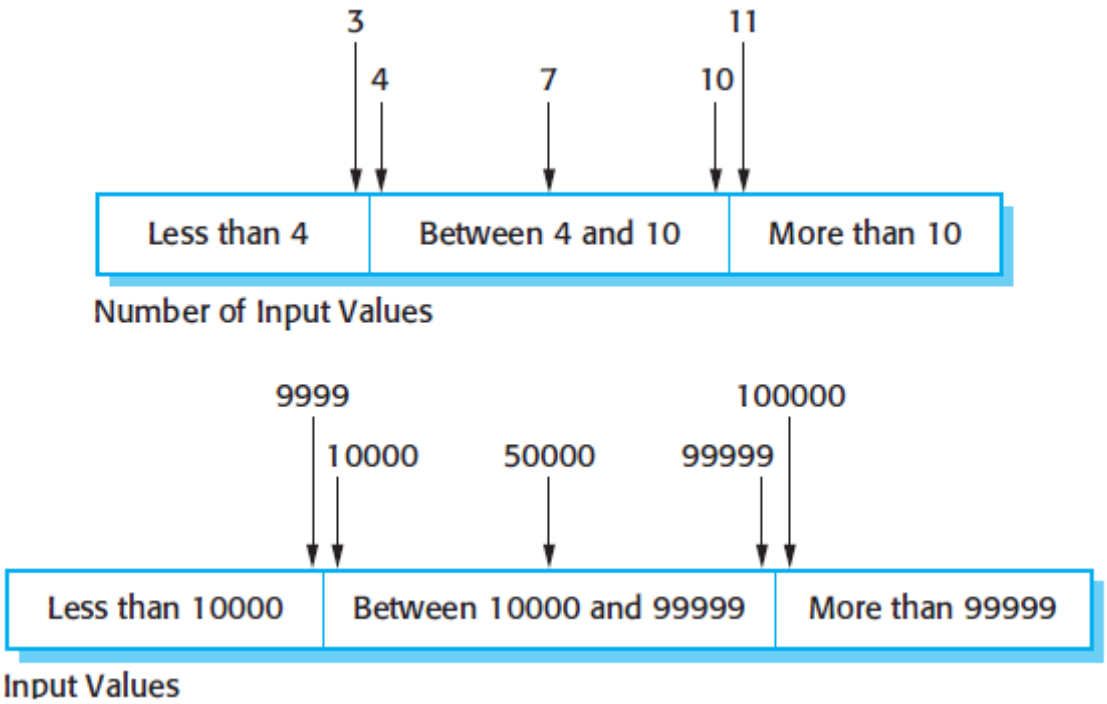

Fig 5: Equivalence Partitioning

➢ **General testing guidelines**

1. Choose inputs that force the system to **generate all error messages**
2. Design inputs that cause **input buffers** to overflow
3. **Repeat** the same input or series of inputs numerous times
4. Force **invalid outputs** to be generated
5. Force **computation results** to be too large or too small.

1.3.2 Component Testing

- Software components are often composite components that are made up of **several interacting objects.**
  - **For example**, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.

  You can assume that unit tests on the individual objects within the component have been completed.

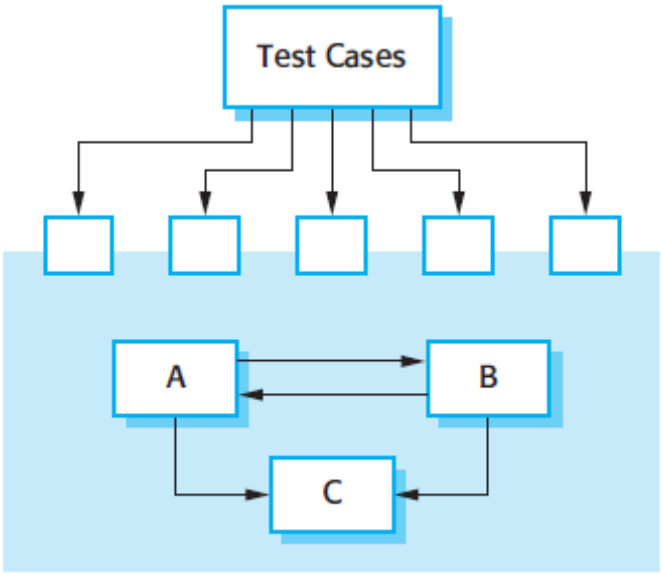

Fig 6 : Interface Testing

- **Interface testing**
  - Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
  - **Interface Types**

1. **Parameter interfaces** Data passed from one method or procedure to another.
2. **Shared memory** interfaces Block of memory is shared between procedures or functions.
3. **Procedural interfaces** Sub-system encapsulates a set of procedures to be called by other sub-systems.
4. **Message passing** interfaces Sub-systems request services from other sub-systems

- **Interface Errors**

1. Interface misuse

A calling component calls another component and makes an error in its use of its interface e.g. parameters in the wrong order.

2. Interface misunderstanding

A calling component embeds assumptions about the behaviour of the called component which are incorrect.

3. Timing errors

- The called and the calling component operate at different speeds and out-of-date information is accessed.

➢ **Interface testing guidelines**

1. Design tests so that **parameters** to a called procedure are at the extreme ends of their ranges.
2. Always test **pointer parameters** with null pointers.
3. Design tests which cause the **component to fail**.
4. Use **stress testing** in message passing systems.
5. In shared memory systems, **vary the order** in which components are activated.

1.3.3 System Testing

◇ System testing during development involves integrating components to create a version of the system and then **testing the integrated system.**

◇ The focus in system testing is testing the **interactions between components.**

◇ System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. System testing tests the emergent behavior of a system. The main goal of the system testing is to test the interaction between the components.

During the system testing:

a. Checks the **components are compatible**
b. Checks that all components **interaction** correctly
c. Checking the **Transfer of correct data** at specified time across the interfaces of units by keeping sequence diagram to know the sequence of activities.

Use-case testing

◇ The use-cases developed to identify system interactions can be used as a basis for system testing.

◇ Each use case usually involves several system components so testing the use case forces these interactions to occur.

The sequence diagrams associated with the use case documents the components and interactions that are being tested

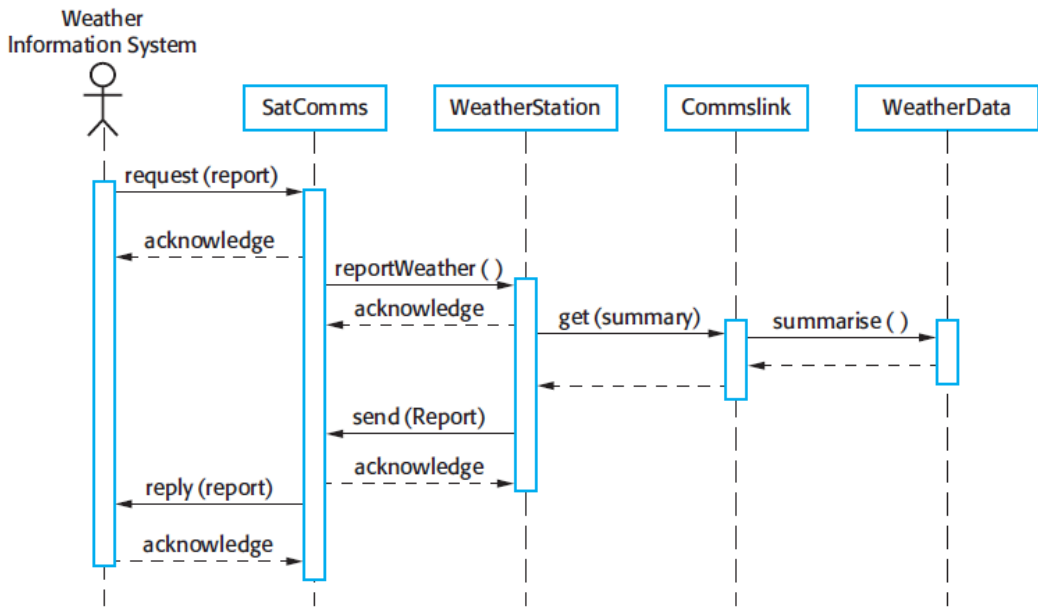

Fig 7 : Collect weather data sequence chart

➢ **Testing policies**

◇ Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.

◇ **Examples of testing policies:**

- All system functions that are accessed through menus should be tested.
- Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
- Where user input is provided, all functions must be tested with both correct and incorrect input.

1.4 Release testing

◇ Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.

◇ The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.

- Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.

◇ Release testing is usually a black-box testing process where tests are only derived from the system specification.

Release Testing	System Testing
A separate team that has not been involved in the system development, should be responsible for release testing.	System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

- **Requirements based testing**
  - Requirements-based testing involves examining each requirement and developing a test or tests for it.
  - MHC-PMS requirements:
    - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user.
    - If a prescriber chooses to ignore an allergy warning, they shall provide a reason why this has been ignored.
- **Requirements tests in MHC-PMS**

1	Set up a patient record with no known allergies, Prescribe medication for allergies that are known to exist	Check that a warning message is not issued by the system
2	Set up a patient record with a known allergy	Check that the warning is issued by the system.
3	Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately	Check that the correct warning for each drug is issued.
4	Prescribe two drugs that the patient is allergic too	Check that two warnings are correctly issued.
5	Prescribe a drug that issues a warning and overrule	Check that the system requires

	that warning	the user to provide information explaining why the warning was overruled.

Features tested by scenario

- It is a kind of release testing in which typical scenario is specified and using this scenario the test cases are designed.
- **Scenario is nothing but a story** that describes the working of the system.

Example : scenario for ATM

1. Authentication by logging on to the system.
2. Downloading and uploading of specified patient records to a laptop.
3. Home visit scheduling.
4. Encryption and decryption of patient records on a mobile device.
5. Record retrieval and modification.
6. Links with the drugs database that maintains side-effect information. The system for call prompting

➢ **Performance testing**

- Part of release testing may involve testing the emergent properties of a system, such as **performance and reliability**.
- Tests should reflect the **profile of use of the system**.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.

1.5 User testing

◇ User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing.

- User testing is essential, even when comprehensive system and release testing have been carried out.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

- **Types of user testing**

1.Alpha testing

- Users of the software work with the development team to test the software at the developer's site.

2.**Beta testing**

- A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

3. **Acceptance testing**

- Customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment. Primarily for custom systems.

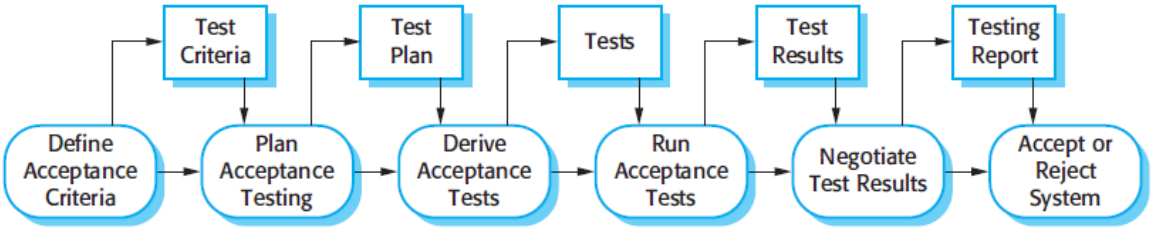

Fig 7:Acceptance Testing

Different stages includes,

1. **Define acceptance criteria:** should be defined at the early stage of process.its normally between customer and software developer.
2. **Plan acceptance testing:** decide the time, budget for acceptance testing.
3. **Derive acceptance tests**: output of this system consisting of various test cases that are designed to meet the acceptable criteria.
4. Run acceptance tests: designed test cases are executed. Test results are documented.
5. **Negotiate test results:** if some minute errors occur then developers and customer should discuss about that and will make some decision about the results.

6. **Reject/accept system:** if the system tested is up to mark , customer expection is met without any error then it is subjected for acceptance. Else rejection

1.6 Test-driven development

- Test-driven development (TDD) is an approach to program development in which you **inter-leave testing and code development.**
- Tests are written before code and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

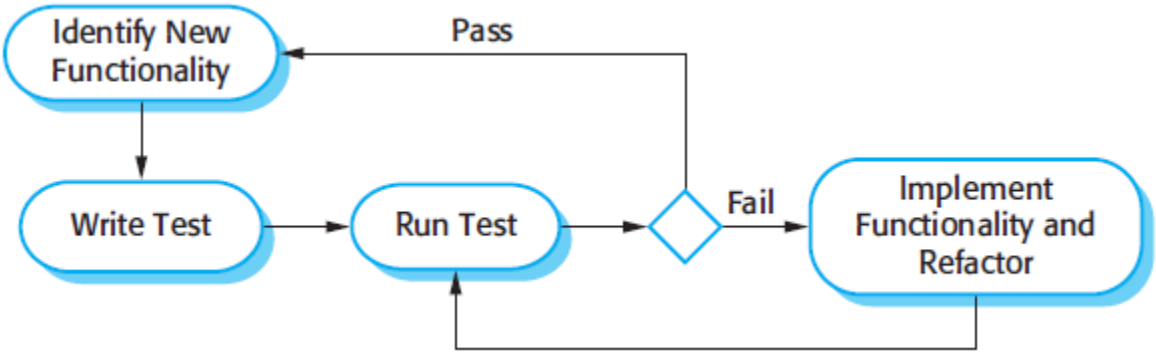

Fig 8: Test Driven Development

TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.

  Once all tests run successfully, you move on to implementing the next chunk of functionality

advantages of test-driven development

1. **Code coverage**

Every code segment that you write has at least one associated test so all code written has at least one test.

2. **Regression testing**

   A regression test suite is developed incrementally as a program is developed.

3. **Simplified debugging**

   When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.

4. **System documentation**

   The tests themselves are a form of documentation that describe what the code should be doing

1.7 Test Automation

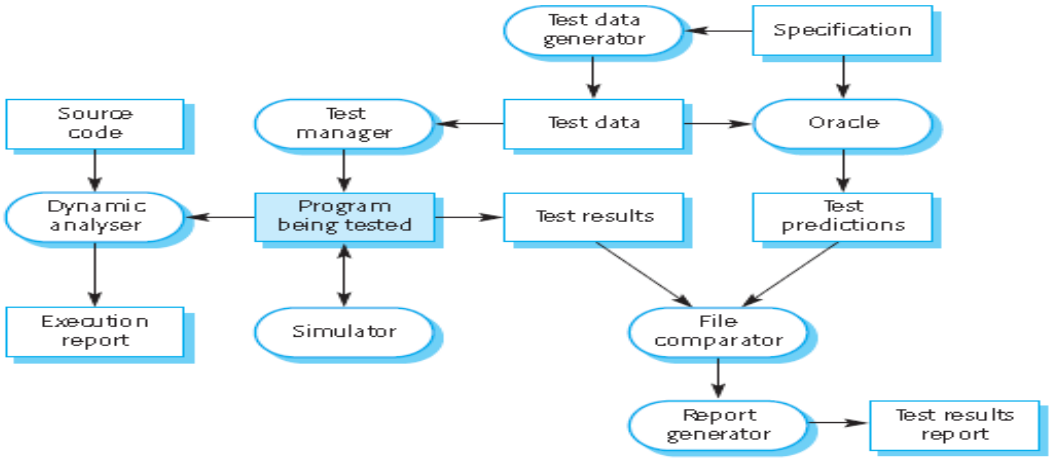

Fig 9:Test Automation

- Testing is an expensive and laborious phase of the software process. As a result, testing tools were among the first software tools to be developed. These tools now offer a range of facilities and their use can significantly reduce the costs of testing.
- A software testing workbench is an integrated set of tools to support the testing process. In addition to testing frameworks that support automated test execution, a workbench may include tools to simulate other parts of the system and to generate system test data.
- Fig 9 shows some of the tools that might be included in such a testing workbench,

1. ***Test manager*** Manages the running of program tests. The test manager keeps track of test data, expected results and program facilities tested.
2. ***Test data generator*** Generates test data for the program to be tested. This may be accomplished by selecting data from a database or by using patterns to generate random data of the correct form.
3. ***Oracle*** Generates predictions of expected test results. Oracles may either be previous program versions or prototype systems.
4. ***File comparator*** Compares the results of program tests with previous test results and reports differences between them.
5. ***Report generator*** Provides report definition and generation facilities for test results.
6. ***Dynamic analyser*** Adds code to a program to count the number of times each statement has been executed.

7 ***Simulator*** Different kinds of simulators may be provided. Target simulators simulate the machine on which the program is to execute

2. Software Evolution

- After systems have been deployed, they i**nevitably have to change if they are to remain useful.** Once software is put into use, new requirements emerge and existing requirements change. Business changes often generate new requirements for existing software.
- The majority of changes are a consequence of **new requirements** that are generated in response to changing business and user needs.
- Consequently, you can think of software engineering as a spiral process with requirements, design, implementation and testing going on throughout the lifetime of the system. This is illustrated in Figure 2.1.

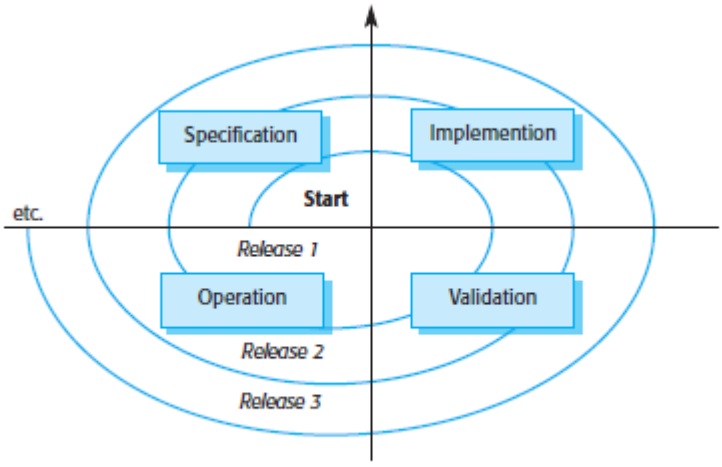

Figure 2.1 A spiral model of development and evolution

2.1 Evolution Processes

- Software evolution processes vary considerably depending on the **type of software being maintained**
- System change proposals are the driver for system evolution in all organizations. These change proposals may involve existing requirements that have not been implemented in the released system, requests for new requirements and bug repairs from system stakeholders,
- As illustrated in Figure 2.2 , the processes of change identification and system evolution are cyclical and continue throughout the lifetime of a system.
- 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.

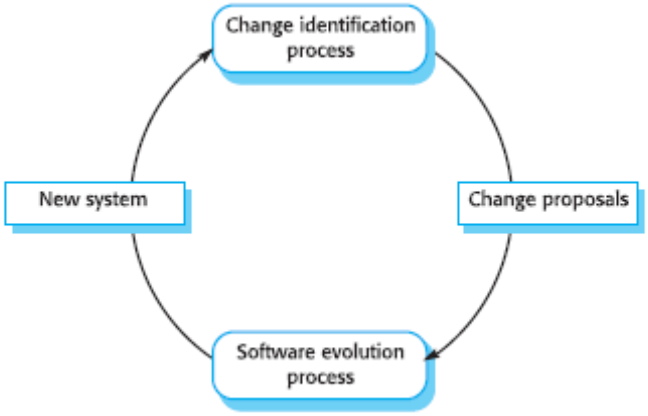

Figure 2.2 Change identification and evolution process

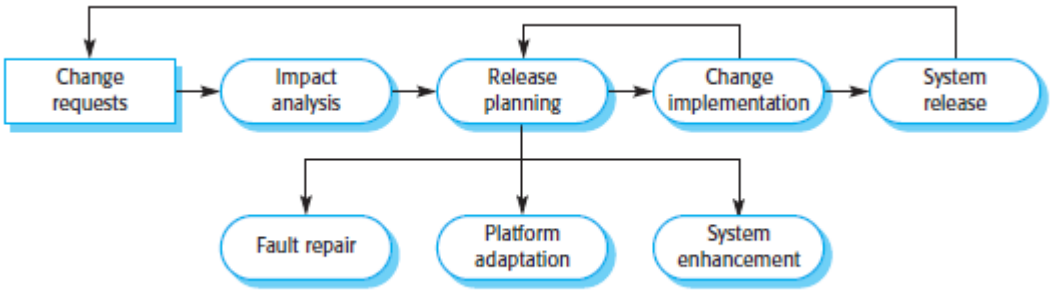

Figure 2.3 The system evolution process

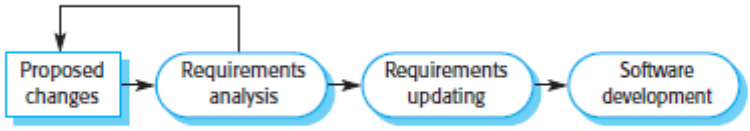

Figure 2.4 change implementation

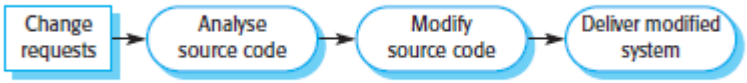

Figure 2.4 The emergency repair process

- 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. The change implementation stage of this process should modify the system specification, design and implementation to reflect the changes to the system (Figure 2.3).
- New requirements that reflect the system changes are proposed, analysed and validated. System components are redesigned and implemented and the system is re-tested As you change software, you develop succeeding releases of the system.

➢ **The 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 system's 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.

- Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem (Figure 2.4).
- When emergency code repairs are made, the change request should remain outstanding after the code faults have been fixed. It can then be re-implemented more carefully after further analysis.

2.2 Program Evolution Dynamics

Program evolution dynamics is the study of system change. The majority of work in this area has been carried out by Lehman and Belady. From these studies, they proposed a set of laws (Lehman's laws) concerning system change. They claim these laws (hypotheses, really) are invariant and widely applicable. Lehman and Belady examined the growth and evolution of a number of large software systems. The proposed laws, shown in Figure 2.5 , were derived from these measurements.

- **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. When the modified system is re-introduced to the environment, this promotes more environmental changes, so the evolution process recycles.
- **The second law** states that, as a system is changed, its structure is degraded. The only way to avoid this happening is to invest in preventative maintenance where you spend time improving the software structure without adding to its functionality. Obviously, this means additional costs, over and above those of implementing required system changes.

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or 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.
Organisational 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 appear to be declining unless they are adapted to changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Figure 2.5 Lehman's Laws

- **The third law** is, perhaps, the most interesting and the most contentious of Lehman's laws. It suggests that large systems have a dynamic of their own that is established at an early stage in the development process. This determines the gross trends of the system maintenance process and limits the number of possible system changes.
- **Lehman's fourth law** suggests that most large programming projects work in what he terms a *saturated* state. That is, a change to resources or staffing has imperceptible 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. Therefore, a large increment in functionality in one system release means that this will have to be followed by a further release where the new system faults are repaired.

- **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.
- The final law reflects the most recent work on feedback processes, although it is not yet clear how this can be applied in practical software development.

2.3 Software Maintenance

Software maintenance is the general process of changing a system after it has been delivered. The term is usually applied to custom software where separate development groups are involved before and after delivery. There are **three different types** of software maintenance:

1. ***Maintenance to repair software faults*** 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 that may be necessary.

2. ***Maintenance to adapt the software to a different operating environment*** 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.

3. ***Maintenance to add to or modify the system's functionality*** This type of maintenance is necessary when the system requirements change in response to organizational or business change. The scale of the changes required to the software is often much greater than for the other types of maintenance.

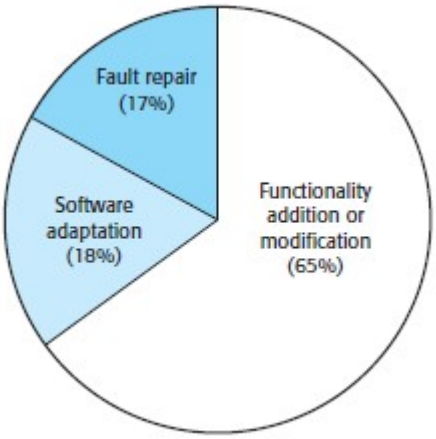

Figure 2.6 Maintenance effort distribution

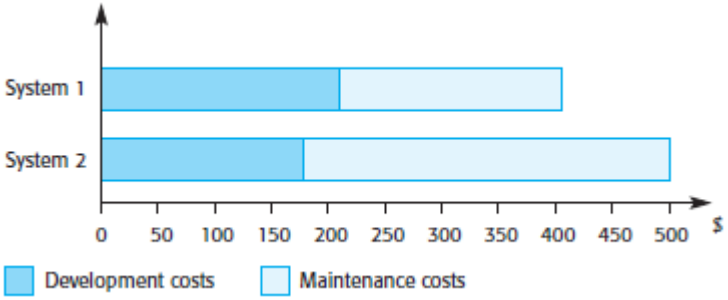

Figure 2.7 Development and maintenance costs

The key factors that distinguish development and maintenance, and which lead to higher maintenance costs, are:

1. ***Team stability*** After a system has been delivered, it is normal for the development team to be broken up and people 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.

2. ***Contractual responsibility*** The contract to maintain a system is usually separate from the system development contract. The maintenance contract may be given to a different company rather than the original system developer. This factor, along with the lack of team stability, means that there is no incentive for a development team to write the software so that it is easy to change.

3. ***Staff skills*** Maintenance staff are often relatively inexperienced and unfamiliar with the application domain. 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

4. ***Program age and structure*** As programs age, their structure tends to be degraded by change, so they become harder to understand and modify. Some systems have been developed without modern software engineering techniques.

2.3.1 Maintenance prediction

Managers hate surprises, especially if these result in unexpectedly high costs. You should therefore try to predict what system changes are likely 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. Figure 2.8 illustrates these predictions and associated questions.

These predictions are obviously closely related:

1. Whether a system change should be accepted depends, to some extent, on the maintainability of the system components affected by that change.
2. Implementing system changes tends to degrade the system structure and hence reduce its maintainability.
3. Maintenance costs depend on the number of changes, and the costs of change implementation depend on the maintainability of system components. Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. To evaluate the relationships between a system and its environment, you should assess:

1. ***The number and complexity of system interfaces*** The larger the number of interfaces and the more complex they are, the more likely it is that demands for change will be made.

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.

process metrics that can be used for assessing maintainability are

1. ***Number of requests for corrective maintenance*** An increase in the number of 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*** This reflects the number of program components that are affected by the change request. If this time increases, it implies that more and more components are affected and maintainability is decreasing.

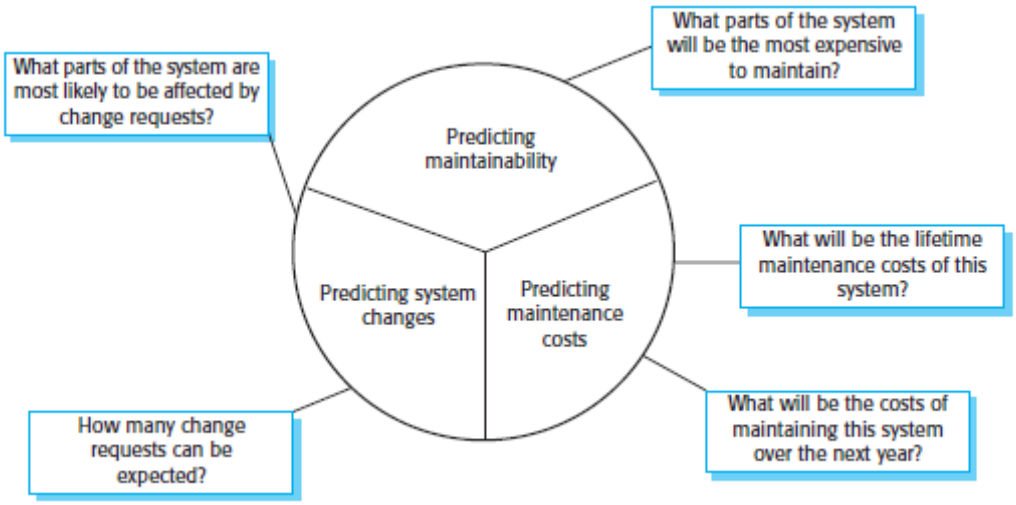

Figure 2.8 Maintenance prediction

3. ***Average time taken to implement a change request*** This is not the same as the time for impact analysis although it may correlate with it. This is the amount of time that you need to actually modify the system and its documentation, after you have assessed which components are affected. 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 System re-engineering

Re-engineering may involve re-documenting the system, organizing and restructuring the system, translating the system to a more modern programming language, and modifying and updating the structure and values of the system's data.

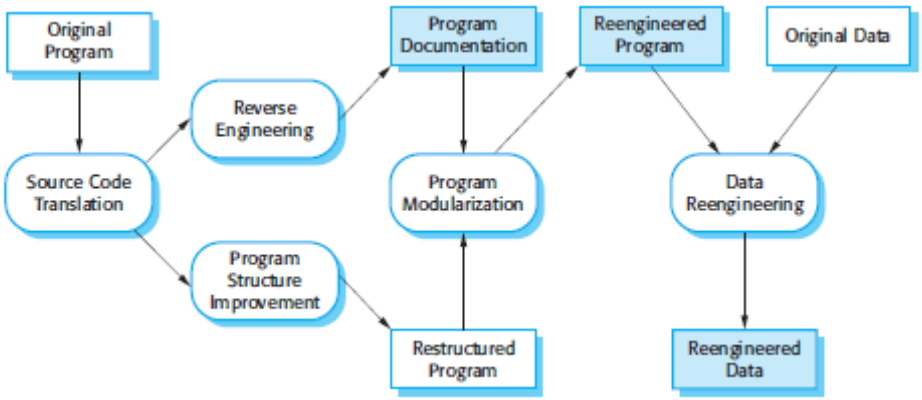

Fig 2.9: Re-engineering

The activities in this re-engineering process are:

1. ***Source code translation*** 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 analysed and information extracted from it. This helps to document its organisation and functionality.

3. ***Program structure improvement*** The control structure of the program is analysed and modified to make it easier to read and understand.

4. ***Program modularisation*** Related parts of the program are grouped together and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural transformation where a centralised system intended for a single computer is modified to run on a distributed platform.

5. ***Data re-engineering*** The data processed by the program is changed to reflect program changes.

Re-engineering advantages are,

1. ***Reduced risk*** There is a high risk in re-developing 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.

2. ***Reduced cost*** The cost of re-engineering is significantly less than the cost of developing new software.

The costs of re-engineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to re-engineering, as shown in Figure 2.10

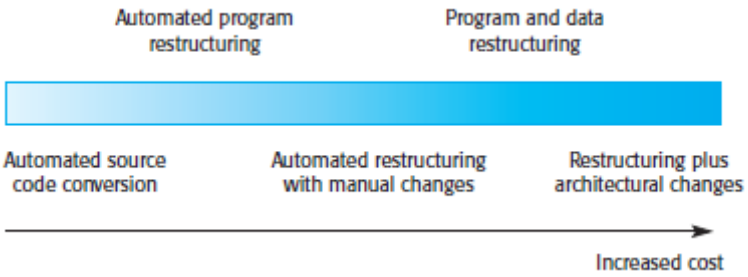

Figure 2.10 Reengineering Approaches

2.4 Legacy system evolution

Organizations that have a limited budget for maintaining and upgrading their legacy systems have to decide how to get the best return on their investment. This means that they have to make a realistic assessment of their legacy systems and then decide what is 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. This occurs when business processes have changed since the system was installed and are no longer completely dependent on the system.

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. ***Re-engineer the system to improve its maintainability*** This option should be chosen when the system quality has been degraded by regular change and where regular change to the system is still required.

4. ***Replace all or part of the system with a new system*** This option should be chosen when other 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.

- **Assessment (judgment) of legacy system.**

1. Relative Business value
2. System quality

From a business perspective, you have to decide whether the business really needs the system. From a technical perspective, you have to assess the quality of the application software and the system's support software and hardware.

From Figure 2.12 , you can see that there **are four clusters of systems**:

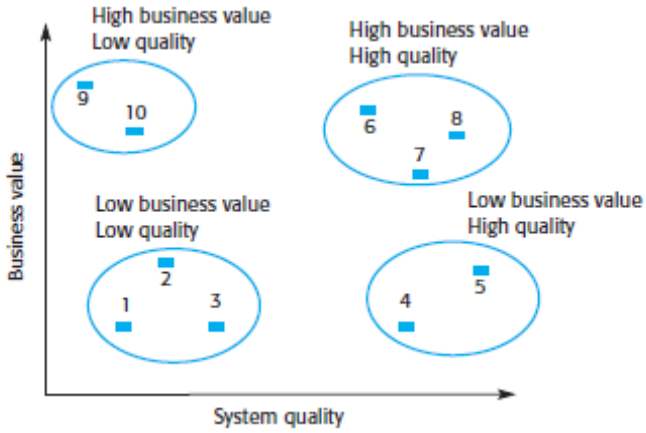

Figure 2.12 Legacy system assessments

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 re-engineered to improve their quality or 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 that may not be very expensive to maintain. It is not worth replacing these systems so normal system maintenance may be continued so long as no expensive changes are required and the system hardware is operational. If expensive changes become necessary, they should be scrapped.

4. ***High quality, high business value*** These systems have to be kept in operation, but their high quality means that you don't have to invest in transformation or system replacement. Normal system maintenance should be continued.

There are **four basic issues for low business value**:

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. A legacy system may have been developed to meet a business need that has either changed or that can now
be met more effectively in other ways.

2. *The business processes that are supported* When a system is introduced, business processes to exploit that system may be designed. However, changing these processes may be impossible because the legacy system can't be adapted. Therefore, a system may have a low business value because new processes can't be introduced.

3. *The system dependability* System dependability is not only a technical problem but also a business problem. 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* The key issue here is the importance of the system outputs to the successful functioning of the business

Factors that you should consider during the environment assessment are shown in Figure 2.13 .

Factor	Questions
Supplier stability	Is the supplier is 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 more modern systems.
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 2.13 Factors used in environment assessment

To assess the technical quality of an application system, you have to assess a range of factors (Figure 2.14 be collected are:

1. *The number of system change requests* System changes tend to 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* This is an important factor in forms-based systems where each form can be considered as a separate user interface. 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 (number of files, size of database, etc.), the more complex the system.

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 only a limited number of people who understand the system?

Figure 2.14 Factors used in application assessment

QUESTION BANK ON MODULE-3

1	Explain lehman's laws	2016,17
2	Explain the following. A. Integration Testing B. Release Testing	2016
3	Write a note on software testing process.	2015
4	List classes of interface errors	2017
5	What is partition testing ? identify equivalence class partitions for automated air condition system having at least four partitions. List also the boundary values for each	2012
6	What is test automation? Explain with a figure the tools that might be included in a testing workbench ?	2011
7	Explain the different types of interfaces between program components	2011
8	Explain software inspection process	2011
9	Explain two goals of software testing	2009
11	List classes of interface errors	
12	What are legacy systems? Explain the components of legacy system with a neat diagram	2016
13	With appropriate block diagram, explain the system evolution process	2015
14	With a appropriate block diagram, explain the system evolution process.	2015,2017
15	Write a short notes on. A.Legacy system B. Software testing process	2015
16	differentiate between black box and white box testing	2010
17	Explain tow distinct goals of a software testing	2009
18	Expalin with illustrations, A.Integration Testing, B Release Testing	2011
19	Explain program evolution dynamics	2012
20	Explain 3 types of testing in detail	
21	Explain evolution process with an example	