3. SOFTWARE TESTING & SOFTWARE EVOLUTION

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.

→ The testing process has two distinct goals:

1. To demonstrate to the developer and the customer that the software meets its requirements. For custom software, this means that there should be at least one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.
2. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption

→ Fig 3.1 helps to explain the differences between validation testing and defect testing.

→ The difference between verification and validation can be mentioned as:

- 'Validation: Are we building the right product?'
- 'Verification: Are we building the product right?'

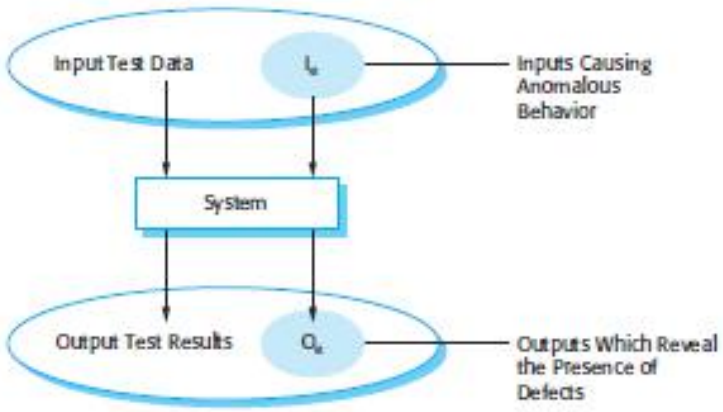

Fig 3.1: An input-output model of program testing

→ The ultimate goal of verification and validation processes is to establish confidence that the software system is 'fit for purpose'.

→ This means that the system must be good enough for its intended use.

→ The level of required confidence depends on the system's purpose, the expectations of the system users, and the current marketing environment for the system:

- **Software purpose:** The more critical the software, the more important that it is reliable. For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.
- **User expectations:** Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails. When a new system is installed, users may tolerate failures because the benefits of use outweigh the costs of failure recovery.
- **Marketing environment:** When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. If a software product is very cheap, users may be willing to tolerate a lower level of reliability.

→ Fig 3.2 shows that software inspections and testing support V & V at different stages in the software process

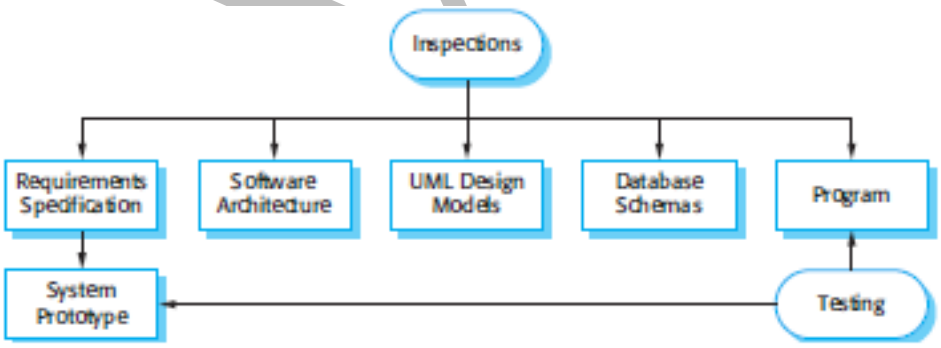

Fig 3.2: Inspections and testing

→ There are three advantages of software inspection over testing:

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, it is not sure that whether the later output anomalies are due to a new error or are side effects of the original error. A single inspection session can discover many errors in a system
2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then it is required to develop specialized test harnesses

to test the parts that are available. This obviously adds to the system development costs.

3. 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

→ Fig 3.3 is an abstract model of the 'traditional' testing process, as used in plan driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested.

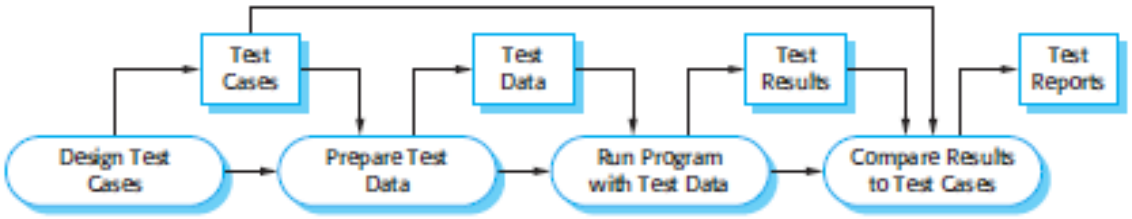

Fig 3.3: A model of the software testing process

→ A commercial software system has to go through three stages of testing:

1. **Development testing**, where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.
2. **Release testing,** where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of system stakeholders.
3. **User testing**, where users or potential users of a system test the system in their own environment. For software products, the 'user' may be an internal marketing group who decide if the software can be marketed, released, and sold. Acceptance testing is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

3.1 Development Testing

→ Development testing includes all testing activities that are carried out by the team developing the system.

→ The tester of the software is usually the programmer who developed that software, although this is not always the case.

→ During development, testing may be carried out at three levels of granularity:

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.

3.1.1 Unit Testing

→ Unit testing is the process of testing program components, such as methods or object classes.

→ Individual functions or methods are the simplest type of component.

→ Your tests should be calls to these routines with different input parameters.

→ The interface of this object is shown in fig 3.4. It has a single attribute, which is its identifier.

→ An automated test has three parts:

- A setup part, where you initialize the system with the test case, namely the inputs and expected outputs
- A call part, where you call the object or method to be tested.
- An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

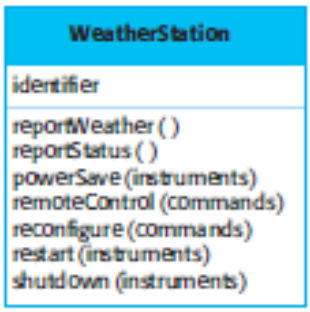

Fig 3.4: The weather station object interface

3.1.2 Choosing unit test cases

→ Testing is expensive and time consuming, so it is important that you choose effective unit test cases. Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that is being tested does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases. 2 test cases includes:
   i. **Partition testing**, where groups of inputs that have common characteristics are identified and should be processed in the same way. Tests must be chosen from within each of these groups.
   ii. **Guideline-based testing,** where testing guidelines are used to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components

→ In fig 3.5, the large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested.

→ The smaller un-shaded ellipses represent equivalence partitions.

→ A program being tested should process all of the members of an input equivalence partitions in the same way.

→ Output equivalence partitions are partitions within which all of the outputs have something in common.

→ The shaded area in the left ellipse represents inputs that are invalid.

→ The shaded area in the right ellipse represents exceptions that may occur

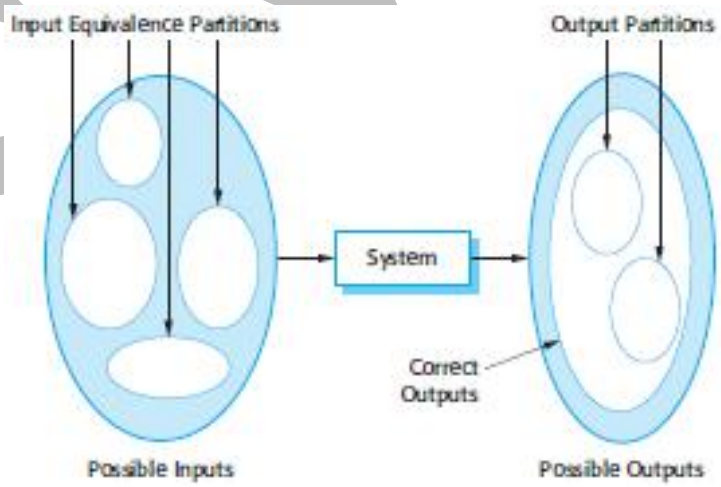

Fig 3.5: Equivalence partitioning

→ Example of equivalence partitioning is as shown in fig 3.6.

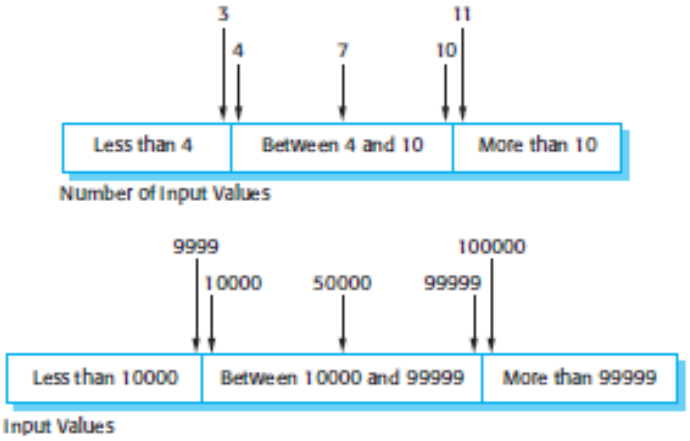

Fig 3.6: Equivalence Partitions

→ For example, say a program specification states that the program accepts 4 to 8 inputs which are five-digit integers greater than 10,000.

→ This information can then be used to identify the input partitions and possible test input values. These are shown in the figure above.

→ Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions.

→ Guidelines that could help reveal defects include:

1. Test software with sequences that have only a single value. Programmers naturally think of sequences as made up of several values and sometimes they embed this assumption in their programs. Consequently, if presented with a single value sequence, a program may not work properly.
2. Use different sequences of different sizes in different tests. This decreases the chances that a program with defects will accidentally produce a correct output because of some accidental characteristics of the input.
3. Derive tests so that the first, middle, and last elements of the sequence are accessed. This approach reveals problems at partition boundaries.

3.1.3 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.

→ Testing composite components should therefore focus on showing that the component interface behaves according to its specification.

→ Fig 3.7 illustrates the idea of component interface testing. Assume that components A, B, and C have been integrated to create a larger component or subsystem.

→ The test cases are not applied to the individual components but rather to the interface of the composite component created by combining these components.

→ Interface errors in the composite component may not be detectable by testing the individual objects because these errors result from interactions between the objects in the component.

→ Different types of interface error that can occur:

- **Parameter interfaces:** These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.
- **Shared memory interfaces**: These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other sub-systems.
- **Procedural interfaces:** These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.
- **Message passing interfaces:** These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service.

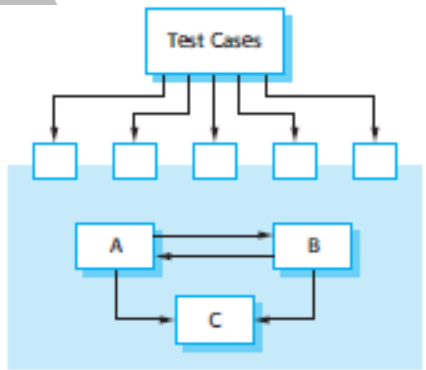

Fig 3.7: Interface testing

→ Interface errors falls into 3 categories:

- **Interface misuse:** A calling component calls some other component and makes an error in the use of its interface. This type of error is common with

parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.

- **Interface misunderstanding:** A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior. The called component does not behave as expected which then causes unexpected behavior in the calling component.
- **Timing errors:** These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds.

→ Some general guidelines for interface testing are:

- Examine the code to be tested and explicitly list each call to an external component. Design a set of tests in which the values of the parameters to the external components are at the extreme ends of their ranges. These extreme values are most likely to reveal interface inconsistencies.
- Where pointers are passed across an interface, always test the interface with null pointer parameters.
- Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.
- Use stress testing in message passing systems. This means that tests should be designed that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems.
- Where several components interact through shared memory, design tests that vary the order in which these components are activated. These tests may reveal implicit assumptions made by the programmer about the order in which the shared data is produced and consumed.

3.1.4 System Testing

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

→ System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces. There are 2 differences:

1. During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
2. Components developed by different team members or groups may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

→ Fig 3.8 shows the sequence of operations in the weather station when it responds to a request to collect data for the mapping system.

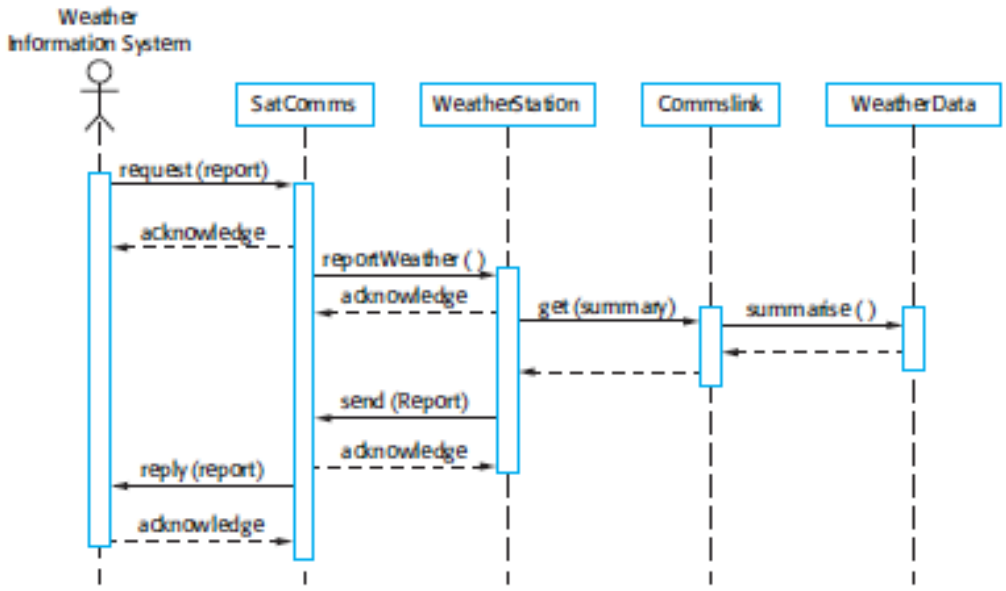

Fig 3.8: Collect weather data sequence chart

3.2 Test Driven Development

→ Test-driven development (TDD) is an approach to program development in which testing and code development are interleaved.

→ Test-driven development was introduced as part of agile methods such as Extreme Programming.

→ The fundamental TDD process is shown in fig 3.9. The steps in the process are as follows:

1. Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.

2. Write a test for this functionality and implement this as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.
3. Run the test, along with all other tests that have been implemented. Initially, the functionality is not implemented, so the new test will fail. This is deliberate as it shows that the test adds something to the test set.
4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what's already there.
5. Once all tests run successfully, then move on to implementing the next chunk of functionality.

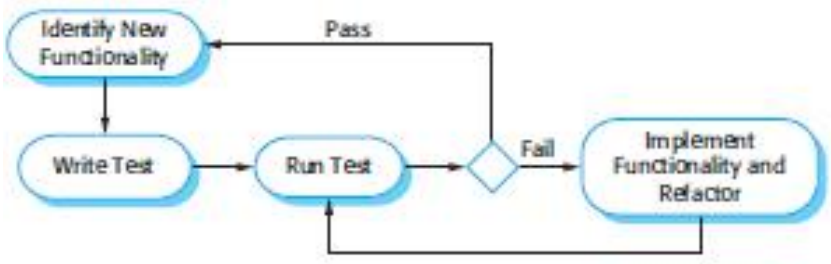

Fig 3.9: Test-driven development

→ Benefits of test-driven development are:

1. **Code coverage:** In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in the system has actually been executed. Code is tested as it is written so defects are discovered early in the development process.
2. **Regression testing**: A test suite is developed incrementally as a program is developed. Regression tests can be run to check that changes to the program have not introduced new bugs.
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. Debugging tools need not be used to locate the problem. Reports of the use of test-driven development suggest that it is hardly ever necessary to use an automated debugger in test-driven development
4. **System documentation**: The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

3.3 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.

→ There are two important distinctions between release testing and system testing during the development process:

1. A separate team that has not been involved in the system development should be responsible for release testing.
2. 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).

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

→ The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs.

3.3.1 Requirements based testing

→ A general principle of good requirements engineering practice is that requirements should be testable; that is, the requirement should be written so that a test can be designed for that requirement.

→ Consider MHC-PMS system (mental health care patient management system)

→ 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.

→ To check if these requirements have been satisfied, it might be necessary to develop several related tests:

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. Prescribe the medication to that the patient is allergic to, and 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 and check that the correct warning for each drug is issued.
4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
5. Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

3.3.2 Scenario testing

→ Scenario testing is an approach to release testing where typical scenarios are devised and are used to develop test cases for the system.

→ A scenario is a story that describes one way in which the system might be used.

→ Scenarios should be realistic and real system users should be able to relate to them.

→ When scenario-based approach is used, normally several requirements within the same scenario are tested.

→ Therefore, as well as checking individual requirements, checking that combinations of requirements do not cause problems.

3.3.3 Performance testing

→ Once a system has been completely integrated, it is possible to test for emergent properties, such as performance and reliability.

→ Performance tests have to be designed to ensure that the system can process its intended load.

→ This usually involves running a series of tests where the load is increased, until the system performance becomes unacceptable.

→ To test whether performance requirements are being achieved, an operational profile may be constructed.

→ An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system.

→ This type of testing has two functions:

1. It tests the failure behavior of the system. Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important

that system failure should not cause data corruption or unexpected loss of user services.

2. It stresses the system and may cause defects to come to light that would not normally be discovered. Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing replicates.

3.4 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.

→ This may involve formally testing a system that has been commissioned from an external supplier, or could be an informal process where users experiment with a new software product to see if they like it and that it does what they need.

→ In practice, there are three different types of user testing:

1. **Alpha testing**, where users of the software work with the development team to test the software at the developer's site.
2. **Beta testing**, where 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**, where 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.

→ There are six stages in the acceptance testing process, as shown in fig 3.10.

→ They are:

1. **Define acceptance criteria:** This stage should, ideally, take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be agreed between the customer and the developer. Detailed requirements may not be available and there may be significant requirements change during the development process.
2. **Plan acceptance testing:** This involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the

requirements and the order in which system features are tested. It should define risks to the testing process, such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

3. **Derive acceptance tests:** Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system.
4. **Run acceptance tests:** The agreed acceptance tests are executed on the system. Ideally, this should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system.
5. **Negotiate test results:** It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be put into use. They must also agree on the developer's response to identified problems.
6. **Reject/accept system:** This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

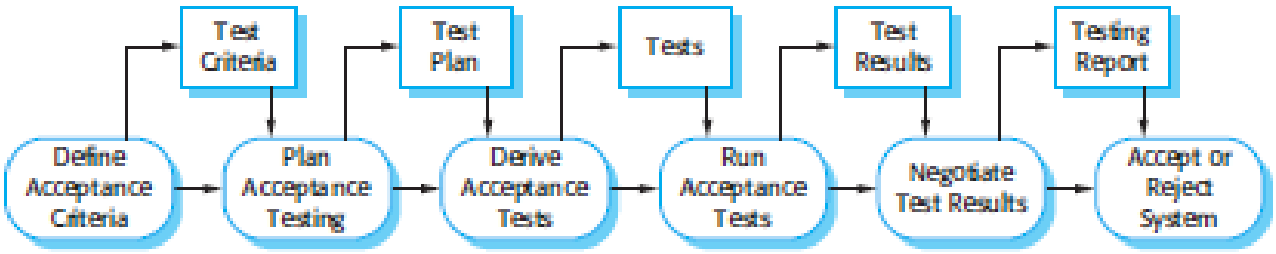

Fig 3.10: The acceptance testing process

3.5 Test Automation

→ Test automation is essential for test-first development.

→ Test-first development is an approach to development where tests are written before the code to be tested

→ Tests are written as executable components before the task is implemented.

→ These testing components should be standalone, should simulate the submission of input to be tested, and should check that the result meets the output specification.

→ An automated test framework is a system that makes it easy to write executable tests and submit a set of tests for execution.

→ Test automation tools such as JUnit that can re-run component tests when new versions of the component are created, can commonly be used

→ JUnit is a set of java classes that the user extends to create an automated testing environment.

→ As testing is automated, there is always a set of tests that can be quickly and easily executed.

→ Whenever any functionality is added to the system, the tests can be run and problems that the new code has introduced can be caught immediately

→ Test-first development and automated testing usually results in a large number of tests being written and executed.

→ This approach does not necessarily lead to thorough program testing. There are three reasons for this:

1. Programmers prefer programming to testing and sometimes they take shortcuts when writing tests.
2. Some tests can be very difficult to write incrementally
3. It becomes difficult to judge the completeness of a set of tests.

→ An automated test has three parts:

1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.
2. A call part, where you call the object or method to be tested.
3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

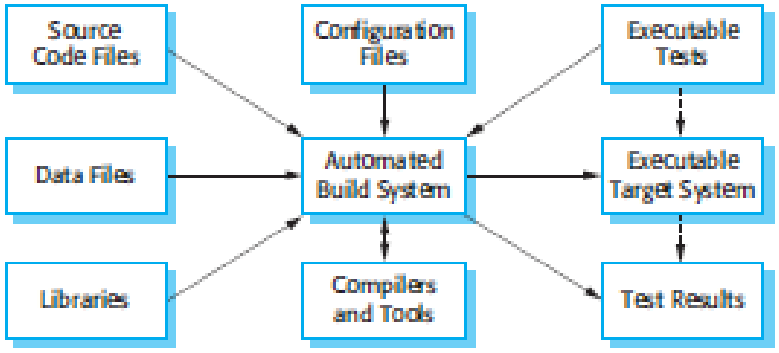

Fig 3.11: System building

→ System building involves assembling a large amount of information about the software and its operating environment.

→ Therefore, for anything apart from very small systems, it always makes sense to use an automated build tool to create a system build as shown in fig 3.11.

SOFTWARE EVOLUTION

→ Software evolution may be triggered by changing business requirements, by reports of software defects, or by changes to other systems in a software system's environment. software engineering as a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system (Figure 3.12).

→ This process can be started by creating release 1 of the system.

→ Once delivered, changes are proposed and the development of release 2 starts almost immediately.

→ This model of software evolution implies that a single organization is responsible for both the initial software development and the evolution of the software

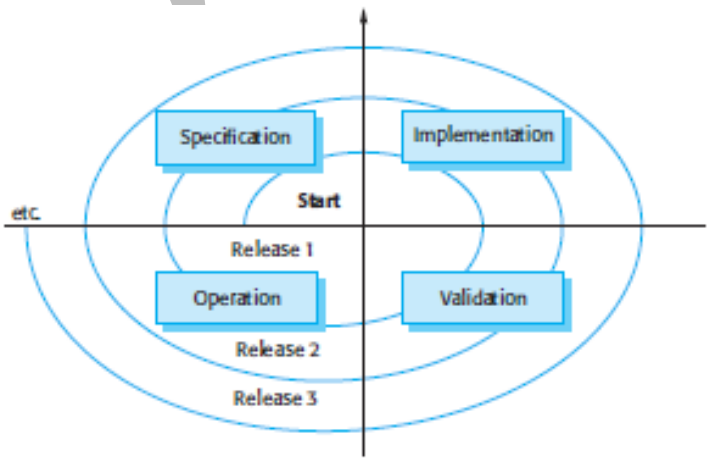

Fig 3.12: A spiral model of development and evolution

→ An alternate view of the software evolution life cycle is as shown in fig 3.13.

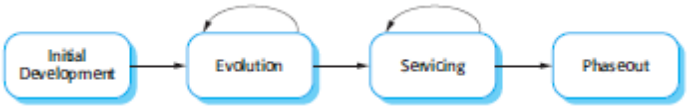

Fig 3.13: Evolution and servicing

3.6 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.

→ In some organizations, evolution may be an informal process where change requests mostly come from conversations between the system users and developers.

→ In other companies, it is a formalized process with structured documentation produced at each stage in the process.

→ The processes of change identification and system evolution are cyclic and continue throughout the lifetime of a system (Fig 3.14).

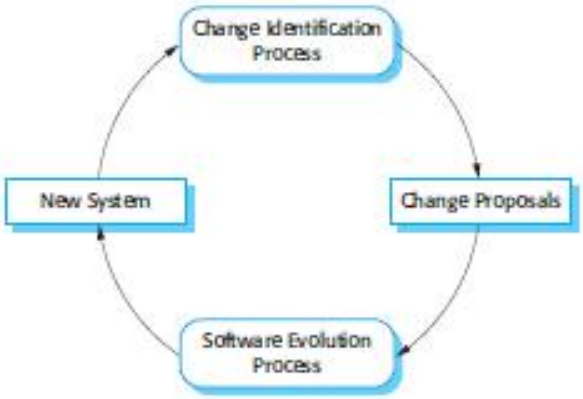

Fig 3.14: Change identification and evolution process

→ Fig 3.15 shows an overview of the evolution process.

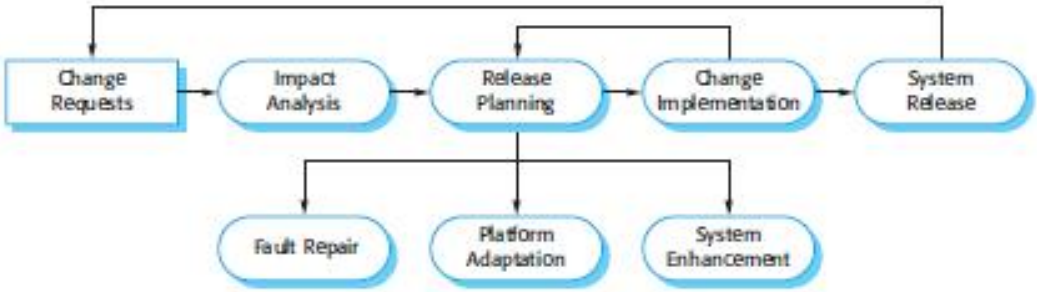

Fig 3.15: The software evolution process

→ The process includes the fundamental activities of change analysis, release planning, system implementation, and releasing a system to customers.

→ The cost and impact of these changes are assessed to see how much of the system is affected by the change and how much it might cost to implement the change.

→ If the proposed changes are accepted, a new release of the system is planned.

→ During release planning, all proposed changes (fault repair, adaptation, and new functionality) are considered.

→ A decision is then made on which changes to implement in the next version of the system.

→ The changes are implemented and validated, and a new version of the system is released.

→ The process then iterates with a new set of changes proposed for the next release.

→ Change implementation can be thought of as an iteration of the development process, where the revisions to the system are designed, implemented, and tested.

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

→ During this program understanding phase, it becomes necessary to understand how the program is structured, how it delivers functionality, and how the proposed change might affect the program.

→ This understanding is required to make sure that the implemented change does not cause new problems when it is introduced into the existing system.

→ The change implementation stage of this process should modify the system specification, design, and implementation to reflect the changes to the system (Fig 3.16).

→ During the evolution process, the requirements are analyzed in detail and implications of the changes emerge that were not apparent in the earlier change analysis process.

→ This means that the proposed changes may be modified and further customer discussions may be required before they are implemented.

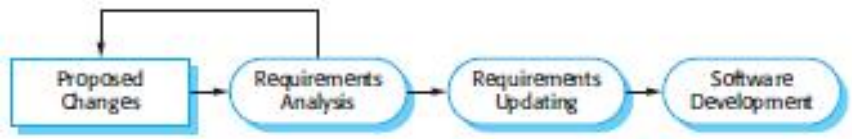

Fig 3.16: Change implementation

→ Change requests can arise for 3 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.

→ Rather than modify the requirements and design, it is possible to make an emergency fix to the program to solve the immediate problem (Fig 3.17).

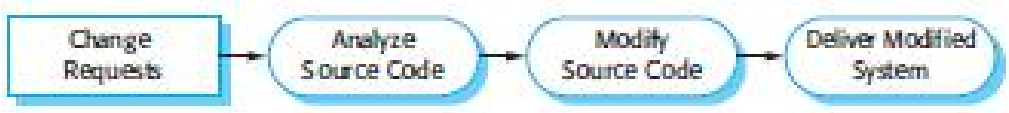

Fig 3.17: The emergency repair process

→ Problems may arise in situations in which there is a handover from a development team to a separate team responsible for evolution.

→ There are two potentially problematic situations:

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

3.7 Program Evolution Dynamics

→ Program evolution dynamics is the study of system change.

→ Lehman and Belady proposed few laws for system change. They claimed these laws are likely to be true for all types of large organizational software systems (what they call Etype systems).

→ These are systems in which the requirements are changing to reflect changing business needs.

→ New releases of the system are essential for the system to provide business value.

→ Fig 3.18 below shows the Lehman's laws.

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

Fig 3.18: Lehman's laws

3.8 Software Maintenance

→ Software maintenance is the general process of changing a system after it has been delivered.

→ 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:** 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. **Functionality addition**: 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.

→ Fig 3.19 shows an approximate distribution of maintenance costs.

→ The specific percentages will obviously vary from one organization to another but, universally; repairing system faults is not the most expensive maintenance activity.

→ Evolving the system to cope with new environments and new or changed requirements consumes most maintenance effort.

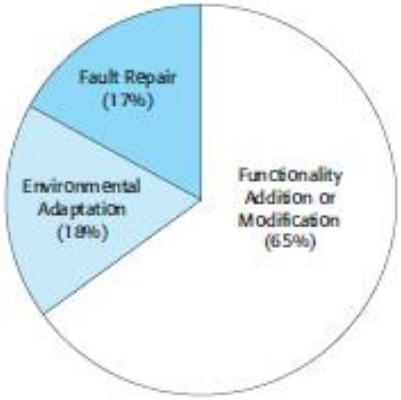

Fig 3.19: Maintenance effort distribution

→ 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:** After a system has been delivered, it is normal for 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. The maintenance contract may be given to a different company rather than the original system developer.
3. **Staff skills:** Maintenance staff are often relatively inexperienced and unfamiliar with the application domain. Maintenance has a poor image among software engineers.
4. **Program age and structure***:* As changes are made to programs, their structure tends to degrade. Consequently, as programs age, they become harder to understand and change.

3.8.1 Maintenance prediction

→ It is essential to estimate the overall maintenance costs for a system in a given time period.

→ Fig 3.20 shows these predictions and associated questions.

→ Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment

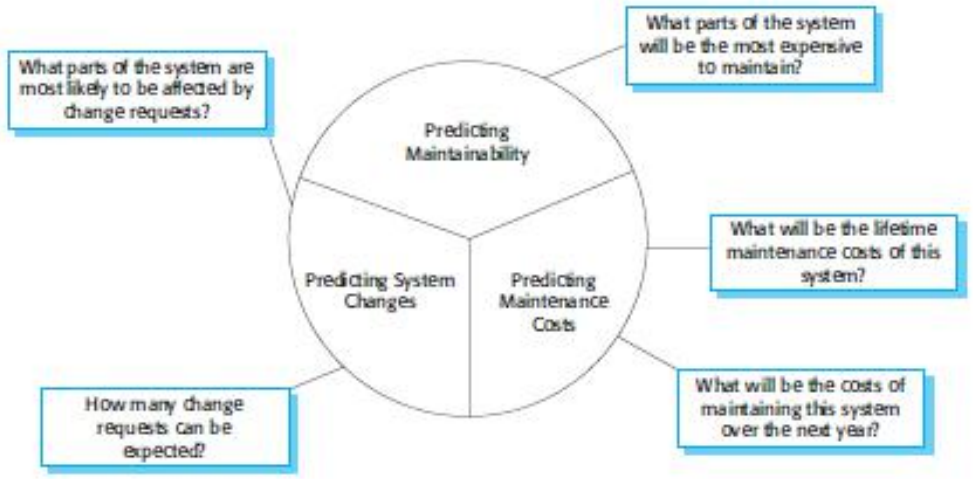

Fig 3.20: Maintenance prediction

→ 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 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:** 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.

→ Examples of 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:** 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 not the same as the time for impact analysis although it may correlate with it. This is the amount of time needed to modify the system and its documentation, after the assessment of which components are affected
4. **Number of outstanding change requests:** An increase in this number over time may imply a decline in maintainability.

3.8.2 Software Engineering

→ To make legacy software systems easier to maintain, these systems can be reengineered to improve their structure and understandability.

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

→ There are two important benefits from reengineering rather than replacement:

1. **Reduced risk:** There is a 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
2. **Reduced cost:** The cost of reengineering may be significantly less than the cost of developing new software.

→ Fig 3.21 is a general model of the reengineering process.

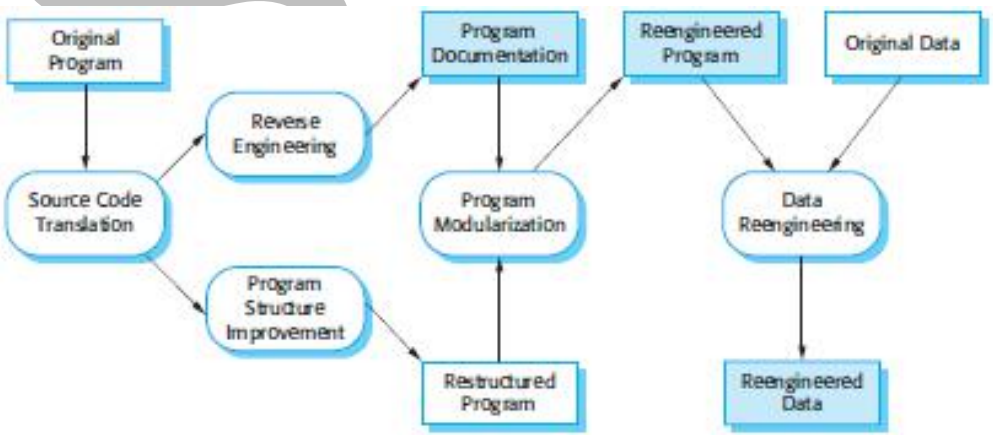

Fig 3.21: The reengineering process

→ 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 this 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. Again, 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 and, where appropriate, redundancy is removed. In some cases, this stage may involve architectural refactoring (e.g., a system that uses several different data stores may be to use a single repository). 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 and converting existing databases to the new structure.

→ The costs of reengineering obviously depend on the extent of the work that is carried out. There is a spectrum of possible approaches to reengineering, as shown in fig 3.22.

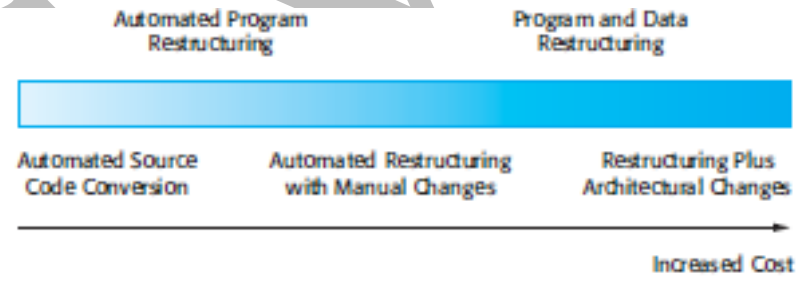

Fig 3.22: Reengineering approaches

3.8.3 Preventative maintenance by refactoring

→ 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 is sometimes considered to be limited to object-oriented development but the principles can be applied to any development approach.

→ Examples of factors that can be improved through refactoring include:

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) reoccurs 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.

3.9 Legacy system management

→ For new software systems developed using modern software engineering processes, such as incremental development and CBSE, it is possible to plan how to integrate system development and evolution.

→ 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 commonly occurs when business processes have changed since the system was installed and are no longer reliant on the legacy 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. **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. This process may include developing new interface components so that the original system can work with other, newer systems.

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.

→ From the fig 3.23, it is observed that there are 4 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.
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.
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.

Fig 3.23: An example of a legacy system assessment

→ Factors used in environment assessment are as shown in fig 3.24.

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?

Fig 3.24: Factors used in environment assessment

→ To assess the technical quality of an application system, you have to assess a range of factors (Fig 3.25) that are primarily related to the system dependability, the difficulties of maintaining the system and the system documentation

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?

Fig 3.25: Factors used in application assessment

→ Data that 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. The higher this accumulated 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 likely that it is that there will be data inconsistencies that reduce the system quality.