MODULE 4

Software Testing

- **Development Testing**
- **Test Driven Development**
- **Release Testing**
- **User 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. You check the results of the test run for errors, anomalies or information about the programs non-functional attributes. **Testing can reveal the presence of errors, but NOT their absence.** Testing is part of a more general verification and validation process, which also includes static validation techniques.

Goals of software testing:

- To **demonstrate** to the developer and the customer that the **software meets its requirements**.
  - Leads to **validation testing**: you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
  - A successful test shows that the system operates as intended.
- To **discover** situations in which the behavior of the software is **incorrect, undesirable or does not conform to its specification**.
  - Leads to **defect testing**: the test cases are designed to expose defects; the test cases can be deliberately obscure and need not reflect how the system is normally used.
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

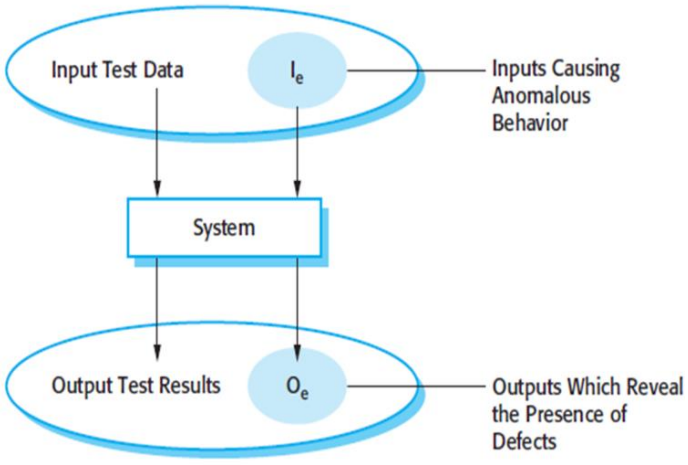

Fig: An input-output model of program testing

Verification and validation

Testing is part of a broader process of software **verification and validation** (V & V).

1. **Verification: Are we building the product right?**
   The software should conform to its specification.
2. **Validation: Are we building the right product?**
   The software should do what the user really requires.

The **goal of V & V** is to establish confidence that the system is **good enough for its intended use**, which depends on:

1. **Software purpose**: the level of confidence depends on how critical the software is to an organization.
2. **User expectations**: users may have low expectations of certain kinds of software.
3. **Marketing environment**: getting a product to market early may be more important than finding defects in the program.

Inspections and Testing

Software inspections 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.

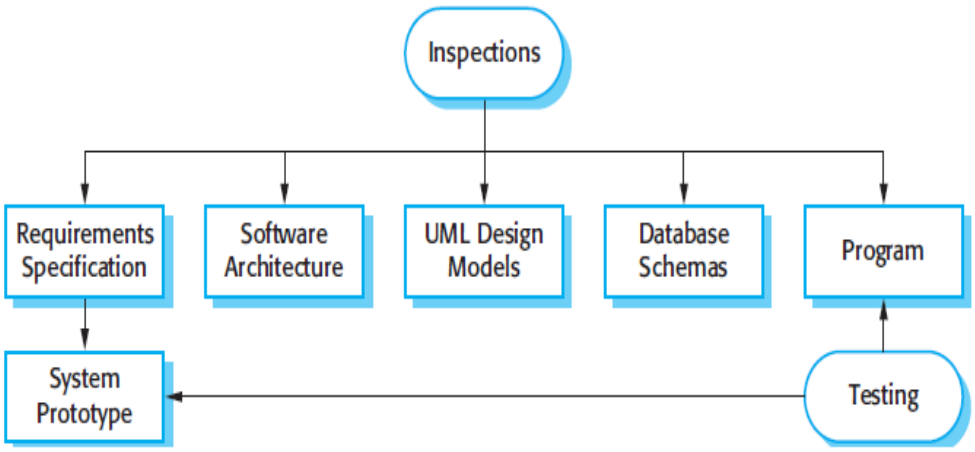

Fig: Inspections and Testing

Advantages of inspections include:

- During testing, errors can mask (hide) other errors. Because inspection is a static process, you don't have to be concerned with **interactions between errors**.

- **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.

Inspections and testing are complementary and not opposing verification techniques. Both should be used during the V & V process. Inspections can check conformance with a specification but not conformance with the customer's real requirements. Inspections cannot check non-functional characteristics such as performance, usability, etc.

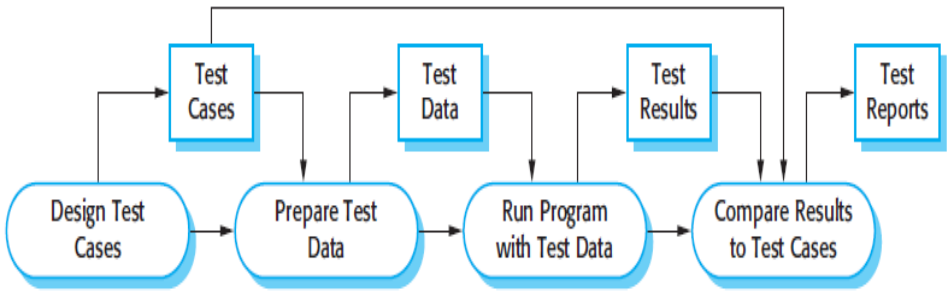

Fig: A model of the software testing process

Typically, a commercial software system has to go through **three stages of testing**:

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

Development testing

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

1. **Unit testing**: individual program units or object classes are tested; should focus on testing the functionality of objects or methods.
2. **Component testing**: several individual units are integrated to create composite components; should focus on testing component interfaces.
3. **System testing**: some or all of the components in a system are integrated and the system is tested as a whole; should focus on testing component interactions.

Unit testing

Unit testing is the process of **testing individual components in isolation**. It is a defect testing process. Units may be:

- **Individual functions** or methods within an object;
- **Object classes** with several attributes and methods;
- **Composite components** with defined interfaces used to access their functionality.

When **testing object classes**, tests should be designed to provide coverage of all of the features of the object:

- Test **all operations** associated with the object;
- Set and check the **value of all attributes** associated with the object;
- Put the object into **all possible states**, i.e. simulate all events that cause a state change.

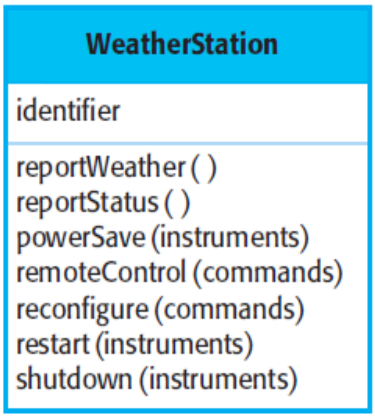

Fig: The weather station object interface

- Generalization or inheritance makes object class testing more complicated. You can't simply test an operation in the class where it is defined and assume that it will work as expected in the subclasses that inherit the operation.
- The operation that is inherited may make assumptions about other operations and attributes. These may not be valid in some subclasses that inherit the operation.
- You therefore have to test the inherited operation in all of the contexts where it is used
- To test the states, we can use state model
- Using this model, you can identify sequences of state transitions that have to be tested and define event sequences to force these transitions
- In principle, you should test every possible state transition sequence, although in practice this may be too expensive.

Examples of state sequences that should be tested in the weather station include:

Shutdown → Running → Shutdown

Configuring → Running → Testing → Transmitting → Running

Running→ Collecting → Running → Summarizing → Transmitting → Running

Whenever possible, **unit testing should be automated** so that tests are run and checked without manual intervention. In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests. Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the 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.

Choosing unit test cases

- Testing is expensive and time consuming, so it is important that you choose effective unit test cases
- The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
- If there are defects in the component, these should be revealed by test cases
- You should therefore write two kinds of test case.
- The first of these should reflect normal operation of a program and should show that the component works

The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are

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

- The input data and output results of a program often fall into a number of different classes with common characteristics.
- Examples of these classes are positive numbers, negative numbers, and menu selections.

- Programs normally behave in a comparable way for all members of a class.
- That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers
- Because of this equivalent behaviour, these classes are sometimes called equivalence partitions or domains
- One systematic approach to test case design is based on identifying all input and output partitions for a system or component.
- Test cases are designed so that the inputs or outputs lie within these partitions.
- Partition testing can be used to design test cases for both systems and components

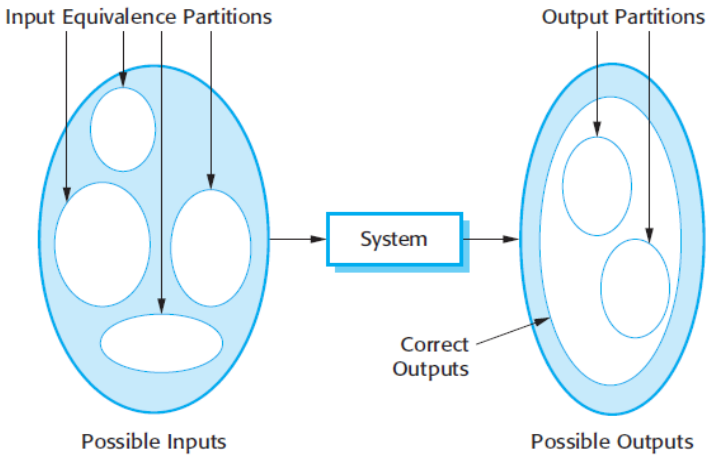

Fig: Equivalence Partitioning

- The large shaded ellipse on the left represents the set of all possible inputs to the program that is being tested.
- The smaller unshaded 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.
- Sometimes there is a 1:1 mapping between input and output equivalence partitions
- The shaded area in the left ellipse represents inputs that are invalid.

- The shaded area in the right ellipse represents exceptions that may occur (i.e., responses to invalid inputs).
- A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition
- The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system.
- You test these by choosing the midpoint of the partition.
- Boundary values are often atypical (e.g., zero may behave differently from other non-negative numbers) so are sometimes overlooked by developers. Program failures often occur when processing these atypical values
- You identify partitions by using the program specification or user documentation and from experience where you predict the classes of input value that are likely to detect errors.
- For example, say a program specification states that the program accepts 4 to 8 inputs which are five-digit integers greater than 10,000.
- You use this information to identify the input partitions and possible test input values.

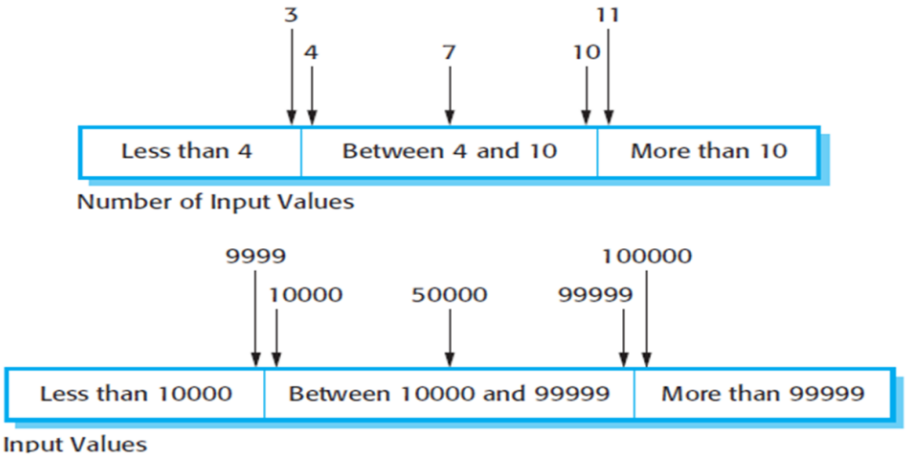

Fig: Equivalence Partitions

- When you use the specification of a system to identify equivalence partitions, this is called 'black-box testing'. Here, you don't need any knowledge of how the system works.
- However, it may be helpful to supplement the black-box tests with 'white-box testing', where you look at the code of the program to find other possible tests.

- For example, your code may include exceptions to handle incorrect inputs. You can use this knowledge to identify 'exception partitions'—different ranges where the same exception handling should be applied.
- You can also use testing guidelines to help choose test cases.
- Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors.
- For example, when you are testing programs with sequences, arrays, or lists, guidelines that could help reveal defects include:
- Test software with sequences that have only a single value
- Use different sequences of different sizes in different tests
- Derive tests so that the first, middle, and last elements of the sequence are accessed. This approach is reveals problems at partition boundaries
- Some of the most general guidelines that Whittaker suggests are:
- Choose inputs that force the system to generate all error messages;
- Design inputs that cause input buffers to overflow;
- Repeat the same input or series of inputs numerous times;
- Force invalid outputs to be generated;
- Force computation results to be too large or too small

Component Testing

Software components are often composite components that are **made up of several interacting objects**. 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. Objectives are to detect faults due to interface errors or invalid assumptions about interfaces. Interface types include:

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.

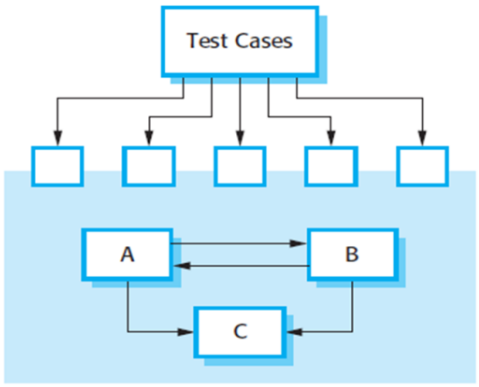

Fig: Interface Testing

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 behavior 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.

General guidelines for interface testing:

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

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.
- 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. Components developed by different team members or sub-teams may be integrated at this stage. System testing is a collective rather than an individual process.
- 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 **document the components and their interactions** that are being tested.

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. This is a differentiating feature of TDD versus writing unit tests after the code is written: it makes the developer focus on the requirements before writing the code.
- **The code is developed 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.
- The **goal of TDD** isn't to ensure we write tests by writing them first, but to **produce working software that achieves a targeted set of requirements using simple, maintainable solutions**. To achieve this goal, TDD provides strategies for **keeping code working, simple, relevant, and free of duplication**.

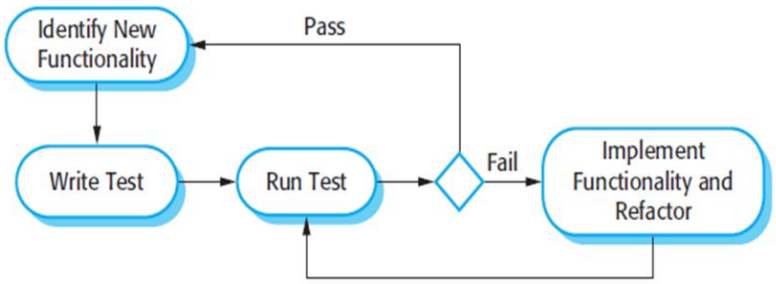

Fig: Test-Driven Development

TDD process includes the following activities:

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.
3. **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.
4. **Implement the functionality and re-run the test**.
5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of test-driven development:

- **Code coverage**: every code segment that you write has at least one associated test so all code written has at least one test.
- **Regression testing**: a regression test suite is developed incrementally as a program is developed.

- **Simplified debugging**: when a test fails, it should be obvious where the problem lies; the newly written code needs to be checked and modified.
- **System documentation**: the tests themselves are a form of documentation that describes what the code should be doing.

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 customer 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 is a form of system testing. Important differences:

- 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 involves examining each requirement and developing a test or tests for it. It is validation rather than defect testing: you are trying to demonstrate that the system has properly implemented its requirements.

Example:

- 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, you may need 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.

Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system. Scenarios should be realistic and real system users should be able to relate to them. If you have used scenarios as part of the requirements engineering process, then you may be able to reuse these as testing scenarios.

Kate is a nurse who specializes in mental health care. One of her responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, Kate logs into the MHC-PMS and uses it to print her schedule of home visits for that day, along with summary information about the patients to be visited. She requests that the records for these patients be downloaded to her laptop. She is prompted for her key phrase to encrypt the records on the laptop.

One of the patients that she visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. Kate looks up Jim's record and is prompted for her key phrase to decrypt the record. She checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect so she notes the problem in Jim's record and suggests that he visits the clinic to have his medication changed. He agrees so Kate enters a prompt to call him when she gets back to the clinic to make an appointment with a physician. She ends the consultation and the system re-encrypts Jim's record.

After, finishing her consultations, Kate returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for Kate of those patients who she has to contact for follow-up information and make clinic appointments.

Fig: A usage Scenario for the MHC-PMS

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.
7. The system for call prompting

- If you are a release tester, you run through this scenario, playing the role of Kate and observing how the system behaves in response to different inputs.
- As 'Kate', you may make deliberate mistakes, such as inputting the wrong key phrase to decode records.
- This checks the response of the system to errors

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 you increase the load until the system performance becomes unacceptable
- To test whether performance requirements are being achieved, you may have to construct an operational profile.
- An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system.
- Therefore, if 90% of the transactions in a system are of type A; 5% of type B; and the remainder of types C, D, and E, then you have to design the operational profile so that the vast majority of tests are of type A
- Experience has shown that an effective way to discover defects is to design tests around the limits of the system.
- In performance testing, this means stressing the system by making demands that are outside the design limits of the software. This is known as 'stress testing'
- For example, say you are testing a transaction processing system that is designed to process up to 300 transactions per second. You start by testing this system with fewer than 300 transactions per second. You then gradually increase the load on the system beyond 300 transactions per second until it is well beyond the maximum design load of the system and the system fails

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. Types of user testing include:

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.

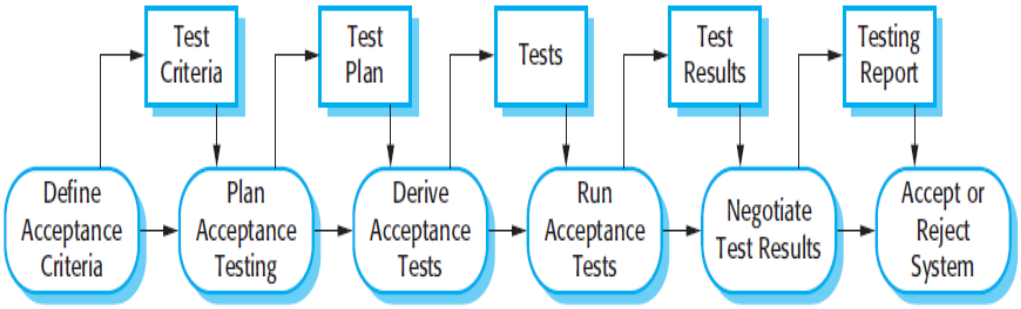

Fig: The Acceptance Testing Process

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.

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

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. They should, ideally, provide complete coverage of the system requirements

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

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.

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

****

Software Evolution

- **Evolution Process**
- **Program Evolution Dynamics**
- **Software Maintenance**
- **Legacy System Management**

****

There are many reasons why **software change is inevitable**:

- New requirements emerge when the software is used;
- The business environment changes;
- Errors must be repaired;
- New computers and equipment is added to the system;
- The performance or reliability of the system may have to be improved.

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

Organizations have **huge investments in their software systems** - they are critical business assets. To maintain the value of these assets to the business, they must be changed and updated. The majority of the software budget in large companies is devoted to changing and evolving existing software rather than developing new software. A **spiral model** of development and evolution represents how a **software system evolves through a sequence of multiple releases**.

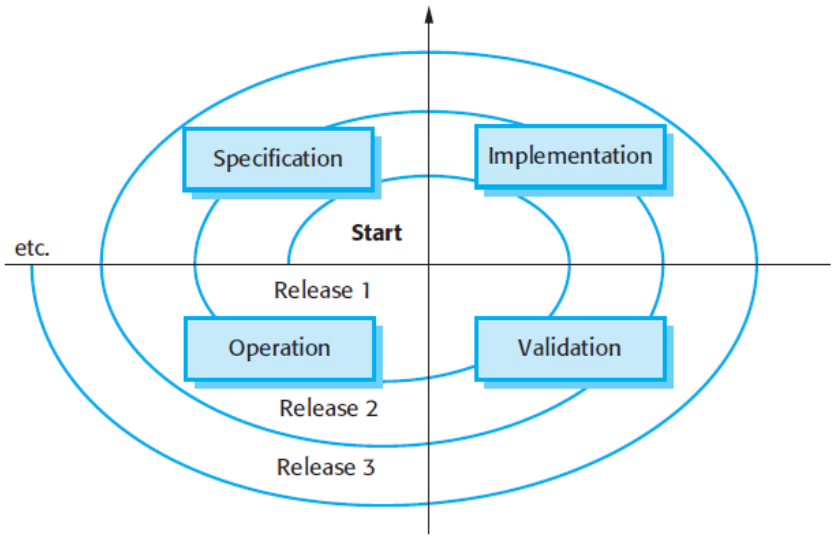

Fig: A spiral model of development and evolution

- Software evolution is important because organizations have invested large amounts of money in their software and are now completely dependent on these systems.

- Their systems are critical business assets and they have to invest in system change to maintain the value of these assets.
- Consequently, most large companies spend more on maintaining existing systems than on new systems development
- Useful software systems often have a very long life time
- Software cost a lot of money so a company has to use a software system for many years to get a return on its investment.
- Obviously, the requirements of the installed systems change as the business and its environment change.
- Therefore, new releases of the systems, incorporating changes, and updates, are usually created at regular intervals
- You should, therefore, think of software engineering as a spiral process with requirements, design, implementation, and testing going on throughout the lifetime of the system
- You start by creating release 1 of the system. Once delivered, changes are proposed and the development of release 2 starts almost immediately.
- In fact, the need for evolution may become obvious even before the system is deployed so that later releases of the software may be under development before the current version has been released.

Evolution and servicing

- Rajlich and Bennett proposed an alternative view of the software evolution life cycle
- They distinguish between evolution and servicing.
- Evolution is the phase in which significant changes to the software architecture and functionality may be made.
- During servicing, the only changes that are made are relatively small, essential changes

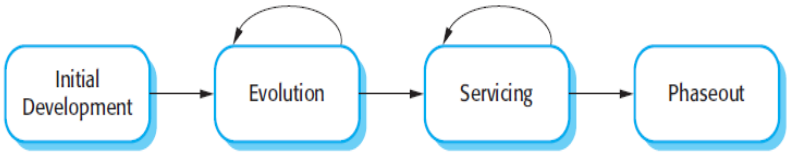

Fig: Evolution and Servicing

- During evolution, the software is used successfully and there is a constant stream of proposed requirements changes.
- However, as the software is modified, its structure tends to degrade and changes become more and more expensive.
- This often happens after a few years of use when other environmental changes, such as hardware and operating systems, are also often required.
- At some stage in the life cycle, the software reaches a transition point where significant changes, implementing new requirements, become less and less cost effective
- At that stage, the software moves from evolution to servicing.
- During the servicing phase, the software is still useful and used but only small tactical changes are made to it.
- During this stage, the company is usually considering how the software can be replaced.
- In the final stage, phase-out, the software may still be used but no further changes are being implemented.
- Users have to work around any problems that they discover

Evolution processes

Software evolution processes depend on:

- The **type of software** being maintained;
- The **development processes** used;
- The **skills and experience** of the people involved.

Proposals for change are the driver for system evolution. These should be linked with components that are affected by the change, thus allowing the cost and impact of the change to be estimated. Change identification and evolution continues throughout the system lifetime.

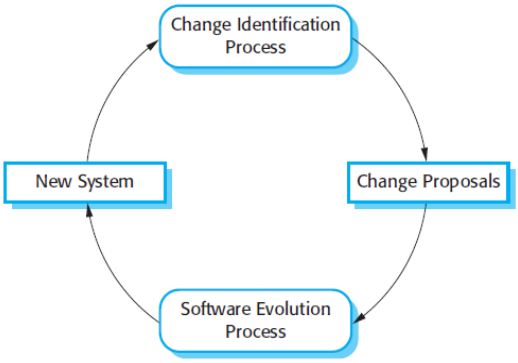

Fig: change identification and evolution processes

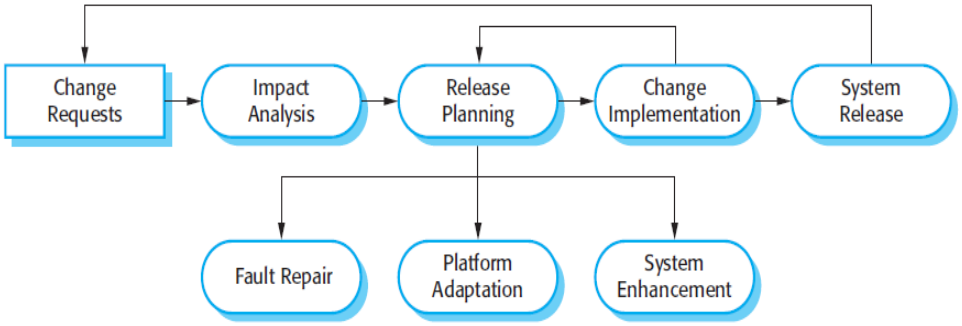

Fig: the software evolution process

- 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
- Ideally, the change implementation stage of this process should modify the system specification, design, and implementation to reflect the changes to the system
- New requirements that reflect the system changes are proposed, analyzed, and validated.

- System components are redesigned and implemented and the system is retested.
- If appropriate, prototyping of the proposed changes may be carried out as part of the change analysis process

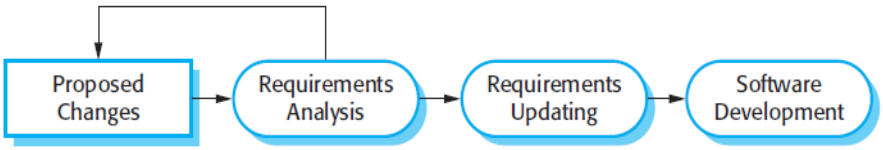

Fig: change implementation

- Change requests sometimes relate to system problems that have to be tackled urgently.
- These urgent changes can arise for three reasons:

1. If a serious system fault occurs that has to be repaired to allow normal operation to continue.
2. If changes to the systems operating environment have unexpected effects that disrupt normal operation
3. If there are unanticipated changes to the business running the system, such as the emergence of new competitors or the introduction of new legislation that affects the system
4. In these cases, the need to make the change quickly means that you may not be able to follow the formal change analysis process.
5. Rather than modify the requirements and design, you make an emergency fix to the program to solve the immediate problem
6. However, the danger is that the requirements, the software design, and the code become inconsistent

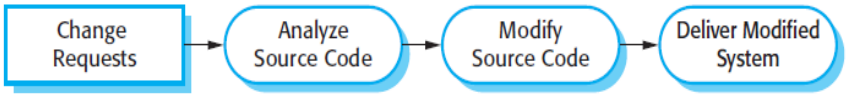

Fig: The Emergency Repair Process

- Emergency system repairs usually have to be completed as quickly as possible.
- You chose a quick and workable solution rather than the best solution as far as system structure is concerned.

- This accelerates the process of software ageing so that future changes become progressively more difficult and maintenance costs increase

Program Evolution Dynamics

- Program evolution dynamics is the study of system change.
- Lehman and Belady carried out several empirical studies of system change with a view to understanding more about characteristics of software evolution
- Lehman and Belady claim these laws are likely to be true for all types of large organizational software systems (what they call E-type 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

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 multi agent, multi loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

Fig: Lehman's laws

- 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 reintroduced to the environment, this promotes more environmental changes, so the evolution process starts again
- 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.
- 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
- The **third law** 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 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 in which the new system faults are repaired.
- Relatively little new functionality should be included in this release. This law suggests that you should not budget for large functionality increments in each release without taking into account the need for fault repair.
- 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

Software Maintenance

Software maintenance focuses on **modifying a program after it has been put into use**. The term is mostly used for changing custom software. Generic software products are said to evolve to create new versions. Maintenance does not normally involve major changes to the system's architecture. Changes are implemented by modifying existing components and adding new components to the system.

Types of software maintenance include:

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

- In practice, there is not a clear-cut distinction between these types of maintenance.
- When you adapt the system to a new environment, you may add functionality to take advantage of new environmental features.
- Software faults are often exposed because users use the system in unanticipated ways.
- Changing the system to accommodate their way of working is the best way to fix these faults
- Different people sometimes give them different names
- **'Corrective maintenance'** is universally used to refer to maintenance for fault repair.
- **'Adaptive maintenance'** sometimes means adapting to a new environment and sometimes means adapting the software to new requirements.
- '**Perfective maintenance'** sometimes means perfecting the software by implementing new requirements; in other cases it means maintaining the functionality of the system but improving its structure and its performance
- The surveys broadly agree that software maintenance takes up a higher proportion of IT budgets than new development (roughly two-thirds maintenance, one-third development).
- They also agree that more of the maintenance budget is spent on implementing new requirements than on fixing bugs

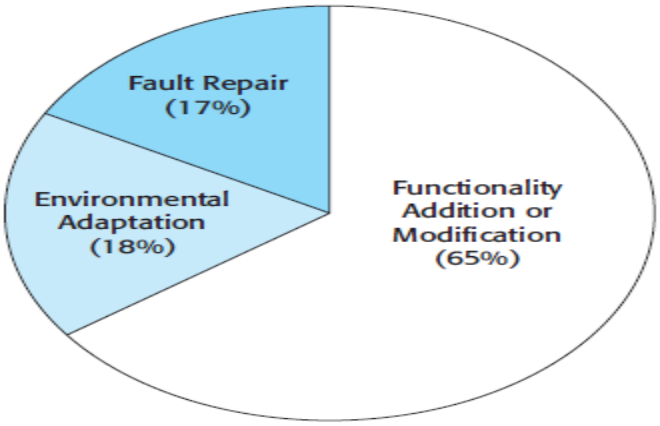

Fig: Maintenance Effort Distribution

Maintenance costs are usually greater than development costs (2x to 100x depending on the application). Costs are affected by both technical and non-technical factors; they tend to increase as software is maintained.
Maintenance corrupts the software structure making further maintenance more difficult. Aging software can have high support costs (e.g. old languages, compilers etc.).

Maintenance cost factors include:

1. **Team stability**: maintenance costs are reduced if the same staff are involved with them for some time.
2. **Contractual responsibility**: the developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
3. **Staff skills**: maintenance staff are often inexperienced and have limited domain knowledge.
4. **Program age and structure**: as programs age, their structure is degraded and they become harder to understand and change.

Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs. Predicting the number of changes requires and understanding of the relationships between a system and its environment. Tightly coupled systems require changes whenever the environment is changed. Factors influencing this relationship are:

- Number and complexity of **system interfaces**; 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
- 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.

- The **business processes** where 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

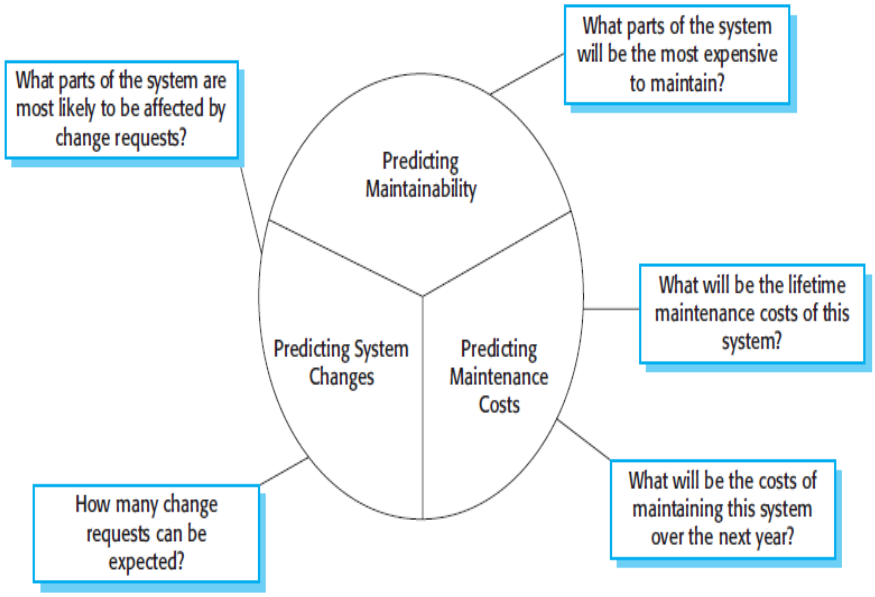

Fig: Maintenance Prediction

- After a system has been put into service, you may be able to use process data to help predict maintainability.
- 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 the amount of time that you need to 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

System reengineering refers to restructuring or rewriting part or all of a legacy system **without changing its functionality**. It is applicable where some but not all sub-systems of a larger system require frequent maintenance. Reengineering involves adding effort to make them easier

to maintain. The system may be restructured and re-documented. **Advantages** of reengineering include:

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

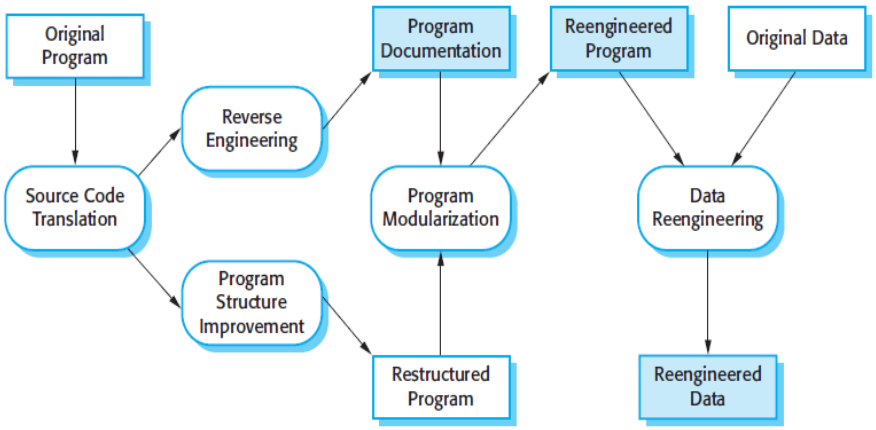

Fig: The reengineering process

Reengineering process **activities** include:

- **Source code translation**: convert code to a new language;
- **Reverse engineering**: analyze the program to understand it;
- **Program structure improvement**: restructure automatically for understandability;
- **Program modularization**: reorganize the program structure;
- **Data reengineering**: clean-up and restructure system data.

Refactoring is the process of making improvements to a program to **slow down degradation through change**. You can think of refactoring as **'preventative maintenance'** that reduces the problems of future change. Refactoring involves modifying a program to improve its structure, reduce its complexity or make it easier to understand. When you refactor a program, you should not add functionality but rather concentrate on program improvement. Reengineering takes place after a system has been maintained for some time and maintenance costs are increasing. You use automated tools to process and reengineer a legacy system to create a new system that is more maintainable. Refactoring is a **continuous process of improvement** throughout the development and evolution process. It is intended to avoid the structure and code degradation that increases the costs and difficulties of maintaining a system.

'Bad smells' of code are stereotypical situations in which the code of a program can be improved through refactoring:

- **Duplicate code** or very similar code may be included at different places in a program; it can be removed and implemented as a single method or function that is called as required.
- **Long methods** should be redesigned as a number of shorter methods.
- **Switch (case) statements** often involve duplication, where the switch depends on the type of a value or the switch statements may be scattered around a program. In object-oriented languages, you can often use polymorphism to achieve the same thing.
- **Data clumping** occurs when the same group of data items (fields in classes, parameters in methods) re-occur in several places in a program. These can often be replaced with an object that encapsulates all of the data.
- **Speculative generality** occurs when developers include generality in a program in case it is required in the future. This can often simply be removed.

Legacy system management

- There are still many legacy systems that are critical business systems.
- These have to be extended and adapted to changing e-business practices
- Most organizations usually have a portfolio of legacy systems that they use, with a limited budget for maintaining and upgrading these systems.
- They have to decide how to get the best return on their investment.
- This involves making a realistic assessment of their legacy systems and then deciding on the most appropriate strategy for evolving these systems
- There are four strategic options:

1. ***Scrap the system completely: This*** *option should be chosen when the system is* not making an effective contribution to business processes. 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
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

Organizations that rely on legacy systems must choose a strategy for evolving these systems. The chosen strategy should depend on the system quality and its business value:

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

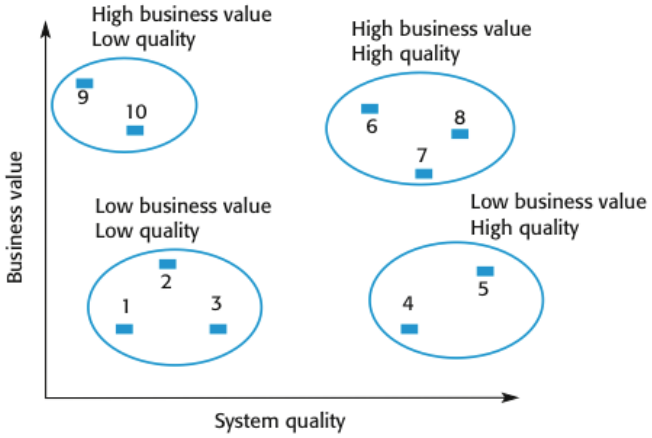

Fig: An example of a legacy system assessment

- To assess the business value of a system, you have to identify system stakeholders, such as end-users of the system and their managers, and ask a series of questions about the system.
- There are four basic issues that you have to discuss:
- ***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.
- You have to be careful, however, about occasional but important use of systems.
- For example, in a university, a student registration system may only be used at the beginning of each academic year. However, it is an essential system with a high business value
- ***The business processes that are supported***
- *When a system is introduced, business* processes are designed to exploit the system's capabilities. If the system is inflexible, changing these business processes may be impossible
- ***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
- ***The system outputs***
- *The key issue here is the importance of the system outputs to* the successful functioning of the business.
- If the business depends on these outputs, then the system has a high business value.
- Conversely, if these outputs can be easily generated in some other way or if the system produces outputs that are rarely used, then its business value may be low

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: factors used in environment assessment

Factor	Questions
Understandability	How difficult is it to understand the source code of the current system? How complex are the control structures that are used? Do variables have meaningful names that reflect their function?
Documentation	What system documentation is available? Is the documentation complete, consistent, and current?
Data	Is there an explicit data model for the system? To what extent is data duplicated across files? Is the data used by the system up-to-date and consistent?
Performance	Is the performance of the application adequate? Do performance problems have a significant effect on system users?
Programming language	Are modern compilers available for the programming language used to develop the system? Is the programming language still used for new system development?
Configuration management	Are all versions of all parts of the system managed by a configuration management system? Is there an explicit description of the versions of components that are used in the current system?
Test data	Does test data for the system exist? Is there a record of regression tests carried out when new features have been added to the system?
Personnel skills	Are there people available who have the skills to maintain the application? Are there people available who have experience with the system?

Fig: Factors used in application assessment