MODULE 2

Rapid Software Development

Objectives:

- Understand how an iterative, incremental software development approach leads to faster delivery of more useful software.
- Understand the differences between agile development methods and software development methods that rely on documented specifications and designs.
- Know the principles, practices and some of the limitations of extreme programming.
- Understand how prototyping can be used to help resolve requirements and design uncertainties when a specification-based.
- Rapid software development processes are designed to produce useful software quickly. Generally, they are iterative processes where specification, design, development and testing are interleaved. Although there are many approaches to rapid software development, they share some fundamental characteristics:
    - The processes of specification, design and implementation are concurrent. There is no detailed system specification, and design documentation is minimized or generated automatically by the programming environment used to implement the system.
    - The system is developed in a series of increments. End-users and other system stakeholders are involved in specifying and evaluating each increment.

- System user interfaces are often developed using an interactive development system that allows the interface design to be quickly created by drawing and placing icon; on the interface.

- **The two main advantages to adopting an incremental approach to software development are:**
  - Accelerated delivery of customer services: early increments of the system can deliver high-priority functionality so that customers can get value from the system early in its development.
  - User engagement with the system Users of the system have to be involved in the incremental development process because they have to provide feedback to the development team on delivered increments. Their involvement does not just mean that the system is more likely to meet their requirements; it also means that the system end-users have made a commitment to it and are likely to want to make it work.
- A general process model for incremental development is illustrated in Figure Notice that the early stages of this process focus on architectural design.

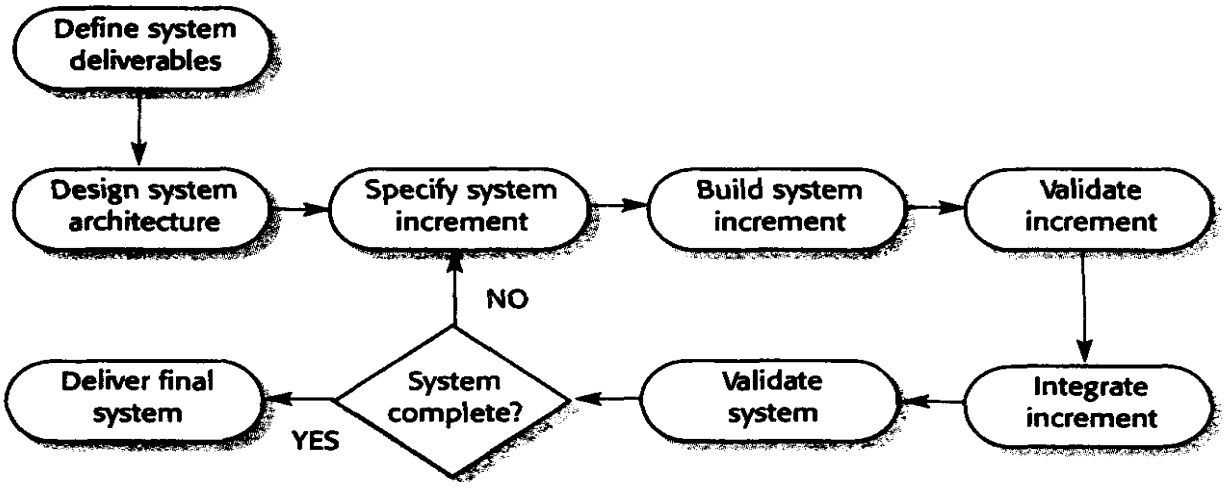


Figure An iterative development process

- However, there can be real difficulties with this approach, particularly in large companies with fairly rigid procedures and in organizations where software development is usually outsourced to an external contractor.
- **The major difficulties with iterative development and incremental delivery are:**
  - **Management problems** Software management structures for large systems are set up to deal with a software process model that generates regular deliverables to assess progress. Furthermore, incremental development may sometimes require unfamiliar technologies to be used to ensure the most rapid delivery of the software. Managers may find it difficult to use existing staff in incremental development processes because they lack these skills.
  - **Contractual problems** The normal contractual model between a customer and a software developer is based around a system specification. When there is no such specification, it may be difficult to design a contract for the system development.
  - **Validation problems** In a specification-based process, verification and validation are geared towards demonstrating that the system meets its specification. An independent V& V team can start work as soon as the specification is available and can prepare tests in parallel with the system implementation.
  - **Maintenance problems** Continual change tends to corrupt the structure of any software system. This means that anyone apart from the original developers may find the software difficult to understand. One way to reduce this problem is to use refactoring, where software structures are continually improved during the development process.

- **Figure shows that incremental development and prototyping have different objectives:**
  - The objective of incremental development is to deliver a working system to end-users. This means that you should normally start with the user requirements that are best understood and that have the highest priority
  - The objective of throw-away prototyping is to validate or derive the system requirements. You should start with requirements that are not well understood because you need to find out more about them.

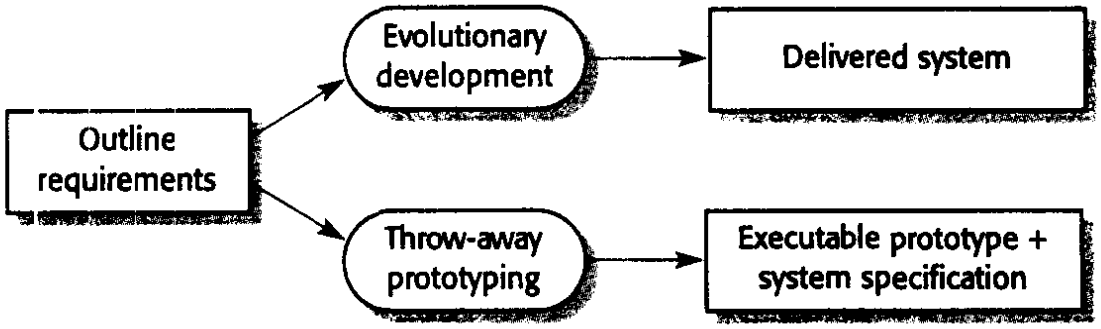


Figure Incremental development and prototyping

Agile methods

- Dissatisfaction with the overheads involved in design methods led to the creation of agile methods.
- These methods:
  - Focus on the code rather than the design;
  - Are based on an iterative approach to software development;
  - Are intended to deliver working software quickly and evolve this quickly to meet changing requirements.

- Agile methods are probably best suited to small/medium-sized business systems or PC products.

Principle	Description
Customer involvement	The customer should be closely involved throughout the development process. Their role is provide and prioritise new system requirements and to evaluate the iterations of the system.
Incremental delivery	The software is developed in increments with the customer specifying the requirements to be included in each increment.
People not process	The skills of the development team should be recognised and exploited. The team should be left to develop their own ways of working without prescriptive processes.
Embrace change	Expect the system requirements to change and design the system so that it can accommodate these changes.
Maintain simplicity	Focus on simplicity in both the software being developed and in the development process used. Wherever possible, actively work to eliminate complexity from the system.

The principles of agile methods

Problems with agile methods:

- It can be difficult to keep the interest of customers who are involved in the process.
- Team members may be unsuited to the intense involvement that characterizes agile methods.
- Prioritizing changes can be difficult where there are multiple stakeholders.
- Maintaining simplicity requires extra work.
- Contracts may be a problem as with other approaches to iterative development.

Extreme programming

- Perhaps the best-known and most widely used agile method.
- Extreme Programming (XP) takes an ‘extreme’ approach to iterative development and customer involvement.

- New versions may be built several times per day;
- Increments are delivered to customers every 2 weeks;
- All tests must be run for every build and the build is only accepted if tests run successfully.

Figure illustrates the XP process to produce an increment of the system that is being developed.

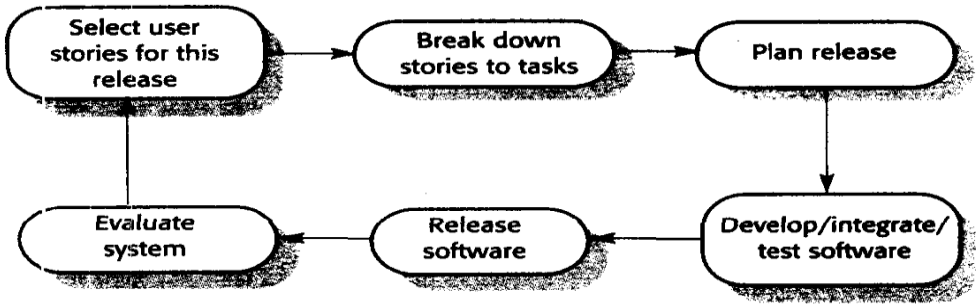


Figure: The extreme programming release cycle

Extreme programming involves a number of practices, summarized in Figure that fit into the principles of agile methods:

- Incremental development is supported through small, frequent releases of the system and by an approach to requirements description based on customer stories or scenarios that can be the basis for process planning.
- Customer involvement is supported through the full-time engagement of the customer in the development team. The customer representative takes part in the development and is responsible for defining acceptance tests for the system.
- People, not process, are supported through pair programming, collective ownership of the system code, and a sustainable development process that does not involve excessively long working hours.

- Change is supported through regular system releases, test-first development and continuous integration.
- Maintaining simplicity is supported through constant refactoring to improve code quality and using simple designs that do not anticipate future changes to the system.

Incremental planning	Requirements are recorded on Story Cards and the Stories to be included in a release are determined by the time available and their relative priority. The developers break these Stories into development ÒTasksÓ
Small Releases	The minimal useful set of functionality that provides business value is developed first. Releases of the system are frequent and incrementally add functionality to the first release.
Simple Design	Enough design is carried out to meet the current requirements and no more.
Test first development	An automated unit test framework is used to write tests for a new piece of functionality before that functionality itself is implemented.
Refactoring	All developers are expected to refactor the code continuously as soon as possible code improvements are found. This keeps the code simple and maintainable.
Pair Programming	Developers work in pairs, checking each otherÕs work and providing the support to always do a good job.
Collective Ownership	The pairs of developers work on all areas of the system, so that no islands of expertise develop and all the developers own all the code. Anyone can change anything.
Continuous Integration	As soon as work on a task is complete it is integrated into the whole system. After any such integration, all the unit tests in the system must pass.
Sustainable pace	Large amounts of over-time are not considered acceptable as the net effect is often to reduce code quality and medium term productivity
On-site Customer	A representative of the end-user of the system (the Customer) should be available full time for the use of the XP team. In an extreme programming process, the customer is a member of the development team and is responsible for bringing system requirements to the team for implementation.

XP and agile principles

- Incremental development is supported through small, frequent system releases.
- Customer involvement means full-time customer engagement with the team.
- Programmers work in pair and develop tests for each task before writing the code. All tests must be successfully executed when the new code is integrated to the system.
- Requirements scenarios
- In XP, user requirements are expressed as scenarios or user stories.
- These are written on cards and the development team break them down into implementation tasks. These tasks are the basis of schedule and cost estimates.
- The customer chooses the stories for inclusion in the next release based on their priorities and the schedule estimates.
- Story card for document downloading

Downloading and printing an article

First, you select the article that you want from a displayed list. You then have to tell the system how you will pay for it - this can either be through a subscription, through a company account or by credit card.

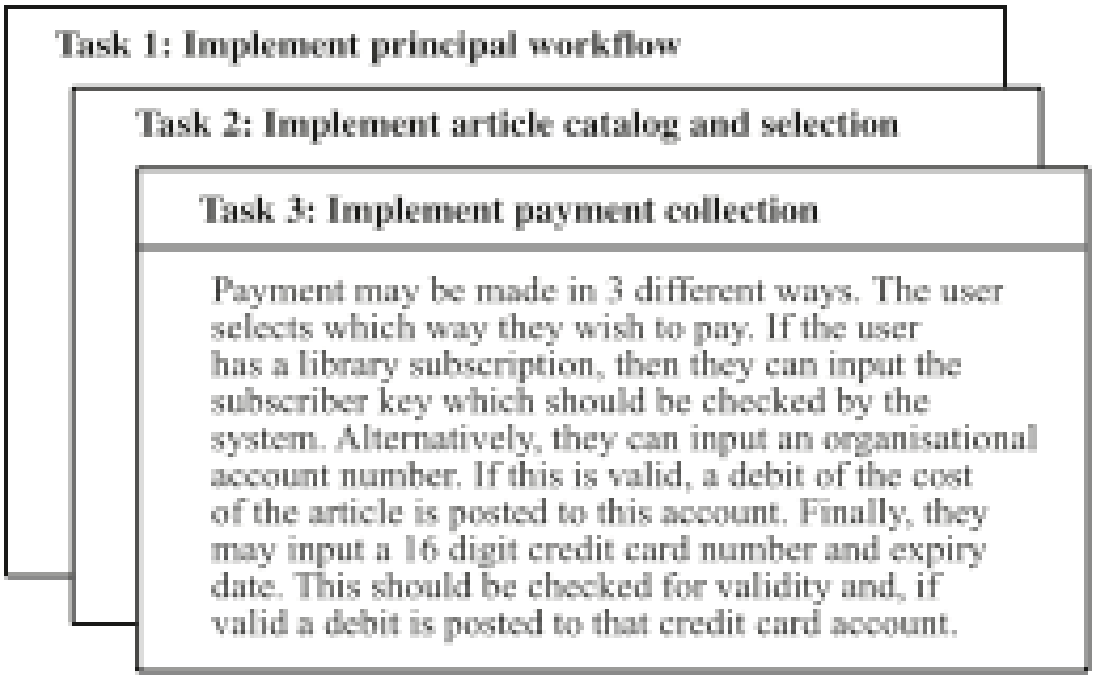
After this, you get a copyright form from the system to fill in and, when you have submitted this, the article you want is downloaded onto your computer.

You then choose a printer and a copy of the article is printed. You tell the system if printing has been successful.

If the article is a print-only article, you can't keep the PDF version so it is automatically deleted from your computer.

XP and change

- Conventional problem in software engineering is to design for change. It is worth spending time and effort anticipating changes as this reduces costs later in the life cycle.
- XP, however, tackles this problem.
- The programming team looks for possible improvements to the software and implements the immediately.
- Therefore, the software should always be easy to understand and change as new stories are implemented.
- Testing in XP
- Test-first development: Biggest difference between iterative development and plan-based development. In XP, the tests are written first. Then, problems of requirements and interface misunderstandings are reduced.
- Incremental test development from scenarios.
- User involvement in test development and validation.
- Automated test harnesses are used to run all component tests each time that a new release is built.
- Task cards for document downloading



Test case description

Test 4: Test credit card validity
Input: A string representing the credit card number and two integers representing the month and year when the card expires **Tests:** Check that all bytes in the string are digits Check that the month lies between 1 and 12 and the year is greater than or equal to the current year. Using the first 4 digits of the credit card number, check that the card issuer is valid by looking up the card issuer table. Check credit card validity by submitting the card number and expiry date information to the card issuer **Output:** OK or error message indicating that the card is invalid

Test-first development

- Writing tests before code clarifies the requirements to be implemented.
- Tests are written as programs rather than data so that they can be executed automatically. The test includes a check that it has executed correctly.

Pair programming

- In XP, programmers work in pairs, sitting together to develop code. These are not always the same people that sit together.
- This helps develop common ownership of code and spreads knowledge across the team -> All programmers may work with other members in a programming pair during the project.
- It serves as an informal review process as each line of code is looked at by more than 1 person.
- It encourages software improving as the whole team can benefit from this.
- Measurements suggest that development productivity with pair programming is similar to that of two people working independently.

Rapid Application Development

Rapid application development (RAD) techniques evolved from so-called fourth-generation languages in the 1980s and are used for developing applications that are data-intensive.

Figure 17.9 illustrates a typical organisation for a RAD system.

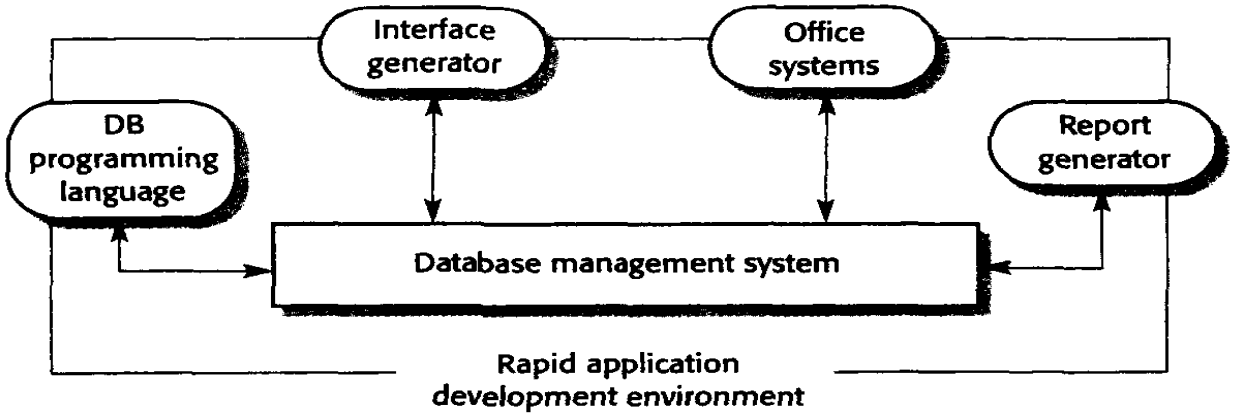
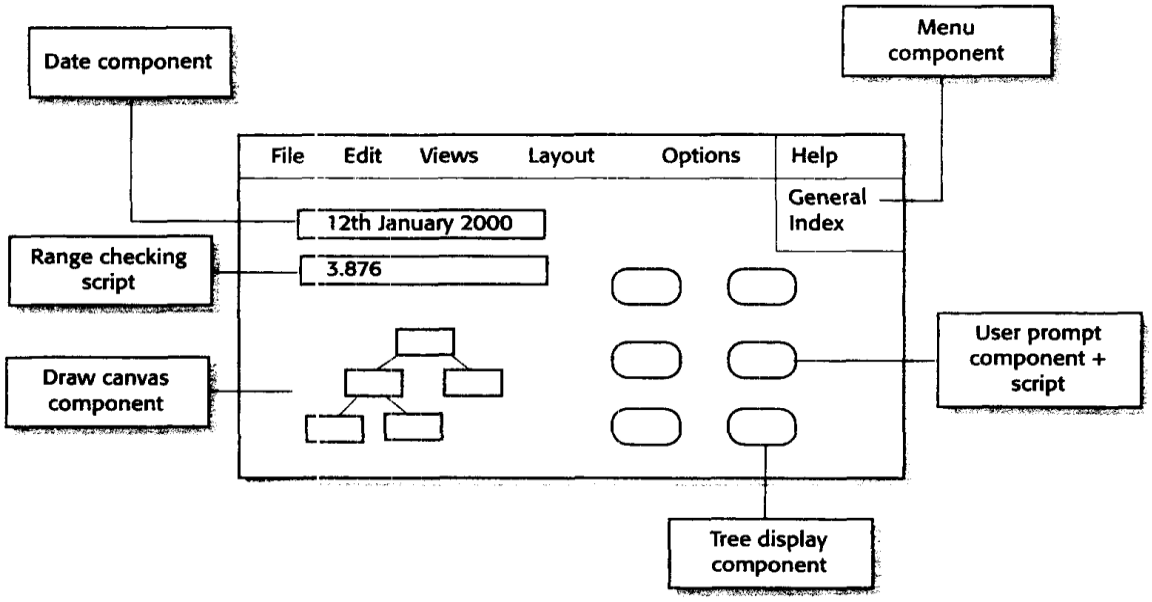


Figure: A rapid application development environment

The tools that are included in a RAD environment are:

- A Database programming language that embeds knowledge of the database structure; and includes fundamental database manipulation operations. SQL(Groff et a., 2002) is the standard database programming language.
- The SQL commands may be input directly or generated automatically from forms filled in by an end-user.. An interface generator, which is used to create forms for data input and display.
- Links to office applications such as a spreadsheet for the analysis and manipulation of numeric information or a word processor for report template creation.
- A report generator, which is used to define and create reports from information in the database.
- Visual development systems such as Visual Basic support this approach to application development. Application programmers build the system interactively by defining the interface in terms of screens, fields, buttons and menus. These are named and processing scripts are associated with individual parts of the interface (e.g., a tutton named Simulate).

Figure below, illustrate this approach which shows an application screen including menus along the top, input fields (the white fields on the left of the display), output fields (the grey field on the left of the display) and buttons (the rounded rectangles on the light of the display).



Visual programming with reuse

Figure above illustrates an application system made up of a compound document that includes text elements, spreadsheet elements and sound files. Text elements are processed by the word processor, tables by the spreadsheet application and sound files by an audio player. When a system user accesses an object of a particular type, the assonated application is called to provide user functionality. For example, when objects of type sound are accessed, the audio player is called to process them.

The **main advantage** of this approach is that a lot of application functionality can be implemented quickly at a very low cost. Users who are already familiar with the applications making up the system do not have to learn how to use new features. However, if they do not know how to use the applications, learning may be difficult, especially as they may be confused by application functionality that isn't necessary. There may also be performance problems with the application because of the need to switch from one application system to another. The switching overhead depends on the: operating system support that is provided.

Software Evolution

Objectives:

- To explain why change is inevitable if software systems are to remain useful
- To discuss software maintenance and maintenance cost factors
- To describe the processes involved in software evolution
- To discuss an approach to assessing evolution strategies for legacy systems

Software change:

- 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 organisations is implementing and managing change to their existing software systems.

Importance of evolution:

- 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 evolving existing software rather than developing new software.
- Post-deployment changes are not simply concerned with repairing fault; in the software. The majority of changes are a consequence of new

requirements that are generated in response to changing business and user needs.

- Consequently, you can think of software engineering as a spiral process with requirements, design, implementation and testing going on throughout the lifetime of the system. This is illustrated in Figure below.

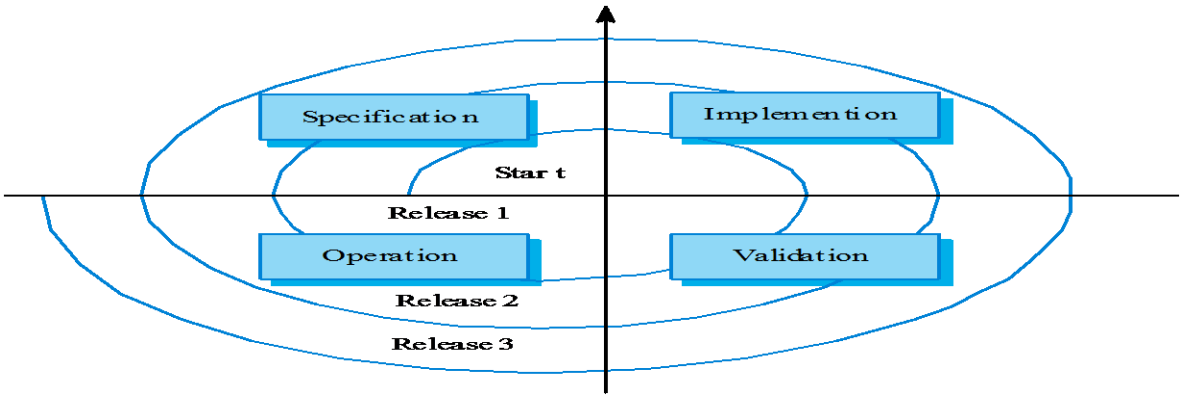


Figure: A spiral model of development and evolution

Program evolution dynamics

- Program evolution dynamics is the study of system change. The majority of work in this area has been carried out by Lehman and Belady, initially in the 1970s and 1980s (Lehman and Belady, 1985).
- The work continued in the 1990s as Lehman and others investigated the significance of feedback in evolution processes (Lehman, 1996; Lehman, et al., 1998; Lehman, et al., 2001).

Law	Description
Continuing change	A program that is used in a real-world environment necessarily must change or become progressively less useful in that environment.
Increasing complexity	As an evolving program changes, its structure tends to become more complex. Extra resources must be devoted to preserving and simplifying the structure.
Large program evolution	Program evolution is a self-regulating process. System attributes such as size, time between releases and the number of reported errors is approximately invariant for each system release.
Organisational stability	Over a program's lifetime, its rate of development is approximately constant and independent of the resources devoted to system development.
Conservation of familiarity	Over the lifetime of a system, the incremental change in each release is approximately constant.
Continuing growth	The functionality offered by systems has to continually increase to maintain user satisfaction.
Declining quality	The quality of systems will appear to be declining unless they are adapted to changes in their operational environment.
Feedback system	Evolution processes incorporate multi-agent, multi-loop feedback systems and you have to treat them as feedback systems to achieve significant product improvement.

- Figure: 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 re-introduced to the environment, this promotes more environmental changes, so the evolution process recycles.
- The second law states that, as a system is changed, its structure is degraded. The only way to avoid this happening is to invest in preventative maintenance where you spend time improving the software structure without adding to its functionality.
- The third law is, perhaps, the most interesting and the most contentious of Lehman's laws. It suggests that large systems have a dynamic of their own that is established at an early stage in the development process.

- Lehman's fourth law suggests that most large programming projects work in what he terms a *saturated* state. That is, a change to resources or staffing has imperceptible effects on the long-term evolution of the system. This is consistent with the third law, which suggests that program evolution is largely independent of management decisions.
- 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 first five laws were in Lehman s initial proposals; the remaining laws were added after further work.
- 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

- Modifying a program after it has been put into use.
- 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.
- The system requirements are likely to change while the system is being developed because the environment is changing. Therefore a delivered system won't meet its requirements!

- Systems are tightly coupled with their environment. When a system is installed in an environment it changes that environment and therefore changes the system requirements.
- Systems MUST be maintained therefore if they are to remain useful in an environment.
- There are three different Types of Software maintenance:
- Maintenance to repair software faults
    - Changing a system to correct deficiencies in the way meets its requirements.
- Maintenance to adapt software to a different operating environment
    - Changing a system so that it operates in a different environment (computer, OS, etc.) from its initial implementation.
- Maintenance to add to or modify the system's functionality
    - Modifying the system to satisfy new requirements.

Figure: shows the Distribution of maintenance effort

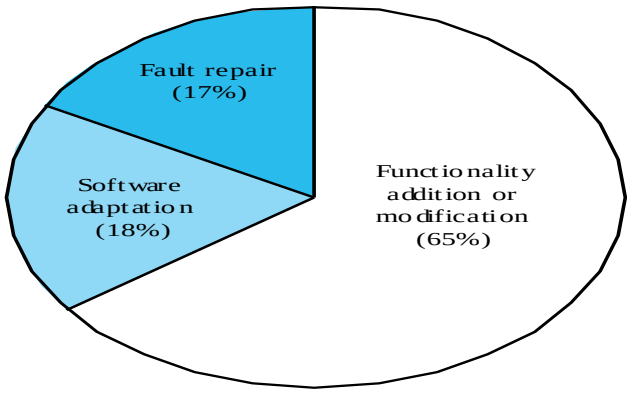


Figure: Distribution of maintenance effort

Maintenance costs:

Maintenance costs as a proportion of development costs vary from one application domain to another.

- Usually greater than development costs (2* to 100* depending on the application).
- Affected by both technical and non-technical factors.
- Increases as software is maintained. Maintenance corrupts the software structure so makes further maintenance more difficult. Ageing software can have high support costs (e.g. old languages, compilers etc.).
- Figure shows how overall lifetime costs may decrease as more effort is expended during system development to produce a maintainable system. Because of the potential reduction in costs of understanding, analysis and testing, there is a significant multiplier effect when the system is developed for maintainability.

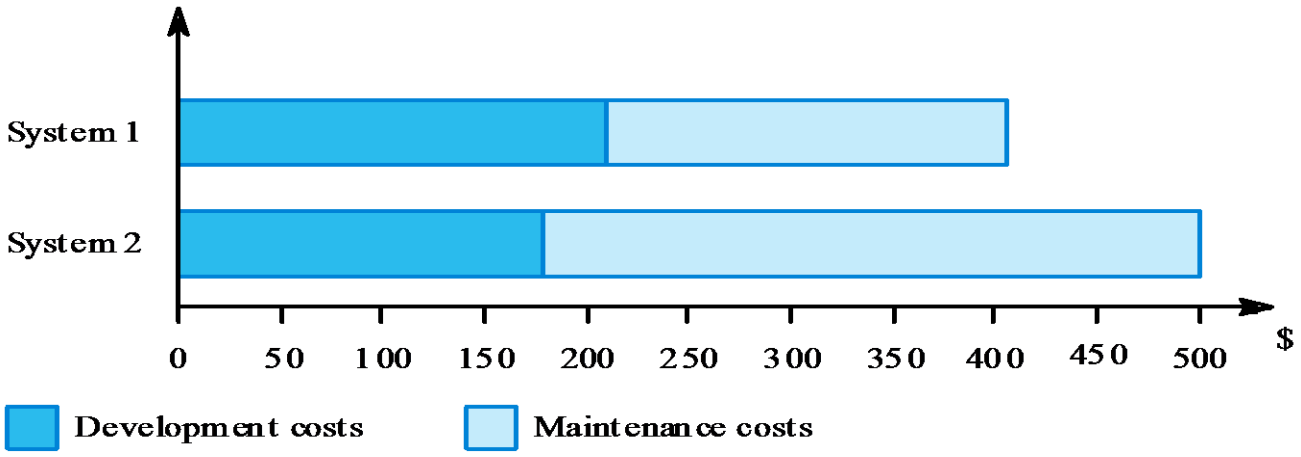


Figure Development and maintenance costs

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

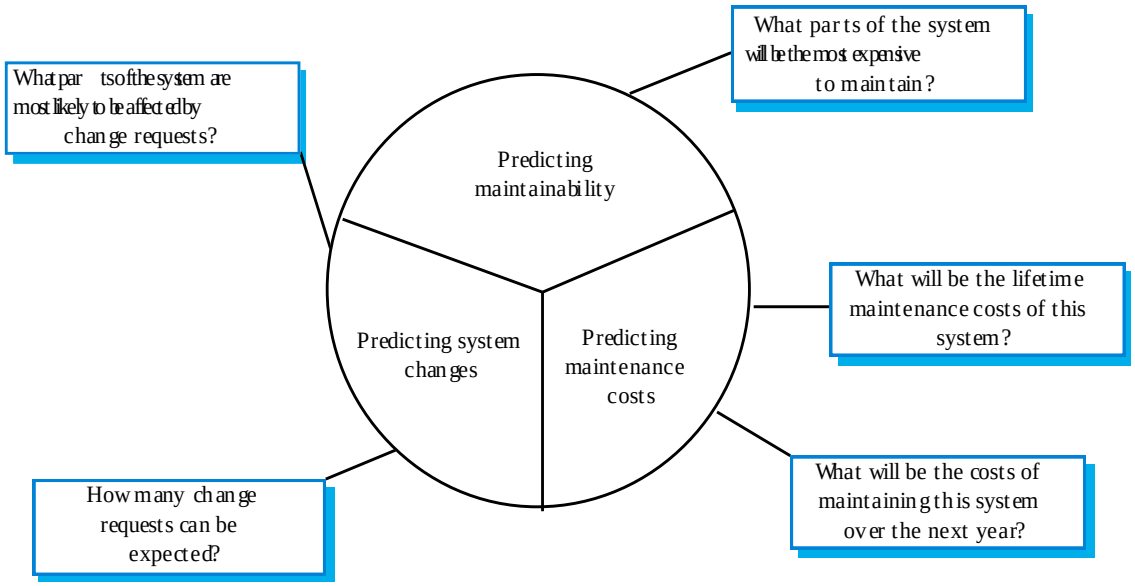
- **Team stability**
    - Maintenance costs are reduced if the same staffs are involved with them for some time.
- **Contractual responsibility**
    - The developers of a system may have no contractual responsibility for maintenance so there is no incentive to design for future change.
- **Staff skills**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

Maintenance prediction

- Maintenance prediction is concerned with assessing which parts of the system may cause problems and have high maintenance costs
- Whether a system change should be accepted depends, to some extent, on the maintainability of the system components affected by that change.
- Implementing system changes tends to degrade the system structure and hence reduce its maintainability.
- Maintenance costs depend on the number of changes, and the costs of change implementation depend on the maintainability of system components.

- Predicting the number of change requests for a system requires an understanding of the relationship between the system and its external environment. Some systems have a very complex relationship with their external environment and changes to that environment inevitably result in changes to the system.
- The number and complexity of system interfaces The larger the number of interfaces and the more complex they are, the more likely it is that demands for change will be made.
- The number of inherently volatile system requirements As I discussed in Chapter 7, 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 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.

Maintenance prediction



Change prediction:

- 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;
    - Number of inherently volatile system requirements;
    - The business processes where the system is used.

Complexity metrics:

- Predictions of maintainability can be made by assessing the complexity of system components.
- Studies have shown that most maintenance effort is spent on a relatively small number of system components.
- Complexity depends on
    - Complexity of control structures;
    - Complexity of data structures;
    - Object, method (procedure) and module size.

Process metrics:

- Process measurements may be used to assess maintainability
    - Number of requests for corrective maintenance;
    - Average time required for impact analysis;
    - Average time taken to implement a change request;
    - Number of outstanding change requests.
- If any or all of these is increasing, this may indicate a decline in maintainability.