Module-5
AGILE SOFTWARE DEVELOPMENT

Coping with Change

→ The system requirements change as the business procuring the system responds to external pressures and management priorities change.

→ Change adds to the costs of software development because it usually means that work that has been completed has to be redone. This is called rework.

→ There are two related approaches that may be used to reduce the costs of rework:

1. Change avoidance, where the software process includes activities that can anticipate possible changes before significant rework is required. For example, a prototype system may be developed to show some key features of the system to customers. They can experiment with the prototype and refine their requirements before committing to high software production costs.
2. Change tolerance, where the process is designed so that changes can be accommodated at relatively low cost. This normally involves some form of incremental development.

→ There are 2 ways of coping with change and changing system requirements

1. **System prototyping**, where a version of the system or part of the system is developed quickly to check the customer's requirements and the feasibility of some design decisions. This supports change avoidance as it allows users to experiment with the system before delivery and so refine their requirements.
2. **Incremental delivery**, where system increments are delivered to the customer for comment and experimentation. This supports both change avoidance and change tolerance. It avoids the premature commitment to requirements for the whole system and allows changes to be incorporated into later increments at relatively low cost.

Prototyping

→ A prototype is an initial version of a software system that is used to demonstrate concepts, try out design options, and find out more about the problem and its possible solutions.

→ A software prototype can be used in a software development process to help anticipate changes that may be required:

1. In the requirements engineering process, a prototype can help with the elicitation and validation of system requirements.
2. In the system design process, a prototype can be used to explore particular software solutions and to support user interface design.

→ System prototypes allow users to see how well the system supports their work.

→ They may get new ideas for requirements, and find areas of strength and weakness in the software. They may then propose new system requirements.

→ A system prototype may be used while the system is being designed to carry out design experiments to check the feasibility of a proposed design.

→ For example, a database design may be prototyped and tested to check that it supports efficient data access for the most common user queries.

→ A process model for prototype development is shown in fig 5.1.

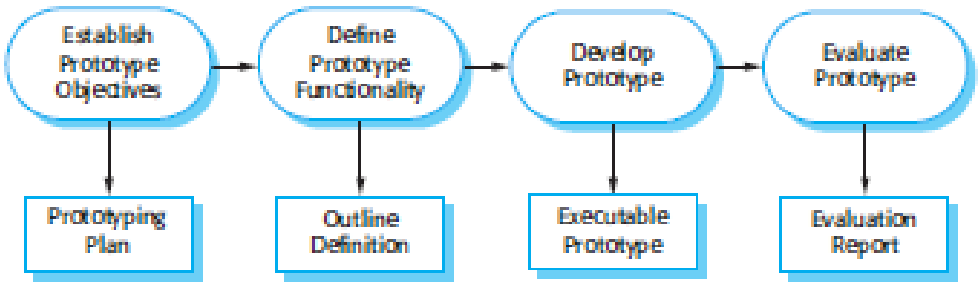

Fig 5.1: The process of prototype development

→ The objectives of prototyping should be made explicit from the start of the process.

→ These may be to develop a system to prototype the user interface, to develop a system to validate functional system requirements, or to develop a system to demonstrate the feasibility of the application to managers.

→ The next stage in the process is to decide what to put into and, perhaps more importantly, what to leave out of the prototype system.

→ The final stage of the process is prototype evaluation.

→ Developers are sometimes pressured by managers to deliver throwaway prototypes, particularly when there are delays in delivering the final version of the software.

→ However, this is usually unwise:

1. It may be impossible to tune the prototype to meet non-functional requirements, such as performance, security, robustness, and reliability requirements, which were ignore during prototype development.

2. Rapid change during development inevitably means that the prototype is undocumented. The only design specification is the prototype code. This is not good enough for long-term maintenance.
3. The changes made during prototype development will probably have degraded the system structure. The system will be difficult and expensive to maintain.
4. Organizational quality standards are normally relaxed for prototype development

Incremental Delivery

→ Incremental delivery (Fig 5.2) is an approach to software development where some of the developed increments are delivered to the customer and deployed for use in an operational environment. In an incremental delivery process, customers identify, in outline, the services to be provided by the system.

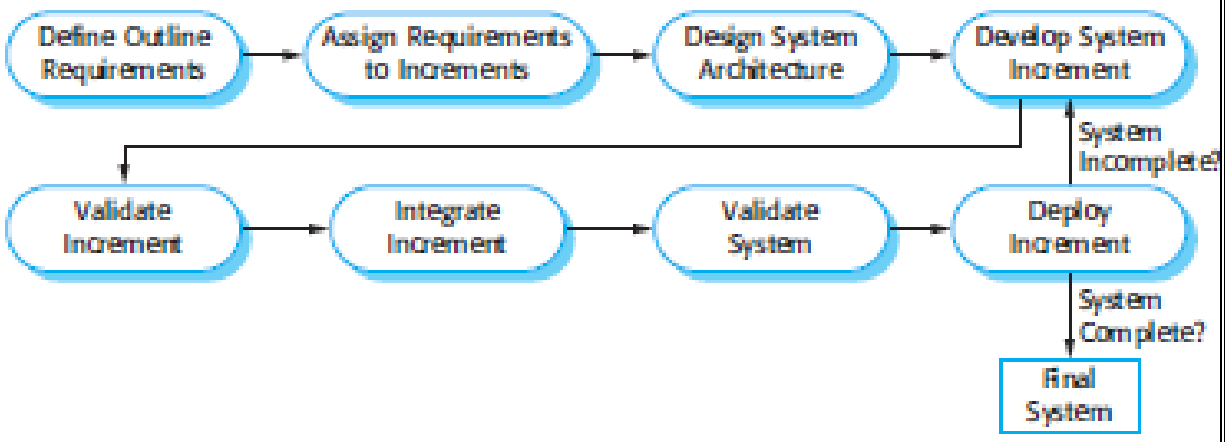

Fig 5.2: Incremental delivery

→ They identify which of the services are most important and which are least important to them.

→ Once the system increments have been identified, the requirements for the services to be delivered in the first increment are defined in detail and that increment is developed.

→ During development, further requirements analysis for later increments can take place but requirements changes for the current increment are not accepted.

→ Incremental delivery has a number of **advantages:**

1. Customers can use the early increments as prototypes and gain experience that informs their requirements for later system increments. Unlike prototypes, these are part of the real system so there is no re-learning when the complete system is available.

2. Customers do not have to wait until the entire system is delivered before they can gain value from it. The first increment satisfies their most critical requirements so they can use the software immediately.
3. The process maintains the benefits of incremental development in that it should be relatively easy to incorporate changes into the system.
4. As the highest-priority services are delivered first and increments then integrated, the most important system services receive the most testing. This means that customers are less likely to encounter software failures in the most important parts of the system.

→ However, there are problems with incremental delivery:

1. Most systems require a set of basic facilities that are used by different parts of the system. As requirements are not defined in detail until an increment is to be implemented, it can be hard to identify common facilities that are needed by all increments.
2. Iterative development can also be difficult when a replacement system is being developed. Users want all of the functionality of the old system and are often unwilling to experiment with an incomplete new system. Therefore, getting useful customer feedback is difficult.
3. In the incremental approach, there is no complete system specification until the final increment is specified. This requires a new form of contract, which large customers such as government agencies may find difficult to accommodate.

Plan-driven and Agile Development

→ Agile approaches to software development consider design and implementation to be the central activities in the software process.

→ They incorporate other activities, such as requirements elicitation and testing, into design and implementation.

→ In a plan-driven approach, iteration occurs within activities with formal documents used to communicate between stages of the process.

→ For example, the requirements will evolve and, ultimately, a requirements specification will be produced.

→ This is then an input to the design and implementation process.

→ In an agile approach, iteration occurs across activities. Therefore, the requirements and the design are developed together, rather than separately

→ A plan-driven software process can support incremental development and delivery.

→ It is perfectly feasible to allocate requirements and plan the design and development phase as a series of increments.

→ An agile process is not inevitably code-focused and it may produce some design documentation.

→ Fig 5.3 shows the distinctions between plan-driven and agile approaches to system specification

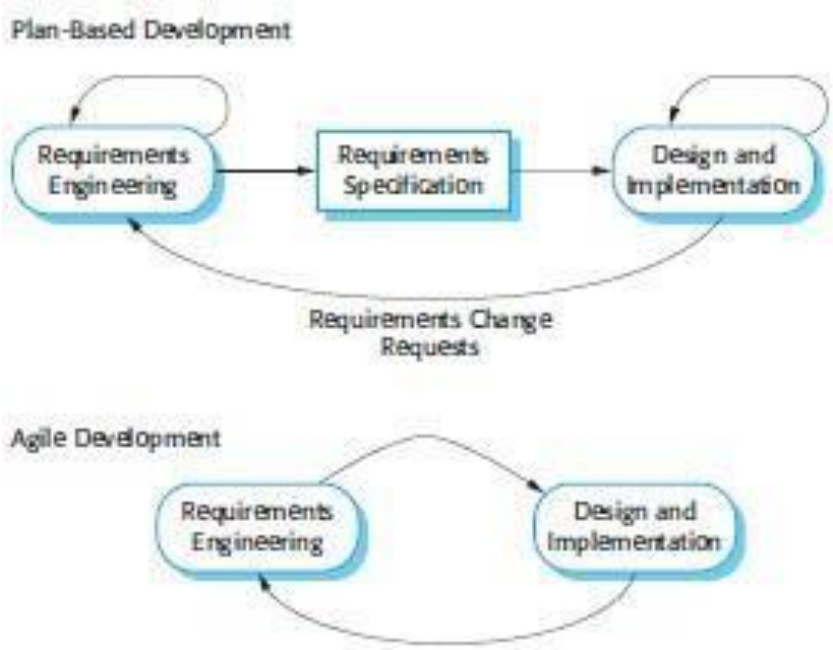

Fig 5.3: Plan- driven and agile specification

Extreme Programming

→ In extreme programming, requirements are expressed as scenarios (called user stories), which are implemented directly as a series of tasks.

→ Programmers work in pairs and develop tests for each task before writing the code.

→ All tests must be successfully executed when new code is integrated into the system.

→ Fig 5.4 illustrates the XP process to produce an increment of the system that is being developed.

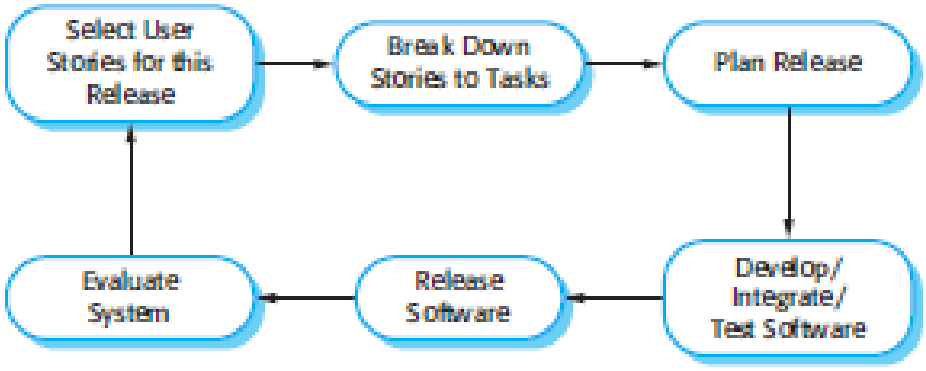

Fig 5.4: The extreme programming release cycle

→ Extreme programming involves a number of practices, summarized in fig 5.5, which reflect the principles of agile methods:

* Incremental development is supported through small, frequent releases of the system. Requirements are based on simple customer stories or scenarios that are used as a basis for deciding what functionality should be included in a system increment.
* Customer involvement is supported through the continuous 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 embraced through regular system releases to customers, test-first development, refactoring to avoid code degeneration, and continuous integration of new functionality.
* Maintaining simplicity is supported by constant refactoring that improves code quality and by using simple designs that do not unnecessarily anticipate future changes to the system.

Principle or practice	Description
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 'Tasks'. See Figures 3.5 and 3.6.
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 other's 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 take responsibility for all of the code. Anyone can change anything.
Continuous integration	As soon as the 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 overtime 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.

Fig 5.5: Extreme programming practices

Testing in XP

→ XP includes an approach to testing that reduces the chances of introducing undiscovered errors into the current version of the system.

→ The key features of testing in XP are:

- Test-first development,
- Incremental test development from scenarios,
- User involvement in the test development and validation, and
- The use of automated testing frameworks.

→ Fig 5.6 is a shortened description of a test case that has been developed to check that the prescribed dose of a drug does not fall outside known safe limits.

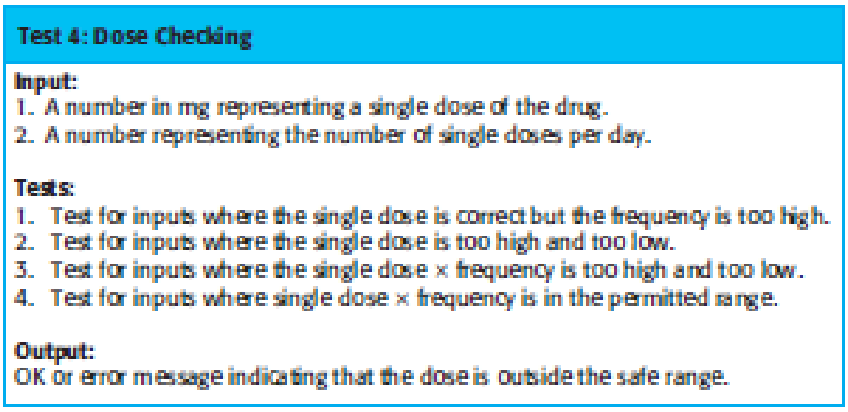

Fig 5.6: Test case description for dose checking

→ The role of the customer in the testing process is to help develop acceptance tests for the stories that are to be implemented in the next release of the system.

→ Test automation is essential for test-first development.

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

→ However, 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. For example, they may write incomplete tests that do not check for all possible exceptions that may occur.
2. Some tests can be very difficult to write incrementally. For example, in a complex user interface, it is often difficult to write unit tests for the code that implements the 'display logic' and workflow between screens.
3. It's difficult to judge the completeness of a set of tests. Although there are lot of system tests, the test set may not provide complete coverage. Crucial parts of the system may not be executed and so remain untested.

Pair Programming

→ Another innovative practice that has been introduced in XP is that programmers work in pairs to develop the software.

→ They actually sit together at the same workstation to develop the software.

→ The use of pair programming has a number of advantages:

* It supports the idea of collective ownership and responsibility for the system.
* It acts as an informal review process because each line of code is looked at by at least two people. Code inspections and reviews are very successful in discovering a high percentage of software errors.
* It helps support refactoring, which is a process of software improvement. The difficulty of implementing this in a normal development environment is that effort in refactoring is expended for long-term benefit. An individual who practices refactoring may be judged to be less efficient than one who simply carries on developing code. Where pair programming and collective ownership are used, others benefit immediately from the refactoring so they are likely to support the process.

Agile Project Management

→ The principal responsibility of software project managers is to manage the project so that the software is delivered on time and within the planned budget for the project.

→ They supervise the work of software engineers and monitor how well the software development is progressing.

→ Fig 5.7 is a diagram of the Scrum management process.

Fig 5.7: The Scrum process

→ Scrum does not prescribe the use of programming practices such as pair programming and test-first development.

→ It can therefore be used with more technical agile approaches, such as XP, to provide a management framework for the project.

→ There are three phases in Scrum.

→ The first is an outline planning phase where you establish the general objectives for the project and design the software architecture.

→ This is followed by a series of sprint cycles, where each cycle develops an increment of the system.

→ Finally, the project closure phase wraps up the project, completes required documentation such as system help frames and user manuals, and assesses the lessons learned from the project.

→ A Scrum sprint is a planning unit in which the work to be done is assessed, features are selected for development, and the software is implemented.

→ At the end of a sprint, the completed functionality is delivered to stakeholders. Key characteristics of this process are as follows:

1. Sprints are fixed length, normally 2–4 weeks. They correspond to the development of a release of the system in XP.
2. The starting point for planning is the product backlog, which is the list of work to be done on the project. During the assessment phase of the sprint, this is reviewed, and priorities and risks are assigned. The customer is closely involved in this process and can introduce new requirements or tasks at the beginning of each sprint.
3. The selection phase involves all of the project team who work with the customer to select the features and functionality to be developed during the sprint.
4. Once these are agreed, the team organizes themselves to develop the software. Short daily meetings involving all team members are held to review progress and if necessary, reprioritize work.
5. At the end of the sprint, the work done is reviewed and presented to stakeholders. The next sprint cycle then begins.

→ Advantages:

* The product is broken down into a set of manageable and understandable chunks.
* Unstable requirements do not hold up progress.
* The whole team has visibility of everything and consequently team communication is improved.

* Customers see on-time delivery of increments and gain feedback on how the product works.
* Trust between customers and developers is established and a positive culture is created in which everyone expects the project to succeed.

Scaling Agile Methods

→ Agile methods were developed for use by small programming teams who could work together in the same room and communicate informally.

→ Agile methods have therefore been mostly used for the development of small and medium-sized systems.

→ Large software system development is different from small system development in a number of ways:

* Large systems are usually collections of separate, communicating systems, where separate teams develop each system. Frequently, these teams are working in different places, sometimes in different time zones. It is practically impossible for each team to have a view of the whole system.
* Large systems are 'brownfield systems' that is they include and interact with a number of existing systems. Many of the system requirements are concerned with this interaction and so don't really lend themselves to flexibility and incremental development
* Where several systems are integrated to create a system, a significant fraction of the development is concerned with system configuration rather than original code development. This is not necessarily compatible with incremental development and frequent system integration.
* Large systems and their development processes are often constrained by external rules and regulations limiting the way that they can be developed, that require certain types of system documentation to be produced, etc.
* Large systems have a long procurement and development time. It is difficult to maintain coherent teams who know about the system over that period as, inevitably, people move on to other jobs and projects.
* Large systems usually have a diverse set of stakeholders. For example, nurses and administrators may be the end-users of a medical system but senior

medical staff, hospital managers, etc. are also stakeholders in the system. It is practically impossible to involve all of these different stakeholders in the development process.

→ There are two perspectives on the scaling of agile methods:

1. A 'scaling up' perspective, which is concerned with using these methods for developing large software systems that cannot be developed by a small team
2. A 'scaling out' perspective, which is concerned with how agile methods can be introduced across a large organization with many years of software development experience.

→ It is difficult to introduce agile methods into large companies for a number of reasons:

* Project managers who do not have experience of agile methods may be reluctant to accept the risk of a new approach, as they do not know how this will affect their particular projects.
* Large organizations often have quality procedures and standards that all projects are expected to follow and, because of their bureaucratic nature, these are likely to be incompatible with agile methods. Sometimes, these are supported by software tools (e.g., requirements management tools) and the use of these tools is mandated for all projects.
* Agile methods seem to work best when team members have a relatively high skill level. However, within large organizations, there are likely to be a wide range of skills and abilities, and people with lower skill levels may not be effective team members in agile processes.
* There may be cultural resistance to agile methods, especially in those organizations that have a long history of using conventional systems engineering processes

The Agile manifesto: Values and Principles

→ Values:

* Individuals and interactions over processes and tools
* Working software over comprehensive documentation
* Customer collaboration over contract negotiation
* Responding to change over following a plan

→ Principles:

* Highest priority is to satisfy the customer through early and continuous delivery of valuable software
* Welcome changing requirements, even late in development. Agile processes harness change for the customers competitive advantage
* Deliver working software frequently, from a couple of weeks to couple of months, with a preference to the shortest timescale
* Business people and developers must work together daily throughout the project.
* Build projects around motivated individuals. Give them the environment and the support they need, and trust them to get the job done
* The most efficient and effective method of conveying information to and within a development team is face-to-face conversation
* Working software is the primary measure of progress
* Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely
* Continuous attention to technical excellence and good design enhances agility
* Simplicity – the art of maximizing the amount of work not done – is essential
* The best architectures, assignments and designs emerge from self organizing teams.
* At regular intervals the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.