Unit 3

4.SELECTION OF AN APPROPRIATE PROJECT APPROACH

The development of software in-house usually means that:

- the developers and the users belong to the same organization;
- the application will slot into a portfolio of existing computer-based systems;
- the methodologies and technologies are largely dictated by organizational standards and policies, including the existing enterprise architecture.

The relevant part of the Step Wise approach is Step 3: Analyse project characteristics. The selection of a particular process model could add new products to the Project Breakdown Structure or new activities to the activity network. This will generate inputs for Step 4: Identify the products and activities of the project.

Build or Buy?

Software development can be seen from two differing viewpoints: that of the developers and that of the clients or users. With in-house development, the developers and the users are in the same organization. Where the development is outsourced, they are in different organizations.

The development of a new IT application within an organization would often require the recruitment of technical staff who, once the project has been completed, will no longer be required. Because this project is a unique new development for the client organization there may be a lack of executives qualified to lead the effort.

An option which is increasingly taken is to obtain a licence to run off-the-shelf software.

Advantages

- the supplier of the application can spread the cost of development over a large number of customers and thus the cost per customer should be reduced;
- the software already exists and so
    1. it can be examined and perhaps even trialed before acquisition,
    2. there is no delay while the software is being built;
- where lots of people have already used the software, most of the bugs are likely to have been reported and removed, leading to more reliable software.

Disadvantages

- As you have the same application as everyone else, there is no competitive advantage.
- Modern off-the-shelf software tends to be very customizable: the characteristics of the application can be changed by means of various parameter tables. However, this flexibility has limits and you may end up having to change your office procedures in order to fi t in with the computer system.
- You will not own the software code. This may rule out making modifications to the application in response to changes in the organization or its environment.
- Once you have acquired your off-the-shelf system, your organization may come to be very reliant upon it. This may create a considerable barrier to moving to a different application. The supplier may be in a position to charge inflated licence fees because you are effectively a captive customer.

Choosing Methodologies and Technologies

The term methodology describes a collection of methods. Techniques and methods are sometimes distinguished.

Techniques tend to involve the application of scientific, mathematical or logical principles to resolve a particular kind of problem. They often require the practice of particular personal skills (the word ‘technique’ is derived from the Greek for skilful) – software design is a good example.

Methods often involve the creation of models. A model is a representation of a system which abstracts certain features but ignores others. For example, an entity relationship diagram (ERD) is a model of the structure of the data used by a system.

Project analysis should select the most appropriate methodologies and technologies for a project. Methodologies include approaches like the Unifi ed Software Development Process (USDP), Structured Systems Analysis and Design Method (SSADM) and Human-Centred Design, while technologies might include appropriate application-building and automated testing environments. The analysis identifies the methodology, but also selects the methods within the methodology that are to be deployed.

As well as the products and activities, the chosen methods and technologies will affect:

- the training requirements for development staff;
- the types of staff to be recruited;
- the development environment – both hardware and software;
- system maintenance arrangements

The steps of project analysis.

- **Identify project as either objective-driven or product-driven**
  A product-driven project creates products defi ned before the start of the project. An objective-driven project will often have come first which will have defi ned the general software solution that is to be implemented.
- **Analyse other project characteristics**
  The following questions can be usefully asked.
  Is a data-oriented or process-oriented system to be implemented? Data-oriented systems generally mean information systems that will have a substantial database. Process-oriented systems refer to embedded control systems.
  Will the software that is to be produced be a general tool or application specific? An example of a general tool would be a spreadsheet or a word processing package. An application-specific package could be, for example, an airline seat reservation system
  Are there specific tools available for implementing the particular type of application?
  Is the system to be created safety critical? For instance, could a malfunction in the system endanger human life? If so, among other things, testing would become very important
  Is the system designed primarily to carry out predefined services or to be engaging and entertaining? With software designed for entertainment, design and evaluation will need to be carried out differently from more conventional software products.
  What is the nature of the hardware/software environment in which the system will operate? The environment in which the final software will operate could be different from that in which it is to be developed.
- **Identify high-level project risks**
  At the beginning of a project, some managers might expect elaborate plans even though we are ignorant of many important factors affecting the project.
  One suggestion is that uncertainty can be associated with the products, processes, or resources of a project.
  Product uncertainty How well are the requirements understood? The users themselves could be uncertain about what a proposed information system is to do.
  Process uncertainty The project under consideration might be the first where an organization is using an approach like extreme programming (XP) or a new application-building tool. Any change in the way that the systems are developed introduces uncertainty.
  Resource uncertainty The main area of uncertainty here is likely to be the availability of staff of the right ability and experience. The larger the number of resources needed or the longer the duration of the project, the more inherently risky it will be.
- **Take into account user requirements concerning implementation**
  staff planning a project should try to ensure that unnecessary constraints are not imposed on the way that a project's objectives are to be m
- **Select general life-cycle approach**
  Control systems A real-time system will need to be implemented using an appropriate methodology. Real-time systems that employ concurrent processing may have to use techniques such as Petri nets.
  Information systems Similarly, an information system will need a methodology, such as SSADM or Information Engineering, that matches that type of environment.

Availability of users Where the software is for the general market rather than application and user specific, then a methodology which assumes that identifiable users exist who can be quizzed about their needs would have to be thought about with caution.
Specialized techniques For example, expert system shells and logic-based programming languages have been invented to expedite the development of knowledge-based systems. Similarly, a number of specialized techniques and standard components are available to assist in the development of graphics based systems.
Hardware environment The environment in which the system is to operate could put constraints on the way it is to be implemented. The need for a fast response time or restricted computer memory might mean that only low-level programming languages can be used.
Safety-critical systems Where safety and reliability are essential, this might justify the additional expense of a formal specification using a notation such as OCL. Extremely critical systems could justify the cost of having independent teams develop parallel systems with the same functionality. The operational systems can then run concurrently with continuous cross-checking. This is known as n-version programming.
Imprecise requirements Uncertainties or a novel hardware/software platform mean that a prototyping approach should be considered. If the environment in which the system is to be implemented is a rapidly changing one, then serious consideration would need to be given to incremental delivery. If the users have uncertain objectives in connection with the project, then a soft systems approach might be desirable.

Software Processes and Process Models

A software product development process usually starts when a request for the product is received from the customer. For a generic product, the marketing department of the company is usually considered as the customer. This expression of need for the product is called product inception. From the inception stage, a product undergoes a series of transformations through a few identifiable stages until it is fully developed and released to the customer.
After release, the product is used by the customer and during this time the product needs to be maintained for fixing bugs and enhancing functionalities. This stage is called the maintenance stage.
When the product is no longer useful to the customer, it is retired. This set of identifiable stages through which a product transits from inception to retirement form the life cycle of the product. The software life cycle is also commonly referred to as Software Development Life Cycle (SDLC) and software process.

A life cycle model (also called a process model) of a software product is a graphical or textual representation of its life cycle. Additionally, a process model may describe the details of various types of activities carried out during the different phases and the documents produced.

Choice of Process Models

The word ‘process’ emphasizes the idea of a system in action. In order to achieve an outcome, the system will have to execute one or more activities: this is its process. This applies to the development of computerbased applications. A number of interrelated activities have to be undertaken to create a final product. These activities can be organized in different ways and we can call these process models.

The planner selects methods and specifies how they are to be applied. Not all parts of a methodology such as USDP or SSADM will be compulsory.

Rapid Application Development

Rapid Application Development (RAD) model is also sometimes referred to as the rapid prototyping model. This model has the features of both the prototyping and the incremental delivery models.

The major aims of the RAD model are as follows:

- to decrease the time taken and the cost incurred to develop software systems; and
- to limit the costs of accommodating change requests by incorporating them as early as possible before large investments have been made on development and testing.

One of the major problems that has been identified with the waterfall model is the following. Clients often do not know what they exactly want until they see a working system. However, they do not see the working system until the development is complete in all respects and delivered to them. As a result, the exact requirements are brought out only through the process of customers commenting on the installed application. The required changes are incorporated through subsequent maintenance efforts.

In the RAD model, development takes place in a series of short cycles called iterations. Plans are made for one iteration at a time only. The time planned for each iteration is called a time box. Each iteration enhances the implemented functionality of the application a little. During each iteration, a quick and dirty prototype for some functionality is developed. The customer evaluates the prototype and gives feedback, based on which the prototype is refi ned. Thus, over successive iterations, the full set of functionalities of the software takes shape.

The development team is also required to include a customer representative to clarify the requirements. Thus, conscious attempts are made to bridge the communication gap between the customer and the development team and to tune the system to the exact customer requirements

Agile Method

Agile methods are designed to overcome the disadvantages we have noted in our discussions on the heavyweight implementation methodologies. One of the main disadvantages of the traditional heavyweight methodologies is the difficulty of efficiently accommodating change requests from customers during execution of the project. Note that the agile model is an umbrella term that refers to a group of development processes, and not any single model of software development.

There are various agile approaches such as the following:

- Crystal Technologies
- Atern (formerly DSDM)
- Feature-driven Development
- Scrum
- Extreme Programming (XP)

In the agile model, the feature requirements are decomposed into several small parts that can be incrementally developed. The agile model adopts an iterative approach. Each incremental part is developed over an iteration. Each iteration is intended to be small and easily manageable, and lasts for a couple of weeks only. At a time, only one increment is planned, developed, and then deployed at the customer's site. No long-term plans are made. The time taken to complete an iteration is called a time box. The implication of the term 'time box' is that the end date for an iteration does not change. The development team can, however, decide to reduce the delivered functionality during a time box, if necessary, but the delivery date is considered sacrosanct.

Besides the delivery of increments after each time box, a few other principles discussed below are central to the agile model.

- Agile model emphasizes face-to-face communication over written documents. Team size is deliberately kept small (5–9 people) to help the team members effectively communicate with each other and collaborate. This makes the agile model well-suited to the development of small projects, though large projects can also be executed using the agile model. In a large project, it is likely that the collaborating teams might work at different locations. In this case, the different teams maintain daily contact through video conferencing, telephone, e-mail, etc.
- An agile project usually includes a customer representative in the team. At the end of each iteration, the customer representative along with the stakeholders review the progress made, re-evaluate the requirements, and give suitable feedback to the development team.
- Agile development projects usually deploy pair programming. In this approach, two programmers work together at one work station. One types the code while the other reviews the code as it is typed. The two programmers switch their roles every hour or so. Several studies indicate that programmers working in pairs produce compact well-written programs and commit fewer errors as compared to programmers working alone.

Extreme Programming (XP)

The ideas were largely developed on the C3 payroll development project at Chrysler. The approach is called 'extreme programming' because, according to Beck, 'XP takes commonsense principles to extreme levels'. Four core values are presented as the foundations of XP.

1. **Communication and feedback.** It is argued that the best method of communication is face-to-face communication. Also, the best way of communicating to the users the nature of the software under production is to provide them with frequent working increments. Formal documentation is avoided.
2. **Simplicity.** The simplest design that implements the users' requirements should always be adopted. Effort should not be spent trying to cater for future possible needs – which in any case might never actually materialize.
3. **Responsibility.** The developers are the ones who are ultimately responsible for the quality of the software – not, for example, some system testing or quality control group.
4. **Courage**. This is the courage to throw away work in which you have already invested a lot of effort, and to start with a fresh design if that is what is called for. It is also the courage to try out new ideas – after all, if they do not work out, they can always be scrapped. Beck argues that this attitude is more likely to lead to better solutions.

Among the core practices of XP are the following.

The planning exercise the components of the system that users could actually use. XP refers to these as releases. Within these releases code is developed in iterations, periods of one to four weeks' duration during which specifi c features of the software are created. Note that these are not usually 'iterations' in the sense that they are new, improved, versions of the same feature – although this is a possibility.

Small releases The time between releases of functionality to users should be as short as possible. Beck suggests that releases should ideally take a month or two.

Metaphor The system to be built will be software code that reflects things that exist and happen in the real world. A payroll application will calculate and record payments to employees. The terms used to describe the corresponding software elements should, as far as possible, refl ect real-world terminology – at a very basic level this would mean using meaningful names for variables and procedures such as 'hourly_rate' and 'calculate_ gross_pay'. In this context 'architecture' refers to the use of system models such as class and collaboration diagrams to describe the system. The astute reader might point out that the use of the term 'architecture' is itself a metaphor

Simple design This is the practical implementation of the value of simplicity that was described above

Testing: Testing is done at the same time as coding. The test inputs and expected results should be scripted so that the testing can be done using automated testing tools. These test cases can then be accumulated so that they can be used for regression testing to ensure that later developments do not insert errors into existing working code.

Refactoring A threat to the target of striving to have always the simplest design is that over time, as modifi cations are made to code, the structure tends to become more spaghetti-like. The answer to this is to have the courage to resist the temptation to make changes that affect as little of the code as possible and be prepared to rewrite whole sections of code if this will keep the code structured.

Pair programming All software code is written by pairs of developers, one actually doing the typing and the other observing, discussing and making comments and suggestions about what the other is doing. At intervals, the developers can swap roles. The ideal is that you are constantly changing partners so that you get to know about a wide range of features that are under development.

Collective ownership This is really the corollary of pair programming. The team as a whole takes collective responsibility for the code in the system. A unit of code does not 'belong' to just one programmer who is the only one who can modify it.

Continuous integration This is another aspect of testing practice. As changes are made to software units, integrated tests are run regularly – at least once a day – to ensure that all the components work together correctly

Forty-hour weeks The principle is that normally developers should not work more than 40 hours a week. It is realistic to accept that sometimes there is a need for overtime work to deal with a particular problem – but in this case overtime should not be worked for two weeks in a row.

On-site customers Fast and effective communication with the users is achieved by having a user domain expert on-site with the developers.

Coding standards If code is genuinely to be shared, then there must be common, accepted, coding standards to support the understanding and ease of modification of the code.

Limitations of XP

The successful use of XP is based on certain conditions. If these do not exist, then its practice could be difficult. These conditions include the following.

- There must be easy access to users, or at least a customer representative who is a domain expert. This may be difficult where developers and users belong to different organizations.
- Development staff need to be physically located in the same office.
- As users find out about how the system will work only by being presented with working versions of the code, there may be communication problems if the application does not have a visual interface.
- For work to be sequenced into small iterations of work, it must be possible to break the system functionality into relatively small and self-contained components.
- Large, complex systems may initially need significant architectural effort. This might preclude the use of XP.

XP does also have some intrinsic potential problems

- There is a reliance on high-quality developers which makes software development vulnerable if staff turnover is significant.
- Even where staff retention is good, once an application has been developed and implemented, the tacit, personal, knowledge of the system may decay. This might make it difficult, for example, for maintenance staff without documentation to identify which bits of the code to modify to implement a change in requirements.
- Having a repository of comprehensive and accurate test data and expected results may not be as helpful as might be expected if the rationale for particular test cases is not documented. For example, where a change is made to the code, how do you know which test cases need to be changed?
- Some software development environments have focused on encouraging code reuse as a means of improving software development productivity. Such a policy would seem to be incompatible with XP.

Scrum

In the Scrum model, projects are divided into small parts of work that can be incrementally developed and delivered over time boxes that are called sprints. The product therefore gets developed over a series of manageable chunks. Each sprint typically takes only a couple of weeks. At the end of each sprint, stakeholders and team members meet to assess the progress and the stakeholders suggest to the development team any changes and improvements they feel necessary.

In the scrum model, the team members assume three fundamental roles, viz., product owner, scrum master, and team member. The product owner is responsible for communicating the customer's vision of the product to the development team. The scrum master acts as a liaison between the product owner and the team, and facilitates the development work.