Management,
nber – 2006]

Management,
, New course]

(1) – (i) (G)

UNIT – 1 Part-1

4

SOFTWARE DEVELOPMENT LIFE CYCLE MODELS

In this Chapter

4.1 INTRODUCTION

The study of project progression over time is called *Life Cycle Modelling*. We will look at several of the more famous Life Cycle Models, including *disadvantages* and *advantages* for each.

One of the many informed decisions Software Program Managers need to make is to *determine which Life Cycle model is most appropriate for a given project.*

4.2 THE GENERAL MODEL

Software life cycle models describe phases of the software cycle and the order in which those phases are executed. There are tons of models, and many companies adopt their own, but all have very similar patterns. The general, basic model is shown below :

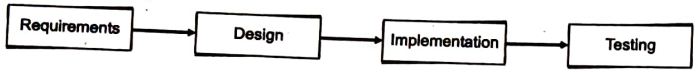

Figure 4.1 General Life Cycle Model.

Each phase produces *deliverables* required by the next phase in the life cycle. *Requirements* are translated into *design*. *Code* is produced during *implementation* that is forced by the *design*. *Testing* verifies the *deliverable* of the *implementation* phase against *requirements*.

Requirements

Business requirements are gathered in this phase. This phase is the main focus of the *project managers* and *stake holders*. Meetings with managers, stake holders and users are held in order to *determine the requirements. Who is going to use the system ? How will they use the system ? What data should be input into the system ? What data should be output by the system ?* These are general questions that get answered during a requirements gathering phase. This produces a big list of *functionality* that the system should provide, which describes *functions the system should perform, business logic that processes data, what data is stored and used by the system,* and *how the user interface should work.* The overall result is the *system as a whole* and *how it performs*, not *how it is actually going to do it.*

Design

The software system design is produced from the *results* of the requirements phase. Architects have the ball in their court during this phase and this is the phase in which their focus lies. This is where the details on *how the system will work* are produced. Architecture, including *hardware and software, communication, software design* (UML is produced here) are all part of the deliverables of a design phase.

Implementation

Code is produced from the *results* of the design phase during implementation, and this is the longest phase of the software development life cycle. For a developer, this is the main

focus of the life cycle because this is where the code is produced. Implementation by overlap with both the design and testing phases. Many tools exist (CASE tools) to actually automate the production of code using information gathered and produced during the design phase.

Testing

During testing, the implementation is tested against the requirements to make sure that the product is actually solving the needs addressed and gathered during the requirements phase. *Unit tests* and *system/acceptance tests* are done during this phase.

Unit tests act on a specific component of the system, while system tests act on the system as a whole.

So that is a very basic overview of the general software development life cycle model. Now let's study about some of the traditional and widely used variations.

4.3 WATERFALL MODEL

This is the most common and classic of life cycle models, also referred to as a *linear-sequential life cycle model*. It is very simple to understand and use. In a waterfall model, *each phase must be completed in its entirety before the next phase can begin.* At the end of each phase, a review takes place to determine if the project is on the right path and whether or not to continue or discard the project. Unlike in the general model, phases do not overlap in a waterfall model.

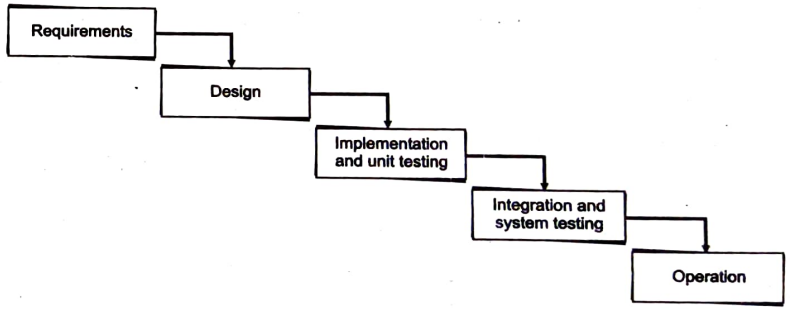

Figure 4.2 Waterfall Life Cycle Model.

Advantages

- Simple and easy to use.
- Easy to manage due to the rigidity (*Synonyms : hardness, firm, strict*) of the model – each phase has specific deliverables and a review process.
- Phases are processed and completed one at a time.
- Works well for smaller projects where requirements are very well understood.

Disadvantages

- Adjusting scope during the life cycle can kill a project.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

4.4 V-SHAPED MODEL

Just like the waterfall model, the V-Shaped life cycle is a sequential path of execution processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though. The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.

Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well.

The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.

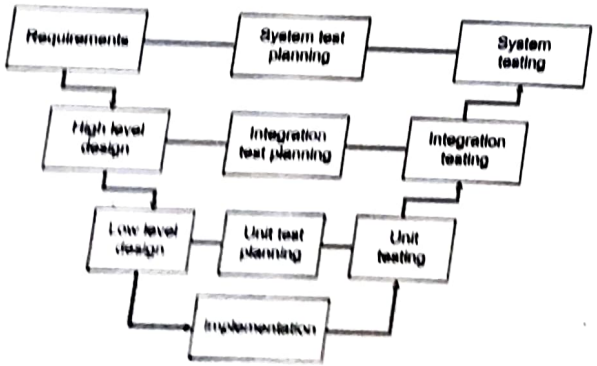

Figure 4.3 V-Shaped Life Cycle Model.

Advantages

- Simple and easy to use.
- Each phase has specific deliverables.
- Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
- Works well for small projects where requirements are easily understood.

Disadvantages

- Very tough, like the waterfall model.
- Little flexibility and adjusting scope is difficult and expensive.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- Model doesn't provide a clear path for problems found during testing phases.

4.5 INCREMENTAL MODEL

The *incremental model* is an apprehensive (Synonyms : conception, arrest, capture) approach to the waterfall model. Multiple development cycles take place here, making the life cycle a *multi-waterfall* cycle. Cycles are divided up into *smaller, more easily managed iterations*. Each iteration passes through the requirements, design, implementation and testing phases.

A working version of software is produced during the first iteration, so we have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

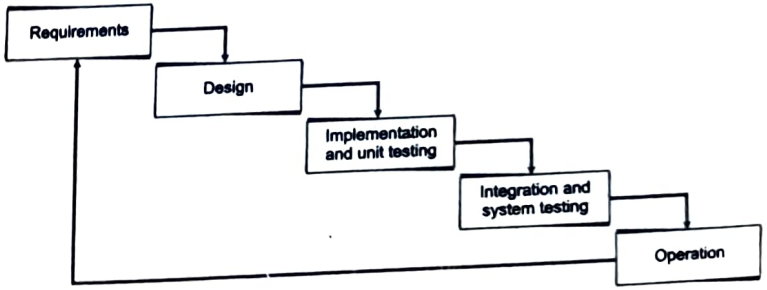

Figure 4.4 Incremental Life Cycle Model.

Advantages

- It generates working software quickly and early during the software life cycle.
- It is more flexible thus less costly to change scope and requirements.

Disadvantages

- Adjusting scope during the life cycle can kill a project.
- No working software is produced until late during the life cycle.
- High amounts of risk and uncertainty.
- Poor model for complex and object-oriented projects.
- Poor model for long and ongoing projects.
- Poor model where requirements are at a moderate to high risk of changing.

4.4 V-SHAPED MODEL

Just like the waterfall model, the V-Shaped life cycle is a *sequential path of execution of processes. Each phase must be completed before the next phase begins. Testing is emphasized in this model more so than the waterfall model though. The testing procedures are developed early in the life cycle before any coding is done, during each of the phases preceding implementation.*

Requirements begin the life cycle model just like the waterfall model. Before development is started, a system test plan is created. The test plan focuses on meeting the functionality specified in the requirements gathering.

The high-level design phase focuses on system architecture and design. An integration test plan is created in this phase as well in order to test the pieces of the software systems ability to work together.

The low-level design phase is where the actual software components are designed, and unit tests are created in this phase as well.

The implementation phase is, again, where all coding takes place. Once coding is complete, the path of execution continues up the right side of the V where the test plans developed earlier are now put to use.

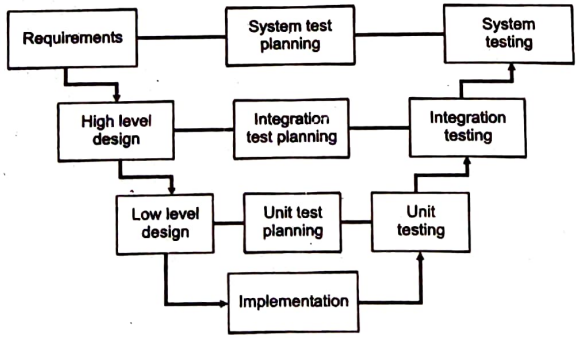

Figure 4.3 V-Shaped Life Cycle Model.

Advantages

- Simple and easy to use.
- Each phase has specific deliverables.
- Higher chance of success over the waterfall model due to the development of test plans early on during the life cycle.
- Works well for small projects where requirements are easily understood.

Disadvantages

- Very tough, like the waterfall model.
- Little flexibility and adjusting scope is difficult and expensive.
- Software is developed during the implementation phase, so no early prototypes of the software are produced.
- Model doesn't provide a clear path for problems found during testing phases.

4.5 INCREMENTAL MODEL

The *incremental model* is an apprehensive (Synonyms : conception, arrest, capture) approach to the waterfall model. Multiple development cycles take place here, making the life cycle a *multi-waterfall* cycle. Cycles are divided up into *smaller, more easily managed iterations*. Each iteration passes through the requirements, design, implementation and testing phases.

A working version of software is produced during the first iteration, so we have working software early on during the software life cycle. Subsequent iterations build on the initial software produced during the first iteration.

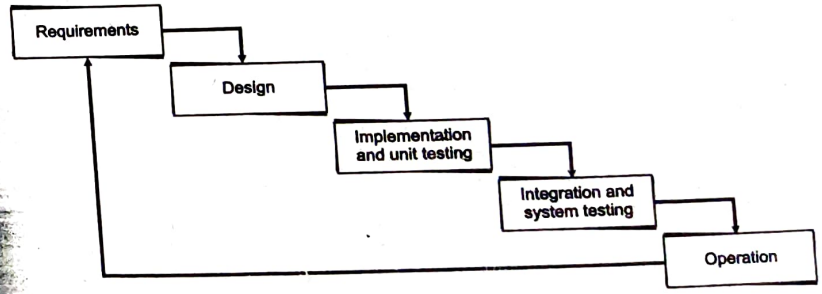

Figure 4.4 Incremental Life Cycle Model.

Advantages

- It generates working software quickly and early during the software life cycle.
- It is more flexible thus less costly to change scope and requirements.

- Easier to test and debug during a smaller iteration.
- Easier to manage risk because risky pieces are identified and handled during its iteration.
- Each iteration is an easily managed milestone.

Disadvantages

- Each phase of an iteration is hard and do not overlap each other.
- Problems may arise belong to system architecture because not all requirements are gathered up front for the entire software life cycle.

4.6 SPIRAL MODEL

The *spiral model* is *similar* to the incremental model, with more emphases placed on risk analysis. The spiral model has *four* phases : *Planning, Risk Analysis, Engineering* and *Evaluation*. A software project repeatedly passes through these phases in iterations (called Spirals in this model). *The baseline spiral, starting in the planning phase, requirements are gathered and risk is assessed (Synonyms : estimate, compute, rate)*. Each subsequent spiral builds on the baseline spiral.

Requirements are gathered during the planning phase. In the risk analysis phase, a process is undertaken to identify risk and alternate solutions. A prototype is produced at the end of the risk analysis phase.

Software is produced in the engineering phase, along with testing at the end of the phase. The evaluation phase allows the customer to evaluate the output of the project to date before the project continues to the next spiral.

In the spiral model, the *angular component* represents *progress*, and the *radius* of the spiral represents *cost*.

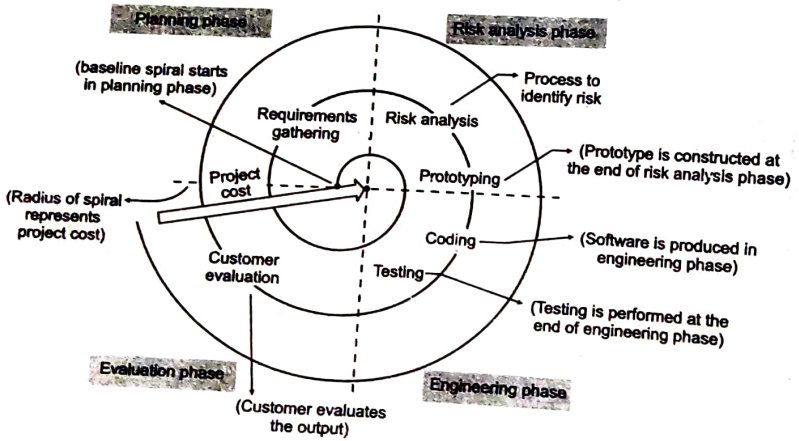

Figure 4.5 Spiral Life Cycle Model.

Advantages

- High amount of risk analysis
- Good for large and mission-critical projects.
- Software is produced early in the software life cycle.

Disadvantages

- Can be a costly model to use.
- Risk analysis requires highly specific expertise.
- Project's success is highly dependent on the risk analysis phase.
- Doesn't work well for smaller projects.

4.7 SOFTWARE PROTOTYPING

The goal of *prototyping* based development is to *oppose the limitations of the waterfall model*. The basic idea here is that *rather than freezing the requirements before a design or coding can proceed, a throwaway prototype is built to understand the requirements*. This prototype is developed based on the currently known requirements. Development of the prototype obviously undergoes design, coding and testing. But each of these phases is not done very formally or thoroughly. By using this prototype, the client can get an *actual feel* of the system, since the interactions with prototype can enable the client to better understand the requirements of the desired system.

Prototyping is an attractive idea for complicated and large systems for which there is no manual process or existing system to help determining the requirements. In such situations letting the client *plan with the prototype* provides invaluable and intangible *(Synonyms : indefinite, imperceptible)* inputs which helps in determining the requirements for the system. It is also an effective method to demonstrate the feasibility of a certain approach. This might be needed for new systems where it is not clear that compulsions can be met or that *algorithms* can be developed to implement the requirements. The process model of the prototyping approach is shown in the figure below.

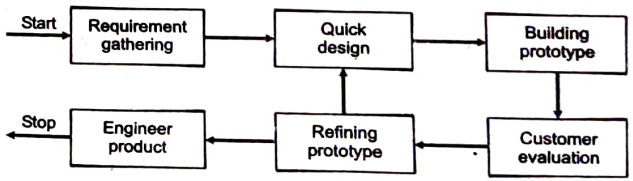

Figure 4.6 Prototyping Model.

The basic reason for little common use of prototyping is the cost involved in this *built-it-twice* approach. However, some argue that prototyping need not be very costly and can actually reduce the overall development cost. The prototypes are usually not complete systems and many of the details are not built in the prototype. *The goal is to provide a system with overall functionality.* In addition, the cost of testing and writing detailed documents is reduced. These factors help to reduce the cost of developing the prototype. On the other hand, the experience of *developing the prototype will be very useful for developers when developing the final system.* This experience helps to *reduce the cost of development of the final system* and *results in a more reliable and better designed system.*

Advantages

- Users are actively involved in the development
- It provides a better system to users, as users have natural tendency to change their mind in specifying requirements and this method of developing systems supports this user tendency.
- Since in this methodology a working model of the system is provided, the users get a better understanding of the system being developed.
- Errors can be detected much earlier as the system is mode side by side.
- Quicker user feedback is available leading to better solutions.

Disadvantages

- Leads to implementing and then repairing way of building systems.
- Practically, this methodology may increase the complexity of the system as scope of the system may expand beyond original plans.

4.7.1 Prototype Models Types

There are *four* types of Prototype Models based on their development planning : the *Patch-Up* Prototype, *Nonoperational* Prototype, *First-of-a-Series* Prototype and *Selected Features* Prototype.

Patch Up Prototype

This type of Prototype Model encourages cooperation of different developers. *Each developer will work on a specific part of the program. After everyone has done their part, the program will be integrated with each other resulting in a whole new program.* Since everyone is working on a different field, Patch Up Prototype is a fast development model. If each developer is highly skilled, there is no need to overlap in a specific function of work. This type of *software development model* only needs a *strong project manager* who can monitor the development of the program. The manager will control the work flow and ensure there is no overlapping o functions among different developers.