Jaipur Engineering College & Research Centre, Jaipur

Notes

Software Engineering

[3CS4 - 07]

Prepared By:
Manju Vyas
Abhishek Jain
Geerija Lavania

VISION AND MISSION OF INSTITUTE

VISION

To become renowned centre of outcome based learning and work towards academic, professional, cultural and social enrichments of the lives of individual and communities”

MISSION

M1. Focus on evaluation of learning outcomes and motivate students to inculcate research aptitude by project based learning.

M2. Identify areas of focus and provide platform to gain knowledge and solutions based on informed perception of Indian, regional and global needs.

M3. Offer opportunities for interaction between academia and industry.

M4. Develop human potential to its fullest extent so that intellectually capable and imaginatively gifted leaders can emerge in a range of professions.

VISION AND MISSION OF DEPARTMENT

VISION

To become renowned Centre of excellence in computer science and engineering and make competent engineers & professionals with high ethical values prepared for lifelong learning.

MISSION

M1: To impart outcome based education for emerging technologies in the field of computer science and engineering.

M2: To provide opportunities for interaction between academia and industry.

M3: To provide platform for lifelong learning by accepting the change in technologies

M4: To develop aptitude of fulfilling social responsibilities.

COURSE OUTCOMES

CO1) understand the purpose of designing a system and evaluate the various models suitable as per its requirement analysis

CO2) understand and apply software project management, effort estimation and project scheduling.

CO3) formulate requirement analysis, process behaviour and software designing.

CO4) Implement the concept of object oriented analysis modelling with the reference of UML and advance SE tools

Program Outcomes (PO)

1. **Engineering knowledge:** Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.
2. **Problem analysis**: Identify, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
3. **Design/development of solutions**: Design solutions for complex engineering problemsand design system components or processes that meet thespecified needs with appropriate consideration for the public health and safety, andthe cultural, societal, and environmental considerations.
4. **Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.
5. **Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6. **The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issuesand the consequent responsibilities relevant to the professional engineering practice.
7. **Environment and sustainability**: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
8. **Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
9. **Individual and team work**: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10. **Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11. **Project management and finance**: Demonstrate knowledge and understanding of the engineering and management principles and apply these to one’s own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12. **Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

Program Educational Objectives (PEO)

1.To provide students with the fundamentals of Engineering Sciences with more emphasis in Computer Science & Engineering by way of analyzing and exploiting engineering challenge

2. To train students with good scientific and engineering knowledge so as to comprehend, analyze, design, and create novel products and solutions for the real life problems.

3. To inculcate professional and ethical attitude, effective communication skills, teamwork skills, multidisciplinary approach, entrepreneurial thinking and an ability to relate engineering issues with social issues.

4. To provide students with an academic environment aware of excellence, leadership, written ethical codes and guidelines, and the self-motivated life-long learning needed for a successful professional career.

5. To prepare students to excel in Industry and Higher education by Educating Students along with High moral values and Knowledge.

MAPPING CO-PO

Cos/POs	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12
CO1	3	3	3	3	3	2	1	2	1	1	2	3
CO2	3	3	3	3	2	2	1	2	2	2	3	3
CO3	3	3	3	2	2	2	1	2	1	2	2	3
CO4	3	3	3	3	3	1	0	1	1	2	2	3

PSO

PSO1: Ability to interpret and analyze network specific and cyber security issues, automation in real word environment.

PSO2: Ability to Design and Develop Mobile and Web-based applications under realistic constraints.

SYLLABUS

UNIT 1: Introduction, software life-cycle models, software requirements specification, formal requirements specification, verification and validation.

UNIT 2: Software Project Management: Objectives, Resources and their estimation, LOC and FP estimation, effort estimation, COCOMO estimation model, risk analysis, software project scheduling.

UNIT 3: Requirement Analysis: Requirement analysis tasks, Analysis principles. Software prototyping and specification data dictionary, Finite State Machine (FSM) models. **Structured Analysis**: Data and control flow diagrams, control and process specification behavioral modeling

UNIT 4: Software Design: Design fundamentals, Effective modular design: Data architectural and procedural design, design documentation.

UNIT 5: Object Oriented Analysis: Object oriented Analysis Modeling, Data modeling. **Object Oriented Design**: OOD concepts, Class and object relationships, object modularization, Introduction to Unified Modeling Language

UNIT - 1 - INTRODUCTION TO SOFTWARE ENGINEERING

The term software engineering is composed of two words, software and engineering.

Software is more than just a program code. A program is an executable code, which serves some computational purpose. Software is considered to be a collection of executable programming code, associated libraries and documentations. Software, when made for a specific requirement is called **software product.**

Engineering on the other hand, is all about developing products, using well-defined, scientific principles and methods. So, we can define software engineering as an engineering branch associated with the development of software product using well-defined scientific principles, methods and procedures. **The outcome of software engineering is an efficient and reliable software product.**

IEEE defines software engineering as: The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software.

We can alternatively view it as a systematic collection of past experience. The experience is arranged in the form of methodologies and guidelines. A small program can be written without using software engineering principles. But if one wants to develop a large software product, then software engineering principles are absolutely necessary to achieve a good quality software cost effectively.

[Reference - R1]

NEED OF SOFTWARE ENGINEERING

The need of software engineering arises because of higher rate of change in user requirements and environment on which the software is working.

Large software - It is easier to build a wall than to a house or building, likewise, as the size of software become large engineering has to step to give it a scientific process.

Scalability- If the software process were not based on scientific and engineering concepts, it would be easier to re-create new software than to scale an existing one.

Cost- As hardware industry has shown its skills and huge manufacturing has lower down the price of computer and electronic hardware. But the cost of software remains high if proper process is not adapted.

Dynamic Nature- The always growing and adapting nature of software hugely depends upon the environment in which the user works. If the nature of software is always changing, new enhancements need to be done in the existing one. This is where software engineering plays a good role.

Quality Management- Better process of software development provides better and quality software product. [Reference - R1]

CHARACTERESTICS OF GOOD SOFTWARE

A software product can be judged by what it offers and how well it can be used. This software must satisfy on the following grounds:

- **Operational**
- **Transitional**
- **Maintenance**

Well-engineered and crafted software is expected to have the following characteristics:

Operational: This tells us how well software works in operations. It can be measured on:

- Budget
- Usability
- Efficiency

- Correctness
- Functionality
- Dependability
- Security
- Safety

Transitional: This aspect is important when the software is moved from one platform to another:

- Portability
- Interoperability
- Reusability
- Adaptability

Maintenance: This aspect briefs about how well a software has the capabilities to maintain itself in the ever-changing environment:

- Modularity
- Maintainability
- Flexibility
- Scalability

In short, Software engineering is a branch of computer science, which uses well-defined engineering concepts required to produce efficient, durable, scalable, in-budget and on-time software products.

[Reference - R1]

LAYERS OF SOFTWARE ENGINEERING

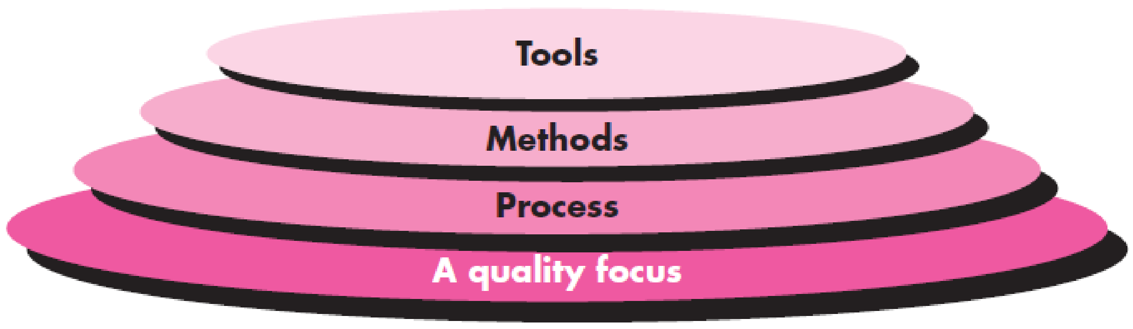


Figure: Layers Of Software Engineering

Software engineering is a layered technology. Software engineering must rest on an organizational commitment to **quality**. Total quality management, Six Sigma, and similar philosophies foster a continuous process improvement culture, and it is this culture that ultimately leads to the development of increasingly more effective approaches to software engineering. The bedrock that supports software engineering is a **quality focus.**

A **Software engineering process** is *not* a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks. The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

Software engineering methods provide the technical how-to’s for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support. Software engineering methods rely on a set of basic principles that govern each area of the technology and include modeling activities and other descriptive techniques.

Software engineering tools provide automated or semi-automated support for the process and the methods. When tools are integrated so that information created by one tool can be used by another, a system for the support of software development, called computer-aided software engineering, is established.

[Reference - R2]

THE SOFTWARE PROCESS

A **process** is a collection of activities, actions, and tasks that are performed when some work product is to be created.

An **activity** strives to achieve a broad objective (e.g., communication with stakeholders) and is applied regardless of the application domain, size of the project, complexity of the effort, or degree of rigor with which software engineering is to be applied.

An **action** (e.g., architectural design) encompasses a set of tasks that produce a major work product (e.g., an architectural design model).

A **task** focuses on a small, but well-defined objective (e.g., conducting a unit test) that produces a tangible outcome.

In the context of software engineering, a process is not a rigid prescription for how to build computer software. Rather, it is an adaptable approach that enables the people doing the work (the software team) to pick and choose the appropriate set of work actions and tasks.

The intent is always to deliver software in a timely manner and with sufficient quality to satisfy those who have sponsored its creation and those who will use it.

[Reference - R3]

A GENERIC PROCESS FRAMEWORK FOR SOFTWARE ENGINEERING: A **process framework** establishes the foundation for a complete software engineering process by identifying a small number of framework activities that are applicable to all software projects, regardless of their size or complexity.

In addition, the process framework encompasses a set of umbrella activities that are applicable across the entire software process.

A generic process framework for software engineering encompasses five activities:

Communication: Before any technical work can commence, it is critically important to communicate and collaborate with the customer and other stakeholders and stakeholders' objectives for the project and to gather requirements that help define software features and functions.

Planning: Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity creates a "map" that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.

Modeling: Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a "sketch" of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

Construction: This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.

Deployment: The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

These five generic framework activities can be used during the development of small, simple programs, the creation of large Web applications, and for the engineering of large, complex computer-based systems. The details of the software process will be quite different in each case, but the framework activities remain the same.

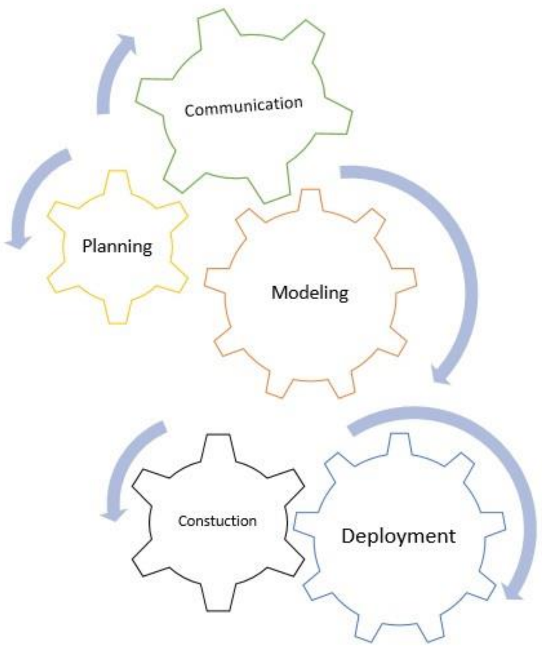


Figure: Activities in Generic process framework for software engineering

[Reference - R3 & R4]

SOFTWARE LIFE CYCLE MODELS/SOFTWARE DEVELOPMENT LIFE CYCLE MODEL (SWDLC MODELS)

A software life cycle model (also called process model) is a descriptive and diagrammatic representation of the software life cycle.

A life cycle model represents all the activities required to make a software product transit through its life cycle phases. It also captures the order in which these activities are to be undertaken. In other words, a life cycle model maps the different activities performed on a software product from its inception to retirement.

Different life cycle models may map the basic development activities to phases in different ways. Thus, no matter which life cycle model is followed, the basic activities are included in all life cycle models though the activities may be carried out in different orders in different life cycle models. During any life cycle phase, more than one activity may also be carried out.

THE NEED FOR A SOFTWARE LIFE CYCLE MODEL

The development team must identify a suitable life cycle model for the particular project and then adhere to it. Without using of a particular life cycle model the development of a software product would not be in a systematic and disciplined manner.

When a software product is being developed by a team there must be a clear understanding among team members about when and what to do. Otherwise it would lead to chaos and project failure.

This problem can be illustrated by using an example. Suppose a software development problem is divided into several parts and the parts are assigned to the team members. From then on, suppose the team members are allowed the freedom to develop the parts assigned to them in whatever way they like. It is possible that one member might start writing the code for his part, another might decide to prepare the test documents first, and some other engineer might begin with the design phase of the parts assigned to him. This would be one of the perfect recipes for project failure.

A software life cycle model defines entry and exit criteria for every phase. A phase can start only if its phase-entry criteria have been satisfied. So without software life cycle model the entry and exit criteria for a phase cannot be recognized. Without software life cycle models it becomes difficult for software project managers to monitor the progress of the project.

Different software life cycle models

Many life cycle models have been proposed so far. Each of them has some advantages as well as some disadvantages. A few important and commonly used life cycle models are as follows:

Types of Software developing life cycles (SDLC)

- Waterfall Model

- Iterative Waterfall Model
- The V – Model
- Incremental Model
- RAD Model
- Prototyping Model
- Spiral Method
- Concurrent Development Model

[Reference - R5]

WATERFALL MODEL

The Waterfall Model is a **LINEAR SEQUENTIAL MODEL**. In which progress is seen as flowing steadily downwards (like a waterfall) through the phases of software implementation. This means that any phase in the development process begins only if the previous phase is complete. The waterfall approach does not define the process to go back to the previous phase to handle changes in requirement. The waterfall approach was the earliest approach and most widely known that was used for software development.

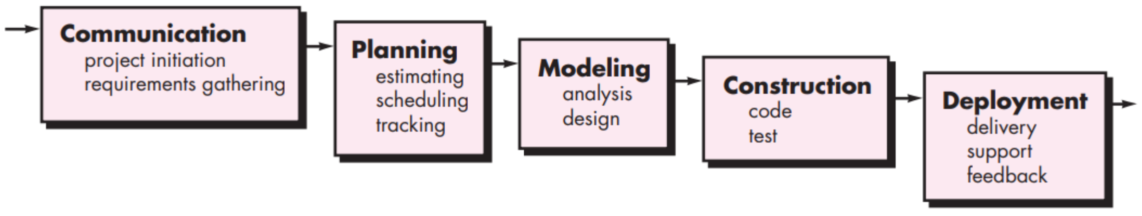


Figure: Generic Waterfall Model

Phases of Waterfall model: All work flows from **communication** towards **deployment** in a reasonably linear fashion.

- **Communication**: In the communication phase, the major task performed is requirement gathering which helps in finding out the exact need of the customer. Once all the needs of the customer are gathered the next step is planning.

- **Planning:** In planning major activities like planning for schedule, keeping tracks on the processes and the estimation related to the project are done. Planning is even used to find the types of risks involved throughout the projects. Planning describes how technical tasks are going to take place and what resources are needed and how to use them.

- **Modeling:** This is one of the important phases of the architecture of the system is designed in this phase. An analysis is carried out and depending on the analysis a software model is designed. Different models for developing software are created depending on the requirements gathered in the first phase and the planning done in the second phase.

- **Construction:** The actual coding of the software is done in this phase. This coding is done based on the model designed in the modeling phase. So, in this phase software is developed and tested..

- **Deployment:** In this last phase, the product is rolled out or delivered & installed at the customer's end and support is given if required. Feedback is taken from the customer to ensure the quality of the product.

Advantages

1. Easy to explain to the users
2. Structures approach.
3. Stages and activities are well defined.
4. Helps to plan and schedule the project
5. Verification at each stage ensures early detection of errors/misunderstanding
6. Each phase has specific deliverables.

Disadvantages

1. Very difficult to go back to any stage after it finished
2. Costly and required more time, in addition to the detailed plan.
3. Assumes that the requirements of a system can be frozen
4. A little flexibility and adjusting scope is difficult and expensive

ITERATIVE WATERFALL MODEL/ WATERFALL MODEL WITH FEEDBACK

To overcome the major shortcomings of the classical waterfall model, we come up with the iterative waterfall model.

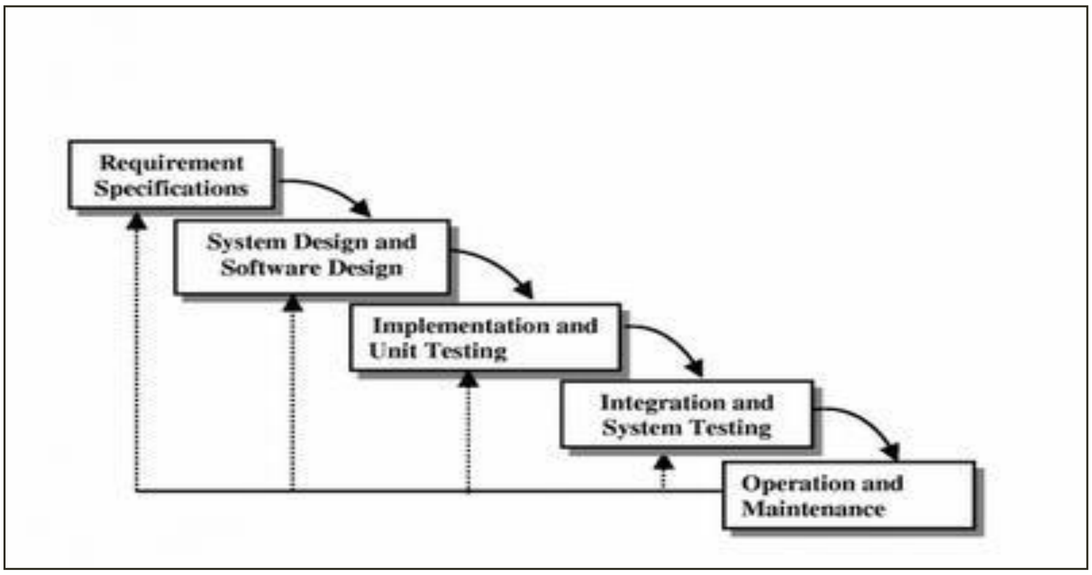

Requirement Specifications
System Design and Software Design
Implementation and Unit Testing
Integration and System Testing
Operation and Maintenance

Figure : Iterative Waterfall Model

Here, we provide feedback paths for error correction as & when detected later in a phase. Though errors are inevitable, but it is desirable to detect them in the same phase in which they occur. If so, this can reduce the effort to correct the bug.

The advantage of this model is that there is a working model of the system at a very early stage of development which makes it easier to find functional or design flaws. Finding issues at an early stage of development enables to take corrective measures in a limited budget. The disadvantage with this SWDLC model is that it is applicable only to large and bulky software development projects. This is because it is hard to break a small software system into further small serviceable increments/modules.

[Reference - R6 Page No. - 79 - 80]

[Reference - R7]

THE V-MODEL

The V-model is a type of SDLC model where process executes in a sequential manner in V-shape. It is also known as Verification and Validation model. It is based on the association of a testing phase for each corresponding development stage. Development of each step directly associated with the testing phase. The next phase starts only after completion of the previous phase i.e. for each development activity, there is a testing activity corresponding to it.

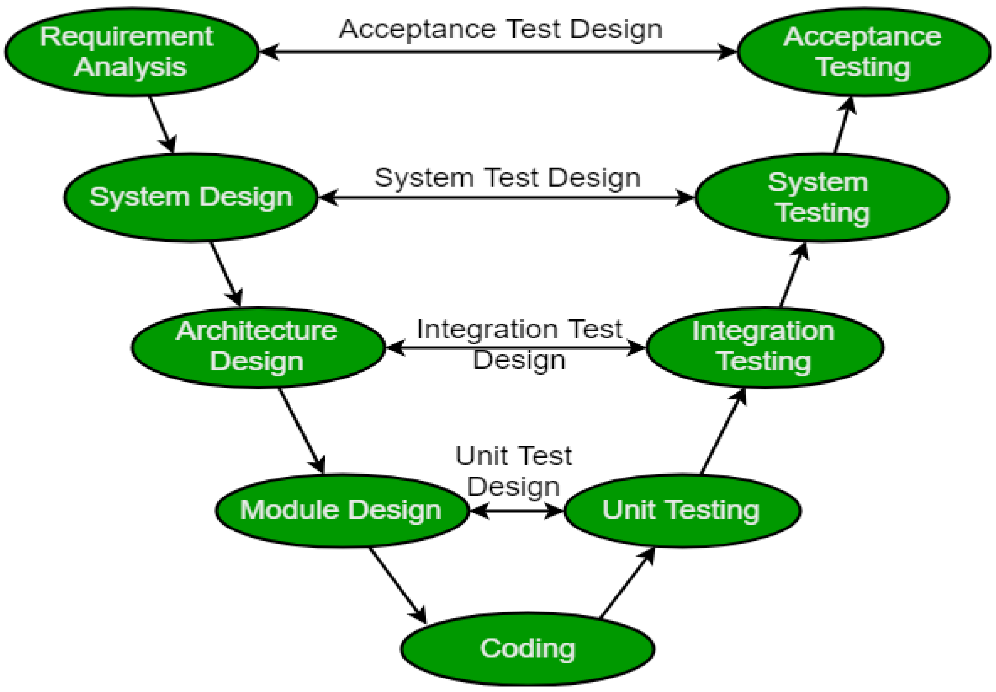


FIGURE: V- MODEL

So V-Model contains Verification phases on one side of the Validation phases on the other side. Verification and Validation phases are joined by coding phase in V-shape. Thus it is called V-Model.

Design Phase:

- **Requirement Analysis:** This phase contains detailed communication with the customer to understand their requirements and expectations. This stage is known as Requirement Gathering.
- **System Design:** This phase contains the system design and the complete hardware and communication setup for developing product.

- **Architectural Design:** System design is broken down further into modules taking up different functionalities. The data transfer and communication between the internal modules and with the outside world (other systems) is clearly understood.
- **Module Design:** In this phase the system breaks dowm into small modules. The detailed design of modules is specified, also known as Low-Level Design (LLD).

Testing Phases:

- **Unit Testing:** Unit Test Plans are developed during module design phase. These Unit Test Plans are executed to eliminate bugs at code or unit level.
- **Integration testing:** After completion of unit testing Integration testing is performed. In integration testing, the modules are integrated and the system is tested. Integration testing is performed on the Architecture design phase. This test verifies the communication of modules among themselves.
- **System Testing:** System testing test the complete application with its functionality, inter dependency, and communication. It tests the functional and non-functional requirements of the developed application.
- **User Acceptance Testing (UAT):** UAT is performed in a user environment that resembles the production environment. UAT verifies that the delivered system meets user’s requirement and system is ready for use in real world.

Advantages:

- This is a highly disciplined model and Phases are completed one at a time.
- V-Model is used for small projects where project requirements are clear.
- Simple and easy to understand and use.
- This model focuses on verification and validation activities early in the life cycle thereby enhancing the probability of building an error-free and good quality product.
- It enables project management to track progress accurately.

Disadvantages:

- High risk and uncertainty.
- It is not a good for complex and object-oriented projects.
- It is not suitable for projects where requirements are not clear and contains high risk of changing.

- This model does not support iteration of phases.
- It does not easily handle concurrent events.

[Reference - R8]

INCREMENTAL PROCESS MODELS

THE INCREMENTAL MODEL

There are many situations in which initial software requirements are well defined, but it is not possible to follow a purely linear process. In addition, there may be a need to provide a limited set of software functionality to users quickly and then refine and expand on that functionality in later software releases. In such cases, you can choose a process model that is designed to produce the software in increments.

The incremental model combines elements of linear and parallel process flows and applies linear sequences in a stepwise manner according to calendar. <u>**Each linear sequence**</u> **produces** <u>**deliverable**</u> **“*increments*”** of the software.

For example, word-processing software developed using the incremental model may deliver:

- basic file management, editing, and document production functions in the **first increment**;
- more sophisticated editing and document production capabilities in the **second increment**;
- spelling and grammar checking in the **third increment**;
- and advanced page layout capability in the **fourth increment**.

It should be noted that the process flow for any increment can incorporate the ***prototyping methods.***

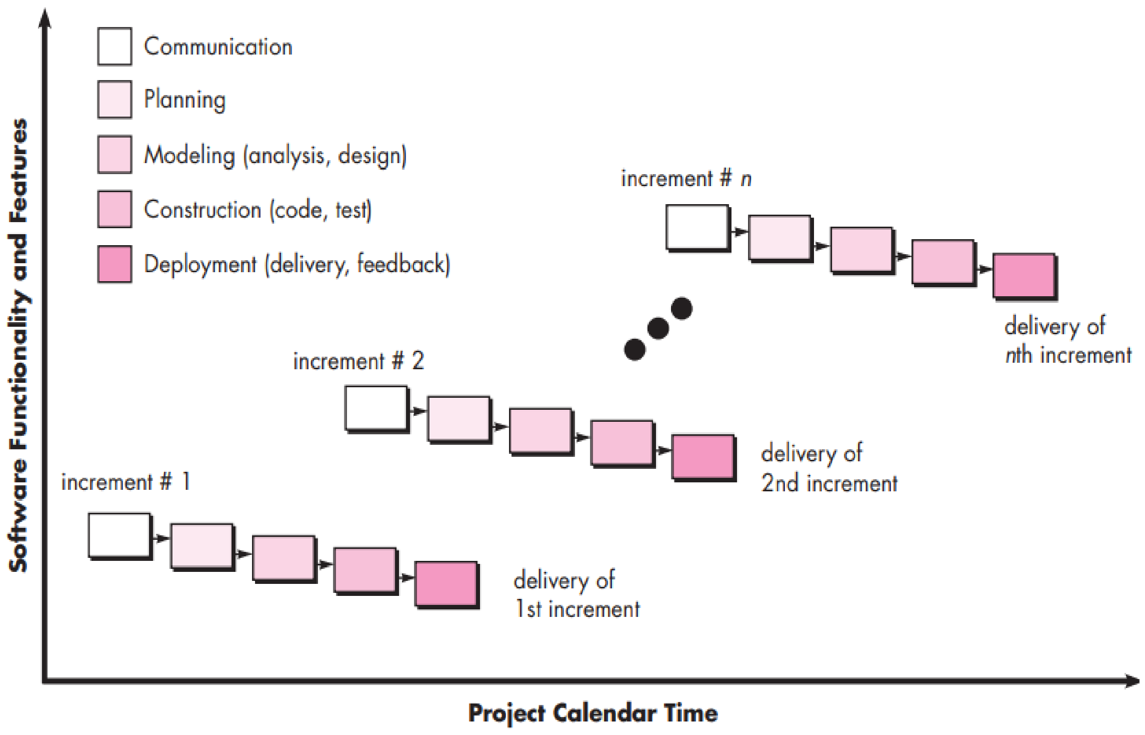

Communication
Planning
Modeling (analysis, design)
Construction (code, test)
Deployment (delivery, feedback)
increment # n
delivery of nth increment
increment # 2
delivery of 2nd increment
increment # 1
delivery of 1st increment
Software Functionality and Features
Project Calendar Time

Figure: The Incremental Model

When an incremental model is used, the **first increment is often a core product**. That is, basic requirements are addressed but many supplementary features (some known, others unknown) remain undelivered. The core product is used by the customer (or undergoes detailed evaluation). As a result of use and/or evaluation, a plan is developed for the next increment.

The plan addresses the modification of the core product to better meet the needs of the customer and the delivery of additional features and functionality. This process is repeated following the delivery of each increment, until the complete product is produced.

The incremental process model focuses on the delivery of an operational product with each increment. Early increments are stripped-down versions of the final product, but they do provide capability that serves the user and also provide a platform for evaluation by the user.

Incremental development is particularly useful when staffing is unavailable for a complete implementation by the business deadline that has been established for the project.

Early increments can be implemented with fewer people. If the core product is well received, then additional staff (if required) can be added to implement the next increment. In addition, increments can be planned to manage technical risks.

For example, a major system might require the availability of new hardware that is under development and whose delivery date is uncertain. It might be possible to plan early increments in a way that avoids the use of this hardware, thereby enabling partial functionality to be delivered to end users without inordinate delay.

Advantages –

- Error Reduction (core modules are used by the customer from the beginning of the phase and then these are tested thoroughly)
- Uses divide and conquer for breakdown of tasks.
- Lowers initial delivery cost.
- Incremental Resource Deployment.

Disadvantages –

- Requires good planning and design.
- Total cost is not lower.
- Well defined module interfaces are required.

[Reference - R6 Page No. - 80 - 81]

THE RAD (RAPID APPLICATION DEVELOPMENT) MODEL:

Rapid Application Development (RAD) is an incremental software development process model which is a “high-speed” adaptation of the linear sequential model in which rapid development is achieved by using component-based construction. If requirements are well understood and project scope is constrained, the RAD process enables a development team to create a “fully functional system” within very short time periods, such as in 60 to 90 days.

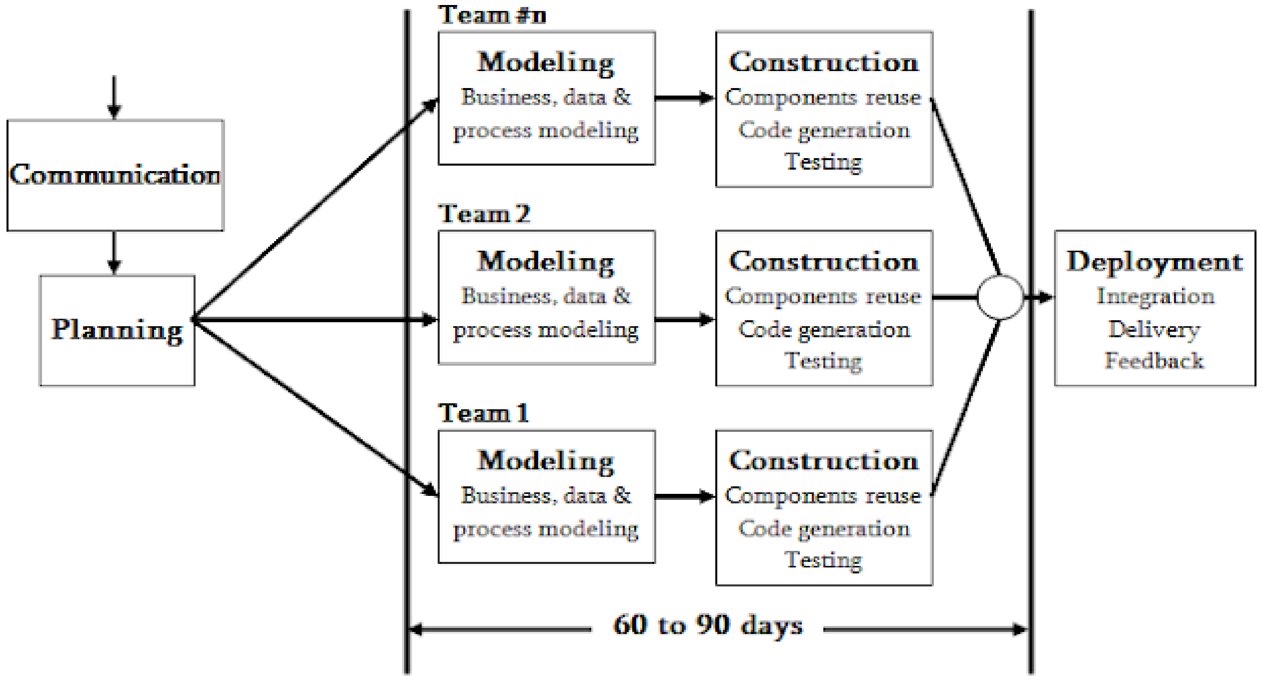


Figure : Flowchart of RAD model

(i) Communication: This step works to understand the business problems and the information characteristics that the software must accommodate.

(ii) Planning: This is very important as multiple teams work on different systems.

(iii) Modeling: Modeling includes the major phases, like business, data, process modeling and establishes design representation that serves as the basis for RAD's construction activity.

(iv) Construction: This includes the use of preexisting software components and the application of automatic code generation.

Advantages –

- Use of reusable components helps to reduce the cycle time of the project.
- Feedback from the customer is available at initial stages.
- Reduced costs as fewer developers are required.
- Use of powerful development tools results in better quality products in comparatively shorter time spans.
- The progress and development of the project can be measured through the various stages.
- It is easier to accommodate changing requirements due to the short iteration time spans.

Disadvantages –

- The use of powerful and efficient tools requires highly skilled professionals.
- The absence of reusable components can lead to failure of the project.
- The team leader must work closely with the developers and customers to close the project in time.
- The systems which cannot be modularized suitably cannot use this model.
- Customer involvement is required throughout the life cycle.
- It is not meant for small scale projects as for such cases, the cost of using automated tools and techniques may exceed the entire budget of the project.

[Reference - R6 Page No. - 82 - 83]

EVOLUTIONALRY PROCESS MODELS

PROTOTYPING MODEL

Prototyping: Often, a customer defines a set of general objectives for software, but does not identify detailed requirements for functions and features. In other cases, the developer may be unsure of the efficiency of an algorithm, the adaptability of an operating system, or the form that human-machine interaction should take. In these, and many other situations, a prototyping paradigm may offer the best approach.

Although prototyping can be used as a stand-alone process model, it is more commonly used as a technique that can be implemented within the context of any one of the process models.

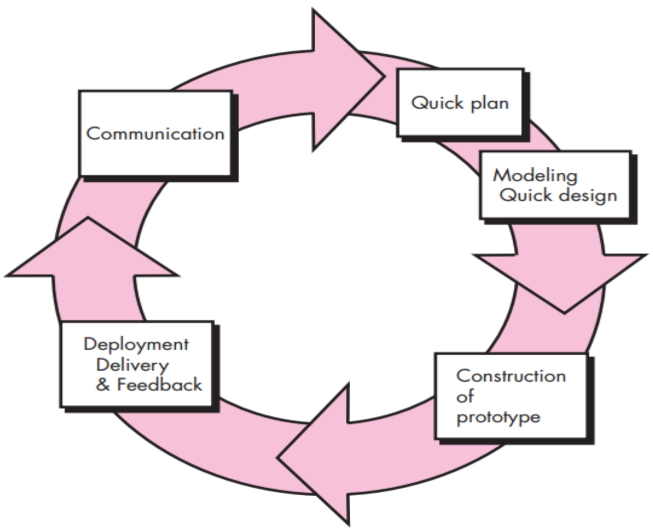


Figure: Prototyping

The prototyping paradigm begins with communication. You meet with other stakeholders to define the overall objectives for the software, identify whatever requirements are known, and outline areas where further definition is mandatory.

A prototyping iteration is planned quickly, and modeling (in the form of a "quick design") occurs. A quick design focuses on a representation of those aspects of the software that will be visible to end users (e.g., human interface layout or output display formats).

The quick design leads to the construction of a prototype. The prototype is deployed and evaluated by stakeholders, who provide feedback that is used to further refine requirements. Iteration occurs as the prototype is tuned to satisfy the needs of various stakeholders, while at the same time enabling you to better understand what needs to be done.

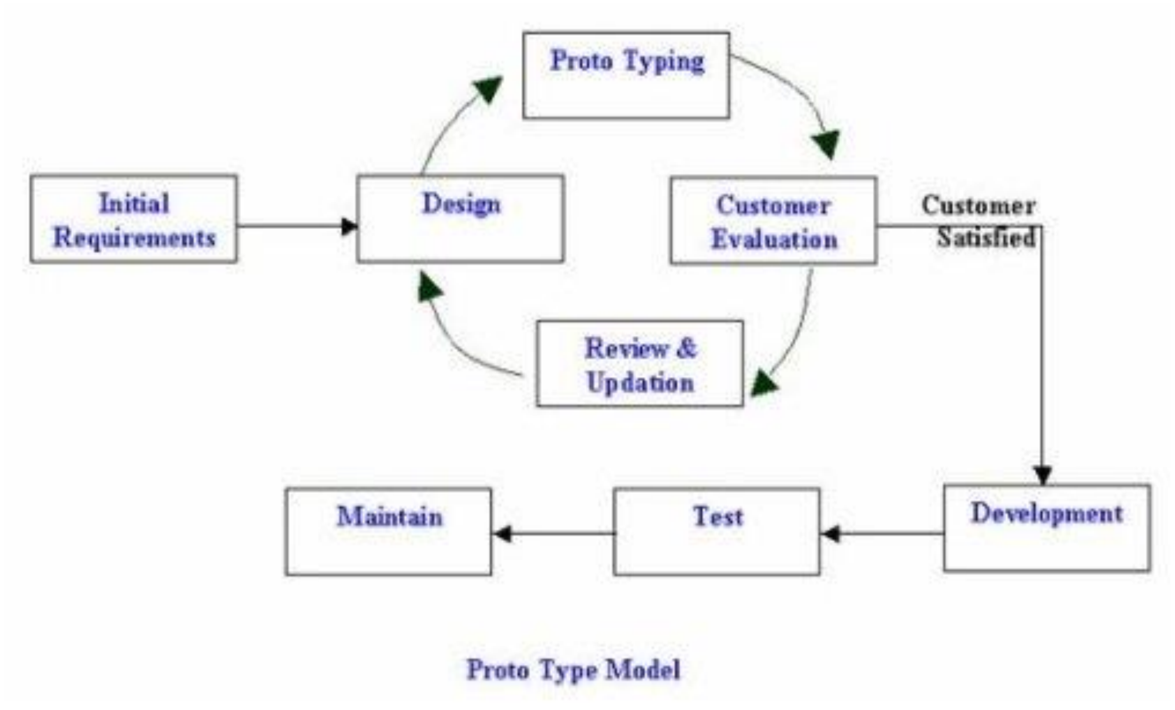


Figure: Prototype Model

Advantages –

- The customers get to see the partial product early in the life cycle. This ensures a greater level of customer satisfaction and comfort.

- New requirements can be easily accommodated as there is scope for refinement.
- Missing functionalities can be easily figured out.
- Errors can be detected much earlier thereby saving a lot of effort and cost, besides enhancing the quality of the software.
- The developed prototype can be reused by the developer for more complicated projects in the future.
- Flexibility in design.

Disadvantages –

- Costly w.r.t time as well as money.
- There may be too much variation in requirements each time the prototype is evaluated by the customer.
- Poor Documentation due to continuously changing customer requirements.
- It is very difficult for the developers to accommodate all the changes demanded by the customer.
- There is uncertainty in determining the number of iterations that would be required before the prototype is finally accepted by the customer.
- After seeing an early prototype, the customers sometimes demand the actual product to be delivered soon.
- Developers in a hurry to build prototypes may end up with sub-optimal solutions.
- The customer might lose interest in the product if he/she is not satisfied with the initial prototype.

[Reference - R6 Page No. - 83 - 85]

THE SPIRAL MODEL

Originally proposed by Barry Boehm, the spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the waterfall model.

It provides the potential for rapid development of increasingly more complete versions of the software. Boehm describes the model in the following manner:

"The spiral development model is a risk-driven process model generator that is used to guide multi-stakeholder concurrent engineering of software intensive systems. It has two main distinguishing features. One is a cyclic approach for incrementally growing a system's degree of definition and implementation while decreasing its degree of risk. The other is a set of anchor point milestones for ensuring stakeholder commitment to feasible and mutually satisfactory system solutions."

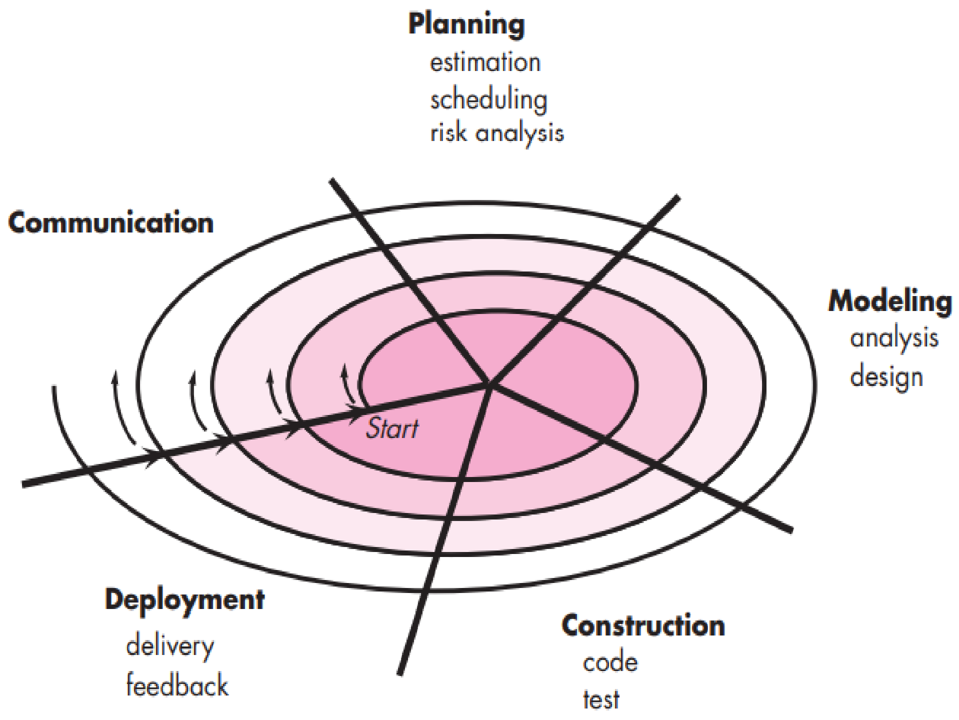


FIGURE: TYPICAL SPIRAL MODEL

[NOTE: The arrows pointing inward along the axis separating the deployment region from the communication region indicate a potential for local iteration along the same spiral path.]

Using the spiral model, software is developed in a series of evolutionary releases. During early iterations, the release might be a model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced.

A spiral model is divided into a set of **framework activities** defined by the software engineering team. Each of the framework activities represent one **segment** of the spiral path illustrated in Figure. The spiral model can be adapted to apply throughout the life of the computer software.

SEGMENTS:

SEGMENT 1 includes following activities: **Communication** (requirement gathering, customer evaluation and understanding)

SEGMENT 2 includes following activities: **Planning** (estimation, scheduling and risk analysis)

SEGMENT 3 includes following activities: **Modeling** (analysis and design)

SEGMENT 4 includes following activities: **Construction** (coding and testing)

SEGMENT 5 includes following activities: **Deployment** (delivery and feedback)

CIRCUITS AROUND THE SPIRAL:

As this evolutionary process begins, the software team performs activities that are implied by a **circuit** around the spiral in a clockwise direction, beginning at the center.

Risk is considered as each revolution is made.

Anchor point milestones, a combination of work products and conditions that are attained along the path of the spiral, are noted for each evolutionary **pass**.

The **first circuit** around the spiral might result in the development of a product specification and concept development of project, that starts at the core of the spiral and continues for multiple iterations until concept development is complete.

subsequent passes around the spiral might be used to develop a prototype and then progressively more sophisticated versions of the software.

Each pass through the planning region results in adjustments to the project plan.

Cost and schedule are adjusted based on feedback derived from the customer after delivery.

In addition, the project manager adjusts the planned number of iterations required to complete the software.

The **version** or **build** or **deliverable** produced at the end of Deployment phase of **the last circuit**, is the **final software product.**

Advantages of Spiral Model: Below are some of the advantages of the Spiral Model.

- **Risk Handling:** The projects with many unknown risks that occur as the development proceeds, in that case, Spiral Model is the best development model to follow due to the risk analysis and risk handling at every phase.
- **Good for large projects:** It is recommended to use the Spiral Model in large and complex projects.
- **Flexibility in Requirements:** Change requests in the Requirements at later phase can be incorporated accurately by using this model.
- **Customer Satisfaction:** Customer can see the development of the product at the early phase of the software development and thus, they habituated with the system by using it before completion of the total product.

Disadvantages of Spiral Model: Below are some of the main disadvantages of the spiral model.

- **Complex:** The Spiral Model is much more complex than other SDLC models.
- **Expensive:** Spiral Model is not suitable for small projects as it is expensive.
- **Too much dependable on Risk Analysis:** The successful completion of the project is very much dependent on Risk Analysis. Without very highly experienced expertise, it is going to be a failure to develop a project using this model.
- **Difficulty in time management:** As the number of phases is unknown at the start of the project, so time estimation is very difficult.

[Reference - R6 Page No. - 86 - 88]

CONCURRENT DEVELOPMENT MODEL

The concurrent development model, sometimes called concurrent engineering.

- It allows a software team to represent iterative and concurrent elements of any of the process model.
- For example, the modeling activity defined for the spiral model is accomplished by invoking one or more of the software engineering actions: prototyping, analysis, and design.
- The activity—modeling—may be in any one of the states noted at any given time.
- Similarly, other activities, actions, or tasks (e.g., communication or construction) can be represented in an similar manner.

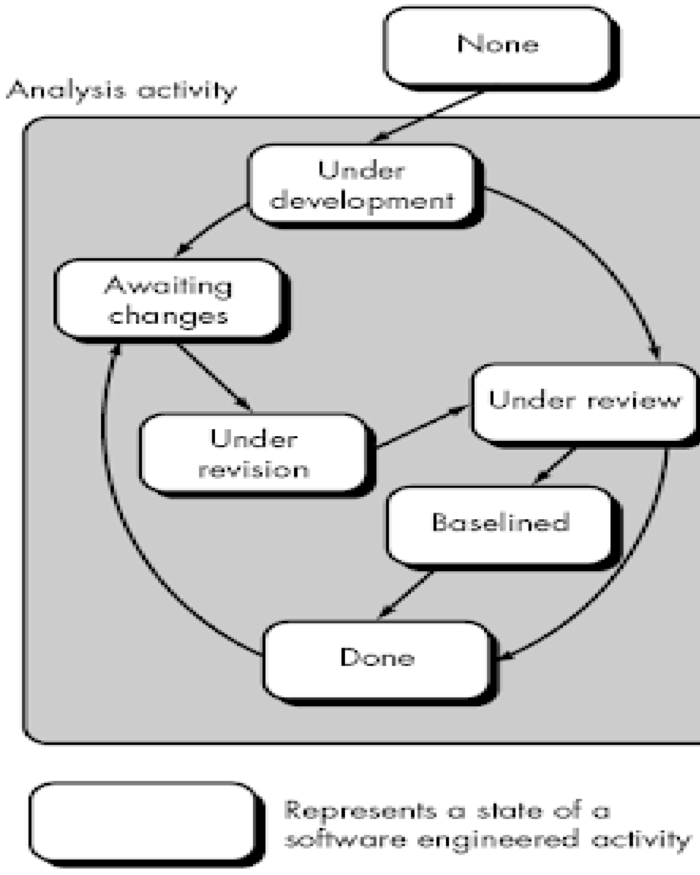


FIGURE: CONCURRENT DEVELOPMENT MODEL

- **All software engineering activities exist concurrently but reside in different states.**
- For example, early in a project the communication activity (not shown in the figure) has completed its first iteration and exists in the awaiting changes state.

- The modeling activity (which existed in the inactive state while initial communication was completed, now makes a transition into the under development state. If, however, the customer indicates that changes in requirements must be made, the modeling activity moves from the under development state into the awaiting changes state.
- Concurrent modeling defines a series of events that will trigger transitions from state to state for each of the software engineering activities, actions, or tasks.

Advantages of the concurrent development model

- This model is applicable to all types of software development processes.
- It is easy for understanding and use.
- It gives immediate feedback from testing.
- It provides an accurate picture of the current state of a project.

Disadvantages of the concurrent development model

- It needs better communication between the team members. This may not be achieved all the time.
- It requires to remember the status of the different activities.

[Reference - R6 Page No. - 88 - 89]

SOFTWARE REQUIREMENTS SPECIFICATION (SRS)

A software requirements specification (SRS) is a detailed description of a software system to be developed with its functional and non-functional requirements.

The SRS is developed based the agreement between customer and contractors. It may include the use cases of how user is going to interact with software system. The software requirement specification document consistent of all necessary requirements required for project development.

To develop the software system we should have clear understanding of Software system. To achieve this we need to continuous communication with customers to gather all requirements.

A good SRS defines the how Software System will interact with all internal modules, hardware, communication with other programs and human user interactions with wide range of real life scenarios.

Using the Software requirements specification (SRS) document on QA lead, managers creates test plan. It is very important that testers must be cleared with every detail specified in this document in order to avoid faults in test cases and its expected results.

It is highly recommended to review or test SRS documents before start writing test cases and making any plan for testing.

Let’s see how to test SRS and the important point to keep in mind while testing it.

1. Correctness of SRS should be checked. Since the whole testing phase is dependent on SRS, it is very important to check its correctness. There are some standards with which we can compare and verify.

2. Ambiguity should be avoided. Sometimes in SRS, some words have more than one meaning and this might confused testers making it difficult to get the exact reference. It is advisable to check for such ambiguous words and make the meaning clear for better understanding.

3. Requirements should be complete. When tester writes test cases, what exactly is required from the application, is the first thing which needs to be clear. For e.g. if application needs to send the specific data of some specific size then it should be clearly mentioned in SRS that how much data and what is the size limit to send.

4. Consistent requirements. The SRS should be consistent within itself and consistent to its reference documents. If you call an input “Start and Stop” in one place, don’t call it “Start/Stop” in another. This sets the standard and should be followed throughout the testing phase.

5. Verification of expected result: SRS should not have statements like “Work as expected”, it should be clearly stated that what is expected since different testers would have different thinking aspects and may draw different results from this statement.

6. Testing environment: some applications need specific conditions to test and also a particular environment for accurate result. SRS should have clear documentation on what type of environment is needed to set up.

7. Pre-conditions defined clearly: one of the most important part of test cases is pre-conditions. If they are not met properly then actual result will always be different expected result. Verify that in SRS, all the pre-conditions are mentioned clearly.

8. Requirements ID: these are the base of test case template. Based on requirement Ids, test case ids are written. Also, requirements ids make it easy to categorize modules so just by looking at them, tester will know which module to refer. SRS must have them such as id defines a particular module.

9. Security and Performance criteria: security is priority when a software is tested especially when it is built in such a way that it contains some crucial information when leaked can cause harm to business. Tester should check that all the security related requirements are properly defined and are clear to him. Also, when we talk about performance of a software, it plays a very important role in business so all the requirements related to performance must be clear to the tester and he must also know when and how much stress or load testing should be done to test the performance.

10. Assumption should be avoided: sometimes when requirement is not cleared to tester, he tends to make some assumptions related to it, which is not a right way to do testing as assumptions could go wrong and hence, test results may vary. It is better to avoid assumptions and ask clients about all the "missing requirements" to have a better understanding of expected results.

11. Deletion of irrelevant requirements: there are more than one team who work on SRS so it might be possible that some irrelevant requirements are included in SRS. Based on the understanding of the software, tester can find out which are these requirements and remove them to avoid confusions and reduce work load.

12. Freeze requirements: when an ambiguous or incomplete requirement is sent to client to analyze and tester gets a reply, that requirement result will be updated in the next SRS version and client will freeze that requirement. Freezing here means that result will not change again until and unless some major addition or modification is introduced in the software.

Most of the defects which we find during testing are because of either incomplete requirements or ambiguity in SRS. To avoid such defects it is very important to test software requirements specification before writing the test cases. Keep the latest version of SRS with you for reference and keep yourself updated with the latest change made to the SRS. Best practice is to go through the document very carefully and note down all the confusions, assumptions and incomplete requirements and then have a meeting with the client to get them clear before development phase starts as it becomes costly to fix the bugs after the software is developed. After all the requirements are cleared to a tester, it becomes easy for him to write effective test cases and accurate expected results.

[Reference - R9]

FORMAL REQUIREMENT SPECIFICATION

A formal software specification is a statement expressed in a language whose vocabulary, syntax, and semantics are formally defined. The need for a formal semantic definition means that the specification languages cannot be based on natural language; it must be based on mathematics.

The advantages of a formal language are:

• The development of a formal specification provides insights and understanding of the software requirements and the software design.

• Given a formal system specification and a complete formal programming language definition, it may be possible to prove that a program conforms to its specifications.

• Formal specification may be automatically processed. Software tools can be built to assist with their development, understanding, and debugging.

• Depending on the formal specification language being used, it may be possible to animate a formal system specification to provide a prototype system.

• Formal specifications are mathematical entities and may be studied and analyzed using mathematical methods.

• Formal specifications may be used as a guide to the tester of a component in identifying appropriate test cases.

Relational and State-Oriented Notations

Relational notations are used based on the concept of entities and attributes.

Entities are elements in a system; the names are chosen to denote the nature of the elements (e.g., stacks, queues).

Attributes are specified by applying functions and relations to the named entities.

Attributes specify permitted operations on entities, relationships among entities, and data flow between entities.

Relational notations include implicit equations, recurrence relations, and algebraic axioms. State-oriented specifications use the current state of the system and the current stimuli presented to the system to show the next state of the system.

The execution history by which the current state was attained does not influence the next state; it is dependent only on the current state and the current stimuli.

State-oriented notations include decision tables, event tables, transition tables, and finite-state tables.

SPECIFICATION PRINCIPLES

Principle 1: Separate functionality from implementation. A specification is a statement of what is desired, not how it is to be realized. Specifications can take two general forms. The first form is that of mathematical functions: Given some set of inputs, produce a particular set of outputs. The general form of such specifications is find [a/the/all] result such that P(input), where P represents an arbitrary predicate. In such specifications, the result to be obtained has been entirely expressed in a “what”, rather than a “how” form, mainly because the result is a mathematical function of the input (the operation has well-defined starting and stopping points) and is unaffected by any surrounding environment.

Principle 2: A process-oriented systems specification language is sometimes required. If the environment is dynamic and its changes affect the behavior of some entity interacting with that environment (as in an embedded computer system), its behavior cannot be expressed as a mathematical function of its input. Rather a process-oriented description must be employed, in which the "what" specification is achieved by specifying a model of the desired behavior in terms of functional responses to various stimuli from the environment.

Principle 3: The specification must provide the implementer all of the information he/she needs to complete the program, and no more. In particular, no information about the structure of the calling program should be conveyed.

Principle 4: The specification should be sufficiently formal that it can conceivably be tested for consistency, correctness, and other desirable properties.

Principle 5: The specification should discuss the program in terms normally used by the user and implementer alike.

SOME SPECIFICATION TECHNIQUES

1. Implicit Equations

Specify computation of square root of a number between 0 and some maximum value Y to a tolerance E.

(0<=X<=Y){ABS_VALUE[(WHAT(X)) 2-X]}<=E

2. Recurrence Relation

Good for recursive computations.

Example, Fibonacci numbers 0, 1, 1, 2, 3, 5, 8,...

FI(0) = 0;

FI(1) = 1;

FI(n) = FI(n-1) + FI(n-2);   for n>=1.

[Reference - R10]

Verification and Validation

We have seen the "V-Model". In the V Model Software Development Life Cycle, based on requirement specification document the development & testing activity is started.

The V-model is also called as Verification and Validation model.

The testing activity is perform in the each phase of Software Testing Life Cycle.

In the first half of the model validations testing activity is integrated in each phase like review user requirements, System Design document & in the next half the Verification testing activity is come in picture.

Verification *(ARE WE BUILDING THE PRODUCT RIGHT?)*

***Definition** : The process of evaluating software to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*

Verification is a static practice of verifying documents, design, code and program. It includes all the activities associated with producing high quality software: inspection, design analysis and specification analysis. It is a relatively objective process.

Verification will help to determine whether the software is of high quality, but it will not ensure that the system is useful. Verification is concerned with whether the system is well-engineered and error-free.

Methods of Verification: Static Testing

- *Walkthrough*
- *Inspection*
- *Review*

Validation *(ARE WE BUILDING THE RIGHT PRODUCT?)*

***Definition**: The process of evaluating software during or at the end of the development process to determine whether it satisfies specified requirements.*

Validation is the process of evaluating the final product to check whether the software meets the customer expectations and requirements. It is a dynamic mechanism of validating and testing the actual product.

Methods of Validation: Dynamic Testing

- *Testing according to End Users*

Difference between Verification and Validation

The distinction between the two terms is largely to do with the role of specifications.

***Verification** is the process of checking that the software meets the specification. "Did I build what I need?"*

***Validation** is the process of checking whether the specification captures the customer's needs. "Did I build what I said I would?"*

Verification	***Validation***
Verification is a static practice of verifying documents, design, code and program.	Validation is a dynamic mechanism of validating and testing the actual product.
It does not involve executing the code.	It always involves executing the code.
It is human based checking of documents and files.	It is computer based execution of program.
Verification uses methods like inspections, reviews, walkthroughs, and Desk-checking etc.	Validation uses methods like black box (functional) testing, gray box testing, and white box (structural) testing etc.

Verification is to check whether the software conforms to specifications.	Validation is to check whether software meets the customer expectations and requirements.
It can catch errors that validation cannot catch. It is low level exercise.	*It can catch errors that verification cannot catch. It is High Level Exercise.*
Target is requirements specification, application and software architecture, high level, complete design, and database design etc.	Target is actual product-a unit, a module, a bent of integrated modules, and effective final product.
Verification is done by QA team to ensure that the software is as per the specifications in the SRS document.	***Validation** is carried out with the involvement of testing team.*
It generally comes first-done before **validation**.	It generally follows after **verification**.
Are we building the system right?	Are we building the right system?
Verification is the process of evaluating products of a development phase to find out whether they meet the specified requirements.	Validation is the process of evaluating software at the end of the development process to determine whether software meets the customer expectations and requirements.
The objective of Verification is to make sure that the product being develop is as per the requirements and design specifications.	The objective of Validation is to make sure that the product actually meet up the user's requirements, and check whether the specifications were correct in the first place.
Following activities are involved in Verification: Reviews, Meetings and Inspections.	Following activities are involved in Validation: Testing like black box testing, white box testing, gray box

	testing etc.
Verification is carried out by QA team to check whether implementation software is as per specification document or not.	Validation is carried out by testing team.
Execution of code is not comes under Verification.	Execution of code is comes under Validation.
Verification process explains whether the outputs are according to inputs or not.	Validation process describes whether the software is accepted by the user or not.
Verification is carried out before the Validation.	Validation activity is carried out just after the Verification.
Following items are evaluated during Verification: Plans, Requirement Specifications, Design Specifications, Code, Test Cases etc,	Following item is evaluated during Validation: Actual product or Software under test.
Cost of errors caught in Verification is less than errors found in Validation.	Cost of errors caught in Validation is more than errors found in Verification.
It is basically manually checking the of documents and files like requirement specifications etc.	It is basically checking of developed program based on the requirement specifications documents & files.

[Reference - R10]

UNIT -2 SOFTWARE PROJECT MANAGEMENT

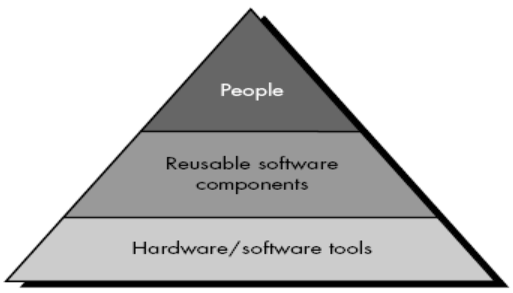
OBJECTIVES

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded. The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

RESOURCES

The second software planning task is estimation of the resources required to accomplish the software development effort. Figure illustrates development resources as a pyramid. The *development environment*—hardware and software tools—sits at the foundation of the resources pyramid and provides the infrastructure to support the development effort. At a higher level, we encounter reusable *software components*— software building blocks that can dramatically reduce development costs and accelerate delivery. At the top of the pyramid is the primary resource—*people*. Each resource is specified with four characteristics: description of the resource, a statement of availability, time when the resource will be required; duration of time that resource will be applied. The last two characteristics can be viewed as a time window.

Availability of the resource for a specified window must be established at the earliest practical time.



HUMAN RESOURCES

The planner begins by evaluating scope and selecting the skills required to complete development. Both organizational position (e.g., manager, senior software engineer) and specialty (e.g., telecommunications, database, client/server) are specified. For relatively small projects (one person-year or less), a single individual may perform all software engineering tasks, consulting with specialists as required.The number of people required for a software project can be determined only after an estimate of development effort (e.g., person-months) is made.

REUSABLE SOFTWARE RESOURCES

Component-based software engineering (CBSE)5 emphasizes reusability—that is, the creation and reuse of software building blocks [HOO91]. Such building blocks, often called *components*, must be cataloged for easy reference, standardized for easy application, and validated for easy integration.

Bennatan [BEN92] suggests four software resource categories that should be considered as planning proceeds:

Off-the-shelf components. Existing software that can be acquired from a third party or that has been developed internally for a past project. COTS (commercial off-the-shelf) components are purchased from a third party, are ready for use on the current project, and have been fully validated.

Full-experience components.

Existing specifications, designs, code, or test data developed for past projects that are similar to the software to be built for the current project. Members of the current software team have had full experience in the application area represented by these components. Therefore, modifications required for full-experience components will be relatively low-risk.

Partial-experience components.

Existing specifications, designs, code, or test data developed for past projects that are related to the software to be built for the current project but will require substantial modification. Members of the current software team have only limited experience in the application area represented by these components. Therefore, modifications required for partial-experience components have a fair degree of risk.

New components.

Software components that must be built by the software team specifically for the needs of the current project.The following guidelines should be considered by the software planner when reusable components are specified as a resource.

DECOMPOSITION TECHNIQUES

Software Sizing

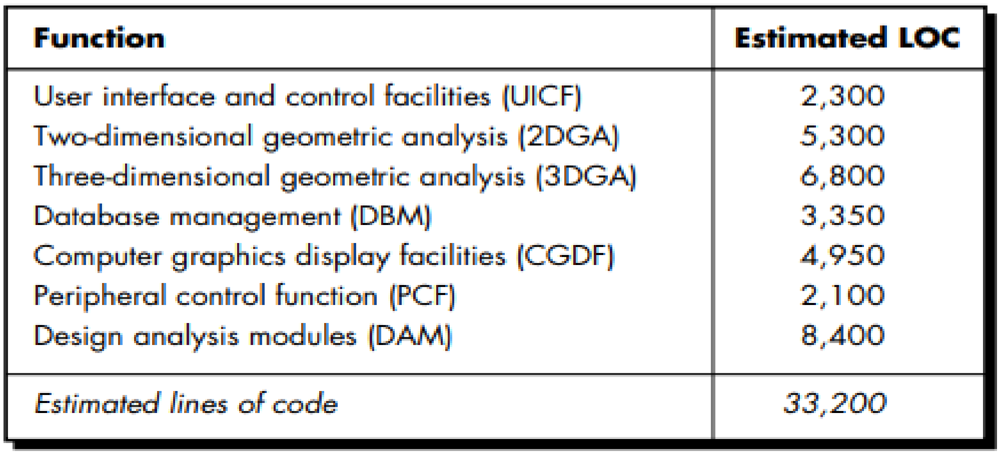
"Fuzzy logic" sizing. This approach uses the approximate reasoning techniques that are the cornerstone of fuzzy logic. To apply this approach, the planner must identify the type of application, establish its magnitude on a qualitative scale, and then refine the magnitude within the original range. Although personal experience can be used, the planner should also have access to a historical database of projects so that estimates can be compared to actual experience.

Function point sizing. Standard component sizing. For example, the standard components for an information system are subsystems, modules, screens, reports, interactive programs, batch programs, files, LOC, and object-level instructions. The project planner estimates the number of occurrences of each standard component and then uses historical project data to determine the delivered size per standard component. To illustrate, consider an information systems application. The planner estimates that 18 reports will be generated. Historical data indicates that 967 lines of COBOL are required per report. This enables the planner to estimate that 17,000 LOC will be required for the reports component. Similar estimates and computation are made for other standard components, and a combined size value (adjusted statistically) results.

Change sizing. This approach is used when a project encompasses the use of existing software that must be modified in some way as part of a project. The planner estimates the number and type (e.g., reuse, adding code, changing code, deleting code) of modifications that must be accomplished. Using an “effort ratio”for each type of change, the size of the change may be estimated

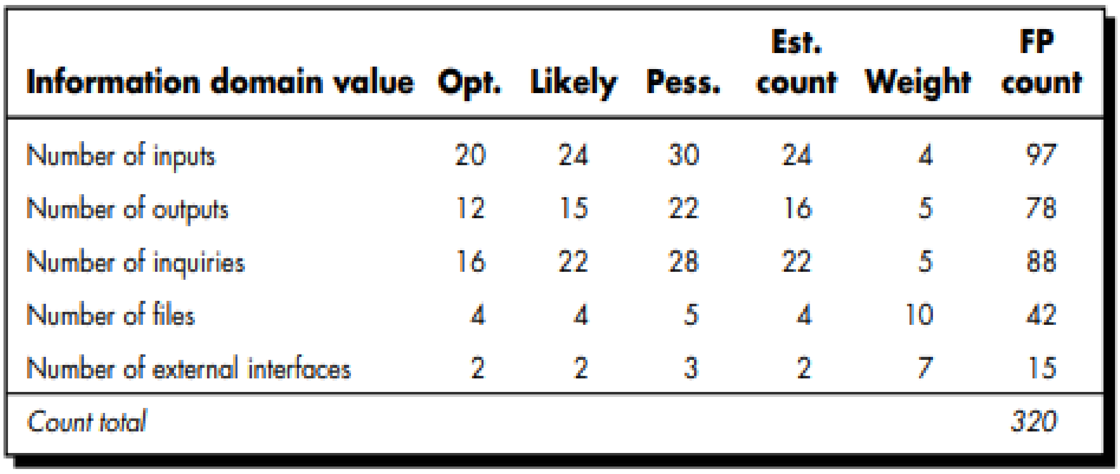
Problem-Based Estimation

LOC-Based Estimation



Function	Estimated LOC
User interface and control facilities (UICF)	2,300
Two-dimensional geometric analysis (2DGA)	5,300
Three-dimensional geometric analysis (3DGA)	6,800
Database management (DBM)	3,350
Computer graphics display facilities (CGDF)	4,950
Peripheral control function (PCF)	2,100
Design analysis modules (DAM)	8,400
Estimated lines of code	*33,200*

FP BASED ESTIMATION:



Information domain value	Opt.	Likely	Pess.	Est. count	Weight	FP count
Number of inputs	20	24	30	24	4	97
Number of outputs	12	15	22	16	5	78
Number of inquiries	16	22	28	22	5	88
Number of files	4	4	5	4	10	42
Number of external interfaces	2	2	3	2	7	15
Count total						*320*

Factor	Value
Backup and recovery	4
Data communications	2
Distributed processing	0
Performance critical	4
Existing operating environment	3
On-line data entry	4
Input transaction over multiple screens	5
Master files updated on-line	3
Information domain values complex	5
Internal processing complex	5
Code designed for reuse	4
Conversion/installation in design	3
Multiple installations	5
Application designed for change	5
Complexity adjustment factor	1.17

Finally, the estimated number of FP is derived:

$$FP_{\text{estimated}} = \text{count-total} \times [0.65 + 0.01 \times \Sigma (F_i)]$$
$$FP_{\text{estimated}} = 375$$

Process-Based Estimation

Activity →	CC	Planning	Risk analysis	Engineering		Construction release		CE	Totals
Task →				Analysis	Design	Code	Test		
Function ↓									
UICF				0.50	2.50	0.40	5.00	n/a	8.40
2DGA				0.75	4.00	0.60	2.00	n/a	7.35
3DGA				0.50	4.00	1.00	3.00	n/a	8.50
CGDF				0.50	3.00	1.00	1.50	n/a	6.00
DBM				0.50	3.00	0.75	1.50	n/a	5.75
PCF				0.25	2.00	0.50	1.50	n/a	4.25
DAM				0.50	2.00	0.50	2.00	n/a	5.00
Totals	0.25	0.25	0.25	3.50	20.50	4.50	16.50		46.00
% effort	1%	1%	1%	8%	45%	10%	36%		

CC = customer communication   CE = customer evaluation

PROJECT PLANNING OBJECTIVES

The objective of software project planning is to provide a framework that enables the manager to make reasonable estimates of resources, cost, and schedule. These estimates are made within a limited time frame at the beginning of a software project and should be updated regularly as the project progresses. In addition, estimates should attempt to define best case and worst case scenarios so that project outcomes can be bounded.

The planning objective is achieved through a process of information discovery that leads to reasonable estimates. In the following sections, each of the activities associated with software project planning is discussed.

THE COCOMO MODEL

In his classic book on "software engineering economics," Barry Boehm [BOE81] introduced a hierarchy of software estimation models bearing the name COCOMO, for *COnstructive COst MOdel*. The original COCOMO model became one of the most widely used and discussed software cost estimation models in the industry. It has evolved into a more comprehensive estimation model, called *COCOMO II* .Like its predecessor, COCOMO II is actually a hierarchy of estimation models that address the following areas:

Application composition model. Used during the early stages of software engineering, when prototyping of user interfaces, consideration of software and system interaction, assessment of performance, and evaluation of technology maturity are paramount.

Early design stage model. Used once requirements have been stabilized and basic software architecture has been established.

Post-architecture-stage model. Used during the construction of the software. Like all estimation models for software, the COCOMO II models require sizing information.Three different sizing options are available as part of the model hierarchy:

object points, function points, and lines of source code. The *object point* is an indirect software measure that is computed using counts of the number of

(1) screens (at the user interface)

(2) reports, and

(3) Components likely to be required to build the application. Each object instance (e.g., a screen or report) is classified into one of three complexity levels (i.e.,simple, medium, or difficult) using criteria suggested by Boehm [BOE96]. In essence,complexity is a function of the number and source of the client and server data tables that are required to generate the screen or report and the number of views or sections presented as part of the screen or report.

The Software Equation

The software equation Is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model has been derived from productivity data collected for over 4000 contemporary software projects. Based on these data, an estimation model of the form

E = [LOC _ *B0.333*/*P*]3 _ (1/*t4*) (5-3)

where *E* = effort in person-months or person-years

t = project duration in months or years

B = “special skills factor”16

P = “productivity parameter” that reflects:

• Overall process maturity and management practices

• The extent to which good software engineering practices are used

• The level of programming languages used

• The state of the software environment

• The skills and experience of the software team

• The complexity of the application

Typical values might be *P* = 2,000 for development of real-time embedded software; *P* = 10,000 for telecommunication and systems software; *P* = 28,000 for business systems applications.17 The productivity parameter can be derived for local conditions using historical data collected from past development efforts. It is important to note that the software equation has two independent parameters:

(1) an estimate of size (in LOC) and (2) an indication of project duration in calendar months or years.

RISK ANALYSIS

First, risk concerns future happenings. Today and yesterday are beyond active concern, as we are already reaping what was previously sowed by our past actions. The question is, can we, therefore, by changing our actions today, create an opportunity for a different and hopefully better situation for ourselves tomorrow. This means second, that risk involves change, such as in changes of mind, opinion, actions, or places . . . [Third,] risk involves choice, and the uncertainty that choice itself entails.

What is it?

Risk analysis and management are a series of steps that help a software team to understand and manage uncertainty. Many problems can plague a software project. A risk is a potential problem—it might happen, it might not. But, regardless of the outcome, it’s a really good idea to identify it, assess its probability of occurrence, estimate its impact, and establish a contingency plan should the problem actually occur.

Who does it?

Everyone involved in the software process—managers, software engineers, and customers participate in risk analysis and management.

SOFTWARE RISKS

There is general agreement that risk always involves two characteristics

• *Uncertainty*—the risk may or may not happen; that is, there are no 100% probable risks.

• *Loss*—if the risk becomes a reality, unwanted consequences or losses will occur.

When risks are analyzed, it is important to quantify the level of uncertainty and the degree of loss associated with each risk. To accomplish this, different categories of risks are considered.

Project risks threaten the project plan. That is, if project risks become real, it is likely that project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, customer, and requirements problems and their impact on a software project.project complexity, size, and the degree of structural uncertainty were also defined as project (and estimation) risk factors.

Technical risks threaten the quality and timeliness of the software to be produced.If a technical risk becomes a reality, implementation may become difficult or impossible.Technical risks identify potential design, implementation, interface, verification,and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and "leading-edge" technology are also risk factors.Technical risks occur because the problem is harder to solve than we thought it would be.

Business risks threaten the viability of the software to be built. Business risks often jeopardize the project or the product. Candidates for the top five business risks are

(1) building a excellent product or system that no one really wants (market risk), (2)building a product that no longer fits into the overall business strategy for the company (strategic risk)

(3) building a product that the sales force doesn't understand

how to sell

(4) losing the support of senior management due to a change in focus or

a change in people (management risk)

(5) losing budgetary or personnel commitment (budget risks). It is extremely important to note that simple categorization won't always work. Some risks are simply unpredictable in advance.

Another general categorization of risks has been proposed by Charette [CHA89]. *Known risks* are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

Predictable risks are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

Unpredictable risks are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

RISK IDENTIFICATION

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories that have been presented earlier : generic risks and product-specific risks.

Generic risks are a potential threat to every software project.

Product-specific risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the project at hand. To identify product-specific risks, the project plan and the software statement of scope are examined and an answer to the following question is developed: "What special characteristics of this product may threaten our project plan?"

One method for identifying risks is to create a *risk item checklist.* The checklist can be used for risk identification and focuses on some subset of known and predictable risks in the following generic subcategories:

• *Product size*—risks associated with the overall size of the software to be built or modified.

• *Business impact*—risks associated with constraints imposed by management or the marketplace.

• *Customer characteristics*—risks associated with the sophistication of the customer and the developer's ability to communicate with the customer in a timely manner.

• *Process definition*—risks associated with the degree to which the software process has been defined and is followed by the development organization.

• *Development environment*—risks associated with the availability and quality of the tools to be used to build the product.

• *Technology to be built*—risks associated with the complexity of the system to be built and the "newness" of the technology that is packaged by the system.

• *Staff size and experience*—risks associated with the overall technical and project experience of the software engineers who will do the work.

The risk item checklist can be organized in different ways. Questions relevant to each of the topics can be answered for each software project. The answers to these questions allow the planner to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of "risk components and drivers" [AFC88] are listed along with their probability *Although generic risks are important to consider, usually the product-specific risks cause the most headaches. Be certain to spend the time to identify as many product-specific risks as possible.*

<u>RISK COMPONENT & DRIVERS</u>

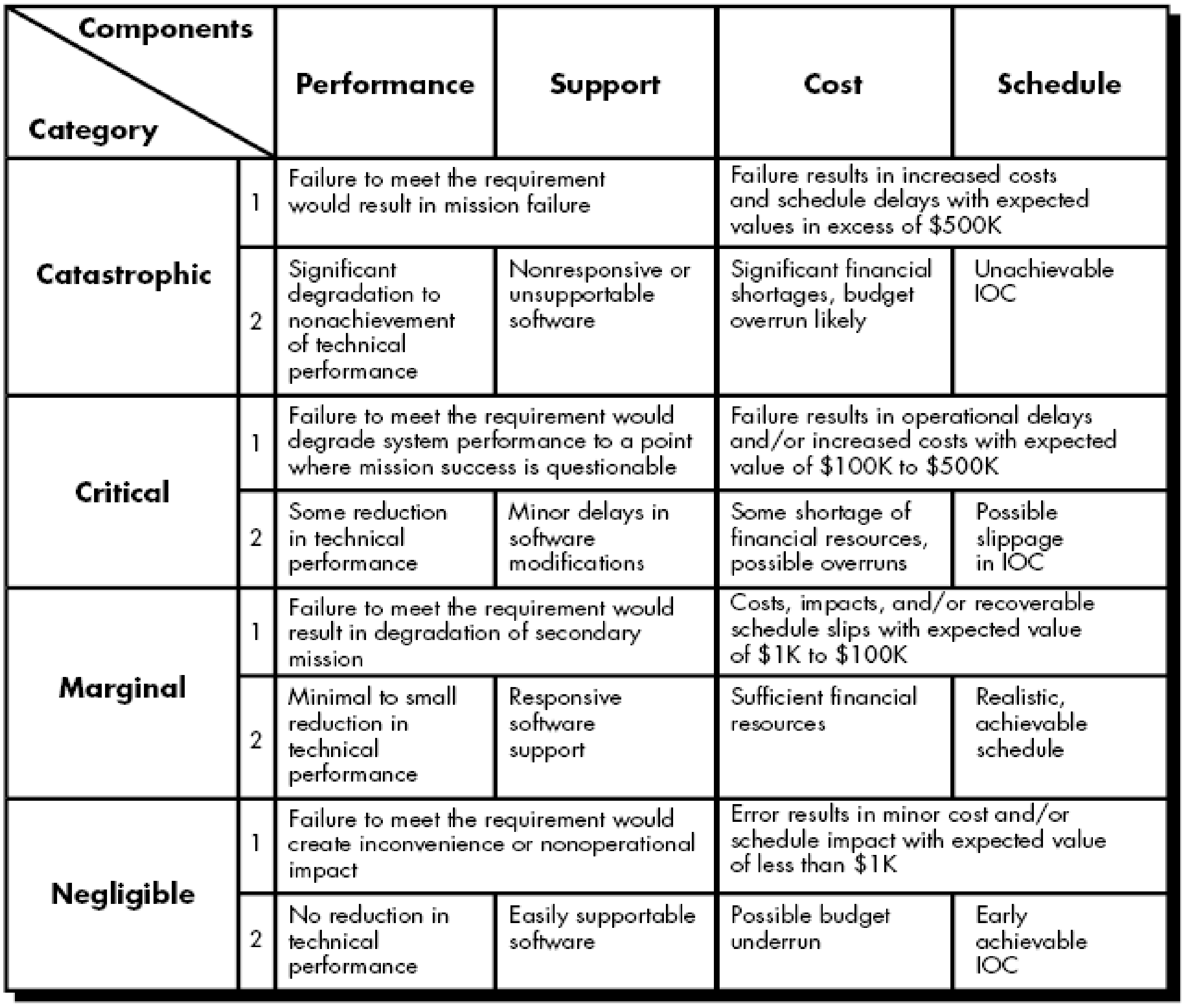
The risk components are defined in the following manner:

• *Performance risk*—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.

• *Cost risk*—the degree of uncertainty that the project budget will be maintained.

• *Support risk*—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.

• *Schedule risk*—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.



Components / Category		Performance	Support	Cost	Schedule
Catastrophic	1	Failure to meet the requirement would result in mission failure		Failure results in increased costs and schedule delays with expected values in excess of $500K	
	2	Significant degradation to nonachievement of technical performance	Nonresponsive or unsupportable software	Significant financial shortages, budget overrun likely	Unachievable IOC
Critical	1	Failure to meet the requirement would degrade system performance to a point where mission success is questionable		Failure results in operational delays and/or increased costs with expected value of $100K to $500K	
	2	Some reduction in technical performance	Minor delays in software modifications	Some shortage of financial resources, possible overruns	Possible slippage in IOC
Marginal	1	Failure to meet the requirement would result in degradation of secondary mission		Costs, impacts, and/or recoverable schedule slips with expected value of $1K to $100K	
	2	Minimal to small reduction in technical performance	Responsive software support	Sufficient financial resources	Realistic, achievable schedule
Negligible	1	Failure to meet the requirement would create inconvenience or nonoperational impact		Error results in minor cost and/or schedule impact with expected value of less than $1K	
	2	No reduction in technical performance	Easily supportable software	Possible budget underrun	Early achievable IOC

RISK MITIGATION, MONITORING, AND MANAGEMENT

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues:

• risk avoidance

• risk monitoring

• risk management and contingency planning

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for *risk mitigation.* For example, assume that high staff turnover is noted as a project risk, *r1.* Based on past history and management intuition, the likelihood, *l1,* of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact, *x1,* is projected at level 2. That is, high turnover will have a critical impact on project cost and schedule.

To mitigate this risk, project management must develop a strategy for reducing turnover. Among the possible steps to be taken are

• Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, competitive job market).

• Mitigate those causes that are under our control before the project starts.

• Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

• Organize project teams so that information about each development activity is widely dispersed.

• Define documentation standards and establish mechanisms to be sure that documents are developed in a timely manner.

• Conduct peer reviews of all work (so that more than one person is "up to speed”).

• Assign a backup staff member for every critical technologist.As the project proceeds, risk monitoring activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the following factors can be monitored:

• General attitude of team members based on project pressures.

• The degree to which the team has jelled.

• Interpersonal relationships among team members.

• Potential problems with compensation and benefits.

• The availability of jobs within the company and outside it.

In addition to monitoring these factors, the project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of documentation standards and mechanisms to be sure that documents are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor documents carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

Risk management and contingency planning assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well underway and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, the project manager may temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to speed.” Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.” This might include video-based knowledge capture, the development of “commentary documents,” and/or meeting with other team members who will remain on the project.

It is important to note that RMMM steps incur additional project cost. For example,spending the time to "backup" every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, the project planner performs a classic cost/benefit analysis. If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is "backup," management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself! For this reason, we adapt the Pareto 80–20 rule to software risk. Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help the planner to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don't fall into the critical 20 percent (the risks with highest project priority).

SOFTWARE PROJECT SCHEDULING

Software project scheduling is an activity that distributes estimated effort across the planned project duration by allocating the effort to specific software engineering tasks. During early stages of project planning, a *macroscopic schedule* is developed. This type of schedule identifies all major software engineering activities and the product functions to which they are applied. As the project gets under way, each entry on the macroscopic schedule is refined into a *detailed schedule*. Here, specific software tasks (required to accomplish an activity) are identified and scheduled. Scheduling for software engineering projects can be viewed from two rather different perspectives. In the first, an end-date for release of a computer-based system has already (and irrevocably) been established. The software organization is constrained to distribute effort within the prescribed time frame. The second view of software scheduling assumes that rough chronological bounds have been discussed but that the end-date is set by the software engineering organization. Effort is distributed to make best use of resources and an end-date is defined after careful analysis of the software. Unfortunately, the first situation is encountered far more frequently than the second. Like all other areas of software engineering, a number of basic principles guide

software project scheduling:

Compartmentalization. The project must be compartmentalized into a number of manageable activities and tasks. To accomplish compartmentalization, both the product and the process are decomposed .

Interdependency. The interdependency of each compartmentalized activity or task must be determined. Some tasks must occur in sequence while others can occur in parallel. Some activities cannot commence until the work product produced by another is available. Other activities can occur independently.

Time allocation. Each task to be scheduled must be allocated some number of work units (e.g., person-days of effort). In addition, each task must be assigned a start date and a completion date that are a function of the interdependencies and whether work will be conducted on a full-time or part-time basis.

Effort validation. Every project has a defined number of staff members. As time allocation occurs, the project manager must ensure that no more than the allocated number of people have been scheduled at any given time. For example, consider a project that has three assigned staff members (e.g., 3 person-days are available per day of assigned effort5). On a given day, seven concurrent tasks must be accomplished. Each task requires 0.50 person days of effort. More effort has been allocated than there are people to do the work.

Defined responsibilities. Every task that is scheduled should be assigned to a specific team member.

Defined outcomes. Every task that is scheduled should have a defined outcome. For software projects, the outcome is normally a work product (e.g. the design of a module) or a part of a work product. Work products are often combined in deliverables.

Defined milestones. Every task or group of tasks should be associated with a project milestone. A milestone is accomplished when one or more work products has been reviewed for quality and has been approved.

Each of these principles is applied as the project schedule evolves.

SCHEDULING

Scheduling of a software project does not differ greatly from scheduling of any multitask engineering effort. Therefore, generalized project scheduling tools and techniques can be applied with little modification to software projects.

Program evaluation and review technique (PERT) and ***critical path method*** (CPM)

[MOD83] are two project scheduling methods that can be applied to software development. Both techniques are driven by information already developed in earlier project planning activities:

• Estimates of effort

• A decomposition of the product function

• The selection of the appropriate process model and task set

• Decomposition of tasks

Interdependencies among tasks may be defined using a task network. Tasks, sometimes called the project *work breakdown structure* (WBS), are defined for the product as a whole or for individual functions.

Both PERT and CPM provide quantitative tools that allow the software planner to

(1) determine the *critical path*—the chain of tasks that determines the duration of the project;

(2) establish “most likely” time estimates for individual tasks by applying statistical models;

(3) calculate “boundary times” that define a time "window" for a particular task.

Boundary time calculations can be very useful in software project scheduling. Slippage in the design of one function, for example, can retard further development of other functions.

Riggs describes important boundary times that may be discerned from a PERT or CPM network:

(1) the earliest time that a task can begin when all preceding tasks are completed in the shortest possible time,

(2) the latest time for task initiation before the minimum project completion time is delayed,

(3) the earliest finish—the sum of the earliest start and the task duration,

(4) the latest finish the latest start time added to task duration, and

(5) the *total float*—the amount of surplus time or leeway allowed in scheduling tasks so that the network critical path is maintained on schedule. Boundary time calculations lead to a determination of critical path and provide the manager with a quantitative method for evaluating progress as tasks are completed.