NAME: - KAMLESH KUMAR SAHU

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING, DIPLOMA

SEMSETER: 5TH **SEM**

SUBJENCT NAME: - **SOFTWARE ENGINEERING**

SUBJECT CODE: - **2022572(022)**

CHHATTISGARH INSTITUTE OF TECHNOLOGY, JASHPUR

SESSION 2025-26

CHAPTER 01
FUNDAMENTALS OF SOFTWARE ENGINEERING

SOFTWARE ENGINEERING (सॉफ्टवेयर इंजीनियरिंग): -

सॉफ्टवेयर इंजीनियरिंग उपयोगकर्ता (User) की जरूरतों का विश्लेषण कर उसके अनुसार सॉफ्टवेयर को डिजाइन (Design), विकास (Development), लागू (Deployment) और उनका परीक्षण (Testing) करने की प्रक्रिया है जो प्रोग्रामिंग भाषाओं (Programming Languages) के उपयोग से किया जाता है। सॉफ्टवेयर इंजीनियरिंग एक Step by Step प्रोसेस है। जिसमे किसी कम्प्यूटर कंपोनेंट, हार्डवेयर डिवाइसेस तथा विभिन्न एप्लीकेशन प्रोग्राम के लिए लिए सॉफ्टवेयर का निर्माण किया जाता है। यह एक सिस्टमेटिक एप्रोच होता है जिसमे सॉफ्टवेयर विभिन्न चरणों से होते हुए विकसित किया जाता है।

SOFTWARE (सॉफ्टवेयर): -

सॉफ्टवेयर एक कंप्यूटर प्रोग्राम होता है जिसमे Associated Documents के साथ साथ Configuration Data तथा Instruction (Commands) भी होते है जो की Programs को सही तरीके से ऑपरेट करने में मदद करता है। अन्य शब्दों में, Software कुछ Instruction के समूह होते है जो किसी Specific Task को Execute करने के लिए प्रयोग किये जाते है।

SOFTWARE CHARACTERSTICS (सॉफ्टवेयर कैरक्टरस्टिक्स): -

एक अच्छे Software के निम्नलिखित गुणधर्म होते है: -

01. **Functionality:** किसी भी सॉफ्टवेयर के निष्पादन की क्षमता को अपने इच्छित उद्देश्य से संदर्भित करता है।
02. **Reliability:** किसी दी गई स्थितियों के तहत वांछित कार्यक्षमता को प्रदान करने के लिए सॉफ्टवेयर की क्षमता का उल्लेख करता है।
03. **Usability:** इस बात को संदर्भित करता है कि सॉफ्टवेयर आसानी से कैसे इस्तेमाल किया जा सकता है।
04. **Efficiency:** सबसे प्रभावी और कुशल तरीके से सिस्टम संसाधनों का उपयोग करने के लिए सॉफ्टवेयर की क्षमता को दर्शाता है।
05. **Maintainability:** किसी सॉफ्टवेर को आसानी से अपनी कार्यक्षमता बढ़ाने, इसके प्रदर्शन को बेहतर बनाने, या Error को सुधारने के लिए उसके Software System में संशोधन किया जा सकता है।
06. **Robustness:** सॉफ्टवेयर में Robustness होनी चाहिए जिससे कि सॉफ्टवेयर में Defects तथा Failure को Detect किया जा सकें।
07. **Security:** सॉफ्टवेयर में Security बेहतर होनी चाहिए जिससे कि सॉफ्टवेयर सम्पूर्ण सुरक्षा बनी रहें और वह Threats से सुरक्षित रहें।
08. **Flexibility:** सॉफ्टवेयर Flexible होना चाहिए अर्थात अगर सॉफ्टवेयर में कुछ बदलाव करने है तो वह आसानी से किये जा सकें।
09. **Testability**: सॉफ्टवेयर की टेस्टिंग आसानी से की जा सकें।
10. **Platform independent:** सॉफ्टवेयर Platform Independent होना चाहिए अर्थात वह किसी भी सिस्टम में Run हो सकें। Software Developers आसानी से या कम से कम परिवर्तनों के द्वारा, एक Platform से दूसरे Platform में Software Transfer कर सकते हैं।

SOFTWARE APPLICATION (सॉफ़्टवेयर एप्लीकेशन): -

कुछ सामान्य प्रकार के एप्लीकेशन सॉफ़्टवेयर: -

1. Productivity Software: प्रोडक्टिविटी सॉफ्टवेयर, जिसमें अधिकांश कंप्यूटर यूजर्स के द्वारा उपयोग किए जाने वाले वर्ड प्रोसेसर, स्प्रेडशीट और टूल शामिल हैं।
2. Presentation Software: ग्राफिक डिजाइनर के लिए ग्राफिक्स सॉफ्टवेयर।
3. Security Software: कंप्यूटर सिस्टम के लिए Anti-Virus Software और Firewall जैसे Security सॉफ्टवेयर।
4. Enterprise Site License: एक इंटरप्राइजेस लाइसेंस, अनलिमिटेड साइट लाइसेंस की तरह है लेकिन यह केवल एक फिजिकल लोकेशन तक सीमित नहीं है।
5. Open Source Software: ओपन सोर्स सॉफ़्टवेयर, मुफ्त सॉफ्टवेयर लाइसेंस के साथ आते है, जो यूजर्स को सॉफ्टवेयर को मोडीफाई करने और रिडिस्ट्रीब्यूट का अधिकार प्रदान करता है।
6. GNU – General Public License: यह एक विशिष्ट प्रकार का लाइसेंस है जो कि शुल्क या नि:शुल्क हो सकते है, लेकिन यूजर्स सॉफ्टवेयर को उनकी इच्छा के अनुसार मोडीफाई कर सकते हैं।
7. Shareware: Shareware एक कॉपीराइट सॉफ्टवेयर है जो डाउनलोड करने के लिए फ्री है, लेकिन इसका उपयोग किसी तरह से लिमिटेड होता है।
8. Customer Relationship Management (CRM): CRM सॉफ्टवेयर का उपयोग बिक्री और सेवा से संबंधित दोनों प्रश्नों में ग्राहकों के साथ इंटरप्राइजेस एंगेजमेंट में किया जाता है।
9. Educational Software: ये ऐसे Content प्रदान करता है जिसकी जरुरत Student तथा Teachers को Education में होती है।
10. Media Development Software: यह सॉफ्टवेयर इलेक्ट्रॉनिक मीडिया जनरेट तथा विभिन्न मीडिया को प्रिंट करने के लिए किया जाता है।

SOFTWARE ENGINEERING APPROACH:

किसी सॉफ्टवेयर के निर्माण के लिए सॉफ्टवेयर इंजीनियरिंग में दो प्रकार के एप्रोच है: -

1. LAYERED TECHNOLOGY OF SOFTWARE ENGINEERING.
2. GENERIC VIEW OF SOFTWARE ENGINEERING.

1. LAYERED TECHNOLOGY: - सॉफ्टवेयर इंजीनियरिंग के लेयर्ड टेक्नोलॉजी चार चरण में होते है।
    a. Quality Focus
    b. Process
    c. Methods
    d. Tools

a. Quality Focus: यह Software Engineering का सबसे महत्वपूर्ण Layer है। दुनिया में कोई भी Product में Quality सबसे जरुरी होता है। जितना अच्छा Product का Quality होगा उतना ही ज्यादा User को पसंद आएगा। इसीलिए जब Product बनाया जाता है सबसे पहले Quality का ख्याल रखा जाता है। इस Layer को Software Development का Bedrock भी कहा जाता है। Quality Focus के अंदर भी कई सारे Parameters को देखा जाता है।

i. Degree Of Goodness: यह parameter तय करता है की software के quality कितनी सही और यह user को कितना पसंद आएगा।

ii. Correctness: इस में यह पता लगाया जाता है की Software को जिस मकसद से बनाया गया था वो पूरा हो रहा है या नहीं। अगर नहीं हो रहा है तो क्या क्या कमी है और उसे कैसे ठीक किया जा सकता है ये भी तय किया जाता है।

iii. Maintainability: यह Parameter से हमे यह पता चलता है की एक बार Software Develop होने के बाद उसे Maintain करना कितना आसान होता है। एक Software का Maintenance जितना सहज होता है उस Software का Quality उतना बढ़ जाता है।

iv. Usability: यह सबसे महत्वपूर्ण Parameter है, क्योँ की Software चाहे जितना भी अच्छा हो लेकिन अगर उसको आसानी से इस्तमाल नहीं किया जा सकता है तो उस Software का कोई Value ही नहीं रहता है। User ज्यादातर उस Product को पसंद करते हैं जिसे आसानी से उपयोग किया जा सकता है।

b. Process: इस Layer मे एक Software को बनाने के लिए अलग अलग तरीके के सवाल पूछा जाता है। इसको हम अगर आसान भाषा में समझेंगे तो "What & How Questions"। एक साधारण Product बनाने से पहले भी कई सारे सवाल खड़े होते हैं। जैसे की Product और Software "कैसे काम करेगा?", "क्या काम करेगा?", "क्योँ बनाया गया?", "कौन से Device पर काम करेगा?" इन सारे सवालों को एकत्रित करके इन सबका उत्तर निकला जाता है।

c. Method: इस Layer में Process Layer के सवालों का जवाब ढूंढा जाता है। यहां पर कई Parameters को उपयोग करके उन सवालों का जवाब निकले जाते हैं।

i. Communication: Method Layer का सबसे जरुरी Parameter Communication है। इसका मतलब है की Customer और Organization के बीच एक बेहतरीन Communication होना चाहिए। इससे Developers को ये पता चलता है की Customer किस तरह का Software बनवाना चाहता है।

ii. Requirement & Design Model Analysis: एक बार Customer का Requirements पता चलने के बाद उनको नजर में रख के Software को सही तरीके से बनवाना होता है और Design भी इस तरह होना चाहिए ता की हमारे Quality Focus Layer के सारे Condition को Satisfy करता हो।

iii. Use Of Programming Tools: एक बार Requirement & Design Model Analysis होने के बाद अब हमे ये तय करना होता है की कौन से Programming Language का उपयोग करके Software को बनाया जायेगा और कौन से Tools का उपयोग किया जायेगा।

iv. Testing & Support: एक बार Software बनने के बाद Testing की बारी आती है । यह Software Development सबसे जरुरी हिस्सा है। कुछ Sample डाटा के साथ Testing करने के बाद ही पता चलता है की सॉफ्टवेयर के अंदर क्या क्या खामियां है और उन्हें जरुरी Supports के मदद से फिर से ठीक किया जाता है।

d. Tools: एक Software Develop करने में हमेशा कई सारे Tools की जरुरत पड़ती है, चाहे वो Code लिखने केलिए हो या Design करने के लिए हो या Testing करने के लिए या फिर Sell करने के लिए हो। ये Tools आपको Software Development के दौरान एक Environment प्रदान करते हैं जो आपको Automated या फिर Semi Automate रूप से आपको मदद करते हैं।

2. GENERIC VIEW: - सॉफ्टवेयर इंजीनियरिंग के जेनेरिक व्यू टेक्नोलॉजी तीन चरण में होते है ।
   a. Definition Phase: - यह लेयर सॉफ्टवेयर निर्माण के लिए सॉफ्टवेयर इंजीनियर "किस Information को Processed करना है" निर्धारित किया जाता है। इस लेयर में किसी Function और उससे Desired Performance का निर्धारण, सिस्टम के व्यवहार का निर्धारण तथा विभिन्न प्रकार के Validation Criteria का निर्धारण किया जाता है।
   b. Devolopment Phase: - यह लेयर सॉफ्टवेयर निर्माण के लिए सॉफ्टवेयर इंजीनियर "किस प्रकार से डाटा को स्ट्रक्चर प्रदान करना है" का निर्धारण किया जाता है। इस लेयर में कोई Function का Performance, Software Architechture का निर्माण तथा प्रोग्रामिंग लैंग्वेज का निर्धारण किया जाता है।
   c. Support Phase: - यह लेयर सॉफ्टवेयर निर्माण के लिए होने वाले परिवर्तन पर Error Correction , सॉफ्टवेयर का मेंटनेस, तथा नवीन Version के सॉफ्टवेयर निर्माण के लिए किया जाता है।

SOFTWARE PROCESS MODEL (सॉफ्टवेयर प्रोसेस मॉडल): -

किसी सॉफ्टवेयर को डेवेलप करने के लिए सॉफ्टवेयर प्रोसेस मॉडल दो प्रकार के होते है: -

1. **CLASSICAL MODEL (क्लासिकल मॉडल): -**
   a. Water Fall Model / Linear Model / Sequential Model / Classical Life Cycle Model.
   b. Incremental Model.
2. **EVOLUTIONARY MODEL (इवोलुशनरी मॉडल): -**
   a. Spiral Model.
   b. Prototype Model.

a. CLASSICAL WATER FALL MODEL (वॉटर फॉल मॉडल): -

- सॉफ्टवेयर इंजीनियरिंग के लिए Waterfall Model (वॉटर फॉल मॉडल) SDLC (सिस्टम डेवलपमेंट लाइफ साइकिल) का एक Popular और Good Version है। Waterfall Model (वॉटर फॉल मॉडल) को Linear तथा Sequential मॉडल भी कहते है।
- इस मॉडल में किसी एक फेज का एक बार डेवलपमेंट पूरा हो जाता है तो हम अगले Phase में चले जाते है औरअगले फेज में जाने के बाद वापस पिछले Phase में नही जा सकते है। हम इस मॉडल में किसी Phases को Overlap नही कर सकते है।
- वॉटरफॉल मॉडल में, एक Phase का आउटपुट दूसरे Phase के लिए इनपुट की तरह कार्य करता है। एक डेवलपमेंट Phase तब तक शुरू नही हो सकता जब तक कि उसका पिछला वाला Phase पूरा नही हो जाता है।
- क्लासिक वॉटरफॉल मॉडल में निम्नलिखित Phases होते है: -
  a. Requirement Phase: - Requirement Phase वॉटर फॉल मॉडल का सबसे पहला Phase है। इस फेज में सॉफ्टवेयर को किस कार्य को करने के किये बनाया जा रहा है वो सभी Requiredment कस्टमर/End User से कलेक्ट करके एक डॉक्यूमेंट तैयार किया जाता है। यह फेज बहुत Crucial होता है क्योंकि इसी फेज पर अगले Phase आधारित होते हैं। यह फेज इस कारण से भी महत्वपूर्ण है अगर हमें कस्टमर की Requiredment की सही तरह से पता नहीं होगी तो हम सॉफ्टवेयर का अगला फेज शुरू नहीं कर सकेंगे।
  b. Design Phase: - Design Phase इस तथ्य पर आधारित होता है कि सॉफ्टवेयर का निर्माण किस प्रकार होगा। डिज़ाइन फेज का मुख्य उद्देश्य सॉफ्टवेयर सिस्टम का Blueprint तैयार करना है जिससे कि आने वाले Phases पर

किसी प्रकार की कोई दिक्कत का सामना ना करना पड़े और Requirement Phase में जो भी Requirements है उनका Solution निकाल लिया जाएँ।

c. Implementation Phase: - इस फेज में हार्डवेयर, सॉफ्टवेयर तथा एप्लीकेशन प्रोग्राम्स को Install किया जाता है तथा डेटाबेस डिज़ाइन को Implement किया जाता है। इससे पहले की डेटाबेस डिज़ाइन को Implement किया जाएं सॉफ्टवेयर को टेस्टिंग, कोडिंग, तथा Debugging प्रोसेस से होकर गुजरना पड़ता है। वॉटर फॉल में यह सबसे लम्बे समय तक चलने वाला Phase है।

d. Verification Phase: - इस फेज में सॉफ्टवेयर को Verify किया जाता है और यह Evaluate किया जाता है कि हमने सही Product बनाया है। इस फेज में विभिन्न प्रकार की टेस्टिंग की जाती है तथा सॉफ्टवेयर के हर Area में Check किया जाता है। माना अगर हमने सॉफ्टवेयर को अच्छी तरह Verify नही किया और इसमें कोई Defect रह जाता है तो इसका इस्तेमाल कोई नही करेगा इसलिए Verification अत्यंत महत्वपूर्ण है। Verification का एक Advantage यह है कि इससे सॉफ्टवेयर के Fail होने का Risk कम हो जाती है।

e. Maintenance Phase: - यह वॉटरफॉल का सबसे अंतिम Phase है। जब सिस्टम बनके तैयार हो जाता है तथा यूजर उसका प्रयोग करना शुरू कर देते है तब जो Problems उसमें आती है उनको time-to-time हल करना पड़ता है। तैयार सॉफ्टवेयर को समय अनुसार उसका ख्याल रखना तथा उसे Maintain रखना ही Maintenance कहलाता है। SDLC में तीन प्रकार के Maintenance होते है:-

I. Corrective maintenance.
II. Adaptive maintenance.
III. Perfective maintenance.

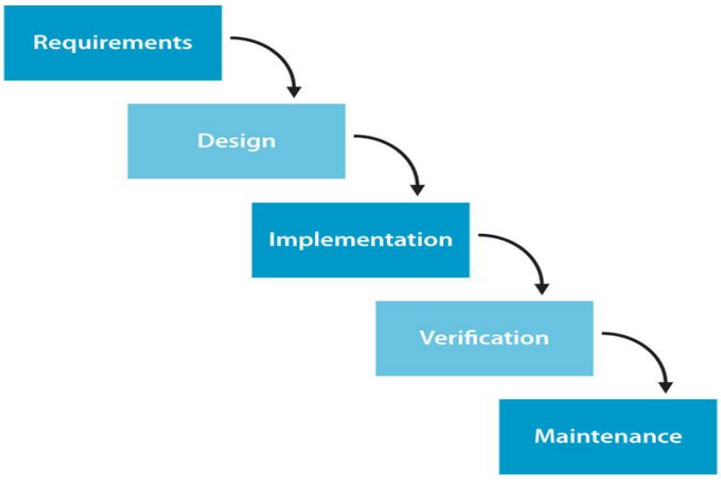


Fig: Water Fall Model

क्लासिक वॉटर फॉल मॉडल के लाभ: -

1. इसे समझने की दृष्टि से बहुत ही आसान व सिंपल Software Development मॉडल है।
2. इसमें एक समय केवल एक चरण प्रोसेस्ड किया जा सकता है।
3. इसमें प्रक्रिया, Action और परिणाम बहुत अच्छी तरह से Documented होते है।
4. यह मॉडल छोटे प्रोजेक्ट में बहुत अच्छी तरह से कार्य करता है साथ ही जहां आवश्यकता को अच्छे से समझ लिया जा सके।
5. इसमें प्रत्येक चरण बहुत अच्छी तरह से तथा स्पष्ट रूप परिभाषित होते है।

क्लासिक वॉटर फॉल मॉडल के हानि: -

1. क्लासिक वॉटर फॉल मॉडल में सॉफ्टवेर का विकास एक चरण से दूसरे चरण में झरने के सामान होता है, Classical Water Fall Model में यह मान के चलना पड़ता है कि डेवलपर द्वारा किसी भी चरण में कोई भी Error नही होगी। अतः इसमें कोई भी Mechanism, Error Correction के लिए नही प्रदान करता है।
2. इस मॉडल में यह मान के चला जाता है कि कस्टमर की जो भी आवश्यकता है उसे प्रोजेक्ट के प्रारम्भ में ही पूरा और सही परिभाषित किया जा सकता है। अतः कस्टमर की आवश्यकता समय के साथ किसी भी फेज के पूरा होने के बाद उसमे बदलाव करना डिफिकल्ट होता है।
3. यह मॉडल इस बात की बात की सिफारिश करता है कि नया चरण तभी प्रारम्भ हो सकता है जब पिछले चरण पूर्ण हो जाये।

b. <u>PROTOTYPE MODEL (प्रोटोटाइप मॉडल): -</u>

- Prototype Model (प्रोटोटाइप मॉडल) एक ऐसी Activity है जिसमें सॉफ्टवेयर एप्लीकेशन के Prototypes का निर्माण किया जाता है। सबसे पहले प्रोटोटाइप का निर्माण किया जाता है फिर उस प्रोटोटाइप पर आधारित अंतिम Product का निर्माण किया जाता है।
- वॉटरफॉल मॉडल की कमियों की दूर करने के लिए Prototype Model को विकसित किया गया था। इस मॉडल का निर्माण तब किया जाता है जब हमें Requirements का अच्छी तरह से पता नही होता है।
- इस मॉडल की खासियत यह है कि इस मॉडल को दूसरे Models के साथ भी प्रयोग में लाया जा सकता है और अकेले भी।
- इस मॉडल में एक दिक्कत यह है कि End Users अगर Prototype मॉडल से संतुष्ट नही है तो दुबारा नया Prototype मॉडल बनाया जाता है जिससे इस मॉडल में बहुत पैसा तथा समय लगता है।
- Prototype model में निम्नलिखित फेज होते है:-

a. <u>Requirement Gathering: -</u> Prototype मॉडल का सबसे पहला स्टेप Requirements को एकत्रित करना होता है, वैसे तो कस्टमर को Requirements का कुछ खास पता नही होता लेकिन जो प्रमुख Requirements है उनको विस्तार पूर्वक Define कर लिया जाता है।

b. <u>Build Prototype: -</u> इस फेज में Initial Prototype का निर्माण किया जाता है। इसमें कुछ बेसिक Requirements को प्रदर्शित किया जाता है तथा यूज़र इंटरफ़ेस उपलब्ध किया जाता है।

c. <u>Evaluate Review: -</u> जब प्रोटोटाइप का निर्माण पूरा हो जाता है तो End Users या Customer को इसे प्रस्तुत किया जाता है और उनसे इस प्रोटोटाइप के बारें में Feedback लिया जाता है। इस फीडबैक का प्रयोग सिस्टम को और बेहतर बनाने में किया जाता है और प्रोटोटाइप में जो सम्भव परिवर्तन हो सकें वह किया जाता है।

d. <u>Refine & Improve: -</u> जब End Users तथा Customer से फीडबैक लिया जाता है तो फीडबैक के आधार पर प्रोटोटाइप को Improve (बेहतर) किया जाता है। अगर कस्टमर प्रोटोटाइप से संतुष्ट नही है तो एक नये प्रोटोटाइप का निर्माण किया जाता है और यह प्रक्रिया तब तक चलती रहती है जब तक की कस्टमर को अपनी इच्छानुसार प्रोटोटाइप नही मिलता।

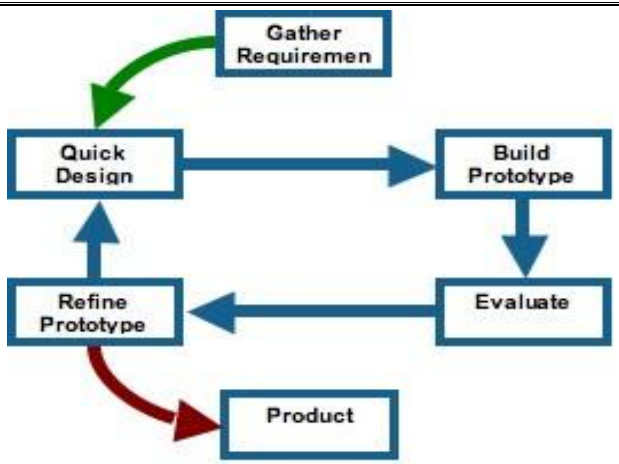


Fig: Proto Type Model

<u>प्रोटोटाइप मॉडल मॉडल के लाभ: -</u>

1. इस मॉडल में End User भी डेवलपमेंट में शामिल रहते है।
2. चूँकि यह मॉडल डेवलप किये जाने वाले सिस्टम का वर्किंग नमूना तैयार करता है जिससे End User सिस्टम को आसानी से समझ पाते है।
3. Error को प्रारंभिक फेज में ही Detect कर लिया जाता है।
4. कम समय में तैयार होने के कारण End User से फीडबैक लेकर बेहतर Solution प्राप्त कर सकते है।
5. छुट्टे हुए Function तथा Requirements की पहचान कर सकते है।

<u>प्रोटोटाइप मॉडल के हानि: -</u>

1. बार बार सिस्टम का प्रोटोटाइप निर्माण होने के कारण Complexity बढ़ जाती है।
2. Ideal प्रोटोटाइप का निर्माण न होने पैर सही End Product का निर्माण नहीं हो पता है।
3. यह मॉडल बहुत अधिक Time Consume करता है।

c. <u>INCREMENTAL MODEL (इंक्रीमेंटल मॉडल): -</u>

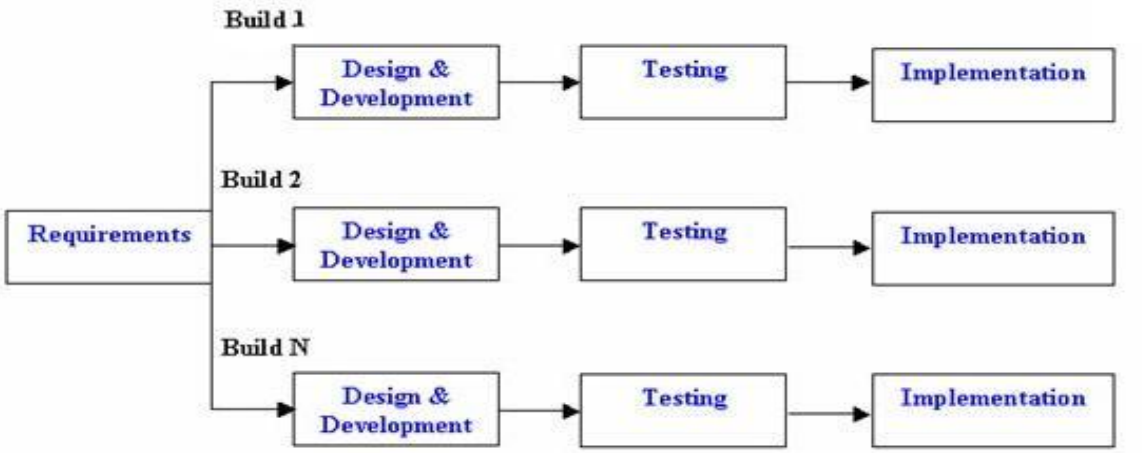
- इंक्रीमेंटल मॉडल में Process Adaptability तथा Customer Satisfaction पर ध्यान दिया जाता है।
- पूर्व में Water Fall Model का प्रयोग सॉफ्टवेयर को बनाने में किया जाता था। परन्तु आज के समय में Software Development के बीच में Customer, सॉफ्टवेयर में Changes करने को कहता है। इन Changes को करने में बहुत सारा Time तथा Money लगता है। इंक्रीमेंटल मॉडल को मुख्यत: Software Development के बीच में Changes करने के लिए ही बनाया गया था जिससे की सॉफ्टवेयर प्रोजेक्ट को जल्दी से पूरा किया जा सकें ।
- इंक्रीमेंटल मॉडल में Software Product को छोटे छोटे भागों (Parts) में बाँट दिया जाता है। इसमें सबसे पहले सबसे छोटे Part को विकसित किया जाता है और उसके बाद उससे बड़ा और प्रत्येक Incremental Part को Iteration पर विकसित किया जाता है। प्रत्येक Iteration को छोटा रखा जाता है जिससे की उसे आसानी से Manage किया जा सके।

इंक्रीमेंटल मॉडल के निम्नलिखित Phases होते है: -

a. <u>Requirement Phase: -</u> इस फेज में सिस्टम की Requirements को एकत्रित तथा Documented किया जाता है। यह फेज बहुत Crucial होता है क्योंकि इसी फेज पर अगले Phase आधारित होते हैं।
b. <u>Design & Developement Phase: -</u> Design Phase इस तथ्य पर आधारित होता है कि सॉफ्टवेयर का निर्माण किस प्रकार होगा। डिज़ाइन फेज का मुख्य उद्देश्य सॉफ्टवेयर सिस्टम का Blueprint तैयार करना है जिससे की आने वाले Phases पर किसी प्रकार की कोई दिक्कत का सामना ना करना पड़े और Requirement Phase में जो भी Requirements है उनका Solution निकाल लिया जाएँ।

c. Testing Phase: - इस फेज में प्रत्येक पार्ट को टेस्ट किया जाता है और यह Evaluate किया जाता है कि हमने सही Product बनाया है। इस फेज में विभिन्न प्रकार की टेस्टिंग की जाती है तथा सॉफ्टवेयर के हर Area में Check किया जाता है। यदि सॉफ्टवेयर को अच्छी तरह टेस्ट नही किया और इसमें कोई Defect रह जाता है तो यह सॉफ्टवेयर का परिणाम गलत प्रदान होगा। इसलिए Verification अत्यंत महत्वपूर्ण है।

d. Implementation Phase: - इस फेज में हार्डवेयर, सॉफ्टवेयर तथा एप्लीकेशन प्रोग्राम्स को Install किया जाता है तथा डेटाबेस डिज़ाइन को Implement किया जाता है। इससे पहले की डेटाबेस डिज़ाइन को Implement किया जाएं सॉफ्टवेयर को टेस्टिंग, कोडिंग, तथा Debugging प्रोसेस से होकर गुजरना पड़ता है।



Incremental Life Cycle Model

इंक्रीमेंटल मॉडल के लाभ: -

1. इसमें दो Programmers एक साथ काम करते है जिससे बहुत ही कम गलतियाँ होती है।
2. इसमें Software Project को बहुत कम समय में पूरा कर लिया जाता है।
3. यह सॉफ्टवेर Development की बहुत ही वास्तविक Approach है।
4. इसमें Rules बहुत कम होते है और Documentation भी ना के बराबर होता है।
5. इसमें Planning की जरुरत नही पड़ती है. इसे आसानी से Manage किया जा सकता है।
6. यह Developers को Flexibility प्रदान करता है।

इंक्रीमेंटल मॉडल के हानि: -

1. यह Complex Dependencies को Handle नहीं कर पाता है।
2. इसमें Formal Documentation की कमी की वजह से Development में Confusion हो जाता है।
3. यह ज्यादातर Customer Representative पर निर्भर रहता है अगर Customer Representative कोई गलत Information दे देता है तो सॉफ्टवेयर गलत बन सकता है।
4. इसमें केवल Experienced Programmers ही कोई Decision ले सकते है. नए Programmers कोई भी Decision नहीं ले सकते है।
5. इसमें Software Development के शुरुवात में Effort तथा Time पता नहीं लग पाता है।

d. **SPIRAL MODEL (स्पाइरल मॉडल): -**

- स्पाइरल मॉडल एक प्रकार का Sequential और Prototype मॉडल का Combination होता है। इनका प्रयोग मुख्य रूप से बड़े प्रोजेक्ट्स के लिए किया जाता है, जिसमें निरंतर वृद्धि शामिल होती है।

- ये विशेष प्रकार की गतिविधियां होती हैं, जिन्हे एक Iteration (Spiral) में किया जाता है। इनका प्रयोग वहां होता है, जहां आउटपुट बड़े सॉफ़्टवेयर का एक छोटा प्रोटोटाइप होता है। जब तक प्रोजेक्ट पूर्ण रूप से पूरा नहीं हो जाता, तब तक सभी Spirals के लिए समान गतिविधियों को दोहराया जाता है।
- स्पाइरल मॉडल में प्रत्येक स्पाइरल एक लूप के समान होता है और हर लूप अपनी अपनी अलग डेवलपमेंट प्रक्रिया रखता है।
- इनका उपयोग बड़े प्रोजेक्ट करने के लिए बहुत अच्छा है जहां आप छोटे प्रोटोटाइप विकसित और वितरित कर सकते हैं और इसे बड़ा सॉफ्टवेयर बनाने के लिए बढ़ा सकते हैं। लेकिन छोटे प्रोजेक्ट के लिए ये महंगा और उतना कारगर सिद्ध नहीं होता है।

एक स्पाइरल मॉडल के निम्नलिखित Phases होते है: -

a. Planning Phase: -जिस जिस चीज की भी आवश्यकता होती है, उन सब से जुड़ा अध्ययन और उन्हें एकत्र किया जाता है साथ ही इसमें व्यवहार्यता अध्ययन भी आवश्यक है। आवश्यकताओं को सुव्यवस्थित करने के लिए समीक्षा और पूर्वाभ्यास भी जरूरी होती है। जरूरत के सभी दस्तावेज को समझना आवश्यकताओं की अंतिम सूची को बनाना ।

b. Risk analysis Phase: - सभी जरूरत का अध्ययन किया जाता है और संभावित जोखिमों की पहचान करने के लिए ब्रेन स्टॉर्मिंग सत्र किए जाते हैं एक बार जोखिमों की पहचान हो जाने के बाद, जोखिम कम करने की रणनीति की योजना बनाई जाती है और उसे अंतिम रूप दिया जाता है, जिसे जोखिम न के बराबर हो। उन दस्तावेज़ को रखा जाता है, जो सभी जोखिमों और इसकी शमन योजनाओं पर प्रकाश डालता है।

c. Engineering Phase: - इस चरण के दौरान सॉफ्टवेयर का वास्तविक विकास और परीक्षण किया जाता है । कोडिंग पर ध्यान दिया जाता है। टेस्ट सारांश रिपोर्ट और दोष रिपोर्ट बनाई जाती है।

d. Evaluation Phase: - इस चरण को अंतिम चरण कहा जाता है इसमें ग्राहक सॉफ़्टवेयर का मूल्यांकन करता है और अपनी प्रतिक्रिया और अनुमोदन प्रदान करता है।

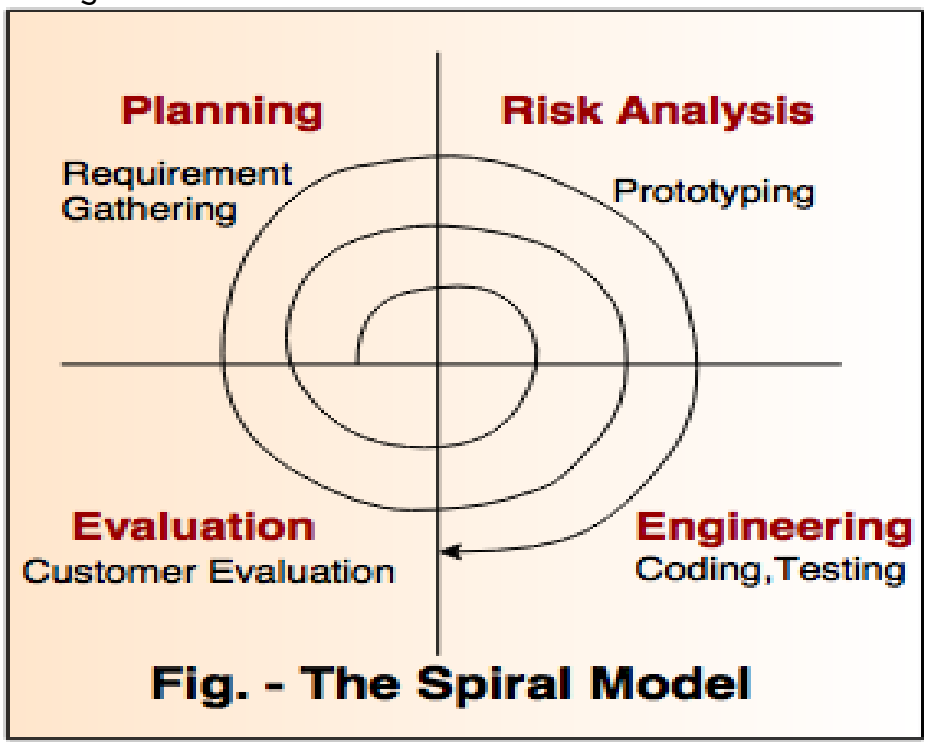


Fig. - The Spiral Model

स्पाइरल मॉडल के लाभ: -

1. सॉफ्टवेयर के विकास की प्रक्रिया तेज हो जाती है।

2. बड़े प्रोजेक्ट या सॉफ्टवेयर को एक सही तरीके से संभाला जा सकता है।
3. जोखिमों का बार बार मूल्यांकन कर के उन्हें कम किया जा सकता है।
4. विकास के सभी चरणों की ओर नियंत्रण रहता है।
5. ज्यादा से ज्यादा सुविधाओं को एक व्यवस्थित तरीके से जोड़ा जाता है।
6. सॉफ्टवेयर का निर्माण जल्दी किया जा सकता है।

<u>स्पाइरल मॉडल के हानि: -</u>

1. इस मॉडल के जरिये सॉफ्टवेयर का निर्माण महंगा हो सकता है।
2. जोखिम विश्लेषण के लिए अत्यधिक विशिष्ट विशेषज्ञता की आवश्यकता पड़ेगी।
3. प्रोजेक्ट की सफलता जोखिम विश्लेषण चरण पर अत्यधिक निर्भर रहती है।
4. छोटे प्रोजेक्ट के लिए ये अच्छा काम नहीं करता है।
5. स्पाइरल लिमिट से बाहर जा सकता है।
6. डॉक्यूमेंटेशन बहुत बढ़ जाती है क्यूंकि इसमें बीच में भी कई चरण आ जाते हैं।

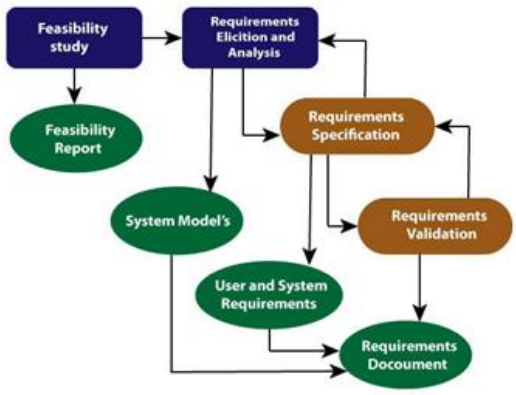
UNIT 02
SOFTWARE ENGINEERING PRACTIES & REQUIREMENT ANALYSIS

Requirement Engineering Process Or Task (REP or RET): -

जब किसी भी संगठन में, प्रत्येक नए उत्पाद (Product) या सेवा (Service) को व्यवसाय (Business) की आवश्यकता के जवाब (Response) में बनाया जाता है। तब नए उत्पाद (Product) या सेवा (Service) के विकास (Development) पर समय और संसाधन खर्च करने के बाद भी आवश्यक उत्पाद (Required Product) और अंतिम उत्पाद (Final Product) के बीच में काफी अंतर हो सकता है।

भविष्य में बड़ी समस्याओं से बचने के लिए किसी भी परियोजना के प्रारंभिक चरणों में एक केंद्रित (Focused) और विस्तृत (Detailed) आवश्यकताओं के विश्लेषण की आवश्यकता है। ऐसी प्रक्रिया को ही Requirment Engineering Process (रेक्विरेमेंट इंजीनियरिंग प्रोसेस) या Requirement Engineering Task (रेक्विरेमेंट इंजीनियरिंग टास्क) कहते है। Requirment Engineering Process चार चरण में किया जाता है: -

1. Feasibility Study (फिजिबिलिटी स्टडी)
2. Requirement Elicitation and Analysis (रेक्विरेमेंट एलिसिटेशन एंड एनालिसिस)
3. Software Requirement Specification (सॉफ्टवेयर रेक्विरेमेंट स्पेसिफिकेशन)
4. Software Requirement Validation (सॉफ्टवेयर रेक्विरेमेंट वेलिडेशन)



Requirement Engineering Process

1. **Feasibility Study (फिजिबिलिटी स्टडी): -** किसी भी सिस्टम की सफलता के लिए सिस्टम की क्षमता का परीक्षण करना Feasibility Study कहलाती है। Feasibility Study का मुख्य उद्देश्य यह निर्धारित करना है कि सिस्टम को विकसित करना Financially तथा Technically रूप से संभव है या नही। इसका उद्देश्य Problem को Solve करना नही होता है बल्कि यह निर्धारित करना है कि प्रॉब्लम को Solve किया जा सकता है या नही। इसमें सिस्टम में आने वाली सभी दिक्कतों को Analyze किया जाता है, इसमें सिस्टम की जानकारी को गहराई पूर्वक जाँचा जाता है और यह निर्धारित किया जाता है कि सिस्टम को विकसित करने में किस प्रकार से सफलता पायी जा सकती है। Feasibility Study निम्नलिखित 03 प्रकार की होती है: -

- Technical Feasibility: -इस Feasibility में सिस्टम की टेक्निकल जरूरतों को निर्धारित किया जाता है। इसमें यह देखा जाता है कि जो प्रस्तावित सिस्टम है उसके लिए जो टेक्नोलॉजी चाहिए वह उपलब्ध हो तथा उस टेक्नोलॉजी को सिस्टम में किस प्रकार Integrate किया जायें। नयी टेक्नोलॉजी में आने वाली सभी प्रकार की Complexities को Handle करने के लिए तकनीकी रूप से सक्षम एक्सपर्ट की जरुरत भी होती है।
- Operational Feasibility: - इस Feasibility में यह निर्धारित किया जाता है कि एक प्रस्तावित सिस्टम किस प्रकार Problems को Solve करेगा तथा सिस्टम में किस प्रकार के बदलाव आये है? इसमें यह देखा जाता है कि जो सिस्टम है क्या वह यूज़र्स के लिए लाभकारी है या नही तथा क्या वह यूज़र्स के जरूरतों को पूरा करने में सक्षम है या नही।
- Economical Feasibility: - इस Feasibility में यह निर्धारित किया जाता है कि जो प्रस्तावित सिस्टम है उसमें कितना खर्चा आएगा तथा उसमें कितना Benefit मिलेगा? Economic Feasibility को Cost Benefit Analysis भी कहते है।

2. **Requirement Analysis Or Elicitation: -**

Elicitation का अर्थ Collect करना होता है और Requirement Elicitation का मतलब Requirement को Collect करना होता है। Requirement Elicitation एक ऐसी प्रोसेस है जिसमें सॉफ्टवेयर के Requirements को Collect करने के लिए Customers, Stoke Holders तथा End Users के साथ Interact किया जाता है। Requirement Elicitation को कभी कभी Requirement Gathering भी कहते है।

किसी भी Software को बनाने के लिए सबसे पहले Requirements को जानने व समझने की आवश्यकता होती है और इन सभी Requirements को Collect करने के लिए एक Medium (माध्यम) की आवश्यकता होती है जो Software से Related सभी आवश्यक जानकारी को Collect करते है। ये Medium Stock-Holder या End-User होते है।

इन End-User या Stock-Holder द्वारा दी गयी Requirement को Collect करके Software बनाया जाता है। Software Development Process तभी सफल हो सकता है जब Customer और Developer के बीच प्रभावशाली Partnership हो। Requirement Elicitation Technique की सफलता Customer, Developer, User और Analyst के समझदारी के ऊपर Depend करता है।

Requirement Elicitation Methods Or Communication Methods: -

Requirement Elicitation के निम्नलिखित तरीके है: –

i. **Interview: –** Interview Method के द्वारा यह समझा जाता है कि Software से Customer की क्या उम्मीदें है। इस Method में End-User और Stock-Holder से Interview लिया जाता है। यह Interview Software की कार्यसूची के Base पर लिया जाता है और Software से सम्बंधित सभी Requirements को Collect करते है। इसमें पहले User से Interview लिया जाता है उसके बाद Stock-Holder से लिया जाता है।

ii. **Scenario: -** Scenario एक Step by Step Process होता है Scenario का मतलब Steps की Series होती है। Scenario में यह देखा जाता है कि किस प्रकार की और कितनी Input की Requirement होगी। इसमें User की जरुरत के अनुसार Step by Step Information या Requirement को Collect किया जाता है।

iii. **Questionnaire: -** Questionnaire जो है वह Requirement Collect करने का सबसे अच्छा Method है। इसमें कम समय और कम Cost में सॉफ्टवेयर से सम्बंधित महत्वपूर्ण Information को Collect करते है। इस Method में एक बहुत बड़ी Problem होती है कि सब अलग अलग Information देते है क्योंकि सबका अलग अलग Field होता है। सब अपने अपने Department का पूरा Knowledge देते है और सभी Information को Collect करके Arrange किया जाता है। इसमें सभी Information सही हो यह जरूरी नहीं होता है। जो Information सही होता है उसका Use किया है।

iv. **On Side Observation: –** किसी Company या Organization जहाँ Software को Develop किया जा रहा है वहाँ जाकर Information Collect करना On Side Observation कहलाता है।

v. **Prototype: –** किसी भी Software को बनाने के लिए Use में आने वाले Component के बारे में हमे कोई जानकारी नहीं होती है। इसलिए उस Component के बारे में पूरी Information Collect करने के लिए Prototype Model का Use किया जाता है।

Problem Arises During Requirement Gathering (Requirement को Collect करने में आने वाली परेशानियाँ)

i. Problem of Scope: - जब Software बना रहे होते है तो उसकी Boundary या Scope Fix को Mention किया जाता है। एक Software की Fix Boundary या Scope है या नहीं उसे Mention किया जाता है। एक Software की Boundary Fix

होनी चाहिए। यदि Software की Boundary या Scope Fix नहीं होती है तो Problem Create हो जाती है। उदाहरण के लिए: - Software बनने के लिए Cost व Budget कितना होगा, Developer की Salary कितनी होगी यह सब Mention किया जाता है।

ii. Problem of Understanding: – Software से संबंधित सभी Modules के बारे में पूरी जानकारी होनी चाहिए तभी Software से संबंधित Information Collect कर सकते है। यदि Software के सभी Modules को अच्छे से समझ नहीं पाएंगे तो Information भी Collect नहीं कर सकते।

iii. Problem of Volatility (अस्थायी): - Environment तथा User के जरिये मिली जानकारी Fix नहीं होती है। किसी भी Software को बनने के लिए उसका Scope Fix होना चाहिए यदि Scope Change होगा तो Information Collect नहीं कर पाएंगे।

iv. Problem of Communication: - यह बहुत ही बड़ा Issue है इमसें Miscommunication, Language Barries, गलत Assumption आदि आते है। अगर हम User को समझ ही नहीं पा रहे कि वो कहना क्या चाहता तो यह बहुत बड़ी बाधा बन जाता है।

3. **SRS (Software Requirement Specification) (सॉफ्टवेयर रिक्वायरमेंट स्पेसिफिकेशन): -**

Software Requirement Specification वह किसी सिस्टम या सॉफ्टवेर एप्लीकेशन की जरूरतों का एक पूरा डॉक्यूमेंट या विवरण होता है। अर्थात् दुसरे शब्दों में कहें तो, "SRS एक डॉक्यूमेंट होता है जो कि यह Describe करता है कि सॉफ्टवेर के Features क्या होंगें तथा उसका Behaviour क्या होगा। वह किस प्रकार परफॉर्म करेगा।"

SRS का फायदा यह है कि इससे Developers को सॉफ्टवेर को विकसित करने में कम समय तथा मेहनत लगती है।SRS सॉफ्टवेर के Layout की तरह होता है जिसकी यूजर समीक्षा कर सकता है तथा देख सकता है की वह (SRS) उसकी जरूरतों के हिसाब से बना है या नहीं।"

Characteristics of SRS (विशेषताएं):- SRS की विशेषताएं निम्नलिखित है:-

1. Complete (संपूर्ण): - SRS जो है वह संपूर्ण होना चाहिए अर्थात् जो भी सॉफ्टवेयर की जरूरतें है वह सभी SRS में उल्लेखित होनी चाहिए।
2. Correct (उचित): - यह Correct होना चाहिए अर्थात् जो कस्टमर की जरूरतें है उसके हिसाब होना चाहिए।
3. Clear (स्पष्ट): - यह स्पष्ट होना चाहिए। सॉफ्टवेयर की आवश्यकताओं को स्पष्ट रूप से वर्णित (Declared) होना चाहिए।
4. Accurate (परिशुद्ध): - इसमें Accuracy होनी चाहिए। अगर यह ही Accurate नहीं होगा तो सॉफ्टवेयर का निर्माण नहीं हो सकता।
5. Consistent (निरंतर): - इसको शुरुवात से लेकर अंत तक Consistent (निरंतर) होना चाहिए जिससे कि यूजर आसानी से रिक्वायरमेंट्स को समझ सकें। और Consistency तभी प्राप्त की जा सकती है जब दो रिक्वायरमेंट्स के मध्य कोई विरोधाभास ना हो।
6. Verifiable (सत्यापनीय): - यह Verifiable (सत्यापन योग्य) होना चाहिए। जो Experts तथा Testers होते है उनके द्वारा Requirements को Verify किया जाता है।
7. Modifiable (परिवर्त्य): - SRS का जो डॉक्यूमेंट होता है उसमें Specify की गयी सारी रिक्वायरमेंट्स मॉडिफाई करने योग्य होनी चाहिये। यह तभी हो सकता है जब SRS का स्ट्रक्चर संतुलित हों।

8. Traceable: - यह Traceable होना चाहिए अर्थात् इसमें प्रत्येक Requirement अलग अलग प्रकार से Identify होनी चाहिए। प्रत्येक Requirement की अपनी अलग एक पहचान होनी चाहिए।
9. Testable (परिक्षण योग्य): - यह Testable होना चाहिए अर्थात् इसे किसी भी प्रकार से टेस्ट करने के योग्य होना चाहिए।
10. Unambigous (अमिश्रित): - यह केवल तब Unambigous हो सकता है जब सभी Requirement का केवल एक ही अर्थ हों। अर्थात् केवल एक ही Interpretation (व्याख्या) हो।

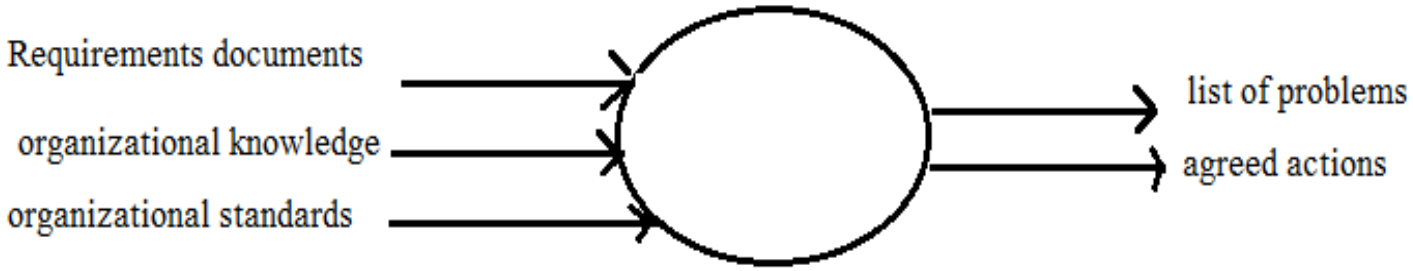
4. **Software Requirement Validation (सॉफ्टवेयर रेक्विरेमेंट वेलिडेशन): -**

Software Designing और Development में किसी भी तरह का कोई Problem ना आये इसलिए Requirement Validation का Use किया जाता है। Software को बनाने के लिए जो Requirements एकत्रित की जाती है उन्हें Analyze किया जाता है कि वह Requirements हमारे Software बनाने के लिए सही है या नहीं तथा Software Development के लिए कौन से Process का प्रयोग किया जायेगा उसे भी Analyze करते है।

Requirements और Process को Analyze करने के बाद एक SRS (Software Requirement Specification) बनाया जाता है। SRS बनाने के बाद यह सुनिश्चित करना जरूरी होता है कि SRS में कोई भी Error नहीं है और यह Specify किया जाता है कि User की Requirements Correct है।

SRS में ज्यादातर Errors Present होते है। यदि यह Error Development Process के बाद या फिर User को Software Deliver करने के बाद Detect हुए तो Software के Cost पर असर पड़ता है। इसलिए SRS से Software Designing और Development के पहले Error Detect कर लेना चाहिए। Requirement से सम्बंधित जितनी भी परेशानी होती है उन सभी को Check करने के लिए Requirement Validation परफॉर्म किया जाता है।

SRS में जो भी Ambiguity, Data Redundancy, Error / Bug की Problem है उसे Software Development के पहले Solve कर लेना चाहिए। इससे हमारे Requirements जो है वह Error Free रहते है और Software भी अच्छी Quality का बनता है। यहाँ Requirement Validation के लिए तीन Inputs होते है: –



i. Requirement Documents: - Organization के Standard के अनुसार Requirement Document को Formulated और Organized होना चाहिए। हर Organization का अपना अलग अलग Standard होता है इसलिए Organization के Standard के अनुसार ही Requirement Document को बनाना चाहिए।
ii. Organizational Knowledge: – Organizational Knowledge का प्रयोग Organization के बारे में पूरी जानकारी रखने के लिए, Software Development का Process ठीक से चल रहा है या नहीं तथा System की Requirements को जानने के लिए किया जाता है।
iii. Organizational Standards: - Organizational Standards में हमे Organization के सभी Rules, Regulations तथा Limits पता होनी चाहिए जिसे Follow करके System Develop कर सकें।

यहाँ Requirement Validation के दो Output होते है: –

i. List of Problem: - List of Problems में उन सभी Problems की List होती है जिन्हें Requirement Document में Discover किया था।
ii. Agreed Action: – Problem List में जितनी भी Problems होती है उनको Solve करने के लिए जो Action परफॉर्म होती है उनको Agreed Action में Display किया जाता है।

Problem Aries During Requirement Validation (Requirement Validation में आने वाली परेशानियां): -

- Data Redundancey / Repetition.
- Accuracy.
- Timeliness.
- Error / Bug.
- Ambiguity.

Requirements Validation Tools & Technique: - Requirement Validation की कुछ Tools तथा Techniques होती है इनका प्रयोग अकेले भी किया जा सकता है या फिर दूसरे Techniques के साथ भी किया जा सकता है. इनका प्रयोग पूरे सॉफ्टवेयर को या फिर सॉफ्टवेयर के Parts को Check करने के लिए किया जाता है कुछ Tools And Techniques नीचे दिए हुए है: –

- Test CASE Gerneration.
- Automated Consistency Analysis.
- Prototyping.

SYSTEM ANALYSIS MODELING (सिस्टम एनालिसिस मॉडलिंग): -

System Ananlysis Modeling को System Model भी कहते है। System Modeling एक Process है। यह प्रोसेस एक System के संपूर्ण Structure को Develop करता है। यह प्रोसेस System के Astract Model को Develop करता है, इसमें प्रत्येक Model सिस्टम के एक अलग-अलग View या Perspective को प्रस्तुत करते है।

System Modeling में बहुत से Models सम्मिलित होते है अर्थात् यह कई सारे Model से मिलकर बना होता है। System Model में जितने भी Models होते है उन्हें Graphical Way (चित्रों के रूप) में Represent किया जाता है।हम किसी भी System के संपूर्ण Model या Structure को Represent करने के लिए Graphical Structure या Graphical Notation का प्रयोग करते है।

System Modeling जो है वह सिस्टम तथा मॉडल की Functionality को Analysis करके समझने में मदद करता है। जो कि Customer के साथ Communicate करने के लिए प्रयोग किया जाता है।

Types of System Analysis Models (सिस्टम मॉडल के प्रकार): -

System Models के कई प्रकार है जिसमे कुछ Models नीचे Describe किये गए है –

i. Data model.
ii. Function Model.
iii. Behavioral model.

i. **DATA MODEL (डाटा मॉडल): –**

Data Model में हम यह Represent करते है कि हमारा Data कैसे एक Module से दुसरे Module में Move कर रहा है। Data modeling, System model का एक प्रकार है। Data Model में Input, Output और Processing Data के Flow को Graphical Notation के द्वारा Represent किया जाता है। इसे E-R Modeling या E-R Notation भी कहते है।

एक Data Model ये Show करता है कि किसी Database के विभिन्न Entities किस प्रकार से आपस में Internally Related हैं। एक Data Model उन Attributes को Show करता है, जो किसी Data Entity को Describe करते हैं।

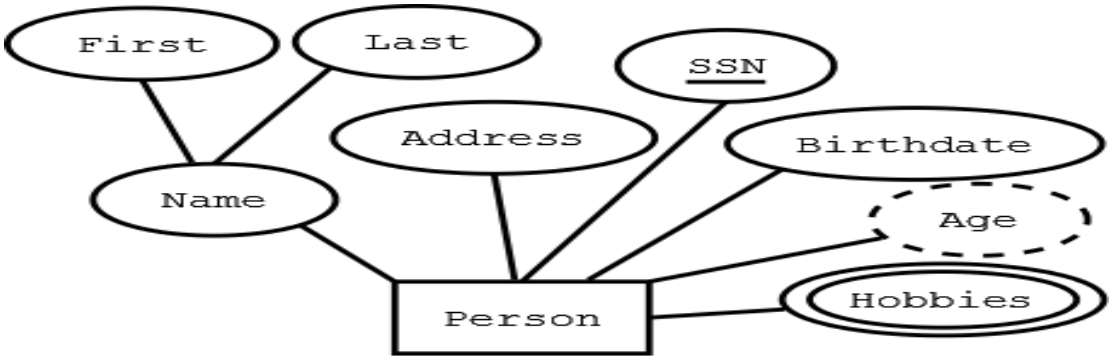
Data के Flow के लिए Notation का भिन्न – भिन्न प्रकार (Data Flow Notation) –

- Rectangle Shape: – इस Shape का Use किसी भी System के External Entities को Represent करने के लिए करते है।
- Data Store: – इस Shape का Use Data Base में कई प्रकार के Information या Data को Store करने के लिए किया जाता है।
- Data Process / Circle: – इस Shape का Use इस Notation में किसी भी Information या Data के Processing के लिए किया जाता है।
- Diamond Shape / Decision Making Digram: – इस Shape का Use Entity के बीच Relationship को Represent करने के लिए किया जाता है।
- Arrow: – इस Shape का प्रयोग Left to Right , Right to Left , Top to Bottom or Bottom to Top डेटा के Movement को Represent करने के लिए किया जाता है।

Perception (SCHEMA) of Data Model: –

Data Model को प्रस्तुत करने के लिए यहाँ तीन प्रकार का Perception है: –

a. Physical Perception – इसमें Database Schema यह Create और Check करता है कि हमारा Database Schema एक दुसरे से कैसे Related है।
b. Conceptual Perception – इसे Logical Perception भी कहते है। इसमें यह देखते है कि हमारा Data कैसे Insert , Delete और Update हो रहा है।
c. External Perception – इसे End User Perception भी कहते है। इसमें यह देखते है कि हमारा Data कैसे और किस Environment में Processing कर रहा है।



ER Model के लाभ: -

a. ER Model बहुत ही सरल होता है।
b. इस मॉडल को डायग्राम के रूप में प्रस्तुत किया जाता है। जिससे हम आसानी से समझ पाते है।
c. इसमें Data Manipulation नहीं होता है।
d. इसका डिजाईन High Level का होता है।

ii. **FUNCTIONAL MODEL (फंक्शनल मॉडल): -**

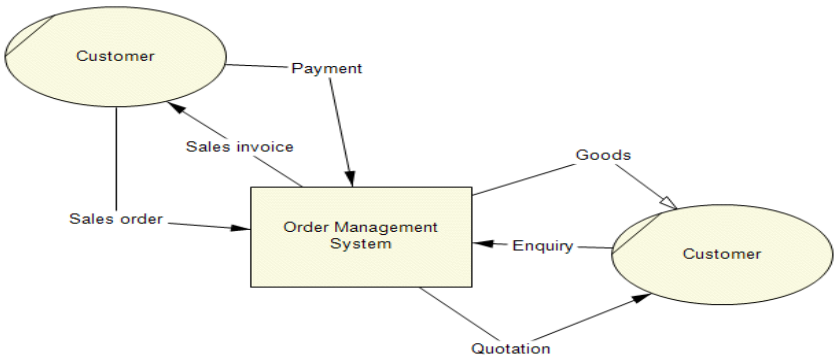
Functional Model एक System Model का प्रकार है। जिसमें Object का प्रयोग करके सभी Real World Problems को Solve किया जाता है। Function Model जो है वह Object तथा Classes के मध्य Association को System को

Describe करता है। Function Model में Classes का प्रयोग Data Information को और उनके Associated Function को Bind करने के लिए करते है।

इसमें कुछ Basic बातों को Decide करना जरूरी होता है जैसे Cost और Time। इसमें यह भी Decide किया जाता है कि सिस्टम में क्या क्या Functionality डालनी है तथा इसमें यह बताया जाता है कि एक Particular Element को कैसे Displayed करना है।

Data Processing Model – Data Processing Model सिस्टम में Data के Flow को Describe करता है। हम Data के Flow को Describe करने के लिए DFD का Use करते है। DFD का पूरा नाम Data Flow Digram है।

- DFD एक Graphical Notation है जिसका Use System में Data के Flow को Represent करने के लिए किया जाता है।
- Data Flow Diagram (DFD) के लिए हम कुछ Guide Lines का प्रयोग कर सकते हैं। ये Guide Lines निम्नानुसार हैं:
- एक DFD ये Show करता है कि Data को कौन Use या Handle कर रहा है।
- एक DFD ये Show करता है कि Business Related Data को किस तरह से व किन माध्यमों (People Inquiry etc.) से Collect किया गया है।
- एक DFD Data पर Perform होने वाले उन Operations को Show करता है, जो Data को एक रूप से दूसरे रूप में Transform करते हैं।



DFD दो प्रकार का होता है: -

a. Logical DFD: - Logical DFD बिज़नेस एक्टिविटी पर केन्द्रित रहता है अर्थात् यह सिस्टम में डेटा के फ्लो तथा सिस्टम के प्रोसेस पर केन्द्रित रहता है ।
b. Physical DFD: - फिजिकल DFD इस बात पर केन्द्रित रहता है कि वास्तव में डेटा फ्लो सिस्टम में किस प्रकार Implement (कार्यान्वित) हुआ है ।

DFD में निम्न चार मुख्य कंपोनेंट्स होते है: -

a. Entities: - डेटा के Source तथा Destination को Entities कहते है, Entities को Rectangle (आयत) के द्वारा प्रदर्शित किया जाता है।
b. Data Flow: - यह डेटा के Movement (गति) को दिखाता है, इसे Arrow द्वारा प्रदर्शित किया जाता है।
c. Process: - यह एक कार्य होता है जो कि सिस्टम के द्वारा किया जाता है, इसे Circle के द्वारा प्रदर्शित किया जाता है।
d. Data Storage: - ऐसी जगह जहाँ डेटा स्टोर होता है डाटा स्टोरेज कहलाता है , इसे Open Rectangle (खुले आयत) के द्वारा प्रदर्शित किया जाता है।

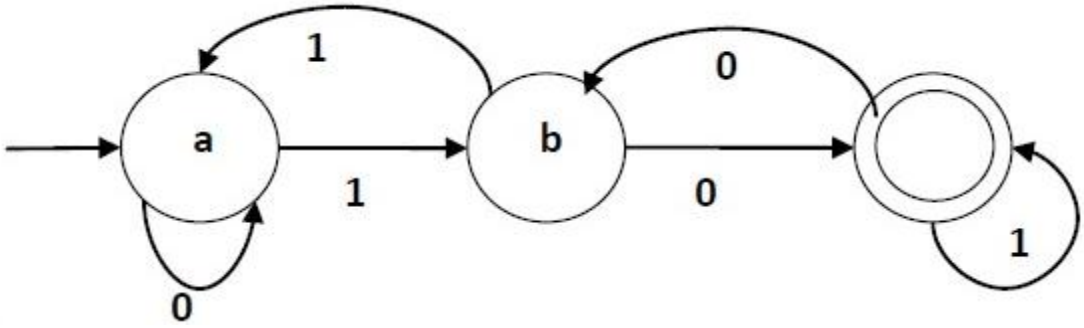
iii. **BEHAVIORAL MODEL (बिहैवियर मॉडल): -**

Behavioral Model सिस्टम मॉडल का एक प्रकार है। Behavioral Model का प्रयोग किसी भी System के संपूर्ण Behavior को Describe करने के लिए किया जाता है। इस Model में हम एक State से दुसरे State तक Data Flow को Describe करते है और प्रोसेस के Execution के दौरान State के Changes को भी Describe करते है। किसी भी System के Behavior को Describe करने के लिए निम्नलिखित Models का Use किया जाता है: –

Data Flow इस बात को Show करता है कि किसी Organization में Data को किस प्रकार से Handle किया जाता है, Data को कहां Store किया जाता है और Data के साथ क्या Processing की जाती है। जबकि Data Modal Data की Internal बातों को तथा Data के बीच की आपसी Relationships को बिना इस बात की परवाह किए Represent करने का काम करता है, कि Data को कौन Handle कर रहा है और Data के साथ किस प्रकार की Processing को Apply किया जा रहा है।

State Machine Model: – State Machine Model किसी भी System के Behavior को Describe करता है। यह External और Internal Events के लिए Response देता है। State Machine Model, सिस्टम State को Node के रूप में और Event को Arcs के रूप में Describe करते है। जब एक Event Occur होता है तब System एक State से दुसरे State में Move करता है।

Finite Automata: - Finite Automata, Patterns को Recognize करने के लिए एक सरल मशीन है। Finite Automata को States Machine या Finite State Machines (FSM) भी कहते है। Finite Automata एक गणितीय मॉडल है जिसका प्रयोग कंप्यूटर प्रोग्राम तथा क्रमबद्ध लॉजिक सर्किटों को डिजाईन करने में किया जाता है।



Finite Automata दो प्रकार के होते है: -

- Deterministic Finite Automata (DFA) एक DFA में, एक विशेष Input Character के लिए Machine केवल एक State में जाती है।
- Non Deterministic Finite Automata (NDFA) को NFA कहते है। NDFA में किसी एक विशेष Input के लिए, मशीन किसी भी State में Move हो सकती है।
  - प्रत्येक DFA, NFA होता है पर प्रत्येक NFA, DFA नहीं होता है।
  - DFA तथा NFA दोनों के पास समान Power होता है। प्रत्येक NFA एक DFA में Translate हो सकता है।
  - DFA तथा NFA में बहुत सारें Final States हो सकते है।
  - DFA का प्रयोग Compiler में Lexical Analysis के लिए किया जाता है।

UNIT 04
SOFTWARE TESTING

SOFTWARE TESTING (सॉफ्टवेयर टेस्टिंग): -

यूजर्स की आवश्यकता के अनुसार सॉफ़्टवेयर / प्रोग्राम में एरर्स (Errors) को खोजने या ढूंढ़ने की प्रक्रिया (Method) को सॉफ़्टवेयर टेस्टिंग कहा जाता हैं।

स्टैंडर्ड्स डेफिनेशन के अनुसार (According to Standard Definition): – एप्लिकेशन की मौजूदा और आवश्यक कंडीशंस (Required Conditions) के बीच डिफरेंसेस (Differences) या दोष (Error) का पता लगाने और सॉफ्टवेयर आइटम के फीचर्स का मूल्यांकन करने के लिए सॉफ़्टवेयर का निरीक्षण करने की प्रक्रिया, ताकि सॉफ्टवेयर ठीक से काम कर सके, को सॉफ्टवेयर टेस्टिंग (Software Testing) कहा जाता है।

सॉफ्टवेयर टेस्टिंग का प्राथमिक उद्देश्य, सॉफ्टवेयर फेलियर को ढूँढना हैं, अगर टेस्टिंग के दौरान सॉफ्टवेयर फेल हो जाता हैं। तो जिन कारणों की वजह से सॉफ्टवेयर फेल हुआ हैं, उन्हें ठीक किया जाता हैं। जिससे सॉफ्टवेयर बिना किसी तकनीकी बाधा के सुचारू रूप से काम कर सकें। डेवलपमेंट टीम के द्वारा सॉफ्टवेयर तैयार किया जाता हैं, उसके बाद तैयार सॉफ्टवेयर को टेस्टिंग टीम के पास भेजा जाता हैं। टेस्टिंग करने वाले व्यक्ती को सॉफ्टवेयर टेस्टर्स कहा जाता हैं।

IMPORTANCE OF SOFTWARE TESTING (सॉफ्टवेयर टेस्टिंग की आवश्यकता): -

निम्न करने के कारण सॉफ्टवेयर टेस्टिंग किया जाता है: -

a. सॉफ्टवेयर में सम्भावित सभी Defects तथा Errors को ढूढ़ने के लिए।
b. सॉफ्टवेयर प्रोडक्ट की क्वालिटी को सुनिश्चित करने के लिए।
c. सॉफ्टवेयर की Performance को बढाने के लिए।
d. यह सिद्ध करने के लिए कि सॉफ्टवेयर में कोई Fault (गलती) नही है।
e. सॉफ्टवेयर की Reliability को बढाने के लिए।
f. यह सुनिश्चित करने के लिए सॉफ्टवेयर Customer की आवश्यकतानुसार बना है या नही ।
g. बिजनेस में बने रहने के लिए।

Test Case Design: -

a. ग्राफ बेस्ड टेस्टिंग मेथड (Graph Based Testing Method): - हर एप्लीकेशन सॉफ्टवेयर के सभी ऑब्जेक्ट्स को पहचान कर (ढूँढ कर) एक आलेख (ग्राफ) तैयार किया जाता हैं। इस ग्राफ के उपयोग से ओबेक्ट रिलेशनशिप को पहचाना जाता हैं और उसी के आधार पर टेस्ट केसेस को एरर ढूँढने के लिए लिखा जाता हैं।
b. एरर गेस्सिंग (त्रुटियों का पूर्वानुमान लगाना): - एरर गेस्सिंग (त्रुटियों का पूर्वानुमान लगाना) यह पूरी तरह से सॉफ्टवेयर टेस्टर और उसके अनुभव पर आधारित होता हैं। जब सिस्टम में एरर नहीं दिखाई देता हैं, तब इसका उपयोग किया जाता हैं। हालांकि ग्राफ बेस्ड टेस्टिंग मेथड में सभी संभावित एरर के बारें में जानकारी मिल जाती हैं।
c. बाउंड्री वैल्यू एनालिसिस (Boundary Value Analysis): - कई बार सॉफ्टवेयर सिस्टम बाउंड्री पर फेल हो जाते हैं, इसी कारण बाउंड्री वैल्यूज की टेस्टिंग करना जरुरी हो जाता हैं। बाउंड्री वैल्यू एनालिसिस में अधिकतम सीमा की वैल्यू तय की जाती हैं। इस डिज़ाइन में जस्ट मैक्सिमम, मिनिमम वैल्यू, इनसाइड/आउटसाइड बाउंड्री, टिपिकल वैल्यू और एरर वैल्यू भी शामिल होते हैं।

d. एक्वीवैलेंस पार्टिशनिंग (Equivalence Partitioning): - एक्वीवैलेंस पार्टिशनिंग, ब्लैक बॉक्स टेस्टिंग की एक मेथड हैं। जिसके अंतर्गत- सिस्टम के इनपुट डोमेन को डाटा क्लासेज में विभाजित किया जाता हैं और इन डाटा क्लासेज से टेस्ट केसेस को तयार किया जाता हैं।

SOFTWARE TESTING STRATEGIES (सॉफ्टवेयर टेस्टिंग स्ट्रैटेजीस): -

किसी सॉफ्टवेयर को टेस्ट करने के लिए किये जाने वाले प्लानिंग को सॉफ्टवेयर टेस्टिंग स्ट्रैटेजीस कहते है। यह कार्य चार चरणों में किया जाता है जो की निम्नानुसार है: -

1. UNIT TESTING (यूनिट टेस्टिंग)
2. INTEGRATION TESTING (इंटीग्रेशन टेस्टिंग)
3. SYSTEM TESTING (सिस्टम टेस्टिंग)
4. ACCEPTANCE TESTING (एक्सेप्टेन्स टेस्टिंग)

1. UNIT TESTING (यूनिट टेस्टिंग): -

यूनिट टेस्टिंग सॉफ्टवेयर टेस्टिंग की एक विधि है जिसमें सॉफ्टवेयर एप्लीकेशन के सबसे छोटे भाग (जिन्हें हम Units कहते है) को Test किया जाता है।आसान शब्दों में कहें तो, "Unit Testing एक ऐसी टेस्टिंग है जिसमें सॉफ्टवेयर को टुकड़ों में तोड़ लिया जाता है, तथा प्रत्येक टुकड़े को बारीकी से Test किया जाता है।"

इस टेस्टिंग का मुख्य उद्देश्य यह सुनश्चित करना है कि सॉफ्टवेयर का प्रत्येक यूनिट का Source कोड सही है तथा इसे Use किया जा सकता है। पहले Adhoc Tools का प्रयोग Units को Test करने के लिए किया जाता था परन्तु आजकल फ्रेमवर्क्स (Java फ्रेमवर्क, .net फ्रेमवर्क तथा PHP फ्रेमवर्क आदि) का प्रयोग किया जाता है।

यूनिट टेस्ट Developers के द्वारा परफॉर्म किये जाते है तथा इसे करने के लिए WHITE BOX TESTING विधि का प्रयोग किया जाता है। यह टेस्टिंग बहुत ही अधिक प्रभावपूर्ण है क्यूंकि इसके प्रयोग के द्वारा अधिकतर Defects को Identify कर लिया जाता है। यूनिट टेस्टिंग में बहुत ही अधिक समय लगता है तथा इसमें बहुत ही धैर्य की आवश्यकता होती है।

यूनिट टेस्टिंग के निम्नलिखित लाभ है: -

a. इस टेस्टिंग के द्वारा हम सॉफ्टवेयर में Defects तथा Bugs को Early Stages में ही ढूँढ लेते है, बाद में Defects तथा Bugs को ढूँढना बहुत अधिक कठिन हो जाता है।
b. यह टेस्टिंग कोडिंग की प्रक्रिया को और अधिक Effective तथा Agile (फुर्तीला) बना देती है जिससे हम सॉफ्टवेयर में ज्यादा से ज्यादा Features को Add कर सकते है।
c. जब हम यूनिट टेस्टिंग कर लेते है तो हमें Manual टेस्टिंग की आवश्यकता बहुत कम रह जाती है। वैसे भी Manual टेस्टिंग बहुत ही अधिक Boring तथा खर्चीली है।
d. हम इस टेस्टिंग के द्वारा सॉफ्टवेयर के Design को बिना Break किये हुए इसके डिज़ाइन को बेहतर बना सकते है।
e. जब हम पहले ही Bugs को Detect कर लेते है तो इससे हमारे Time तथा Cost की बचत हो जाती है।
f. यह टेस्टिंग कोड की Efficiency को बढाता है तथा इसको Maintain करना आसान हो जाता है।

2. INTEGRATION TESTING (इंटीग्रेशन टेस्टिंग): -

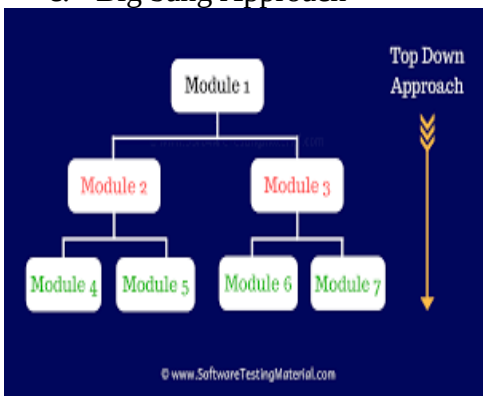
Integration Testing सॉफ्टवेयर टेस्टिंग की एक विधि है जिसमें दो या दो से अधिक सॉफ्टवेयर Units या Modules को एक साथ Combine किया जाता है तथा एक समूह में उनकी टेस्टिंग की जाती है।

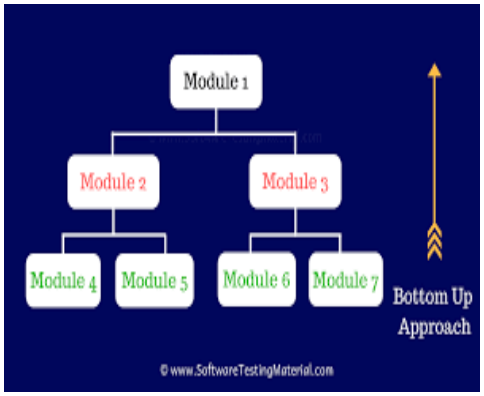
इस टेस्टिंग का मुख्य उद्देश्य Integrated Units के मध्य Bugs तथा Faults को खोजना है तथा Units के मध्य कार्य, Performance तथा विश्वसनीयता को Identify करना है। सामन्यतया Intergation टेस्टिंग को एक Integration टेस्टर के द्वारा Perform किया जाता है।

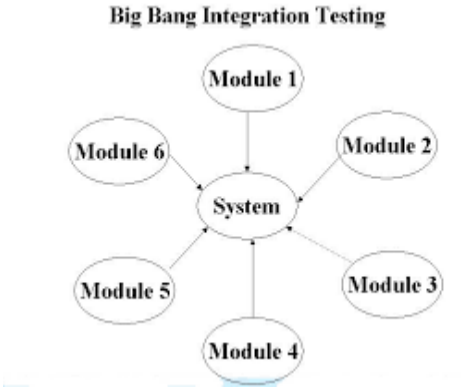
यह टेस्टिंग यूनिट टेस्टिंग के बाद तथा Validation टेस्टिंग के पहले की जाती है। जब यूनिट टेस्टिंग कर ली जाती है तो प्रत्येक Unit को एक-एक करके Integrate किया जाता है और यह क्रिया तब तक चलती है जब तक कि सारें Units को Integrate नही कर लिया जाएँ।

Integration टेस्टिंग की कुछ Approaches निम्नलिखित है:-

a. Top-down Approach
b. Bottom-up Approach
c. Big bang Approach







a. Top-down Approach: - इस Integration टेस्टिंग में Top Level के Integrated Units को सबसे पहले Test किया जाता है तथा उसके बाद उसके नीचे वाले Sub-Units को Test किया जाता है।
b. Bottom-up Approach: - इस Integration टेस्टिंग में सबसे पहले Bottom Level के Sub-Units को Test किया जाता है तथा उसके बाद ऊपर के मुख्य Units को Test किया जाता है।
c. Big Bang Approach: - इस प्रकार की टेस्टिंग में सभी Units को एक साथ Integrate कर लिया जाता है तथा इसके बाद एक समूह में सभी को Test कर लिया जाता है। इस Integration टेस्टिंग में किसी एक यूनिट को तब तक Integrate नही किया जा सकता है जब तक की सभी Units तैयार ना हों।

इस टेस्टिंग का Advantage यह है कि यह छोटे सिस्टम्स के लिए सुविधाजनक है।
इसका Disadvantage यह है कि इसमें Faults तथा Bugs को ढूढ़ पाना बहुत ही मुश्किल होता है अगर कोई Fault मिल भी जाता है तो Fault के होने का मुख्य कारण का पता लगाना बहुत ही Hard हो जाता है तथा इस टेस्टिंग में बहुत ही समय लग जाता है। यह टेस्टिंग बहुत ही Risky होती है क्योंकि अगर हमने सही तरीके से डॉक्यूमेंटेशन नही किया तो इसमें Failure होने का खतरा बढ़ जाता है।

3. SYSTEM TESTING (सिस्टम टेस्टिंग): -

System Testing, Users और System Specifications से एकत्रित की जाने वाली आवश्यकताओं के Against Software का Evaluation (मूल्याकंन) है। System Testing में Validation और Verification शामिल होता है।

a. Software Validation: - Validation यह जांचने की प्रकिया है की Software User के Requirement को पूरा करता है की नही। यह Software Development Life Cycle के अंत में किया जाता है। यदि Software उन Requirements से Match करता है जिसके लिए यह बना है। तो यह Validate है।

Validation Testing सुनिश्चित करता है कि Product Development के तहत User के Requirements के अनुसार है।

Validation Testing इस प्रश्न का उत्तर देता है कि "क्या हम उस उत्पाद को विकसित कर रहे है। जो Users की सभी आवश्यकताओ को इस Software से पूरा करने का Attempts करता है ।

Validation User की आवश्यकताओं पर जोर देती है ।

b. Software Verification: - Verification यह सुनिश्चित करने की प्रकिया है की क्या Software Business Requirements को पूरा कर रहा है, और Proper Specification और तरीको (Methodologies) का पालन करने के लिए विकसित किया गया है।

Verification इस प्रश्न का उत्तर देता है "क्या हम सभी डिजाईन Specifications का पालन करके इस Product को विकसित कर रहे है। Verification Design और System Specification पर केन्द्रित है।

Verification Ensure करता है कि Product Design Specification के अनुसार विकसित किया जा रहा है।

Difference Between Verification and Validation (वेरिफिकेशन और वेलिडेशन में क्या अंतर है)

- वेरिफिकेशन में रिव्यु, मीटिंग और इंस्पेक्शन जैसी प्रोसेस शामिल होती है वहीं वेलिडेशन टेस्टिंग्स में ब्लैक बॉक्स टेस्टिंग, वाइट बॉक्स टेस्टिंग और ग्रे बॉक्स टेस्टिंग आदि होती है।
- वेरिफिकेशन ये सुनिश्चित करता है कि जो प्रोडक्ट आप बना रहे हो वो ठीक से तो बना रहे हो, वहीं वेलिडेशन ये सुनिश्चित करता है कि जो प्रोडक्ट बना है वो ठीक बना है की नहीं।
- वेरिफिकेशन में Quality Assurance टीम होती है जो बनने वाले सॉफ्टवेयर की जाँच करती रहती है जबकि वेलिडेशन में टेस्टिंग टीम होती है जो ये जाँच करती है की प्रोडक्ट अर्थात सॉफ्टवेयर ठीक से ग्राहक की जरूरत और उम्मीद के अनुसार बना है।
- वेरिफिकेशन में Execution Code की आवश्यकता नहीं होती है जबकि वेलिडेशन में Execution Code की आवश्यकता होती है।
- वेरिफिकेशन की प्रक्रिया द्वारा ये पता लगाया जाता है कि जो आउटपुट मिला है वो इनपुट के अनुसार ही है और वहीं वेलिडेशन ये पता लगाता है कि जो इनपुट दिया जा रहा है वो यूजर द्वारा सॉफ्टवेयर ले रहा है की नहीं।
- वेरिफिकेशन, वेलिडेशन से पहले की प्रक्रिया होती है और वेलिडेशन, वेरिफिकेशन के तुरंत बाद की प्रक्रिया होती है।
- वेरिफिकेशन के दौरान Plans, Requirement Specifications, Design Specifications, Code, Test Cases आदि में ध्यान दिया जाता है और वेलिडेशन के दौरान ये पता करते हैं की बना हुआ प्रोडक्ट टेस्ट कंडीशन को full fill कर रहा है या नहीं।
- वेरिफिकेशन में मिले एरर की कॉस्ट वेलिडेशन में पाए गए एरर की तुलना में कम होती है।
- वेरिफिकेशन एक प्रकार की मैन्युअल चेकिंग होती है और वेलिडेशन एक प्रोग्राम बेस्ड होती है।

4. ACCEPTANCE TESTING (एक्सेप्टेन्स टेस्टिंग): -

एक्सेप्टेन्स टेस्टिंग दो प्रकार से किये जाते है: -

a. Alpha Testing: - यह टेस्टिंग Acceptance टेस्टिंग का एक प्रकार है। Customers के लिए सॉफ्टवेयर प्रोडक्ट को Release करने से पहले उसमें संभावित Defects तथा Bugs को Detect करना इस टेस्टिंग का मुख्य उद्देश्य होता है। इस टेस्टिंग को हमेशा Developers के द्वारा Developer Site (जहाँ Engineers काम करते है) पर ही परफॉर्म किया जाता है।

यह टेस्टिंग तब की जाती है जब सॉफ्टवेयर बनकर तैयार हो जाता है अगर Alpha टेस्टिंग के बाद Defects के कारण कुछ बदलाव करने होते है वह कर लिए जाते है।

यह टेस्टिंग दो Steps में पूर्ण होती है, पहले स्टेप में इस टेस्टिंग को Developers के द्वारा परफॉर्म किया जाता है जबकि दुसरे स्टेप को Software Quality Assurance (SQA) टीम के द्वारा परफॉर्म किया जाता है।

यह टेस्टिंग यूजर या कस्टमर के लिए बंद होती है अर्थात् इस टेस्टिंग में यूजर या कस्टमर का कोई Involvement नही होता है। यह टेस्टिंग हमेशा Virtual Environment में ही की जाती है।

b. Beta Testing: - Beta Testing को निम्नलिखित बिन्दुओं के आधार पर आसानी से समझ सकते है। यह भी एक Acceptance टेस्टिंग का एक प्रकार है तथा इसे Field टेस्टिंग भी कहते है। इस टेस्टिंग को End Users or Custmore के द्वारा Real Environment में परफॉर्म किया जाता है।

इसे टेस्टिंग टीम के द्वारा परफॉर्म नही किया जाता है। Beta टेस्टिंग का उद्देश्य Users के Perspective के आधार पर सॉफ्टवेयर में उपस्थित दोष तथा समस्या को Detect करना है तथा Users से उपलब्ध फीडबैक को प्राप्त करना है, ताकि सॉफ्टवेयर के अगले Updated Version में इन समस्याओं को दूर किया जा सकें। यह टेस्टिंग Real Time Environment में की जाती है। Alpha टेस्टिंग के बाद Beta टेस्टिंग की जाती है और Beta टेस्टिंग के बाद कोई और टेस्टिंग नही होती है।

TYPE OF SOFTWARE TESTING (सॉफ्टवेयर टेस्टिंग के प्रकार): -

सॉफ्टवेयर टेस्टिंग के मुख्य रूप से 3 प्रकार हैं-

a. ब्लैक बॉक्स टेस्टिंग (Black Box Testing).
b. वाइट बॉक्स टेस्टिंग (White Box Testing).
c. परफॉरमेंस टेस्टिंग (Performance Testing).

1. Black Box Testing (ब्लैक बॉक्स टेस्टिंग): -

- ब्लैक बॉक्स टेस्टिंग में सॉफ्टवेयर सिस्टम को ब्लैक बॉक्स माना जाता हैं। ब्लैक बॉक्स का मतलब टेस्टिंग के दौरान, सॉफ्टवेयर टेस्टर को सॉफ्टवेयर की कमियों को जांचने के लिए सॉफ्टवेयर के कोड की जानकारी की जरुरत नहीं होती।
- ब्लैक बॉक्स टेस्टिंग का मुख्य उद्देश्य सॉफ्टवेयर की कार्यक्षमता को परखना होता हैं। ब्लैक बॉक्स टेस्टिंग का उपयोग सॉफ्टवेयर डेवलपमेंट लाइफ साइकिल और टेस्टिंग लाइफ साइकिल के दौरान कई स्तर पर किया जाता हैं, जैसे की-यूनिट, इंटीग्रेशन, एक्सेपटन्स एंड रिग्रेशन टेस्टिंग।
- ब्लैक बॉक्स टेस्टिंग प्रक्रिया में टेस्ट डिज़ाइनर वैलिड और इनवैलिड इनपुट्स को सेलेक्ट कर, सॉफ्टवेयर द्वारा दिए जानेवाले उत्तर(आउटपुट) के आधार पर सॉफ्टवेयर की अचूकता और कार्यक्षमता को परखता हैं।

नीचे दिए गए Errors (त्रुटियों) के लिए ब्लैक बॉक्स टेस्टिंग की जाती हैं: -

i. गलत फंक्शन (Incorrect Function).
ii. इंटरफ़ेस एरर (Interface Error).
iii. एक्सटर्नल डेटाबेस या डेटा स्ट्रक्चर में एरर (Errors in External Database or Data Structures).
iv. परफॉरमेंस एरर (Performance Errors).
v. इनिशियलाइज़ेशन (सॉफ्टवेयर की शुरुवात) और टर्मिनेशन (अंत) में तकनिकी समस्या (Initialization and Termination Problems).

ब्लैक बॉक्स टेस्टिंग को निम्नलिखित प्रश्नों के उत्तर पाने के लिए डिजाईन किया जाता हैं: -

i. सॉफ्टवेयर सिस्टम की फंक्शनलिटी को किस प्रकार जांचा गया हैं?
ii. क्या सॉफ्टवेयर सिस्टम की कार्यक्षमता सही हैं?
iii. क्या सॉफ्टवेयर सिस्टम कुछ विशिष्ट इनपुट देने पर ही, सही आउटपुट देता हैं?
iv. सॉफ्टवेयर सिस्टम के काम करने योग्य छोर के डाटा वैल्यू और डाटा वॉल्यूम क्या हैं?
v. सॉफ्टवेयर सिस्टम के कार्यक्षमता पर डाटा के स्पेसिफिक कॉम्बिनेशन का क्या असर होगा?

Black Box Testing Tools (ब्लैक बॉक्स टेस्टिंग में उपयोग किए जाने वाले टूल्स): -

ब्लैक बॉक्स टेस्टिंग में मुख्य रूप से रिकॉर्ड और प्लेबैक टूल्स का उपयोग किया जाता हैं। इन टूल्स का उपयोग यह जांचने के लिए किया जाता हैं कि नए सुधारों ने पहले से सुचारू रूप से काम कर रहे सॉफ्टवेयर सिस्टम में एरर तो नहीं निर्माण किया। रिकॉर्ड और प्लेबैक टूल्स को सामान्यतः टीएसएल (TSL), वीबी स्क्रिप्ट (VB Script), जावा स्क्रिप्ट (Java Script), पर्ल(Perl) जैसी प्रोग्रामिंग लैंग्वेज में लिखा जाता हैं।

Advantages of Black Box Testing (ब्लैक बॉक्स टेस्टींग के फायदे): -

a. ब्लैक बॉक्स टेस्टिंग में टेस्टर और डेवलपर एक दुसरे पर निर्भर नहीं होते, जिसके कारण सॉफ्टवेयर सिस्टम की निष्पक्ष जांच होती हैं और त्रुटियों को खोज कर डेवलपर को सूचित किया जाता हैं।
b. इस टेस्टिंग में, ग्लास बॉक्स टेस्टिंग के तुलना में बड़े कोड के टेस्टिंग के लिए अधिक प्रभावी हैं।
c. इस टेस्टिंग के लिए सॉफ्टवेयर टेस्टर को सॉफ्टवेयर कोड की जानकारी होना आवश्यक नहीं हैं।
d. इस टेस्टिंग में डेवलपमेंट टीम द्वारा सॉफ्टवेयर सिस्टम के पूरा किए जाने के तुरंत बाद ब्लैक बॉक्स टेस्टिंग की जा सकती हैं।
e. इस टेस्टिंग में टेस्टर नॉन-टेक्निकल भी हो सकता हैं।
f. इस टेस्टिंग का उपयोग सॉफ्टवेयर सिस्टम और अपेक्षित परिणामों के बीच के विरोधाभास को जांचने के लिए किया जाता हैं।

2. White Box Testing (वाइट बॉक्स टेस्टिंग): -

- वाइट बॉक्स टेस्टिंग को ओपन बॉक्स टेस्टिंग, क्लियर बॉक्स टेस्टिंग, गलास बॉक्स टेस्टिंग के नाम से भी जाना जाता हैं। वाइट बॉक्स टेस्टिंग-सॉफ्टवेयर टेस्टिंग की एक ऐसी पद्धति हैं जिसमे सिस्टम में एरर ढूँढने के लिए सॉफ्टवेयर सिस्टम की कार्यप्रणाली की जानकारी होना आवश्यक हैं।
- ब्लैक बॉक्स टेस्टिंग के विपरीत, वाइट बॉक्स टेस्टिंग में सॉफ्टवेयर के कोड को ध्यान में लेकर इंटरनल स्ट्रक्चर को जांचने के लिए टेस्ट केसेस निर्धारित किए जाते हैं। वाइट बॉक्स टेस्टिंग के लिए टेस्टर को प्रोग्रामिंग लैंग्वेजेज का ज्ञान होना आवश्यक हैं।
- वाइट बॉक्स टेस्टिंग में टेस्टर, टेस्ट केस इनपुट की मदद से कोड पाथ को निर्धारित करता हैं और टेस्ट के अंत में प्राप्त होने वाले अपेक्षित आउटपुट को तय करता हैं। उदहारण के तौर पर-इन सर्किट टेस्टिंग, जोकि इलेक्ट्रिकल हार्डवेअर टेस्टिंग हैं। इसमें सर्किट पर उपलब्ध हर नोड की जांच कर आउटपुट को मापा जाता हैं।

वाइट बॉक्स टेस्टिंग को निम्नलिखित प्रश्नों के उत्तर पाने के लिए डिजाईन किया जाता हैं: -

i. Structural Test or Interior Testing (संरचनात्मक परीक्षण या आंतरिक परीक्षण) है

ii. मुख्य रूप से Code Structure, Branches, Conditions, Loops etc (कोड संरचना, शाखाओं, परिस्थितियों, लूप आदि) जैसे परीक्षण के तहत सिस्टम के प्रोग्राम कोड के परीक्षण पर ध्यान केंद्रित किया जाता है।

iii. व्हाइट बॉक्स परीक्षण का मुख्य उद्देश्य यह जांचने के लिए कि सिस्टम कैसा प्रदर्शन कर रहा है।

iv. Structural Testing, Logic Testing, Path Testing, Loop Testing, Code Coverage Testing, Open Box Testing (संरचनात्मक परीक्षण, तर्क परीक्षण, पथ परीक्षण, लूप परीक्षण, कोड कवरेज परीक्षण, ओपन बॉक्स परीक्षण) व्हाइट बॉक्स परीक्षण के तहत किया जाता है।

Advantages of White Box Testing (वाइट बॉक्स टेस्टिंग के फायदे): -

a. सॉफ्टवेयर के सोर्स कोड की जांच करके उसी आधार पर टेस्ट केस को लिखा जाता हैं, इससे सॉफ्टवेयर की त्रुटियों को आसानी से ढूँढा जा सकता हैं। उदहारण के तौर पर, सॉफ्टवेयर के सोर्स कोड की जानकारी होने के कारण वाइट बॉक्स टेस्टर, एरर हैन्डेलिंग मैकेनिज्म के जरिए एरर का कम समय में पता लगा सकता हैं।

b. वाइट बॉक्स टेस्टिंग में टेस्टर को सॉफ्टवेयर के आतंरिक ढांचे की जानकारी नहीं होती, इसी कारण अगर एरर हैन्डेलिंग में ज्यादा समय और श्रम की जरुरत पड़ती हैं।

c. वाइट बॉक्स टेस्टिंग का उपयोग, सॉफ्टवेयर डेवलपमेंट के शुरुवाती स्तर में किया जाता हैं, इस प्रक्रिया में टेस्टर द्वारा उपयोग में लाए जानेवाले टेस्ट केसेस की अहम भूमिका होती हैं।

d. वाइट बॉक्स टेस्टिंग में सॉफ्टवेयर टेस्टर को सोर्स कोड की जानकारी होती हैं। अगर कोड में कुछ ज्यादा अनुप्रयुक्त लाइनें लिखी गयी हैं, जिसके कारण एरर उत्पन्न हो सकता हैं। तो उन अतिरिक्त लाइनों को टेस्टर द्वारा हटाया जाता हैं, जिससे सोर्स कोड भी छोटा हो जाता हैं।

वाइट बॉक्स टेस्टिंग और ब्लैक बॉक्स टेस्टिंग में अंतर (difference between white box and black box testing)

क्र.	वाइट बॉक्स टेस्टिंग(WBT)	ब्लैक बॉक्स टेस्टिंग(BBT)
1	वाइट बॉक्स टेस्टिंग, एक सॉफ्टवेयर टेस्टिंग पद्धति हैं- जिसमें सॉफ्टवेयर के आतंरिक ढांचे(इंटरनल स्ट्रक्चर) और कोड की जांच की जाती हैं, और इस विषय में टेस्टर को जानकारी होती हैं।	ब्लैक बॉक्स टेस्टिंग, एक सॉफ्टवेयर टेस्टिंग पद्धति हैं, जिसमें सॉफ्टवेयर के आतंरिक ढांचे(इंटरनल स्ट्रक्चर) और कोड की जांच की जाती हैं, और इस विषय में टेस्टर को जानकारी नहीं होती हैं।
2	वाइट बॉक्स टेस्टिंग में सॉफ्टवेयर के इंटरनल कोड की जांच की जाती हैं।	ब्लैक बॉक्स टेस्टिंग में केवल सॉफ्टवेयर के इनपुट और आउटपुट की जांच की जाती हैं।
3	वाइट बॉक्स टेस्टिंग के दौरान, सोर्स कोड में लिखित हर स्टेटमेंट, लूप की जांच, टेस्ट केसेस के आधार पर की जाती हैं।	ब्लैक बॉक्स टेस्टिंग में, टेस्ट केसेस के आधार पर केवल इनपुट और आउटपुट की जांच की जाती हैं।
4	चूँकि,वाइट बॉक्स टेस्टिंग के दौरान हर स्टेटमेंट की जांच की जाती हैं, इस कारन तयार सॉफ्टवेयर पूरी तरह से त्रुटिरहित(एरर-फ्री) होता हैं।	ब्लैक बॉक्स टेस्टिंग में सोर्स कोड की जांच इनपुट-आउटपुट के आधार पर होती हैं, न की हर लाइन को जांचा जाता हैं। इसी कारन सॉफ्टवेयर को पुरी तरह से त्रुटिरहित(एरर-फ्री) नहीं कहा जा सकता।
5	सॉफ्टवेयर सिस्टम की गुणवत्ता सुधारने में महत्वपूर्ण भूमिका।	सॉफ्टवेयर सिस्टम की गुणवत्ता सुधारने में वाइट बॉक्स टेस्टिंग की तुलना में कम कारगर।
6	प्रोग्रामिंग की जानकारी वाले विशेषज्ञ व्यक्ति ही वाइट बॉक्स टेस्टिंग कर सकते हैं।	ब्लैक बॉक्स टेस्टिंग के लिए प्रोग्रामिंग की जानकारी होना जरुरी नहीं होता।

3. Performance Testing (परफॉरमेंस टेस्टिंग): -

- परफॉरमेंस टेस्टिंग, एक गैर-तकनिकी टेस्टिंग पद्धति हैं, जिसमें सॉफ्टवेयर सिस्टम को स्थिरता, सुचारू रूप से काम करने की काबिलियत जैसे अन्य मानकों पर अलग-अलग वर्कलोड में जांचा जाता हैं।
- परफॉरमेंस टेस्टिंग का मुख्य उद्देश्य सॉफ्टवेयर की विश्वसनीयता, स्थिरता, मापनीयता और उपलब्ध संसाधनों का कम उपयोग करने की क्षमता को परखना हैं।

परफॉरमेंस टेस्टिंग के तरीके

a. लोड टेस्टिंग (Load Testing): - सॉफ्टवेयर सिस्टम एक विशिष्ट प्रकार के कार्यभार(वर्कलोड) में किस प्रकार से काम करता हैं, यह जांचने का सबसे आसान तरीका लोड टेस्टिंग यह हैं। लोड टेस्टिंग से विशिष्ट लोड में काम करने से डेटाबेस, एप्लीकेशन सर्वर पर पड़नेवाला प्रभाव इन जैसी जरुरी चीजों का परीक्षण किया जाता हैं।
b. स्ट्रेस टेस्टिंग (Stress Testing): - स्ट्रेस टेस्टिंग में सॉफ्टवेयर सिस्टम की अधिकतम कार्यभार में काम करने के क्षमता से ज्यादा में उसे (सॉफ्टवेयर) जांचा हैं। जिससे यह पता चल सके अधिकतम सीमा से भार होने पर सॉफ्टवेयर सिस्टम किस प्रकार काम करता हैं। स्ट्रेस टेस्टिंग को सॉफ्टवेयर को नाकाम करने के लिए किया जाता हैं। सिस्टम पर वर्तमान यूजर्स से दोगुना यूजर्स को जोड़ा जाता हैं, और सॉफ्टवेयर को तब तक रन किया जाता हैं जब तक सॉफ्टवेयर नाकाम न हो जाए। इस प्रकार के टेस्ट को सॉफ्टवेयर की मजबूती जांचने के लिए किया जाता हैं।

   उदारहण के तौर पर, वेब सर्वर की स्ट्रेस टेस्टिंग शेल स्क्रिप्ट(शेल स्क्रिप्ट, एक स्क्रिप्ट है, जिसे ऑपरेटिंग सिस्टम के शेल या कमांड लाइन इंटरप्रेटर के लिए इस्तेमाल किया जाता हैं।), बोट्स(बोट्स, सॉफ्टवेयर एप्लीकेशन होते हैं)। आम तौर पर, बोट्स आसन और स्ट्रक्चरली रिपीटेटीव काम तेजीसे करते हैं। इसी बीच कई डिनायल ऑफ़ सर्विस टूल्स का इस्तेमाल, यह जांचने के लिए किया जाता हैं की अधिकतम स्ट्रेस में वे बोट्स किस प्रकार काम करती हैं।

<u>BASIS PATH TESTING (बेसिस पाथ टेस्टिंग): -</u>

Basis Path Testing एक White Box टेस्टिंग तकनीक है जिसे 1980 में McCabe ने प्रस्तावित किया था।
इस टेस्टिंग का प्रयोग Test Cases को डिज़ाइन करने के लिए किया जाता है तथा इसमें Test Cases को डिज़ाइन करने के लिए प्रोग्राम के Flow या Logical Path का प्रयोग किया जाता है।
यह टेस्टिंग सुनिश्चित करती है कि प्रोग्राम के सभी Flow या Logical Path कम से कम एक बार Execute कर लिए गये है। (इसके लिए यह McCabe के Cyclomatic complexity का प्रयोग करता है।) Basis Path टेस्टिंग हमें प्रोग्राम के कोड में उपस्थित सभी त्रुटियों को निर्धारित करने में सहायक होती है।

Basis Path Testing में निम्नलिखित चरण होते है:-
1 सबसे पहले प्रोग्राम के विभिन्न Path को निर्धारित करने के लिए Control Glow Graph को बनाया जाता है।
2 Independent Paths को निर्धारित करने के लिए) Cyclomatic Complexity कैलकुलेट की जाती है।
3 Paths के एक Basis समूह को ढूंढा जाता है।
4:- अंत में प्रत्येक Path के लिए Test Cases को जनरेट किया जाता है।

DEBUGGING (डिबगिंग): -

टेस्टिंग टीम या क्लाइंट से किसी सॉफ्टवेयर में मौजूद खराबियों (Bug) की टेस्ट रिपोर्ट लेने के बाद जब डेवलपमेंट टीम या डेवलपर उस सॉफ्टवेयर में मौजूद Bug को ठीक करने के लिए बैठते हैं, तो इस प्रक्रिया को Debugging कहा जाता है। इसके दौरान डेवेलपर सॉफ्टवेयर में मौजूद नुक्स या डिफेक्ट का कारण जानने की कोशिश करते हैं और उसे ठीक करते हैं।

इसके लिए डेवेलपर कोड के हर लाइन की जाँच करते हैं जिससे पता चल सके कि कोड के किस भाग में डिफेक्ट है। जब उस Bug का पता चल जाता है, वे उस कोड में बदलाव करते हैं और रन करके देखते हैं जिससे पता चल सके कि डिफेक्ट ठीक हुआ या नहीं। इसके बाद डेवेलपर दुबारा उस सॉफ्टवेयर को टेस्टर के पास टेस्टिंग के लिए भेज देते हैं।

Debugging के दौरान कुछ निर्देशों का पालन किया जाता है, जो इस प्रकार हैं:-

- Debugging किसी त्रुटि को हल करने का तरीका है। इस प्रक्रिया को शुरू करने से पहले Debugging टीम के सदस्यों को त्रुटि के सारे कारण समझने चाहिए।
- Debugging के दौरान सॉफ्टवेयर के कोड में कोई दूसरा बदलाव नहीं करना चाहिए। ऐसे में कोड में और भी Bug जुड़ सकते हैं।
- अगर प्रोग्राम के किसी एक हिस्से में Error आता है, तब इस बात की ज्यादा सम्भावना रहती है कि उस प्रोग्राम में और भी जगह Error हो। इसलिए यह सुझाव दिया जाता है कि अगर एक हिस्से में Error मिले तो पूरा प्रोग्राम स्कैन किया जाना चाहिए।
- Error को ठीक करने के लिए अगर प्रोग्राम के किसी कोड में बदलाव किया गया है तो वह सही होना चाहिए और उसकी वजह से बाकि प्रोग्राम के आउटपुट पर प्रभाव नहीं पड़ना चाहिए। इसके लिए रिग्रेशन टेस्टिंग किया जाता है।

Debugging Techniques (Debugging तकनीक)

Debugging के दौरान कई प्रकार के Error सामने आते हैं, जिनमे से कुछ कम हानिकारक होते हैं, जैसे कोड में गलत Function का होना। अगर कोई Error खतरनाक हो तो उससे सिस्टम Failure भी हो सकता है। एक बार जैसे ही सॉफ्टवेयर सिस्टम के Error के बारे में पता चल जाता है, उसको हल करने के लिए निम्नलिखित कदम उठाए जाते हैं: -

a. Defect का पता लगाना: - सिस्टम के error को पहचाना जाता है और उसकी Defect रिपोर्ट बनाई जाती है।
b. डिफेक्ट के सबूत: - एक सॉफ्टवेयर इंजीनियर फिर Defect रिपोर्ट की जाँच करता है। डिफेक्ट की पुष्टि करने से पहले वह कुछ बातें जांचता है जैसे कि क्या सिस्टम में वाकई डिफेक्ट है? क्या वह डिफेक्ट फिर से आ सकता है? डिफेक्ट होने पर सिस्टम कैसे काम कर रहा है आदि।
c. डिफेक्ट का हल: - जैसे ही डिफेक्ट के असली कारण का पता चल जाता है, कोड में बदलाव करके Error को इस तरह सुधारा जाता है कि उसमे दुबारा कोई डिफेक्ट न आये।

(Debugging Strategies): -

Debugging का काम कठिन और समय लेने वाला होता है, इसलिए इसको करने से पहले एक स्ट्रेटेजी बनाना जरूरी होता है। इनमें से कुछ स्ट्रेटेजी इस प्रकार हैं:-

a. Brute Force मेथड: - यह सबसे प्रमुख तरीका है लेकिन कम असरदार है। इसका उपयोग तब होता है जब Debugging के दूसरे तरीके बेअसर हो जाते हैं। यहाँ मेमोरी या स्टोरेज डंप के आधार पर Debugging किया जाता है। प्रोग्राम में ऐसे बहुत से आउटपुट स्टेटमेंट होते हैं जिनमें बहुत से Information होते हैं। इनके जाँच से Error के

कारण का पता चल जाता है। लेकिन यह सब करने के लिए बहुत मात्रा में अनावश्यक (Irrevalent) डाटा की जरूरत होती है इसमें बहुत समय लगता है और कोई एक्सपर्ट ही इसे कर सकता है।

b. Induction स्ट्रेटेजी: – इस प्रक्रिया में यह मान कर चला जाता है कि एक बार जब Error का पता चल जाता है तब उनके बीच का रिलेशन बनाया जाता है फिर Error को इन रिलेशनशिप के आधार पर पहचाना जाता है। इसके अंतर्गत पहले जरुरी डाटा को पहचाना किया जाता है, उनको Organize किया जाता है, डाटा के लिए Hypothesis तैयार किया जाता है फिर उसको Prove करते हैं जिससे Error के होने का सबूत मिलता है। अगर Hypothesis के आधार पर Error नहीं मिलते यानि कि उस Hypothesis में गड़बड़ी है।

c. Deduction स्ट्रेटेजी: – यहां पहले सभी कारण को पहचाना जाता है। फिर हर कारण की जाँच की जाती है। अगर कोई जाँच Invalid निकलता है तो उस कारण को हटा दिया जाता है। यह लगभग Induction स्ट्रेटेजी के समान है।

UNIT 05
SOFTWARE PROJECT MANAGEMENT

SOFTWARE PROJECT MANAGEMENT (सॉफ्टवेयर प्रोजेक्ट मैनेजमेंट): -

किसी सॉफ्टवेयर या प्रोजेक्ट के निर्माण प्रक्रिया को मैनेज करना, Project के management को ही हम software Project management कहते है। किसी भी Software Industries में Software Process को दो भागों में बाटा गया है।

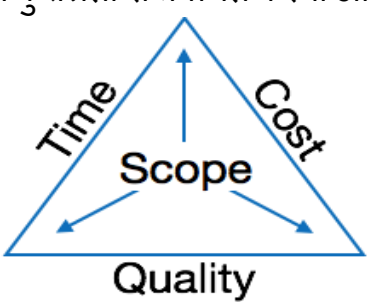
1) Software Creation – किसी सॉफ्टवेयर या प्रोजेक्ट के निर्माण प्रक्रिया को हम software creation कहते है।

2) Software Project Management – जो भी Project develop हो रहा है, उसके Management को ही हम Software Project Management कहेंगे।

Software Project कुछ Segements में Characterized होते है: -

1. हर एक प्रोजेक्ट का अपना Unique Goal होता है।
2. सभी Project अपने अपने Point ऑफ़ View से Design किये जाते हैं ।
3. एक Project का जो भी Programmer होगा वो सिर्फ Creation वाले Part तक ही Operation Perform करेगा, वो day to day operation नहीं करेगा ।
4. हर एक Project के बनने का एक Fix Start Time होगा और एक Fix End Time होता है।
5. जब भी Software Project, Complete हो जायेगा तो वो प्रोजेक्ट किसी Organization का Temporary Phase हो जायेगा ।
6. किसी भी Project को बनाने के लिए कुछ जरूरी Resources होने चाहिए जैसे Time, Manpower, Financial, Material और Knowledge Bank।

अधिकांश Software डेवलपमेंट, Customer की आवश्यकताओं के अनुरूप बने होते हैं। इसलिए एक उत्पाद का अनुभव दूसरे उत्पादों पर लागू नहीं किया जा सकता है। ऐसे सभी Business और (पर्यावरणीय बाधाएं) Software Development में जोखिम लाती हैं इसलिए Software Project को कुशलता से प्रबंधित करना आवश्यक है।



ऊपर दिखाए गए चित्र में software organization के तीन Essential Part है Time, Cost और Quality जो किसी सॉफ्टवेयर के Develop या Scope को दर्शाता है। Quality Product को Delivered करने के लिए यह Software Organization का एक अनिवार्य हिस्सा है। Software Organization ग्राहक को बजट के भीतर एक Particular time पे अपने Customer को उसके लागत के हिसाब से Software प्रदान कराता है। इसलिए, बजट और समय की बाधाओं के साथ Customer आवश्यकताओं को शामिल करने के लिए SOFTWARE प्रोजेक्ट MANAGEMENT आवश्यक है।

अगर हम Time को Consider करे तो वो Cost और Quality पर भी effect लाएगा। जैसे अगर हमको Cost ज्यादा मिलेगा तो software जल्दी और Quality वाला बनेगा। अगर Time ज्यादा है मतलब उस सॉफ्टवेयर का Quality और Cost दोनों कम होगा ।

सॉफ्टवेयर प्रोजेक्ट मैनेजमेंट सिस्टम सामन्यतयः 4P's होते है। जो की निम्नानुसार है: -

i. **PEOPLE: -**

People फैक्टर सॉफ्टवेयर निर्माण के लिए बहुत महत्वपूर्ण होता है। सॉफ्टवेयर को निर्माण करने के लिए विभिन्न Area में बहुत से People कार्य तथा जिम्मेदारी होती है। किसी सॉफ्टवेयर निर्माण के लिए 05 (पांच) प्रकार के People है: -

a. Senior Manager: यह व्यक्ति किसी सॉफ्टवेयर इंडस्ट्रीज के व्यापारिक समस्या को निदान करते है।
b. Project (Technical) Manager: यह व्यक्ति सॉफ्टवेयर निर्माण के लिए Planning, Motivation, Objective, तथा सॉफ्टवेयर टीम (Developers) को Control करने का कार्य करते है।
c. Software Team (Developers): -ये वो टेक्निकल व्यक्ति होते है जो सॉफ्टवेयर के निर्माण के लिए मुख्य रूप से जिम्मेदार होते है।
d. Customore: -ये वो व्यक्ति होते है जो सॉफ्टवेयर के लिए Requirements बताने तथा उसके निर्माण के पश्चात् सॉफ्टवेयर की क्वालिटी टेस्ट करने लिए जिम्मेदार होते है।
e. End User: - ये वो व्यक्ति होते है जो नए निर्मित सॉफ्टवेयर को प्रयोग करते है।

ii. **PRODUCT OR PROBLEM: -**

- किसी प्रोजेक्ट के प्लान करने के पूर्व Product या Problem से संबंधित Objective और Scope निर्धारित करते है। उसके बाद विभिन्न प्रकार के Solution तथा Technical एंड Management Constrains की पहचान करते है।
- बिना इस सूचना के प्राप्त किये किसी प्रोजेक्ट का Estimation Cost ज्ञात करना लगभग असंभव हो जाता है। इन सूचनाओं के आधार पर Project Schedule Risk Calculation तथा Breakdown Point ज्ञात किया जा सकता है।
- Problem Objective के निर्धारण से सॉफ्टवेयर निर्माण की रुपरेखा तैयार किया जा सकता है।
- Scope के द्वारा Primary Data Function तथा सॉफ्टवेयर का Behavior ज्ञात किया जा सकता है।
- Problem Objective और Scope के निर्धारण करने के पश्चात् Problem Solution की पहचान की जाती है।

iii. **PROCESS: -**

- एक सॉफ्टवेयर प्रोसेस किसी सॉफ्टवेयर को निर्मित करने के लिए किसी प्लान को एक फ्रेमवर्क प्रदान करता है।
- सॉफ्टवेयर प्रोजेक्ट को कुछ छोटे छोटे फ्रेम वर्क में परिवर्तित किया जाता है। जिसका आधार सॉफ्टवेयर की Size और Complexity होती है।
- विभिन्न प्रकार के छोटे कार्य, Milestones Work Product और Quality Assurance Point के द्वारा Framework Activity को चालू किया जाता है।
- विभिन्न प्रकार के मॉडल का प्रयोग कर Software Requirement के आधार पर सॉफ्टवेयर का निर्माण किया जाता है।
- प्रोसेस के अंतिम चरण में Umbrell Activity को परफॉर्म किया जाता है जिसमे Software Quality Assurance, Software Configuration Management और प्रोसेस मॉडल का Measurment होता है।

iv. **PROJECT: -**

प्रोजेक्ट एक प्रक्रिया है जो किसी सॉफ्टवेयर प्रोजेक्ट की Cost, Effort, Resources और Time, की गणना करने के लिए किया जाता है। प्रोजेक्ट उच्च Quality और Performance का होना चाहिए। प्रोजेक्ट से सम्बंधित जानकारी निम्न जानकारी होना चाहिए: -

- Meet User Requirements: - किसी प्रोजेक्ट का निर्माण User Requirements को समझकर उसी के अनुसार ही किया जाना चाहिए।
- Meet Schedule Deadline: - किसी प्रोजेक्ट के सभी Milestones को प्लान में Describe किये गए के अनुसार ही किया जाना चाहिए ताकि प्रोजेक्ट सही समय में User को दिया जा सके।
- Be within Budget: - प्रोजेक्ट पर लगने वाला खर्च, दिए गए Budget के अनुसार ही होना चाहिए।
- Produce Quality Deliverables: - प्रोजेक्ट उच्च Quality और Performance का होना चाहिए।

SOFTWARE METRICS (सॉफ्टवेयर मैट्रिक्स): -

- यह एक Measuring Software Performance (सॉफ़्टवेयर प्रदर्शन को मापने), Planning Work item (कार्य आइटम की योजना बनाने), Measuring Productivity (उत्पादकता को मापने), Debugging (डिबगिंग) और Estimation Cost (अनुमान लागत) को मापने के लिए SOFTWARE METRICS (सॉफ़्टवेयर मेट्रिक्स) महत्वपूर्ण हैं। जो भविष्य की परियोजनाओं को अधिक अनुमानित और कुशल बनाता है ।
- यह एक डिग्री का एक माप है जिसमें एक सॉफ्टवेयर प्रणाली कुछ गुण रखती है, जिससे हम किसी सॉफ्टवेयर के क्वालिटी की जाँच करते है।
- यह एक उपाय है जो आवश्यक पैरामीटर को वर्गीकृत और नियंत्रित करता है जो सॉफ्टवेयर विकास को प्रभावित करता है।
- सॉफ्टवेयर मेट्रिक्स प्रबंधन के चार कार्यों से संबंधित हैं: Planning, Organization, Control, or Improvement.

सॉफ्टवेयर मैट्रिक्स के निम्नलिखित गुणधर्म होते: -

1. Simple and computable: - एक सॉफ्टवेयर मैट्रिक्स पढ़ने एवं समझने में आसान होना चाहिए तथा गणना किये जाने में ज्यादा समय तथा ताकत नहीं लगना चाहिए।
2. Consitent: - सॉफ्टवेयर मैट्रिक्स की गणना से प्राप्त परिणाम स्पष्ट तथा निश्चित होना चाहिए।
3. Independent: - यह किसी भी प्रोग्रामिंग लैंग्वेज से स्वतंत्र होना चाहिए।
4. High Quality: - सॉफ्टवेयर मैट्रिक्स से प्राप्त परिणाम के आधार पर विकसित सॉफ्टवेयर उच्च क्वालिटी का होना चाहिए।
5. Reasonable Cost: - सॉफ्टवेयर मैट्रिक्स कम से कम कीमत में किया जाना चाहिए।
6. Robust: - यह requirement परिवर्तन, Error या Defect आने पर भी अपना कार्य सुचारु रूप से करना चाहिए।
7. Margin Error: - इसकी गणना में अंकीय त्रुटि कम से कम होनी चाहिए।

सॉफ्टवेयर मैट्रिक्स तीन (03) प्रकार के होते है: -

a. Product Metrics.
b. Process Metrics.
c. Project Metrics.

a. **PRODUCT METRICS: -** इस मैट्रिक्स में सॉफ्टवेयर प्रोडक्ट के कैरेक्टरस्टिक्स जैसे Size, Complexity Design Features Performance and Quality Level को describe करता है। इस मैट्रिक्स के द्वारा तैयार सॉफ्टवेयर को User Requirement के आधार पर Verify किया जाता है।यह मैट्रिक्स सॉफ्टवेयर इंजीनियर को प्राम्भिक फेज में ही Error को Detect करने में मदद करता है।

प्रोडक्ट मैट्रिक्स निम्न प्रकार के होते है: -

i. Metrics For Analysis Model: - इस प्रकार के मैट्रिक्स में सॉफ्टवेयर के विभ्भिन प्रकार के पहलु जैसे System Functionality, System Size इत्यादि को जांचते है।

ii. Metrics For Design Model:- इस प्रकार के मैट्रिक्स में सॉफ्टवेयर इंजीनियर Quality Design, Architectural Design, Component Level Design इत्यादि का परीक्षण करते है।

iii. Metrics For Source Code: - इस प्रकार के मैट्रिक्स में सॉफ्टवेयर इंजीनियर Source Code Copmlexity, Maintainability इत्यादि का परीक्षण करते है।

iv. Metrics For Testing: - इस प्रकार के मैट्रिक्स में सॉफ्टवेयर इंजीनियर विभ्भिन प्रकार के Test Case Design कर सॉफ्टवेयर का Testing करते है।

v. Metrics for maintenance: - यह मैट्रिक्स सॉफ्टवेयर के Stability का परीक्षण करने के लिए करते है।

b. **PROCESS METRICS: -** इस मैट्रिक्स में सॉफ्टवेयर इंजीनियर सॉफ्टवेयर डेवलपमेंट प्रोसेस के परफॉरमेंस का परीक्षण करते है। किसी सॉफ्टवेयर की गुणवत्ता तथा परिपक्वत्ता का परीक्षण करने पर प्रोसेस मैट्रिक्स निम्न प्रकार से परिणाम प्रदान करता है: -

i. सॉफ्टवेयर के निर्माण के पश्चात प्राप्त होने वाले error की जानकारी।

ii. User के द्वारा Reported defect की जानकारी।

iii. Errors को ठीक करने में लगा समय की जानकारी।

iv. Man - Power की जानकारी।

v. Wait Time की जानकारी।

vi. Estimated Cost और Actual Cost की तुलनात्मक जानकारी।

c. **PROJECT METRICS: -**

यह मैट्रिक्स प्रोजेक्ट मैनेजर के द्वारा किये जाने वाले परीक्षण की जानकारी देता है। इसमें पुराने बने हुए प्रोजेक्ट के आधार पर नए निर्माण किये जाने प्रोजेक्ट का आंकलन किया जाता है। यह मेट्रिक मुख्यतयः दो कारणों से प्रयोग किया जाता है।

1. वर्तमान प्रोजेक्ट के Potential Risk और Problems को दूर करने के लिए जरुरी कदम हेतु।
2. निर्मित होने वाले सॉफ्टवेयर की क्वालिटी को निश्चित अंतराल में परीक्षण करने तथा और बेहतर बनाने के लिए तकनिकी समस्याओं के समाधान के लिए।

SOFTWARE MEASUREMENT (सॉफ्टवेयर मेज़रमेंट): -

Engineer Product (इंजीनियर उत्पाद) या प्रणाली की गुणवत्ता का आकलन करने के लिए और बनाए गए मॉडल को बेहतर ढंग से समझने के लिए, कुछ उपायों का उपयोग किया जाता है। Measurement (मापन) एक सॉफ्टवेयर प्रोजेक्ट में आकलन, गुणवत्ता नियंत्रण, उत्पादकता मूल्यांकन और परियोजना नियंत्रण में मदद करता है।

Software Measurement को दो Category में बांटा गया है: -

I. Direct Measure: - Direct Measure में लागत, प्रयास और, Execution Speed, और अन्य Defects की रिपोर्ट जैसे उत्पादों की प्रक्रिया शामिल है।

II. Indirect Measure: - Indirect Measure में Functionality (कार्यक्षमता), Quality (गुणवत्ता), Complexity (जटिलता), Reliability (विश्वसनीयता), Maintainability (रखरखाव), और कई अन्य उत्पादों जैसे उत्पादों की प्रक्रिया शामिल हैं।

1. **Direct Measurement: -**

- डायरेक्ट मेज़रमेंट को Size Oriented Metrics भी कहते है। इस मेज़रमेंट तकनिकी में Line of Code (KLoC) को Normalization Value के लिए उपयोग किया जाता है।
- यह पद्धति FORTAN और COBOL जैसे प्रोग्रामिनिंग लैंग्वेज के लिए प्रयोग किया जाता है।
- इसमें Productivity(P) को KLOC / Effort से मापते है तथा Effort(E) को Person-Month से मापा जाता है।
- साइज ओरिन्टेड मैट्रिक्स प्रयोग किये गए प्रोग्रामिंग लैंग्वेज पर निर्भर करता है। जिसके कारण High Language Code की तुलना में Assembly Language Code की Prodctivity अधिक होती है।
- यह मेथड Graphical User Interface (GUI) प्रोग्रामिंग प्रोजेक्ट के लिए अनुप्रयुक्त है क्योंकि GUI Project फॉर्म का प्रयोग करते है जिसमे गणना नहीं की जा सकती है।
- प्रोग्रामिंग लैंग्वेज जितना अधिक Expressive होगा Project की Productivity उतना ही कम होगी।
- इस मेथड में Productivity की गणना करने के लिए सभी संस्थाओं को एक जैसे मापदंड प्रयोग करने की आवश्यकता पड़ती है, जो की संभव नही है।
- यह मेथड सॉफ्टवेयर इंडस्ट्रीज में सही Standards स्थापित नहीं करता है जिस कारण इसे सभी जगह मान्य नहीं किया गया है।
- इस मेथड में निम्न की गणना की जा सकती है: -
    i. LOC: - प्रोजेक्ट में कुल कोडिंग लाइन की संख्या।
    ii. Person-Month: - प्रोजेक्ट को पूर्ण करने में लगने वाले की संख्या।
    iii. Cost: - प्रोजेक्ट के कुल लागत।
    iv. Pages: - प्रोजेक्ट में कुल पेजेज की संख्या।
    v. Errors: - प्रोजेक्ट में उपश्तिथ Errors की संख्या।

Formula for Productivity in KLOC **$EV= (S_{opt} + 4 * S_{ml} + S_{Pess}) / 6$**

- जहाँ **EV**: - Stand for the estimation variable. **S_{opt}**: - Stand for the optimistic estimate.
  S_{ml}: - Stands for the most likely estimate. **S_{pess}**: - Stand for the pessimistic estimate.

<u>डायरेक्ट मेजरमेंट से लाभ: -</u>

a. ये मेथड सिंपल और आसान है।
b. यह मेथड Project से Directly सम्बंधित है।
c. इसकी गणना प्रोजेक्ट के Complete होने के बाद की जाती है।
d. यह मेथड सॉफ्टवेयर डेवलपर की ओर से लाभकारी होता है।

<u>डायरेक्ट मेजरमेंट से हानि: -</u>

a. प्रारंभिक फेज में प्रोजेक्ट की LOC की गणना करना संभव नहीं है।
b. Programming Language पर निर्भर होने के कारण विभिन्न Programming Language में प्रोजेक्ट की लागत अलग अलग आती है।
c. इस मेथड के लिए कोई निश्चित मापदंड नहीं है।
d. यह मेथड End User के लिए लाभकारी नहीं है।

<u>Example: -</u> सॉफ्टवेयर में लगने वाला Productivity & Effort ज्ञात कीजिये यदि किसी सॉफ्टवेयर इंजीनियर के द्वारा निर्मित किये गए सॉफ्टवेयर में LOC (S_{opt} = 4000; S_{ml} = 14000; S_{Pess} = 6000) की संख्या दिया गया है।

<u>हल</u>: - दिए गए Formula के अनुसार: - EV = (4 + 4 * 16 + 6) / 6 = 12.33 ~ **12**

2. Indirect Measurment: -

- इनडायरेक्ट मेज़रमेंट को Functional Point Analysis कहते है जिसमे हम Line of Code के बजाये सॉफ्टवेयर के विभिन्न फंक्शन के आधार पर Productivity तथा Effort की गणना की जाती है।
- यह मेथड प्रोग्रामिंग लैंग्वेज पर निर्भर नहीं करता तथा यह मेथड सॉफ्टवेयर इंडस्ट्रीज में प्रयोग किया जाने वाला सबसे प्रचलित है।
- इस मेथड में Fucntion के आधार पर सॉफ्टवेयर के Cost की गणना की जाती है।
- इस मेथड में निम्न Steps के अनुसार FP की गणना की जा सकती है : -
  i. सभी प्रकार Function के लिए Proposed Count Value ज्ञात किया जाता है।
  ii. Unadjusted Function Point (UFP) की गणना की जाती है।
  iii. Total Degree of Influence (TDI) की गणना की जाती है।
  iv. Compute Value Adjustment Factor (VAF) की गणना की जाती है।
  v. Fuction Point की गणना की जाती है।

Step 01: - सभी प्रकार function के लिए Proposed Count Value ज्ञात किया जाता है।

a. External Input = X_1
b. External Output = X_2
c. External Inquiers = X_3
d. Internal Files = X_4
e. External Interface Files = X_5

Step 02: - Unadjusted Function Point (UFP) की गणना की जाती है। UFP की गणना सॉफ्टवेयर की Complexity के आधार पर की जाती है जिसके लिए निमानुसार Table है: -

S.No.	Fuction Type	Value from Step1	SIMPLEX	AVERAGE	COMPLEX
01	External Inputs	X_1 * __	3	4	6
02	External Output	X_2 * __	4	5	7
03	External Inquiers	X_3 * __	3	4	6
04	Internal Files	X_4 * __	7	10	15
05	External Interface Files	X_5 * __	5	7	10
	Total =	SUM (01 to 05)			

Step 03: - Total Degree of Influence (TDI) की गणना की जाती है। इस स्टेप में TDI की values (0 से 5) तक होती है जो कुल 14 Fucntions पर निर्भर करता है। जिसके लिए निमानुसार Table है: -

Values	Software Level
0	No Influent
1	Intial
2	Moderate
3	Average
4	Essential
5	Significant

So **TDI = 14 * (Software Level)**

Step 04: - Compute Value Adjustment Factor (VAF) की गणना की जाती है।

VAF ज्ञात करने के लिए निम्न Formula है: - **VAF = 0.65 + (0.01 x TDI)**

Step 05: - Fuction Point की गणना की जाती है। Fucntional Point ज्ञात करने के लिए निम्न Formula है: -

FP = UFP (Step02) * VAF (Step04)

इनडायरेक्ट मेज़रमेंट से लाभ: -

a. इस मेथड को सॉफ्टवेयर निर्माण के प्रारंभिक फेज में भी गणना की जा सकती है।
b. यह मेथड प्रोग्रामिंग लैंग्वेज से स्वतंत्र होता है।
c. यह वास्तविक Effort से सम्बंधित रहता है।
d. यह मेथड GUI Based Software के लिए भी उपयोगी है।
e. इस मेथड में Fucntion के आधार पर सॉफ्टवेयर के Cost की गणना की जाती है।

इनडायरेक्ट मेज़रमेंट से हानि: -

a. इस मेथड के लिए सॉफ्टवेयर इंजीनियर का प्रशिक्षित होना आवश्यक है।
b. LOC, Max. प्रयोग होने वाला मॉडल है इसलिए FP को LOC में परिवर्तित करने की आवश्यकता होती है।
c. यह मेथड बहुत ही महंगा और समय अधिक लेना वाला है।
d. यह मेथड Software Developer के लिए लाभकारी नहीं है।

PROJECT ESTIMATION (प्रोजेक्ट एस्टिमेशन): -

प्रोजेक्ट एस्टिमेशन एक प्रक्रिया है जो किसी सॉफ्टवेयर प्रोजेक्ट की Cost, Effort, Resources और Time, की गणना करने के लिए किया जाता है। किसी प्रोजेक्ट का एस्टिमेशन निम्न के आधार पर किया जाता है: -

a. पुरानी सूचना / पुराना अनुभव (Past Data / Past Experience)
b. प्रस्तुत दस्तावेज / ज्ञान (Available Documents / Knowledge)
c. अनुमानतः (Assumptions)
d. Risk के पहचान पर (Identification Risk)

प्रोजेक्ट एस्टिमेशन में निम्न चरण में ज्ञात किया जाता है: -

a. किसी निर्मित किये जाने वाले प्रोजेक्ट की Size का एस्टीमेट KLOC या FP में।
b. प्रोजेक्ट में Effort का एस्टीमेट Person - Month या Person - Hours में।
c. प्रोजेक्ट में Schedule का एस्टीमेट Calender Months में।
d. प्रोजेक्ट में Project Cost का एस्टीमेट Currency में।

प्रोजेक्ट एस्टिमेशन की दो तकनीक होती है: -

1. DECOMPOSITION TECHNIQUES: -
2. EMPIRICAL ESTIMATION MODEL: -

1. **DECOMPOSITION TECHNIQUES: -**

डिकम्पोज़िशन तकनीक बहुतायत प्रयोग में आने वाली प्रोजेक्ट एस्टिमेशन तकनीक है। यह Divide and Conquer पद्धति में कार्य करता है। इस पद्धति में प्रोजेक्ट की Size, Effort और Cost को Step by Step Calculate किया जाता है।

Step 1: - सबसे पहले सॉफ्टवेयर के निर्माण के Scope को जाना जाता है।

Step 2: - सॉफ्टवेयर के Size की गणना (KLOC या Functional Point) के द्वारा की जाती है।

Step 3: - किसी प्रोजेक्ट को विभिन्न प्रकार के Software Engineering Activities में Break कर सॉफ्टवेयर प्रोजेक्ट के Estimate और Cost की गणना की जाती है।

Step 4: - Step 2 और Step 3 से प्राप्त परिणामो की तुलना किया जाता है। यदि प्राप्त परिणाम लगभग सामान आये तो गणना सही होती है अन्यथा परिणाम की भिन्नता का पता लगाया जाता है।

Step 5: - परिणाम की भिन्नता का कारन को पता कर उसे दूर करने का प्रयास किया जाता है।

Reliable Estimation के लिए निम्न लिखित दिशा निर्देश होते है: -

- प्रोजेक्ट का एस्टिमेशन पूर्व में किये गए प्रोजेक्ट के आधार पर किया जाना चाहिए।
- डिकम्पोज़िशन तकनीक जितना अधिक सामान्य हो उतना ही प्रोजेक्ट की Cost और Effort ज्ञात करना आसान होता है।
- एक से अधिक मॉडल का प्रयोग कर सॉफ्टवेयर के Cost और Effort के Reliability की गणना किया जाना चाहिए।

डिकम्पोज़िशन तकनीक के Issues: -

- प्रोजेक्ट के एस्टिमेशन की Accuracy प्रोजेक्ट के लिए Collect किये गए सूचनाओं की गुणवत्ता पर निर्धारित करता है।
- उपलब्ध कराये गए Resource का Utilization पूर्ण रूप से नहीं हो पता है।
- समय सीमा में प्रोजेक्ट को पूरा नहीं होने के कारण भी Project Estimation सही से नहीं किए जा पता है।

2. **EMPIRICAL ESTIMATION MODEL: -**

- एम्पिरकाल एस्टिमेशन मॉडल के अंतर्गत CoCoMo (कोकोमो) मॉडल आता है। इस मॉडल के द्वारा ही किसी Software Project का एस्टिमेशन किया जाता है।
- CoCoMo (Constructive Cost Model): - CoCoMo मॉडल का उपयोग किसी भी सॉफ्टवेयर के डेवलपमेंट की कॉस्ट ऐस्टीमेशन (Cost Estimation) के लिए किया जाता है जो सॉफ्टवेयर की Cost को Evaluate करता है।
- Boehm के अनुसार किसी प्रोजेक्ट के डेवलपमेंट (Development) के लिए लगने वाला Cost सॉफ्टवेयर के Characteristics के साथ साथ उसमें लगने वाली Development Team और Development Environment के ऊपर भी निर्भर करता है जो किसी न किसी रूप से उसके Cost को Evaluate करता है।
- COCOMO Model में किसी Project को Develop (विकसित) करने के लिए उसमे लगने वाले Time (Month) और Person (लोगों) की संख्या की जरूरत को Estimate करने के लिए Effort Equation (समीकरण) का प्रयोग किया जाता है। Effort Estimation के Unit के लिए PM (Person - Months) का प्रयोग होता है। यहाँ पर PM (Person - Months) किसी Project को बनाने में किसी एक Person द्वारा एक महीने (months) में उसके Work - Done (कार्य-पूर्ण) से है।
- Boehm के अनुसार, किसी भी Software को Cost Estimation के आधार पर निम्न तीन (Three) Stages में बांटा गया हैं: -
    i. Basic COCOMO Model.
    ii. Intermediate COCOMO Model.
    iii. Complete COCOMO Mode.

i. **BASIC COCOMO MODEL: -**

Basic COCOMO Model: - बेसिक कोकोमो मॉडल केवल एक Single Predictor Variable (Size in DSI) और तीन Software Development Modes का उपयोग कर सॉफ्टवेयर विकास प्रयास का अनुमान लगाता है।

यह मॉडल Static तथा Single Valued होता है जो कि सॉफ्टवेयर डेवलपमेंट Effort तथा Cost को Function के Program Size की तरह Compute करता है तथा Program Size जो है वह अनुमानित (Estimated) lines of Code (LOC) में व्यक्त होता है।

सामान्यतया इस मॉडल का प्रयोग छोटे तथा मध्यम आकर के सॉफ्टवेयर प्रोजेक्ट्स में किया जाता है। इस मॉडल में Cost को Estimate (Effort and Duration) करने के लिए निम्नलिखित सूत्र का प्रयोग किया जाता है: -

Effort (E) = $a^* (KLOC)^b$ Duration (D) = $c^* (E)^d$ जहां: -

- KLOC सॉफ्टवेयर का अनुमानित आकर है।
- a, b, c and d सॉफ्टवेयर की प्रत्येक Category के लिए नियतांक (Constant) है।
- Effort सॉफ्टवेयर को विकसित करने में लगा कुल Effort PM (Person-Month) में है।
- Duration (D) सॉफ्टवेयर को विकसित करने के लिए अनुमानित समय Months में है।

Basic COCOMO Model Constant Table: -

SOFTWARE PROJECT TYPE	a	b	b	d
Organic	2.4	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

Limitations of Basic CoCoMo: -

- इसकी Accuracy सीमित है क्योंकि कारकों की कमी की वजह से सॉफ्टवेयर लागत पर महत्वपूर्ण प्रभाव पड़ता है।
- ये Hardware Issue को Ignore करता हैं।

ii. **INTERMEDIATE COCOMO MODEL: -**

ये Basic COCOMO Model का Extension हैं। Intermediate COCOMO Model अपने Environment को Consider करते हुए किसी Value या Cost को estimate करने के लिए 15 Additional Predictor Use करता हैं।

जिस Development Environment में हमें Project को Develop करना है उसके क्या Attributes है, वो 15 कॉस्ट ड्राइवर, कॉस्ट ऐस्टीमेशन के टाइम पर हमें हेल्प करता है। इससे हमारे एक्यूरेसी Cost Estimation की Increase होती है।

Intermediate COCOMO सॉफ्टवेयर डेवलपमेंट Effort को Program Size के फंक्शन तथा Cost Drivers के समूह की तरह कंप्यूट करता है। Cost Drivers प्रोजेक्ट में लगे Time तथा Effort को निर्धारित करता है।

यह मॉडल बेसिक COCOMO मॉडल से बेहतर परिणाम देता है क्योंकि इसमें Cost Drivers का प्रयोग किया जाता है। इस मॉडल में Cost को Estimate (Effort and Duration) करने के लिए निम्नलिखित सूत्र का प्रयोग किया जाता है: -

Effort (E) = $a^* (KLOC)^b * EAF$ Duration (D) = $c^* (E)^d$ जहां: -

- KLOC सॉफ्टवेयर का अनुमानित आकर है।
- a, b, c and d सॉफ्टवेयर की प्रत्येक Category के लिए नियतांक (Constant) है।
- Effort सॉफ्टवेयर को विकसित करने में लगा कुल Effort PM (Person-Month) में है।
- Duration (D) सॉफ्टवेयर को विकसित करने के लिए अनुमानित समय Months में है।
- EAF (Effort Adjustment Factor) को 15 Cost Drivers की सहायता से गणना किया जाता है।
- Intermediate COCOMO Model Constant Table: -

SOFTWARE PROJECT TYPE	a	b	b	d
Organic	3.2	1.05	2.5	0.38
Semi Detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

iii. **COMPLETE COCOMO MODEL: -**

यह मॉडल Intermediate COCOMO MODEL का Extension होता है। इसे Detailed or Advanced COCOMO Model भी कहते है।यह मॉडल इंटरमीडिएट मॉडल से भिन्न होता है क्योंकि यह प्रोजेक्ट के प्रत्येक Phase के लिए Effort Multipliers का प्रयोग करता है।

Complete कोकोमो में प्रत्येक Subsystem की Cost को अलग-अलग Estimate किया जाता है। इस विधि के कारण त्रुटियाँ बहुत ही कम होती है।

Basic तथा Intermediate COCOMO मॉडल की कमी यह है कि यह सॉफ्टवेयर प्रोजेक्ट को Single Homogeneous Entity की तरह Consider करता है। इस कमी को complete कोकोमो मॉडल दूर करता है।

Complete COCOMO मॉडल Estimation को कैलकुलेट करने के लिए बहुत ही जटिल Procedures का प्रयोग करता है।