

7. SOFTWARE TESTING AND COST ESTIMATION

Contents:

- 7.1 System testing
- 7.2 Component testing
- 7.3 Test case design
- 7.4 Test automation
- 7.5 Estimation techniques
- 7.6 Project duration and staffing

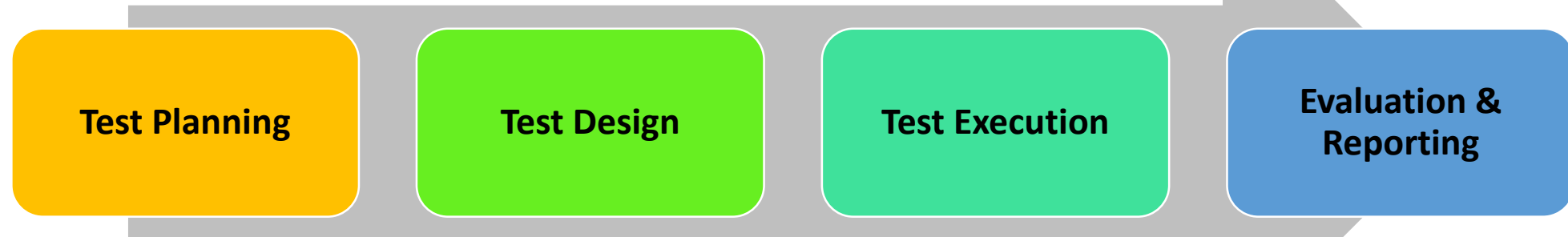
7.1 System testing

7.1 Software Testing

- Software testing is the process of executing a software system **to determine whether if it matches its specification** and executed in its intended environment or not.
- Software testing is done to confirm that:
 - ✓ It is error-free
 - ✓ Working as to our expectation
- When we test software, we execute a program using artificial data.
- Testing is intended to show that a **program does what it is intended to do** and to **discover program defects** before it is put into use.
- **Testing can reveal the presence of errors, but NOT their absence.**

Testing Process

Software testing is the process of executing a software system to determine whether it matches its specification and executed in its intended environment or not.



- Test plan documents the strategy to meet design specification and other requirements.
- It is the act of creating and writing test suites for testing a software.
- Actual testing of softwares.
- Different factors considered.
- test conditions , test environment, etc.
- The test is measured against the exit criteria.
- Exit criteria refer to the benchmark or standard that the quality of software has to meet.
- Finally, a test report is documented.

Goals of Software Testing

1. To **demonstrate** to the developer and the customer that the **software meets its requirements**.
 - Leads to **validation testing**: you expect the system to perform correctly using a given set of test cases that reflect the system's expected use.
 - A successful test shows that the system operates as intended.
2. To **discover** situations in which the behavior of the software is **incorrect, undesirable or does not conform to its specification**.
 - Leads to **defect testing**: the test cases are designed to expose defects; the test cases can be deliberately obscure and need not reflect how the system is normally used.
 - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Validation and Defect testing

- The first goal leads to validation testing, The second goal leads to defect testing
- **Validation testing**
 - To demonstrate to the developer and the system customer that the software meets its requirements
 - A successful test shows that the system operates as intended.
- **Defect testing**
 - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
 - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

Verification Vs Validation

- Testing is part of a broader process of software **verification and validation** (V & V).
- **Verification:** "Are we building the product right".
 - Software verification is the process of checking that the software meets its stated functional and non-functional requirements.
 - The software should conform to its specification.
 - verification focuses on the correctness of the development process, making sure that the product is built according to the specifications.
- **Validation:** "Are we building the right product".
 - The software should do what the user really requires.
 - The aim of software validation is to ensure that the software meets the customer's expectations.
 - Validation, on the other hand, focuses on the end product, ensuring that it meets the customer's needs and functions as intended.
 - It goes beyond checking conformance with the specification to demonstrating that the software does what the customer expects it to do

- *Example:*
- Suppose you are developing a web application that requires user authentication. Verification in this context would involve checking that the implemented authentication system adheres to the specified requirements. This may include reviewing the code, examining design documents, and ensuring that the authentication process aligns with the defined security standards.
- Continuing with the web application scenario, validation would involve testing the implemented authentication system to ensure it functions as intended and meets the user's requirements. This could include conducting user acceptance testing (UAT) to verify that the authentication process aligns with user expectations and provides the desired level of security.

Goals of V & V

- The **goal of V & V** is to establish confidence that the system is **good enough for its intended use**, which depends on:
 - **Software purpose:** the level of confidence depends on how critical the software is to an organization.
 - **User expectations:** users may have low expectations of certain kinds of software.
 - **Marketing environment:** getting a product to market early may be more important than finding defects in the program.

Inspections and testing

- **Software inspections** involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections do not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

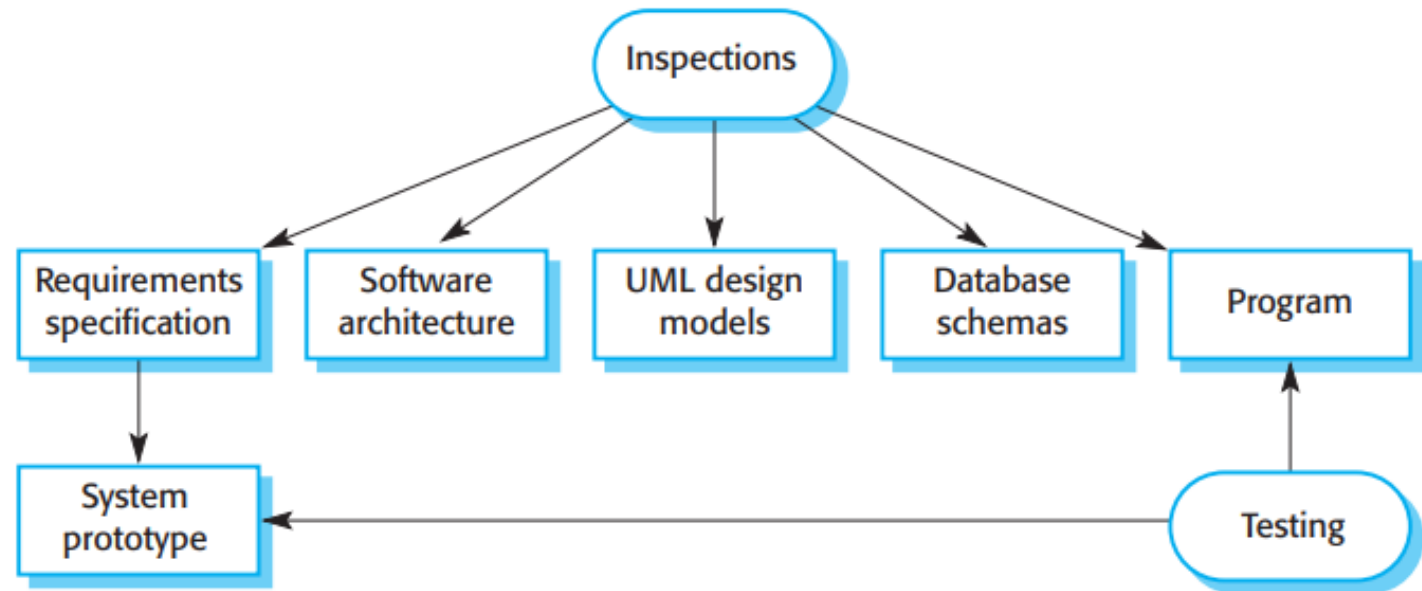


Figure: inspection testing

Inspections and Testing are complementary

- **Inspections and testing are complementary** and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Stages of Testing

- Typically, a commercial software system has to go through **three stages of testing**:
 1. **Development testing**: the system is tested during development to discover bugs and defects.
 2. **Release testing**: a separate testing team test a complete version of the system before it is released to users.
 3. **User testing**: users or potential users of a system test the system in their own environment.

1) Development Testing

- Development testing includes all testing activities that are carried out by the team developing the system:
 - a) **Unit testing:** individual program units or object classes are tested; should focus on testing the functionality of objects or methods.
 - b) **Component testing:** several individual units are integrated to create composite components; should focus on testing component interfaces.
 - c) **System testing:** some or all of the components in a system are integrated and the system is tested as a whole; should focus on testing component interactions.

a) Unit testing

- Unit testing is the process of **testing individual components in isolation**. It is a defect testing process.
- Units may be:
 - **Individual functions** or methods within an object;
 - **Object classes** with several attributes and methods;
 - **Composite components** with defined interfaces used to access their functionality.

b) Component testing

- **Software components** are often composite components that are **made up of several interacting objects**.
- You access the functionality of these objects through the **defined component interface**.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types include:
 - **Parameter interfaces**: data passed from one method or procedure to another.
 - **Shared memory interfaces**: block of memory is shared between procedures or functions.
 - **Procedural interfaces**: sub-system encapsulates a set of procedures to be called by other sub-systems.
 - **Message passing interfaces**: sub-systems request services from other sub-systems.

c) System Testing

- System testing during development involves **integrating components** to create a version of the system and then **testing the integrated system**.
- The focus in system testing is **testing the interactions between components**.
- System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the **emergent behavior** of a system.
- During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components.
- The complete system is then tested.
- Components developed by different team members or sub-teams may be integrated at this stage.
- System testing is a collective rather than an individual process.

2. Release Testing

- Release testing is the process of **testing a particular release** of a system that is **intended for use outside of the development team**.
- The primary goal of the release testing process is to **convince the customer of the system that it is good enough for use**.
- Release testing, therefore, has to show that the system delivers its specified functionality, performance and dependability, and that it does not fail during normal use.
- Release testing is usually a black-box testing process where **tests are only derived from the system specification**.

3. User Testing

- User or customer testing is a stage in the testing process in which **users or customers provide input and advice on system testing**.
- User testing is essential, even when comprehensive system and release testing have been carried out.
- Types of user testing include:
 - **Alpha testing**: users of the software work with the development team to test the software at the developer's site.
 - **Beta testing**: a release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
 - **Acceptance testing**: customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

White Box and Black Box testing

White Box Testing

- Aka **Structural** or **'glass box'** or **'clear-box testing'**
- The structural testing is the **testing of the structure** of the system or component.
- In white-box testing, the testers are **required to have the knowledge of the internal implementations of the code**. Here the testers require knowledge of how the software is implemented, how it works.
- For example, a structural technique wants to know **how loops in the software are working**. Different test cases may be derived to exercise the loop once, twice, and many times. This may be done regardless of the functionality of the software.
- White-box testing can be **used at all levels of testing**. Developers use structural testing in component testing and component integration testing, especially where there is good tool support for code coverage.

Black Box Testing

- Aka **Functional** Testing or **component** Testing or **Specification** Testing
- Black-box Testing is a testing technique that is **used to test the features/functionality of the system or Software.**
- In Black-box Testing we need to check if each components are functioning as expected or not, so it is also called as “**Component Testing**”.
- Generates test cases based on the functionality of software.
- The internal program structure is hidden from the testing process.

7.2 Component Testing

Types of Black Box (Component) testing

- a) Boundary Value analysis (BVA)
- b) Equivalence Class Testing
- c) Decision Table based testing

a) Boundary Value analysis (BVA)

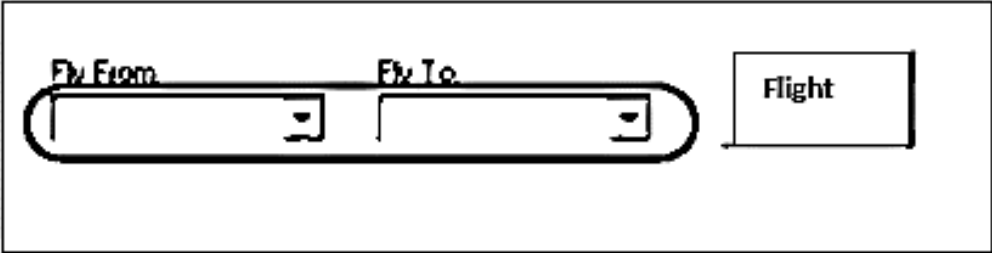
- BVA is the testing technique meant for **testing the boundaries of the input domain for errors**.
- Here, we have both **valid boundaries** (in the valid partitions) and **invalid boundaries** (in the invalid partitions).
- For eg.
 - A form accepts product delivery days: a minimum of 2 days and maximum of 14 days are noted on the form.
 - Boundary value test might put values of 1, 2, 14, 15 to determine how the function reacts to data.

b) Equivalence Class Testing

- It is a software testing technique that **divides the input data of the application under test into each partition** at least once of equivalent data from which test cases can be derived.
- It reduces the time required for performing testing of a software due to less number of test cases.
- **For eg. Consider that a form accepts an integer in the range of 100 to 999.**
 - Valid equivalence class partition : 100 to 999 inclusive
 - Non-valid equivalence class partition :
 - less than 100,
 - more than 999,
 - decimal no,
 - alphabets and non numeric characters.

c) Decision Table based testing

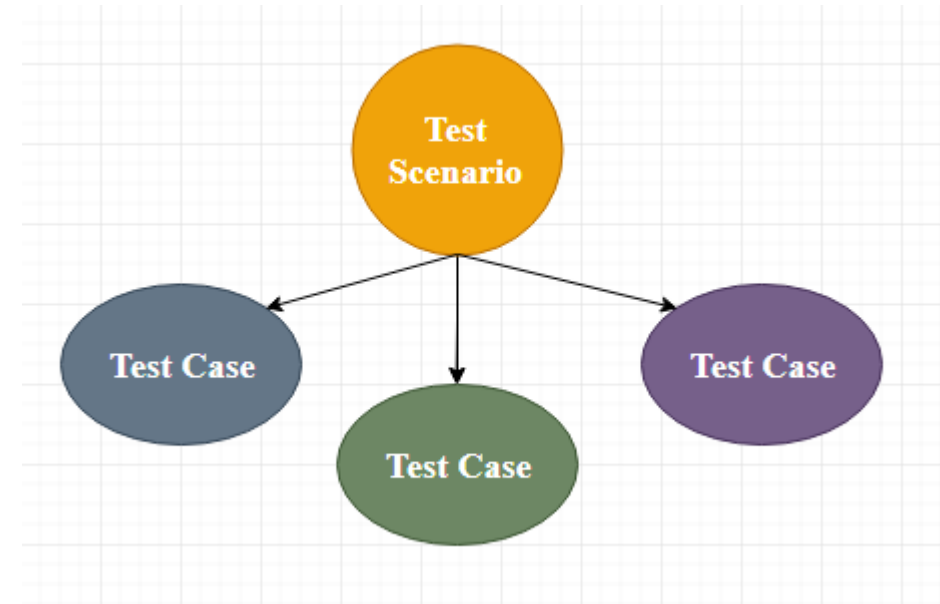
- A decision table based testing **uses a decision table**. A decision table is a good way to deal with combinations of inputs, which produce different results.
- Lets us consider the behavior of “FLIGHT” button for different combination of FLY FROM and FLY TO.

				
Conditions	Rule 1	Rule 2	Rule 3	Rule 4
FLY FROM	F	F	T	T
FLY TO	F	T	F	T
<u>Outcome</u> FLIGHT	F	F	F	T

7.3 Test case design

Test case

- A test case is a **document**, which has a set of test data, preconditions, expected results and post-conditions, developed for a particular test scenario in order to verify compliance against a specific requirement.
- The test case is defined as a group of conditions under which a tester determines whether a software application is working as per the customer's requirements or not.
- Test case designing includes preconditions, case name, input conditions, and expected result.
- A test case is a first level action and derived from test scenarios.



Test Case Example

1. Test suite ID:	TS001
2. Test case ID:	TC001
3. Test case summary:	To verify that clicking the “Generate Coin” button generates coins.
4. Prerequisites:	1. User is authorized. 2. Coin balance is available.
5. Test procedure:	1. Select the coin denomination in the denomination field. 2. Enter the number of coins in the Quantity field. 3. Click Generate coin.
6. Expected result:
Actual result:
7. Status:	Pass / Fail
8. Remarks:	This is a sample test case.
9. Created by:	Er. Shiva Ram Dam
10. Date of creation:	07 /08/ 2016
11. Executed by:	Er. Subash Raj Bhat
12. Date of execution:	07 /08/ 2016
13. Test environment:	OS : Windows 8 Browser : Google chrome

Fig: Test case

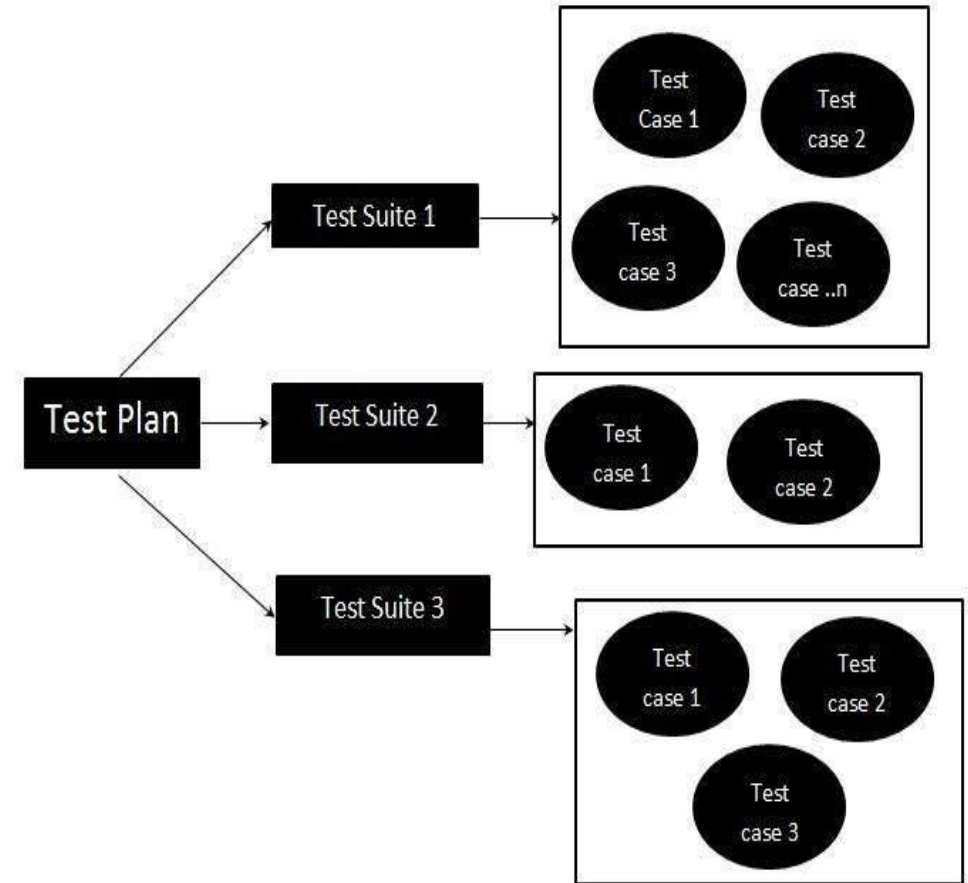


Fig: Test suite

7.4 Test automation

Test Automation

- Test automation refers to **the use of specialized tools and frameworks** to automate the execution of test cases and the validation of software applications.
- The primary goal of test automation is **to improve the efficiency, effectiveness, and coverage of the testing process.**
- Instead of manually executing repetitive and time-consuming test cases, automation allows testers to write scripts that can automatically perform the testing tasks.
- Various tools are available for test automation, ranging from open-source tools like **Selenium** and **Appium** for web and mobile applications, respectively, to commercial tools such as **UFT (Unified Functional Testing)** and **TestComplete**.
- These tools **provide features for recording and playback, scripting, and reporting.**
- In automated unit testing, you make use of a test automation framework (such as **JUnit**) to write and run your program tests.

7.5 Estimation techniques

Estimation of project

- Estimating project schedules is difficult.
- You have to make initial estimates on the basis of an incomplete user requirements definition.
- The software may have to run on unfamiliar platforms or use new development technology.
- The people involved in the project and their skills will probably not be known.
- There are so many uncertainties that it is impossible to estimate system development costs accurately during the early stages of a project.
- Nevertheless, **organizations need to make software effort and cost estimates.**
- Two types of techniques can be used for making estimates:
 - **Experience-based techniques**
 - **Algorithmic cost modeling**

Types of Estimation techniques

1. Experience-based techniques:

- The estimate of future effort requirements is **based on the manager's experience of past projects** and the application domain.
- Essentially, the manager makes an informed judgment of what the effort requirements are likely to be.

2. Algorithmic cost modeling:

- Also known as Heuristic estimation
- In this approach, a **formulaic approach** is used to compute the project effort based on estimates of product attributes, such as size, process characteristics, and experience of staff involved

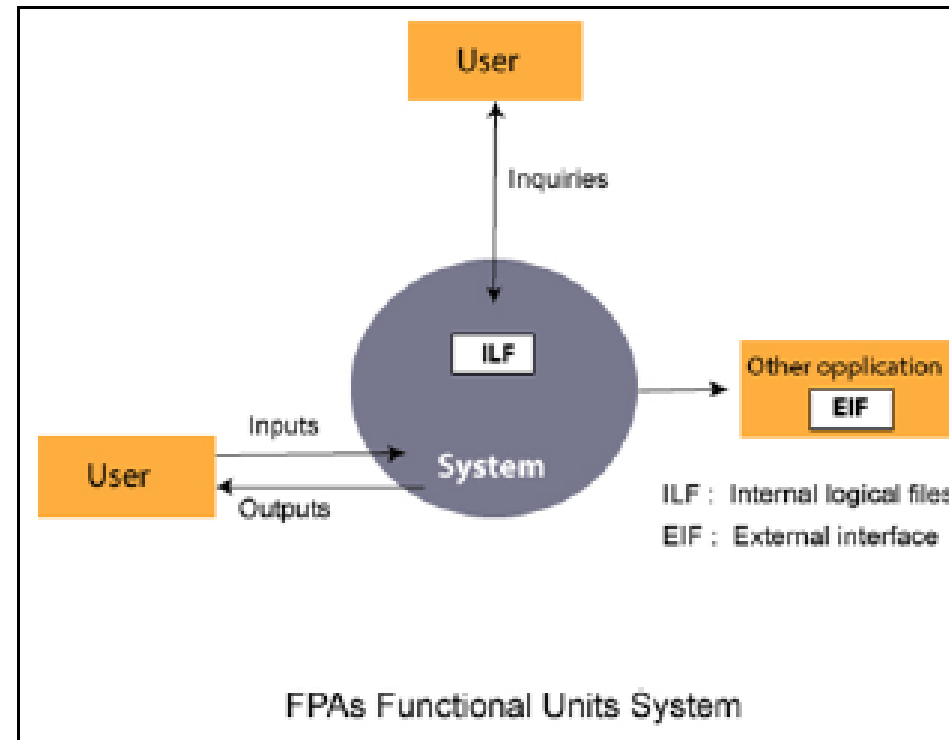
- Most algorithmic models for estimating effort in a software project are based on a simple formula:

$$\text{Effort} = A \times \text{Size}^B \times M$$

- Where:
- **A:** a **constant** factor, which depends on local organizational practices and the type of software that is developed.
- **Size:** an assessment of the **code size** of the software or a functionality estimate expressed in function or application points.
- **B:** represents the **complexity** of the software and usually lies between 1 and 1.5.
- **M:** is a factor that takes into account process, product and development attributes, such as the dependability requirements for the software and the experience of the development team.

Functional Point (FP) based Estimation

- An FP (Functional Point) is a unit of measurement to express the amount of business functionality that a software provides to a user. It measures functionality from user's point of view.
- In such method, the software product is directly dependent on the number of functions or features it supports.
- The approach is to identify and count a number of unique function types:
 - External inputs (eg: file names)
 - External outputs (eg: reports, message)
 - Queries (interactive inputs needing a response)
 - External files or interface (files shared with other software systems)
 - Internal files (invisible outside the system)



Computation of FP:

Mathematically;

$$FP = \text{Count Total} \times CAF$$

Where, Count Total = sum of all FP entries

$$CAF = 0.65 + 0.01 \times \sum F_i$$

CAF → Complexity Adjustment Factor

The F_i ($i=1$ to 14) is the value adjusted factor based on response to 14 questions.

```
string aspects[14] = {  
    "reliable backup and recovery required ?",  
    "data communication required ?",  
    "are there distributed processing functions ?",  
    "is performance critical ?",  
    "will the system run in an existing heavily utilized operation",  
    "on line data entry required ?",  
    "does the on line data entry require the input transaction to",  
    "are the master files updated on line ?",  
    "is the inputs, outputs, files or inquiries complex ?",  
    "is the internal processing complex ?",  
    "is the code designed to be reusable ?",  
    "are the conversion and installation included in the design ?",  
    "is the system designed for multiple installations in differer",  
    "is the application designed to facilitate change and ease of",  
};
```

<u>FACTOR</u>	<u>VALUE</u> <u>(Fi)</u>
1. Back-up and Recovery ?	4
2. Data Communication ?	2
3. Distributed Processing ?	0
4. Performance Critical ?	4
5. Existing Operational Environment ?	3
6. On-line Data Entry ?	4
7. Input transactions over multiple Screens?	5
8. Online Updates ?	3
9. Information Domain Values Complex ?	5
10. Internal Processing Complex?	5
11 Code Designed for reuse?	4
12. Conversion / installation in Design?	3
13. Multiple Installations?	5
14. Application Designed for change ?	5
<hr/>	
$\Sigma (F_i)$	
52	

Numerical: Given the following values, Compute FP when all complexity adjustment factors and weighting factors are average.

Information Domain Value	Count	Weighting Factor		
		Simple	Average	Complex
External Inputs (EI)	50	3	4	6
External Outputs (EO)	40	4	5	7
External Inquiries (EQ)	35	3	4	6
Internal Logical Files (IFL)	6	7	10	15
External Interface Files (EIF)	4	5	7	10

Solution: FP is given by :

$$FP = \text{Count Total} \times CAF$$

$$\text{Now, Count Total} = 50 \times 4 + 40 \times 5 + 35 \times 4 + 6 \times 10 + 4 \times 7 = 628$$

$$CAF = 0.65 + 0.01 \times \sum F_i$$

$$= 0.65 + 0.01 \times (14 \times 3) = 1.07 \quad (\text{Here 14 is taken for 14 questions to be answered})$$

$$\text{Hence FP} = 628 \times 1.07 = 672$$

If FP =672, average productivity = 6.5 FP/PM and labor rate = \$8000 per month, then compute:

- a) Cost per FP**
- b) Total project cost**
- c) Estimated Effort**

Solution:

- a) Cost per FP = $8000 / 6.5 = \$1230$**
- b) Total project cost = $672 * \$1230 = \826560**
- c) Estimated effort = Total FP/ Average productivity= $672/ 6.5 = 103$ persons**

For example:

If FP = 375 FP, average productivity = 6.5 FP/PM and labor rate = \$8000 per month then

- Cost per FP = $(\$8000) / (6.5 \text{ FP/PM}) = \1230
- Total project cost = $375 \times \$1230 = \461250
- Estimated effort = Total FP / Average productivity = $375 \text{ FP} / (6.5 \text{ FP/PM}) = 58$ persons

COConstructive COst MModel (COCOMO II)

- an algorithmic software cost estimation model developed by Barry Boehm
- uses a basic regression formula, with some parameters that are derived from historical project data and current project characteristics.
- COCOMO-II is the successor of COCOMO-81 and is better suited for estimating software development projects.
- It provides more support for modern software development processes and an updated project database.
- The need for new model came as software development technology moved from mainframe and overnight batch processing to desktop development, cost reusability and the use of off-the-self software components.
- COCOMO II is tuned to modern software life cycles.
- The original COCOMO model has been very successful, but it doesn't apply to newer software development practices .
- **COCOMO II** has four different models:
 1. The Application Composition Model
 2. The Early Design Model
 3. The Reuse Model
 4. The Post-Architecture Mode

1.The Application Composition Model

- The application composition model was introduced into COCOMO II to support the estimation of effort required for prototyping projects and for projects where the software is developed by composing **existing components**.
- formula for effort computation for system prototypes is:

$$PM = (NAP \times (1 - \%reuse/100)) / PROD$$

PM: the effort estimate in person-months.

NAP: the total number of application points in the delivered system.

%reuse: an estimate of the amount of reused code in the development.

PROD: the application-point productivity

Figure: Application point productivity

Developers experience & capability	Productivity (PROD)
Very Low	4
Low	7
Nominal	13
High	25
Very High	50

Example 1:

- Consider a database application project with
 - The application has **four** screens with **four** views each and **seven** data tables for **three** servers and **four** clients.
 - Application may generate **two** reports of **six** section each from **seven** data tables for **two** servers and **three** clients.
 - 10% reuse** of object points.
 - Developer's experience and capability in similar environment is **low**.
- Calculate the object point count, New object point and effort to develop such project.

No. of views contain	Sources of data tables		
	Total < 4 (< 2 servers < 3 clients)	Total < 8 (2 - 3 servers 3-5 clients)	Total 8 + (> 3 servers > 5 clients)
< 3	Simple	Simple	Medium
3 - 7	Simple	Medium	Difficult
> 8	Medium	Difficult	Difficult

For Screens

No. of section contain	Sources of data tables		
	Total < 4 (< 2 servers < 3 clients)	Total < 8 (2 - 3 servers 3-5 clients)	Total 8 + (> 3 servers > 5 clients)
0 - 1	Simple	Simple	Medium
2 - 3	Simple	Medium	Difficult
4 +	Medium	Difficult	Difficult

For Reports

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Components	-	-	10

Complexity Weight

Solution:

- **Step 1:** Number of screens = 4 and Number of records = 2
- **Step 2:** For screens,
 - *Number of views = 4*
 - *Number of data tables = 7*
 - *Number of servers = 3*
 - *Number of clients = 4*
- By using above given information and table (For Screens), Complexity level for each screen = **medium**
- For reports,
 - *Number of sections = 6*
 - *Number of data tables = 7*
 - *Number of servers = 2*
 - *Number of clients = 3*
- By using above given information and table (For Reports), Complexity level for each report = **difficult**.

No. of views contain	Sources of data tables		
	Total < 4 (< 2 servers < 3 clients)	Total < 8 (2 - 3 servers 3-5 clients)	Total 8 + (> 3 servers > 5 clients)
< 3	Simple	Simple	Medium
3 - 7	Simple	Medium	Difficult
> 8	Medium	Difficult	Difficult

For Screens

No. of section contain	Sources of data tables		
	Total < 4 (< 2 servers < 3 clients)	Total < 8 (2 - 3 servers 3-5 clients)	Total 8 + (> 3 servers > 5 clients)
0 - 1	Simple	Simple	Medium
2 - 3	Simple	Medium	Difficult
4 +	Medium	Difficult	Difficult

For Reports

- **Step 3:** By using complexity weight table we can assign complexity weight to each object instance depending upon their complexity level.

- **Complexity weight for each screen = 2**
- **Complexity weight for each report = 8**

- **Step 4:**

- **Object point count** = $\text{sigma (Number of object instances) * (its Complexity weight)}$

$$= 4 * 2 + 2 * 8 = 24$$

- **Step 5:**

- **%reuse of object points** = 10% (given)
- **NOP** = $[\text{object points} * (100 - \%reuse)] / 100$
 $= [24 * (100 - 10)] / 100 = 21.6$

- **Step 6:**

- Developer's experience and capability is low (given) Using information given about developer and productivity rate table
- **Productivity rate (PROD) of given project = 7**

- **Step 7:**

- **Effort** = NOP / PROD
 $= 21.6 / 7$
 $= 3.086 \text{ person-month}$

Therefore, effort to develop the given project = 3.086 person-month.

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Components	-	-	10

Complexity Weight

Developers experience & capability	Productivity (PROD)
Very Low	4
Low	7
Nominal	13
High	25
Very High	50

Productivity Rate

Example 2:

- Consider a database application project with **four** screens with medium complexity, and **two** reports with difficult complexity. The application uses **10% reuse** of object points. The Developer's experience and capability in similar environment is **low**.
- Using COCOMO II model, calculate the object point count, New object point and effort to develop such project. The complexity weight and productivity rate are provided for your reference.

Solution:

Here:

- Complexity weight for each screen** = 2
- Complexity weight for each report** = 8
- Object point count** = $\sum (\text{Number of object instances}) * (\text{its Complexity weight})$
 $= 4 * 2 + 2 * 8 = 24$
- %reuse of object points** = 10% (given)
- NOP** = $[\text{object points} * (100 - \%reuse)] / 100$
 $= [24 * (100 - 10)] / 100 = 21.6$
- Productivity rate (PROD) of given project** = 7
- Effort** = NOP / PROD
 $= 21.6 / 7$
 $= 3.086 \text{ person-month}$

Therefore, effort to develop the given project = 3.086 person-month.

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Components	-	-	10

Complexity Weight

Developers experience & capability	Productivity (PROD)
Very Low	4
Low	7
Nominal	13
High	25
Very High	50

Productivity Rate

2. The Early Design Model

- This model may be **used during the early stages of a project**, before a detailed architectural design for the system is available. The early design model assumes that user requirements have been agreed and initial stages of the system design process are underway.
- The estimates produced at this stage are based on the standard formula for algorithmic models, namely:

$$\text{Effort} = A \times \text{Size}^B \times M$$

Where:

Effort is persons-months

$A = 2.94$

B = Scale factor varies from 1.1 to 1.24

Size = size of the software expressed in KSLOC

M = multipliers, and $M = \text{PERS} \times \text{PREX} \times \text{RCPX} \times \text{RUSE} \times \text{PDIF} \times \text{SCED} \times \text{FSIL}$

PERS: personnel capability

PREX: personnel experience

RCPX: product reliability and complexity

RUSE: reuse required

PDIF: platform difficulty

SCED: schedule

FSIL: support facilities

3. The Reuse Model

- COCOMO II **considers two types of reused code.**
 - **Black-box code** is code that can be reused without understanding the code or making changes to it.
 - **White-box code** is reusable code that has to be adapted to integrate it with new code or other reused components.
- the formula to estimate the reuse effort is:

$$\text{Effort} = A \times \text{ESLOC}^B \times M$$

- Here A, B, and M have the same values as used in the early design model.
- The formula used to calculate the source code equivalence is:

$$\text{ESLOC} = (\text{ASLOC} \times (1 - \text{AT}/100) \times \text{AAM})$$

- Where:
 - **ESLOC**: the equivalent number of lines of new source code.
 - **ASLOC**: an estimate of the number of lines of code in the reused components that have to be changed.
 - **AT**: the percentage of reused code that can be modified automatically.
 - **AAM**: an Adaptation Adjustment Multiplier that reflects the additional effort required to reuse components

4. The Post-architecture Model

- The post-architecture model is the most detailed of the COCOMO II models.
- It is used **when you have an initial architectural design for the system.**
- The starting point for estimates produced at the post-architecture level is the same basic formula used in the early design estimates:

$$\mathbf{PM = A \times Size^B \times M}$$

Assignment:

1. Prepare slides on COCOMO II model for cost estimation.
You will be presenting in the classroom.

7.6 Project duration and staffing

Project duration

- Project managers must also estimate **how long the software will take to develop** and when staff will be needed to work on the project.
- The COCOMO model includes a formula to estimate the calendar time required to complete a project:

$$T_{DEV} = 3 \times 3 (PM)^{(0.33 + 0.2*(B - 1.01))}$$

Where,

- TDEV: the nominal schedule for the project, in calendar months, ignoring any multiplier that is related to the project schedule.
 - PM: the effort computed by the COCOMO model.
 - B: a complexity-related exponent
-
- If $B = 1.17$ and $PM = 60$ then $TDEV = 3 \times 3 (60)^{0.36} = 13$ months

Staffing

- There is a complex relationship between the number of people working on a project, the effort that will be devoted to the project and the project delivery schedule.
- If four people can complete a project in 13 months (i.e., 52 person-months of effort), then you might think that by adding one more person, you could complete the work in 11 months (55 person-months of effort).
- However, the COCOMO model suggests that you will, in fact, need six people to finish the work in 11 months (66 person-months of effort).
- The reason for this is that adding people to a project reduces the productivity of existing team members.
- As the **project team increases in size, team members spend more time communicating** and defining interfaces between the parts of the system developed by other people.
- **Doubling** the number of staff (for example) therefore **does not mean** that the duration of the project will be **halved**.
- Consequently, when you add an extra person, the actual increment of effort added is less than one person as others become less productive.
- If the development team is large, adding more people to a project sometimes increases rather than reduces the development schedule because of the overall effect on productivity.

- You cannot simply estimate the number of people required for a project team by dividing the total effort by the required project schedule.
- **Usually, a small number of people are needed at the start** of a project to carry out the initial design.
- The team then **builds up to a peak during the development** and testing of the system, **and then declines in size** as the system is prepared for deployment.
- A very rapid build-up of project staff has been shown to correlate with project schedule slippage.
- **As a project manager, you should therefore avoid adding too many staff to a project early in its lifetime.**

Reference:

- Software Testing:
<https://cs.ccsu.edu/~stan/classes/CS410/Notes16/08-SoftwareTesting.html>

Brief Answer Questions:

1. Define testing. What is its purpose?
2. What are the phases in testing process?
3. What are the goals of software testing? Mention any two.
4. Define validation and verification.
5. What is the importance of verification and validation?
6. What is the main difference between inspection and testing?
7. Mention the different levels of testing.
8. Define test case and test suite.
9. What is test automation?
10. What is heuristic estimation?
11. What are functional points?
12. What is release testing?
13. Differentiate between alpha testing and beta testing.
14. What are unit testing and System testing?

Short Answer Questions:

1. Define software testing. Explain the different phases of software testing process.
2. Differentiate between Validation and Verification with suitable example.
3. How inspection is different from testing? Explain.
4. Inspection and testing are complementary. Justify.
5. Explain the different stages of testing.
6. What are alpha and beta testing? Explain.
7. Differentiate between Whitebox and Blackbox testing.
8. Explain BVA and Equivalence and Decision table based testing.
9. Design a test case sample for testing a button “Sign up” in you newly developed software.
10. What is COCOMO II model? Explain.
11. What is the major distinction between COCOMO I and COCOMO II?
12. Project Staffing is non-linear in behavior. Justify.

13. Design a Test case for a login functionality of an application.
14. Differentiate between component testing and system testing.
15. Why algorithmic cost modeling is better than experience-base technique?
16. Explain FP based estimation approach with suitable example.
17. Explain COCOMO II as an effective cost estimation technique.
18. Given the complexity-related exponent (B) =1.17 and effort required per month (PM)= 60, calculate the total development time (i.e project duration).
19. As a project manager, how do you plan for staffing in software development? Explain.
20. What do you mean by cost estimation and effort estimation? Why do we need to perform such estimation? Explain.

21. What is function point analysis? Consider a project with the following functional units :

- Number of user inputs = 60
- Number of user outputs = 20
- Number of user enquiries = 15
- Number of user files = 05
- Number of external interfaces = 03
- Assuming all complexity adjustment factors and weighing factors as average, Calculate the function point of the project. (*AVERAGE complexity weights = {4, 5, 4, 10, 7} for the 5 complexities respectively. AVERAGE characteristic weight = 3.*)

22. Given the following values, computer FP when all complexity adjustment factors and weighing factors are average. Also given that average characteristic weight = 3.

Information Domain Value	Count	Weighting Factor		
		Simple	Average	Complex
External Inputs (EI)	50	3	4	6
External Outputs (EO)	40	4	5	7
External Inquiries (EQ)	35	3	4	6
Internal Logical Files (IFL)	6	7	10	15
External Interface Files (EIF)	4	5	7	10

- If average productivity = 6.5 FP/PM and labor rate- \$8000 per month, then also compute:
 - a) Cost per FP
 - b) Total project cost
 - c) Estimate effort

23. Consider a database application project with four screens with medium complexity, and two with difficult complexity. The application uses 10% reuse of object points. The Developer's experience and capability in similar environment is low.

Object Type	Complexity Weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL Components	-	-	10

Complexity Weight

Developers experience & capability	Productivity (PROD)
Very Low	4
Low	7
Nominal	13
High	25
High	50

Productivity Rate

Using COCOMO II model, calculate the object point count, New object point and effort to develop such project. The complexity weight and productivity rate are provided for your reference

End of Chapter