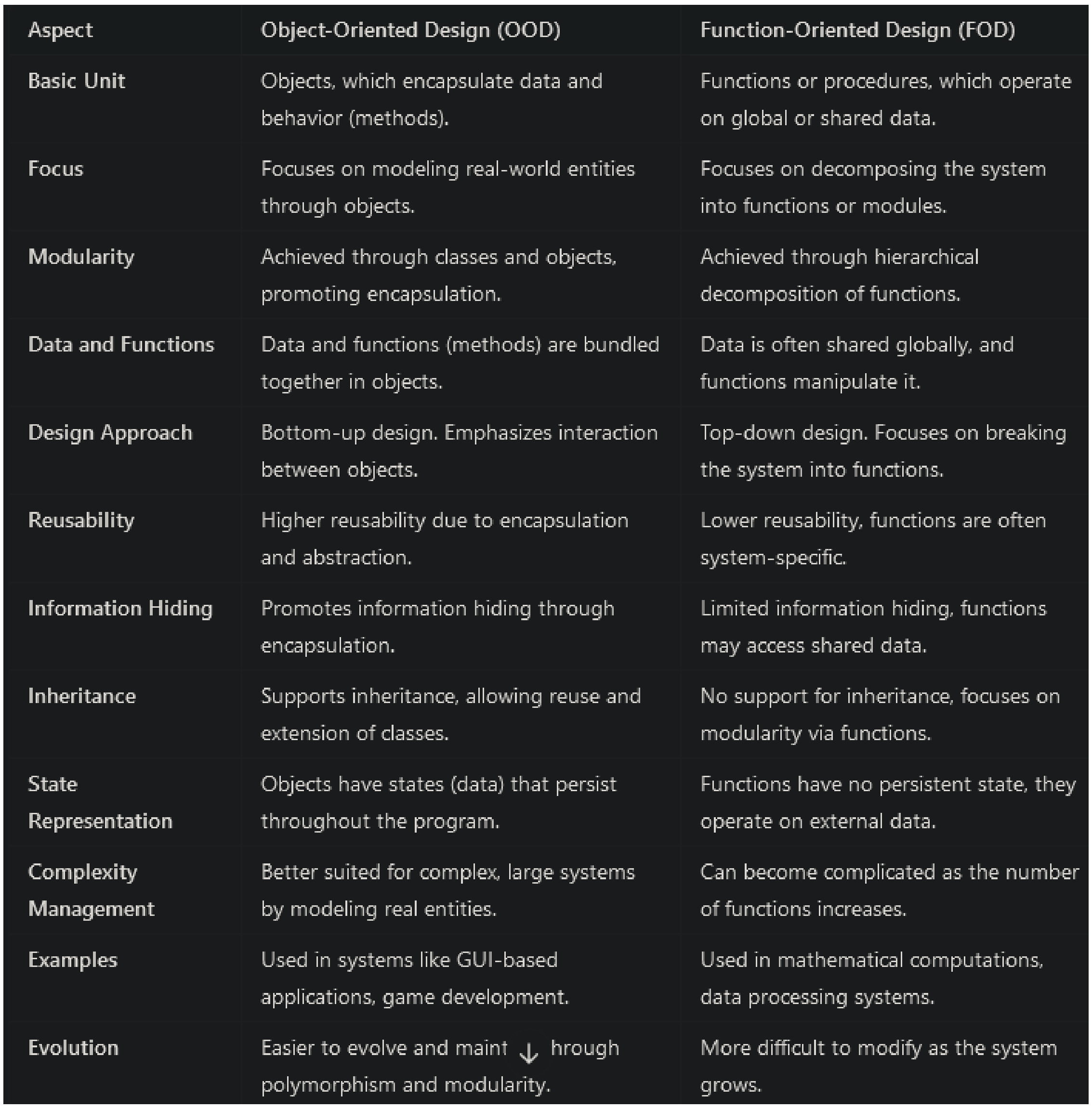
Software engineering assignment 3

Ques 1 : discuss the differences btw object oriented and function oriented design .



Aspect	Object-Oriented Design (OOD)	Function-Oriented Design (FOD)
Basic Unit	Objects, which encapsulate data and behavior (methods).	Functions or procedures, which operate on global or shared data.
Focus	Focuses on modeling real-world entities through objects.	Focuses on decomposing the system into functions or modules.
Modularity	Achieved through classes and objects, promoting encapsulation.	Achieved through hierarchical decomposition of functions.
Data and Functions	Data and functions (methods) are bundled together in objects.	Data is often shared globally, and functions manipulate it.
Design Approach	Bottom-up design. Emphasizes interaction between objects.	Top-down design. Focuses on breaking the system into functions.
Reusability	Higher reusability due to encapsulation and abstraction.	Lower reusability, functions are often system-specific.
Information Hiding	Promotes information hiding through encapsulation.	Limited information hiding, functions may access shared data.
Inheritance	Supports inheritance, allowing reuse and extension of classes.	No support for inheritance, focuses on modularity via functions.
State Representation	Objects have states (data) that persist throughout the program.	Functions have no persistent state, they operate on external data.
Complexity Management	Better suited for complex, large systems by modeling real entities.	Can become complicated as the number of functions increases.
Examples	Used in systems like GUI-based applications, game development.	Used in mathematical computations, data processing systems.
Evolution	Easier to evolve and maint ↓ hrough polymorphism and modularity.	More difficult to modify as the system grows.

Ques 2 : explain halstead theory of software science
Is it significant in today's scenario of component based software development?

Halstead's theory of software science [HAL77] is one of "the best known and most thoroughly studied . . . composite measures of (software) complexity" [CUR80]. Software science proposed the first analytical "laws" for computer software.9 Software science assigns quantitative laws to the development of computer software, using a set of primitive measures that may be derived after code is generated or estimated once design is complete. These follow

n_1 = the number of distinct operators that appear in a program.
n_2 = the number of distinct operands that appear in a program.
N_1 = the total number of operator occurrences.
N_2 = the total number of operand occurrences.

Halstead uses these primitive measures to develop expressions for the overall program length, potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), the language level (a constant for a given language), and other features such as development effort, development time, and even the projected number of faults in the software

Halstead shows that length N can be estimated

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume may be defined

$$V = N \log_2 (n_1 + n_2)$$

It should be noted that V will vary with programming language and represents the volume of information (in bits) required to specify a program. Theoretically, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio L as the ratio of volume of the most compact form of a program to the volume of the actual program. In actuality, L must always be less than 1. In terms of primitive measures, the volume ratio may be expressed as

L = 2/n1 * n2/N2

Halstead's work is amenable to experimental verification and a large body of research has been conducted to investigate software science.

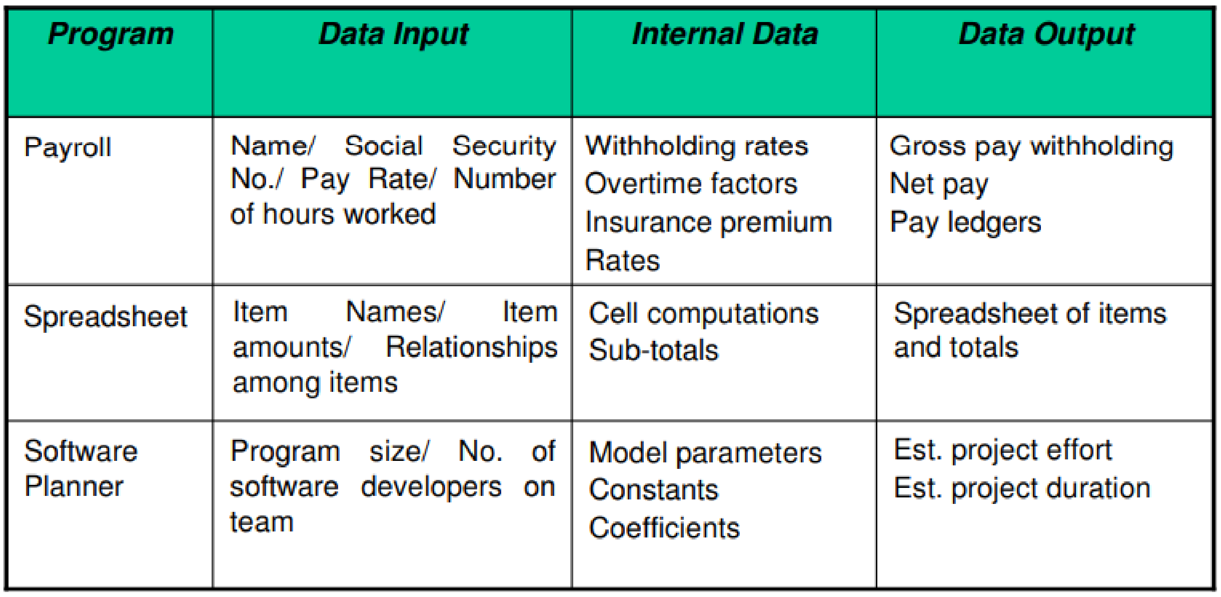
In today's component-based software development, Halstead's theory still holds some relevance, but its significance has decreased. Modern software engineering focuses heavily on reusability, modularity, and abstraction, especially in object-oriented and component-based systems. Halstead's theory, which was developed in the context of procedural programming, does not directly address these modern paradigms.

However, the basic principles of complexity measurement from Halstead's theory can still provide insights into certain aspects of code complexity, especially for traditional procedural code or smaller, standalone components.

Ques 3 : define data structure metrics how can we calculate amount of data in a program ?

Essentially the need for software development and other activities are to process data. Some data is input to a system, program or module; some data may be used internally, and some data is the output from a system, program, or module.

Data Structure Metrics



Program	*Data Input*	*Internal Data*	*Data Output*
Payroll	Name/ Social Security No./ Pay Rate/ Number of hours worked	Withholding rates Overtime factors Insurance premium Rates	Gross pay withholding Net pay Pay ledgers
Spreadsheet	Item Names/ Item amounts/ Relationships among items	Cell computations Sub-totals	Spreadsheet of items and totals
Software Planner	Program size/ No. of software developers on team	Model parameters Constants Coefficients	Est. project effort Est. project duration

Fig.1: Some examples of input, internal, and output data

That's why an important set of metrics which capture in the amount of data input, processed in an output form software. A count of this data structure is called Data Structured Metrics. In these concentrations is on variables (and given constant) within each module & ignores the input-output dependencies.

There are some Data Structure metrics to compute the effort and time required to complete the project. There metrics are:

1. The Amount of Data.
2. The Usage of data within a Module.
3. Program weakness.
4. The sharing of Data among Modules.

1. The Amount of Data: To measure the amount of Data, there are further many different metrics, and these are:

- **Number of variable (VARS):** In this metric, the Number of variables used in the program is counted.
- **Number of Operands (η_2):** In this metric, the Number of operands used in the program is counted.
  η_2 = VARS + Constants + Labels
- **Total number of occurrence of the variable (N2):** In this metric, the total number of occurrence of the variables are computed

2. The Usage of data within a Module: The measure this metric, the average numbers of live variables are computed. A variable is live from its first to its last references within the procedure.

$$\text{Average no of Live variables (LV)} = \frac{\text{Sum of count live variables}}{\text{Sum of count of executable statements}}$$

3. Program weakness: Program weakness depends on its Modules weakness. If Modules are weak(less Cohesive), then it increases the effort and time metrics required to complete the project.

$$\text{Average life of variables } (\gamma) = \frac{\text{Sum of count live variables}}{\text{Sum of count of executable statements}}$$

Module Weakness (WM) = LV* γ

A program is normally a combination of various modules; hence, program weakness can be a useful measure and is defined as:

$$\mathbf{WP} = \frac{\left(\sum_{i=1}^{m} \mathrm{WM}_i\right)}{m}$$

Where

WMi: Weakness of the ith module

WP: Weakness of the program

m: No of modules in the program

4. There Sharing of Data among Module: As the data sharing between the Modules increases (higher Coupling), no parameter passing between Modules also increased, As a result, more effort and time are required to complete the project. So Sharing Data among Module is an important metrics to calculate effort and time.

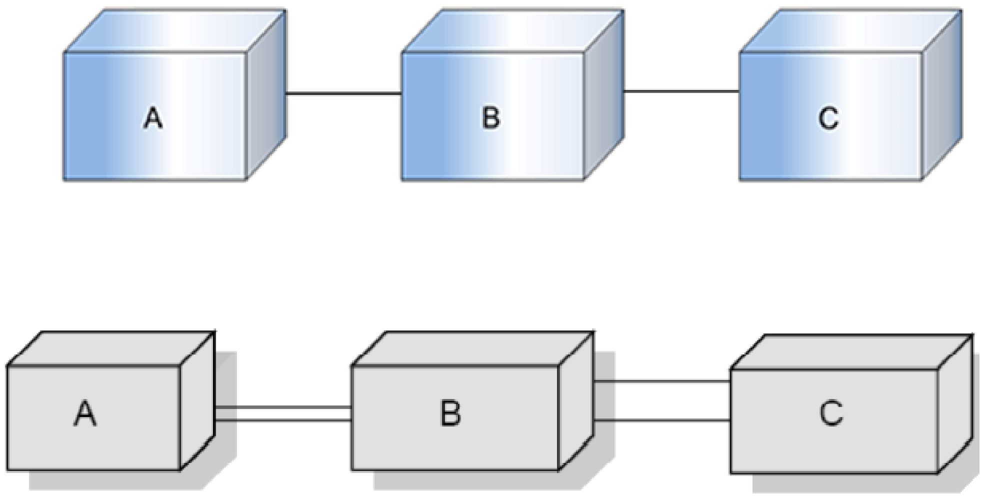


Fig.11: "Pipes" of data shared among the modules

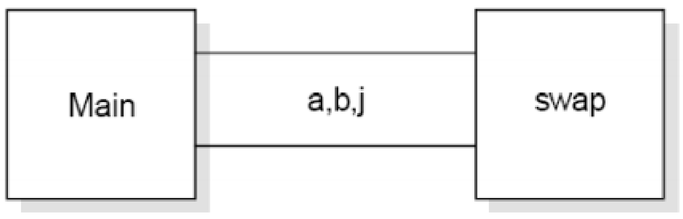


Fig.12: The data shared in program bubble

Ques 4 : discuss object oriented metrics what is the importance of metrics in obejct oriented software development?

Object-Oriented (OO) Metrics are used to measure various aspects of object-oriented software development, such as design quality, complexity, maintainability, and performance. These metrics help in assessing the effectiveness of OO techniques like classes, inheritance, and encapsulation.

Size Metrics:

- **Number of Methods per Class (NOM)**: Counts the total methods in a class.
- **Number of Attributes per Class (NOA)**: Counts the total attributes in a class.
- **Weighted Number of Methods per Class (WMC)**: Measures the complexity by summing the complexities of all methods in a class.

Coupling Metrics:

- **Response for a Class (RFC)**: Measures the number of methods (internal and external) that can be executed in response to a message to an object.
- **Data Abstraction Coupling (DAC)**: Counts the number of abstract data types in a class.
- **Message Passing Coupling (MPC)**: Counts the number of send statements (messages) in a class.
- **Coupling Factor (CF)**: Ratio of actual coupling to the maximum possible coupling in the system.
- **Coupling Between Objects (CBO)**: Measures the number of classes to which a particular class is coupled.

Cohesion Metrics:

- **Lack of Cohesion in Methods (LCOM)**: Measures how closely the methods in a class are related to each other.
- **Tight Class Cohesion (TCC)**: Percentage of pairs of public methods that share common attributes.
- **Loose Class Cohesion (LCC)**: Similar to TCC but also considers indirectly connected methods.
- **Information-based Cohesion (ICH)**: Measures the number of invocations between methods of the same class, weighted by the number of parameters of the invoked methods.

Inheritance Metrics:

- **Depth of Inheritance Tree (DIT)**: The level of a class in the inheritance hierarchy.
- **Number of Children (NOC)**: The number of immediate subclasses of a class.
- **Attribute Inheritance Factor (AIF)**: The ratio of inherited attributes to total attributes in a class.
- **Method Inheritance Factor (MIF)**: The ratio of inherited methods to the total number of methods in a class system.

Importance of metrics in object-oriented software development:

1. **Quality Assurance**: Evaluate code and design quality.
2. **Project Management**: Assist in estimating timelines and resource allocation.
3. **Performance Evaluation**: Gauge component performance for optimization.
4. **Maintainability**: Identify complex or tightly coupled classes for refactoring.
5. **Cost Estimation**: Provide data for better development cost estimates.
6. **Risk Management**: Identify high-risk areas for proactive mitigation.

Ques 5 .explain the basic and logarithmic poision model and there significance in reliabilty studies?

$$\frac{d\lambda}{d\mu} = -\lambda_0 \theta \exp(-\mu\theta) \qquad\qquad \frac{d\lambda}{d\mu} = -\theta\lambda$$

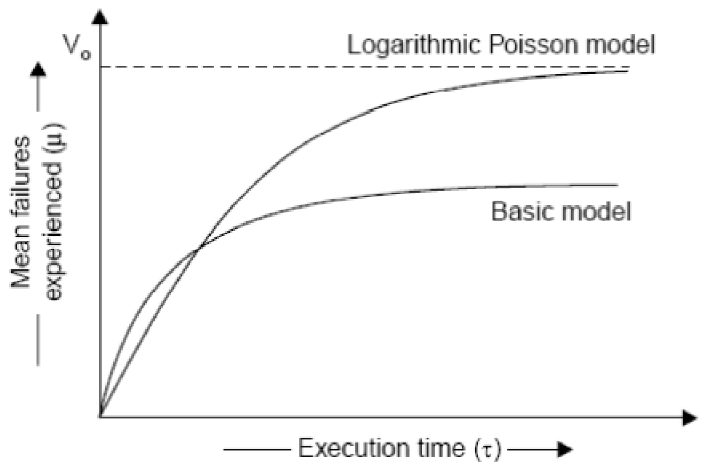
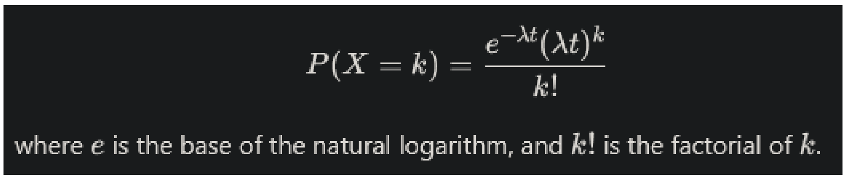


Fig.7.19: Relationship between

Basic Poisson Model

Description:

- The basic Poisson model assumes that events (e.g., system failures) occur randomly and independently over time. It is characterized by a single parameter, λ (lambda), which represents the average rate of occurrence (mean number of events in a specified interval).
- The probability of observing kkk events in a time interval ttt is given by the formula:



$$P(X = k) = \frac{e^{-\lambda t}(\lambda t)^k}{k!}$$

where e is the base of the natural logarithm, and $k!$ is the factorial of k.

Significance in Reliability Studies:

- **Failure Rate Estimation**: The basic Poisson model helps estimate the failure rate of systems, enabling organizations to understand how often failures are likely to occur.
- **Reliability Prediction**: It allows for predicting system reliability over time by modeling how the failure rate may change with time or usage.
- **Maintenance Planning**: By understanding failure patterns, organizations can optimize maintenance schedules and resource allocation, reducing downtime and costs.

Logarithmic Poisson Model

Logarithmic Poisson Execution Time Model

Failure Intensity

$$\lambda(\mu) = \lambda_0 \exp(-\theta\mu)$$

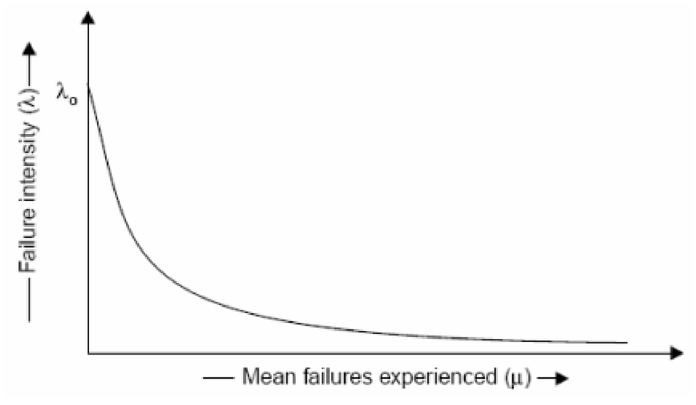
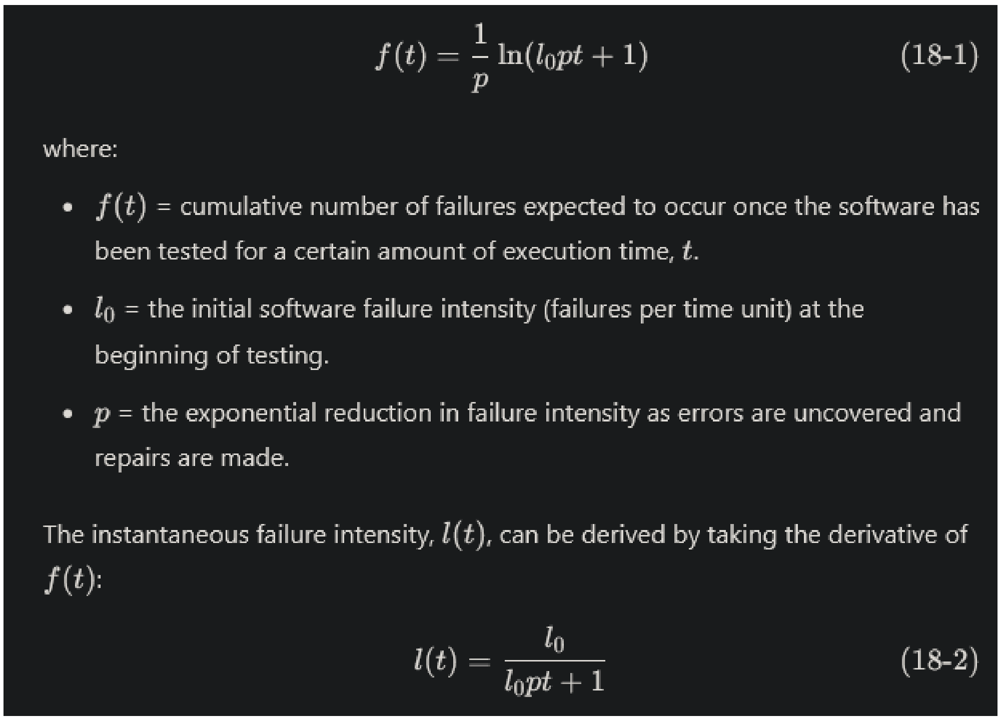


Fig.7.18: Relationship between μ & λ

$$\mu(\tau) = \frac{1}{\theta} Ln(\lambda_0\theta\tau + 1)$$

$$\lambda(\tau) = \lambda_0 /(\lambda_0\theta\tau + 1)$$

Using statistical modeling and software reliability theory, models of software failures (uncovered during testing) as a function of execution time can be developed [MUS89]. A version of the failure model, called a logarithmic Poisson execution-time model, takes the form:



$$f(t) = \frac{1}{p}\ln(l_0 pt + 1) \tag{18-1}$$

where:

- $f(t)$ = cumulative number of failures expected to occur once the software has been tested for a certain amount of execution time, t.
- l_0 = the initial software failure intensity (failures per time unit) at the beginning of testing.
- p = the exponential reduction in failure intensity as errors are uncovered and repairs are made.

The instantaneous failure intensity, $l(t)$, can be derived by taking the derivative of $f(t)$:

$$l(t) = \frac{l_0}{l_0 pt + 1} \tag{18-2}$$

Using the relationship noted in Equation (18-2), testers can predict the drop-off of errors as testing progresses. The actual error intensity can be plotted against the predicted curve (Figure 18.3). If the actual data gathered during testing and the logarithmic Poisson execution time model are reasonably close to one another over a number of data points, the model can be used to predict the total testing time required to achieve an acceptably low failure intensity.