



Object Oriented Software Engineering

1

LECTURE 18

I. PRODUCT METRICS



Product Metrics

3

- Measurement assigns numbers or symbols to attributes of entities in the real world. To accomplish this, a measurement model encompassing a consistent set of rules is required.

McCall's Triangle of Quality (1970s)

Maintainability

Flexibility

Testability

PRODUCT REVISION

Portability

Reusability

Interoperability

PRODUCT TRANSITION

PRODUCT OPERATION

Correctness

Usability

Efficiency

Reliability

Integrity

SW built to conform to these factors will exhibit high quality, even if there are dramatic changes in technology.



Measures, metrics and indicators

- ❑ A **measure** provides a quantitative indication of the extent, amount, dimension, capacity, or size of some attribute of a product or process
- ❑ The IEEE glossary defines a **metric** as “a quantitative measure of the degree to which a system, component, or process possesses a given attribute”
- ❑ An **indicator** is a metric or combination of metrics that provide insight into the software process, a software project, or the product itself

I. Metrics for the requirement model

6

- It is possible to adapt metrics that are often used for project estimation and apply them in this context. These metrics examine the requirements model with the intent of predicting the “size” of the resultant system. Size is sometimes (but not always) an indicator of design complexity and is almost always an indicator of increased coding, integration, and testing effort.



1. Function-Based Metrics

7

- The function point (FP) metric can be used effectively as a means for measuring the functionality delivered by a system.⁴ Using historical data, the FP metric can then be used to (1) estimate the cost or effort required to design, code, and test the software; (2) predict the number of errors that will be encountered during testing; and (3) forecast the number of components and/or the number of projected source lines in the implemented system. Function points are derived using an empirical relationship based on countable (direct) measures of software's information domain and qualitative assessments of software complexity. Information domain values are defined in the following manner.



- **Number of external inputs (EIs).** Each external input originates from a user or is transmitted from another application and provides distinct application-oriented data or control information. Inputs are often used to update internal logical files (ILFs). Inputs should be distinguished from inquiries, which are counted separately.
- **Number of external outputs (EOs).** Each external output is derived data within the application that provides information to the user. In this context external output refers to reports, screens, error messages, etc. Individual data items within a report are not counted separately.



- **Number of external inquiries (EQs).** An external inquiry is defined as an online input that results in the generation of some immediate software response in the form of an online output (often retrieved from an ILF).
- **Number of internal logical files (ILFs).** Each internal logical file is a logical grouping of data that resides within the application's boundary and is maintained via external inputs.
- **Number of external interface files (EIFs).** Each external interface file is a logical grouping of data that resides external to the application but provides information that may be of use to the application.

- To compute function points (FP), the following relationship is used:
- FP count total $[0.65 + 0.01 \sum(F_i)]$
- where count total is the sum of all FP entries obtained

Information Domain Value	Count		Weighting factor				
			Simple	Average	Complex		
External Inputs (EIs)	<input type="text"/>	×	3	4	6	=	<input type="text"/>
External Outputs (EOs)	<input type="text"/>	×	4	5	7	=	<input type="text"/>
External Inquiries (EQs)	<input type="text"/>	×	3	4	6	=	<input type="text"/>
Internal Logical Files (ILFs)	<input type="text"/>	×	7	10	15	=	<input type="text"/>
External Interface Files (EIFs)	<input type="text"/>	×	5	7	10	=	<input type="text"/>
Count total							<input type="text"/>



- The F_i ($i = 1$ to 14) are value adjustment factors (VAF) based on responses to the following questions:
 1. Does the system require reliable backup and recovery?
 2. Are specialized data communications required to transfer information to or from the application?
 3. Are there distributed processing functions?
 4. Is performance critical?
 5. Will the system run in an existing, heavily utilized operational environment?
 6. Does the system require online data entry?
 7. Does the online data entry require the input transaction to be built over multiple screens or operations?
 8. Are the ILFs updated online?
 9. Are the inputs, outputs, files, or inquiries complex?
 10. Is the internal processing complex?
 11. Is the code designed to be reusable?
 12. Are conversion and installation included in the design?
 13. Is the system designed for multiple installations in different organizations?
 14. Is the application designed to facilitate change and ease of use by the user?

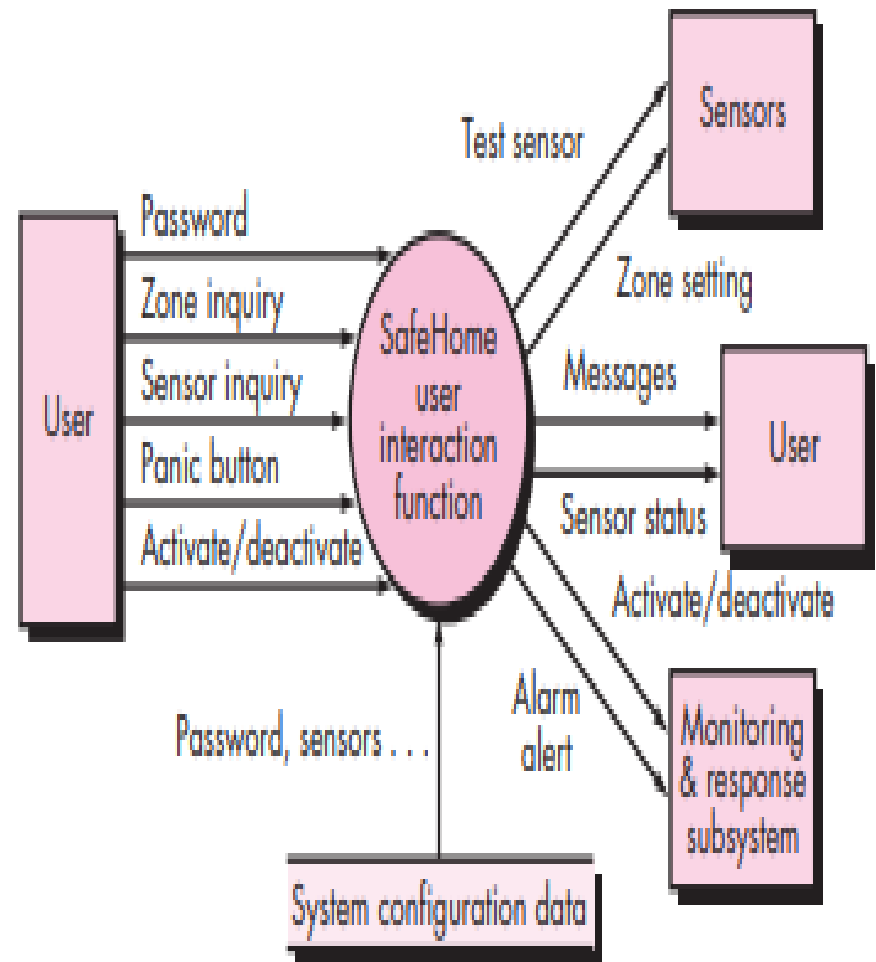


- Each of these questions is answered using a scale that ranges from 0 (not important or applicable) to 5 (absolutely essential).

Example: Function Point

13

The data flow diagram is evaluated to determine a set of key information domain measures required for computation of the function point metric. **Three external inputs—password, panic button, and activate/deactivate—are shown in the figure along with two external inquiries—zone inquiry and sensor inquiry.** One ILF (system configuration file) is shown. Two external outputs (messages and sensor status) and four EIFs (test sensor, zone setting, activate/deactivate, and alarm alert) are also present.



Example: Function Point

14

- The count total shown in previous figure must be adjusted using. For the purpose of this example, we assume that is $\Sigma(F_i)$ 46 (a moderately complex product).
- Therefore, $FP = 50 \times [0.65 + (0.01 \times 46)] = 56$

Information Domain Value	Count		Weighting factor			
			Simple	Average	Complex	
External Inputs (EIs)	3	×	3	4	6	= 9
External Outputs (EOs)	2	×	4	5	7	= 8
External Inquiries (EQs)	2	×	3	4	6	= 6
Internal Logical Files (ILFs)	1	×	7	10	15	= 7
External Interface Files (EIFs)	4	×	5	7	10	= 20
Count total						50

- A list of characteristics that can be used to assess the quality of the requirements model and the corresponding requirements specification: **specificity** (lack of ambiguity), completeness, correctness, understandability, verifiability, internal and external consistency, achievability, concision, traceability, modifiability, precision, and reusability.



II. Metrics for the Design model

16

- Design metrics for computer software, like all other software metrics, are not perfect. Debate continues over their efficacy and the manner in which they should be applied. Many experts argue that further experimentation is required before design measures can be used. And yet, design without measurement is an unacceptable alternative.



a. Architectural Design Metrics

17

- Architectural design metrics focus on characteristics of the program architecture with an emphasis on the architectural structure and the effectiveness of modules or components within the architecture. These metrics are “black box” in the sense that they do not require any knowledge of the inner workings of a particular software component. Card and Glass define three software design complexity measures: structural complexity, data complexity, and system complexity



b. Metrics for Object-Oriented Design

18

- **Size.** Size is defined in terms of four views: population, volume, length, and functionality. Population is measured by taking a static count of OO entities such as classes or operations. Volume measures are identical to population measures but are collected dynamically—at a given instant of time. Length is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). Functionality metrics provide an indirect indication of the value delivered to the customer by an OO application.



- **Complexity.** Like size, there are many differing views of software complexity. Some views the complexity in terms of structural characteristics by examining how classes of an OO design are interrelated to one another.
- **Coupling.** The physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects) represent coupling within an OO system.
- **Sufficiency.** Whitmire defines sufficiency as “the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application.”



- **Completeness.** The only difference between completeness and sufficiency is “the feature set against which we compare the abstraction or design component”. Sufficiency compares the abstraction from the point of view of the current application. Completeness considers multiple points of view.
- **Cohesion.** Like its counterpart in conventional software, an OO component should be designed in a manner that has all operations working together to achieve a single, well-defined purpose.
- **Primitiveness.** A characteristic that is similar to simplicity, primitiveness (applied to both operations and classes) is the degree to which an operation is atomic—that is, the operation cannot be constructed out of a sequence of other operations contained within a class.
- **Similarity.** The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose is indicated by this measure.
- **Volatility.** Design changes can occur when requirements are modified or when modifications occur in other parts of an application, resulting in mandatory adaptation of the design component in question. Volatility of an OO design component measures the likelihood that a change will occur.



- The class is the fundamental unit of an OO system. Therefore, measures and metrics for an individual class, the class hierarchy, and class collaborations will be invaluable when you are required to assess OO design quality.
- Weighted methods per class (WMC)
- Depth of the inheritance tree (DIT)
- Number of children (NOC).
- Coupling between object classes (CBO)
- Response for a class (RFC)
- Lack of cohesion in methods (LCOM).



d. Component-Level Design Metrics

22

- Component-level design metrics for conventional software components focus on internal characteristics of a software component and include measures of the “three Cs”—module cohesion, coupling, and complexity. These measures can help you judge the quality of a component-level design. Component-level design metrics may be applied once a procedural design has been developed and are “glass box” in the sense that they require knowledge of the inner workings of the module under consideration.



e. Operation-Oriented Metrics

23

- Because the class is the dominant unit in OO systems, fewer metrics have been proposed for operations that reside within a class.
- **Average operation size (OSavg).** Size can be determined by counting the number of lines of code or the number of messages sent by the operation. As the number of messages sent by a single operation increases, it is likely that responsibilities have not been well allocated within a class.
- **Operation complexity (OC).** The complexity of an operation can be computed using any of the complexity metrics proposed for conventional software. Because operations should be limited to a specific responsibility, the designer should strive to keep OC as low as possible.
- **Average number of parameters per operation (NPavg).** The larger the number of operation parameters, the more complex the collaboration between objects. In general, NPavg should be kept as low as possible.



f. User Interface Design Metrics

24

- A typical GUI uses layout entities—graphic icons, text, menus, windows, and the like—to assist the user in completing tasks. To accomplish a given task using a GUI, the user must move from one layout entity to the next. The absolute and relative position of each layout entity, the frequency with which it is used, and the “cost” of the transition from one layout entity to the next all contribute to the appropriateness of the interface.



III. Metrics for testing

25

- The majority of metrics proposed focus on the process of testing, not the technical characteristics of the tests themselves. In general, testers must rely on analysis, design, and code metrics to guide them in the design and execution of test cases.



A. Halstead Metrics Applied to Testing

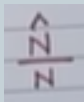
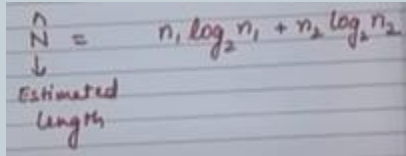
26

- Halstead's software metrics is a set of measures proposed by Maurice Halstead to evaluate the complexity of a software program. These metrics are based on the number of distinct operators and operands in the program, and are used to estimate the effort required to develop and maintain the program.



The Halstead metrics include the following:

27

- **Program length (N):** This is the total number of operator and operand occurrences in the program.
- **Vocabulary size (n):** This is the total number of distinct operators and operands in the program.
- **Purity Ratio:**  
- **Program volume (V):** This is the product of program length (N) and logarithm of vocabulary size (n), i.e., $V = N * \log_2(n)$.
- **Program difficulty (D):** This is the ratio of the number of unique operators to the total number of operators in the program, i.e., $D = (n1/2) * (N2/n2)$.
- **Program effort (E):** This is the product of program volume (V) and program difficulty (D), i.e., $E = V * D$.

Halstead's Software Metrics Example

28

```
if (k<2)
{
    if (k>3)
        x=x*k;
}
```

N1: Total No. of Operators

N2: Total No. of Operands

n1: Distinct operators

n2: Distinct Operands

N1	N2	n1	n2
if	k	if	K
(2	(2
<	k	<	3
)	3)	x
{	x	{	
if	x	>	
(k	=	
>		*	
)		;	
=		}	
*			
;			
}			

N1: 13

N2: 7

n1: 10

n2: 4



$$\text{Length} = N = N_1 + N_2 = 13 + 7 = 20$$

$$\text{Vocabulary} = n = n_1 + n_2 = 10 + 4 = 14$$

$$\begin{aligned} \text{E-Length} = \hat{N} &= n_1 \log_2 n_1 + n_2 \log_2 n_2 \\ &= 10 \log_2 10 + 4 \log_2 4 \\ &= 10 \frac{\log_{10} 10}{0.3010} + 4 \frac{\log_{10} 4}{0.3010} \\ &= \frac{10 \times 1}{0.3010} + 4 \times \frac{0.6020}{0.3010} \\ &= 33.222 + 8 \\ &= 41.222 \end{aligned}$$



$$\text{Purity Ratio} = \frac{N^A}{N} = \frac{41.222}{20} = 2.0611$$

$$\begin{aligned} \text{Program Volume}(v) &= N \log_2 (n_1 + n_2) \\ &= 20 \log_2 14 \\ &= \frac{20 \times \log_{10} 14}{0.3010} \\ &= 76.154 \end{aligned}$$

$$\begin{aligned} \text{Difficulty}(o) &= \frac{n_1}{2} \times \frac{n_2}{n_2} \\ &= \frac{10}{2} \times \frac{7}{4} \\ &= 8.75 \end{aligned}$$



$$\begin{aligned}\text{Effort} &= V \times D \\ &= 76.154 \times 8.75 \\ &= 666.3475\end{aligned}$$



B. Metrics for Object-Oriented Testing

32

- The metrics consider aspects of encapsulation and inheritance.
- Lack of cohesion in methods (LCOM).
- Percent public and protected (PAP)
- Public access to data members (PAD)
- Number of root classes (NOR)
- Fan-in (FIN).
- Number of children (NOC) and depth of the inheritance tree (DIT).

THANK YOU