Software Engineering
CS5704: Class 12 - 4/13/01

Instructor: Shawn A. Bohner
Voice: (703) 538-8374
Email: bohner@nvc.cs.vt.edu

Teaching Assistant: Sapna Georges
Voice: (703) 538-8381
Email: sgeorge@vt.edu

Agenda

▲ Turn in Project Team Results
▲ Class 10 Review
  • Chapter 17 – Testing Techniques
  • Chapter 18 – Testing Strategies
  • Chapter 23 – Object-Oriented Testing

▲ Chapter 19 – Technical Metrics for Software
Break???
▲ Chapter 24 – Technical Metrics for Object-Oriented Systems
▲ Homework Assignments
Spring Semester Timeline

Class Begins
Product & Process

PM Metrics & Estimation

Midterm Exam

Analysis, Design, & Architecture

SW Metrics & Testing Strategies

Maintenance & Evolution

Final Exam

3 weeks, 3 sessions to go… we’re in the home stretch… Still, so much to do and so little time…

Testing Across the Software Process

Test Planning

Acceptance Test Plan
??? Tests
??? Tests

System Test Plan
??? Tests
??? Tests

Integration Test Plan
??? Tests
??? Tests

Unit Test Plan
Unit Tests

Program Coding

Test Execution

Acceptance Testing

System Testing

Integration Testing

Unit Testing
Who Tests the Software?

**Developer**
- Understands the system
- Will test “???”
- Driven by “???”

**Independent Tester**
- Must learn about the system,
- Will attempt to “???” it
- Driven by “???”

White-Box Testing

**Why Coverage?**
- Logic errors and incorrect assumptions are inversely proportional to a path’s execution probability
- We often believe that a path is not likely to be executed; in fact, reality is often counter intuitive
- Typographical errors are random; it’s likely that untested paths will contain some

The goal is to ensure that all ??? and ??? have been executed at least ???

**Good for “???” modules**
What makes Black-Box Testing different from White-Box Testing?

Regression Testing

- Regression Testing is the ??? of tests that have already been used to ensure changed have not ??? unwanted ???

- Tools that support regression testing primarily manage test cases and suites:
  - Enabling staff members to determine which tests have been executed and compare the results
Packaged Applications Testing

▲ Testing Packaged Applications takes place largely at the ??? level (???-box)
▲ ???-level testing seldom indicates how the system behaves when a specific component fails
  • Very few packages support instrumentation to support testing
  • Tracing back to the source is nearly impossible
▲ A more rigorous approach may be needed

---

Debugging: A ??? Process

Test Cases

Regression Tests

Suspected Causes

Identified Causes

New Test Cases

Results

Debugging

Brute Force / Testing

Backtracking

Induction

Deduction

What’s the difference between Testing and Debugging?
Object-Oriented Testing

- Begins by evaluating the correctness and consistency of the OOA and OOD models

Testing Strategy Changes
- Concept of “Unit” due to encapsulation
- Integration focuses on and their execution across a ‘thread’ or in the context of a usage scenario

OOT Strategy

- Class testing is like unit testing
  - Operations within the class are tested
  - State behavior of the class is examined

- 3 Integration testing strategies
  - ??? testing—integrates the set of classes required to respond to one input or event
  - ??? testing—integrates the set of classes required to respond to one use case
  - Cluster testing—integrates the set of classes required to demonstrate one collaboration
Chapter 19: Technical Metrics for SW

The purpose of this chapter is to introduce software metrics as a means of helping to assess the software work product quality.

- Builds on Chapter 8’s context of assessing software process and project management.

Objectives

- Review purpose of software measurement.
- Introduce key software product measurement principles and use/interpretation strategies.
- Illustrate metric usage through examples.

McCall’s Triangle of Quality

McCall’s quality factors were proposed in the early 1970s. They are as valid today as they were in that time!
Attributes of Effective Metrics

▲ **Simple and Computable**
- Relatively easy to learn how to derive the metric
- Computation should not demand inordinate effort or time

▲ **Empirically and Intuitively Persuasive**
- Satisfy engineer’s intuitive notions about the product attribute

▲ **Consistent and Objective**
- Always yield results that are unambiguous

▲ **Consistent in Its Use of Units and Dimensions**
- Computation should use logically consistent measures

▲ **Programming Language Independent**
- Based on the analysis or design model, or the program structure

▲ **An Effective Mechanism for Quality Feedback**
- Provide a software engineer with information that can lead to a higher quality end product

Analysis Metrics

▲ **Function-based metrics:** use the function point as a normalizing factor or as a measure of the “size” of the specification

▲ **Bang metric:** used to develop an indication of software “size” by measuring characteristics of the data, functional and behavioral models

▲ **Specification metrics:** used as an indication of quality by measuring number of requirements by type
Architectural Design Metrics

- **Architectural Design Metrics** (Card and Glass)
  - Structural complexity = S(fan-out)
  - Data complexity = D(input & output variables, fan-out)
  - System complexity = C(structural + data complexity)

**Information Flow Metric:**
(Henry and Kafura)
architectural complexity as a function of fan-in and fan-out

**Morphology Metrics** (Fenton):
a function of the number of modules and the number of interfaces between modules

Component-Level Design Metrics

- **Cohesion metrics:** a function of data objects and the locus of their definition
- **Coupling metrics:** a function of input and output parameters, global variables, and modules called
- **Complexity metrics:** hundreds have been proposed (e.g., cyclomatic complexity)
Interface Design Metrics

Layout Appropriateness: a function of layout entities, the geographic position and the “cost” of making transitions among entities

Goal/Question/Metric Paradigm

1. Generate set of goals for program
2. Derive a set of questions which quantify those goals
3. Develop a set of metrics which provide information to answer those questions
**Typical Measurement Goals**

- A typical goal of a metrics effort is to initiate dialogue between customers and suppliers
  - Cost, Schedule, Quality, and Content
- **Additional goals are to measure**
  - Customer Satisfaction
  - Process Effectiveness
  - Resource Utilization

**Typical Needs**

- **Monitor and assess on-going efforts**
  - Same staff supporting multiple concurrent projects
  - More efficient processes to justify cost of operations
- **Satisfy contractual obligations**
  - Quality requirements
  - Delivery dates
Typical Metric Set Constraints

Key Requirement:
“Monitoring without intruding”

▲ Relevant to environment (i.e., large multi-contractor, multi-process, multi-language, real-time systems)
▲ Minimum cost to collect and analyze data
▲ Multiple metrics across program goals to allow “sanity checks” and monitoring of data and product quality
▲ Strong government /industry background for establishing rule-of-thumb thresholds [Customize thresholds as local experience dictates]

Example SW Development GQM

<table>
<thead>
<tr>
<th>Goal</th>
<th>Questions</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Better Project Estimates</td>
<td>How long should it take to develop a system?</td>
<td>Development Progress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Staffing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Test Case Completion</td>
</tr>
<tr>
<td></td>
<td>Will development and Staffing sustaining costs</td>
<td>Inspection Duration</td>
</tr>
<tr>
<td></td>
<td>be acceptable?</td>
<td>Software Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Density</td>
</tr>
<tr>
<td>Better Control of On-going</td>
<td>Is schedule estimate consistent with progress</td>
<td>Software Reliability</td>
</tr>
<tr>
<td>Projects</td>
<td>indicators?</td>
<td>Requirements Stability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computer Resource Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Development Progress</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Staffing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DR Open Duration</td>
</tr>
<tr>
<td>Will operational performance</td>
<td>Will operational performance be OK?</td>
<td>Test Hours requested/received</td>
</tr>
<tr>
<td>be OK?</td>
<td></td>
<td>Software Reliability</td>
</tr>
<tr>
<td>Are systems being adequately</td>
<td>Are systems being adequately tested?</td>
<td>Test Focus</td>
</tr>
<tr>
<td>tested?</td>
<td></td>
<td>Fault Density</td>
</tr>
<tr>
<td>Minimize Defects</td>
<td>Is the implementation adequate?</td>
<td>Software Complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Density</td>
</tr>
</tbody>
</table>
### NASA SW Development Metric Set

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Size</td>
<td># of source lines of code at delivery</td>
</tr>
<tr>
<td>Staffing</td>
<td>Engineering hours per project, subsystem, and DR/SR closed</td>
</tr>
<tr>
<td>Requirements Stability</td>
<td>Total # of requirements to be implemented by project</td>
</tr>
<tr>
<td>Development Progress</td>
<td>Planned vs actual Number of units designed, coded, and tested</td>
</tr>
<tr>
<td>CRU Performance</td>
<td>Percent of CPU, disk, memory, and I/O channel utilization</td>
</tr>
<tr>
<td>Test Case Completion</td>
<td>Planned and actual # of test cases successfully completed</td>
</tr>
<tr>
<td>Test Focus</td>
<td>Fraction of DRs Closed with a software fix by system</td>
</tr>
<tr>
<td>Facility Utilization</td>
<td># of facility hours requested, scheduled, and used in test</td>
</tr>
<tr>
<td>Fault Density</td>
<td># of DRs per 1000 lines of code over time per system</td>
</tr>
<tr>
<td>DR Open Duration</td>
<td>Time from DR written to closure</td>
</tr>
<tr>
<td>Software Reliability</td>
<td>Probability that the software “works” for a specified time in a specified environment</td>
</tr>
<tr>
<td>Software Complexity</td>
<td># of control paths in the software per module</td>
</tr>
</tbody>
</table>

### Example SW Maintenance GQM

<table>
<thead>
<tr>
<th>Goal</th>
<th>Question</th>
<th>Metric(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximize customer satisfaction</td>
<td>How many problems are affecting the customer?</td>
<td>Discrepancy Report (DR) &amp; Service Request (SR)</td>
</tr>
<tr>
<td></td>
<td>How long does it take to fix a problem?</td>
<td>Open Duration</td>
</tr>
<tr>
<td>Minimize effort and schedule</td>
<td>Where are the bottlenecks?</td>
<td>Software Reliability</td>
</tr>
<tr>
<td></td>
<td>How maintainable is the system?</td>
<td>Break/Fix Ratio</td>
</tr>
<tr>
<td>Minimize defects</td>
<td>Is software sustaining engineering effective?</td>
<td>DR/SR Closure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DR/SR Open Duration</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Staff Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computer Resource Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Type Distribution</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Computer Resource Utilization</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Size</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Density</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Volatility</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Complexity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Fault Density</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Break/Fix Ratio</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Software Reliability</td>
</tr>
</tbody>
</table>
## NASA SW Maintenance Metric Set

<table>
<thead>
<tr>
<th>Metric</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Software Size</td>
<td>Number of source lines of code that must be maintained</td>
</tr>
<tr>
<td>DR/SR Closure</td>
<td>Number of requests received and closed by month</td>
</tr>
<tr>
<td>DR/SR Scheduling</td>
<td>Planned vs actual time to close request</td>
</tr>
<tr>
<td>DR/SR Open Duration</td>
<td>Time from request initiation to delivery to operations</td>
</tr>
<tr>
<td>Staffing</td>
<td>Engineering hours per project, subsystem, and DR/SR closed</td>
</tr>
<tr>
<td>Fault Types</td>
<td>Percent of requests closed by request type (e.g., sw, hw) by phase inserted</td>
</tr>
<tr>
<td></td>
<td>(e.g., req, design, code, test) and by sw fault type (e.g., logic, interface)</td>
</tr>
<tr>
<td>Software Volatility</td>
<td>Number of modules changed per major release</td>
</tr>
<tr>
<td>Fault Density</td>
<td>Number of DRs per 1000 lines of code over time</td>
</tr>
<tr>
<td>Break/Fix Ratio</td>
<td>Number of DRs written against a module divided by the total number of</td>
</tr>
<tr>
<td></td>
<td>changes to the module over time</td>
</tr>
<tr>
<td>CRU Performance</td>
<td>Percent of CPU, disk, memory, and I/O channel utilization</td>
</tr>
<tr>
<td>Software Reliability</td>
<td>Probability that the software &quot;works&quot; for a specified time in a specified</td>
</tr>
<tr>
<td></td>
<td>environment</td>
</tr>
<tr>
<td>Software Complexity</td>
<td>Number of control paths in the software per module</td>
</tr>
</tbody>
</table>

### Metric Definition: Identify Categories

<table>
<thead>
<tr>
<th>Indicator Category</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQA Audit and Review Results</td>
<td>Provides estimation of product quality, compliance of staff to project</td>
</tr>
<tr>
<td></td>
<td>processes, and status of action items from reviews</td>
</tr>
<tr>
<td>Trouble Reports</td>
<td>Provides insight into the quality of the product and processes and the</td>
</tr>
<tr>
<td></td>
<td>effectiveness of testing</td>
</tr>
<tr>
<td>Requirements Stability</td>
<td>Provides visibility into the magnitude and impact of requirements changes</td>
</tr>
<tr>
<td>Size Stability</td>
<td>Provides insight into the stability of the requirements and into the</td>
</tr>
<tr>
<td></td>
<td>capability of the staff to complete project activities</td>
</tr>
</tbody>
</table>

VirginiaTech
Metric Definition: Derive Metric

<table>
<thead>
<tr>
<th>Indicator Category</th>
<th>Indicator</th>
<th>Recommended Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>SQA Audit and Review Results</td>
<td>Risk Management</td>
<td>Number of Items on Risk List</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of Open Issues</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of Action Items</td>
</tr>
<tr>
<td>Trouble Reports</td>
<td>Average Time to Close a Problem Report</td>
<td>Time from Assigned Problem Report Date to Problem Report Closure Date</td>
</tr>
<tr>
<td></td>
<td>Tracking Unplanned Changes</td>
<td>Number of Changes to Release Content (Unanticipated Problem Reports)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Number of Emergency Problem Reports</td>
</tr>
<tr>
<td>Requirements Stability</td>
<td>Tracking Requirements Stability</td>
<td>Number of Requirements Changes</td>
</tr>
<tr>
<td>Size Stability</td>
<td>Tracking Code Growth</td>
<td>Number of New Modules</td>
</tr>
</tbody>
</table>

What is Software Complexity?

- **Complexity** - degree to which a system or component is difficult to analyze, understand, or explain. Key factors include:
  - the # and intricacy of interfaces and branches
  - the degree of nesting
  - the types of data structures

- **Why is it important?**
  - Provides a means of relating the complexity of modules produced by the same process, in the same language, or within some other aspect of interest
  - It provides a systematic way of allocating sparse test or inspection resources to the modules which show the greatest need
  - Furnishes information on where the complexity in the code lies: data structures, flow control, interfaces...
McCabe’s Cyclomatic Complexity

- **Cyclomatic complexity** \( V(G) \)
  - \# of simple decisions + 1
  - \# of enclosed areas + 1
- **Derive the independent paths**
  - Path 1: 1,2,3,6,7,8
  - Path 2: 1,2,3,5,7,8
  - Path 3: 1,2,4,7,8
  - Path 4: 1,2,4,7,2,4,...7,8
- **\( V(G) < 10 \) is a good design guideline**
  - Functions with large CASE statements are often waived
- **\( V(G) > 40 \) should be scrutinized closely and probably redesigned**

Software Reliability

- **Software reliability** is the probability that a software system will perform its intended function without failure for a specified time in a specified environment

Source: George Stark (IBM)
Data Recording to Calculate the Failure Rate

Measuring individual failure times

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>3</td>
<td>22</td>
<td>91</td>
<td>158</td>
<td>212</td>
<td>227</td>
<td>306</td>
</tr>
</tbody>
</table>

Measuring number of failures in a time interval since the last failure

<table>
<thead>
<tr>
<th>Time</th>
<th>0</th>
<th>50</th>
<th>100</th>
<th>150</th>
<th>200</th>
<th>250</th>
<th>300</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Source: George Stark (IBM)

Establishing a Requirement Based on Cost

[Graph showing cost of fixing a fault during test, cost of failures during operations, and cost of testing as functions of test time (days)].

Source: George Stark (IBM)
Reliability Estimation During Test

Source: George Stark (IBM)

Modeling the Reliability Growth to Estimate a Release Date

Source: George Stark (IBM)
Key Principles for Analyzing Metrics

▲ Clearly defined metrics, consistently applied
▲ Metrics are only indicators, use them accordingly
▲ Focus on leading indicators over lagging ones
▲ Recognize indicators of problems
  ● Lack of change
  ● Frequent change
  ● Slow, steady deviation from plans

*Software metrics are navigational instruments giving position, direction, and rate of change*

Chapter 24: Metrics for OO Systems

▲ The purpose of this chapter is to describe several metrics that useful in assessing the quality of object-oriented systems.
  ● Basic process of using metrics is the same for both object-oriented and conventional software, but a change of focus is required
  ● In object design the smallest design unit is the encapsulated class and this is generally the target of OO metrics

▲ Objectives
  ● Examine OO measurement possibilities
  ● Outline key OO measurement techniques
Distinguishing Characteristics of MOOSE

Berard [BER95] argues that the following characteristics require that special OO metrics be developed:

- **Localization**—the way in which information is concentrated in a program
- **Encapsulation**—the packaging of data and processing
- **Information hiding**—the way in which information about operational details is hidden by a secure interface
- **Inheritance**—the manner in which the responsibilities of one class are propagated to another
- **Abstraction**—the mechanism that allows a design to focus on essential details

---

OO and Quality

Source: Paolo Nesi
### Research Citations on OO Metrics

<table>
<thead>
<tr>
<th>Category</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Empirical studies (e.g., case studies, experiments)</td>
<td>14.5%</td>
</tr>
<tr>
<td>Design and/or architecture</td>
<td>13.9%</td>
</tr>
<tr>
<td>Product metrics (e.g., complexity, coupling)</td>
<td>13.3%</td>
</tr>
<tr>
<td>Object-Oriented Languages (Smalltalk, C++, Java)</td>
<td>9.6%</td>
</tr>
<tr>
<td>Management issues (e.g., case for OO)</td>
<td>7.2%</td>
</tr>
<tr>
<td>Quality assurance (e.g., reliability, usability, etc.)</td>
<td>6.7%</td>
</tr>
<tr>
<td>Survey (bibliographies, books etc)</td>
<td>5.5%</td>
</tr>
<tr>
<td>Sizing and effort estimation</td>
<td>5.4%</td>
</tr>
<tr>
<td>Formal methods, theory, graph theory</td>
<td>5.1%</td>
</tr>
<tr>
<td>Reuse and/or reusability measurement</td>
<td>4.4%</td>
</tr>
<tr>
<td>Industrial focus or viewpoint</td>
<td>4.3%</td>
</tr>
<tr>
<td>Automated data collection, analysis etc.</td>
<td>4.2%</td>
</tr>
<tr>
<td>Maintenance or legacy system issues</td>
<td>3.3%</td>
</tr>
<tr>
<td>Testing and/or testability measurement</td>
<td>2.5%</td>
</tr>
</tbody>
</table>

Source: Empirical Software Engineering Research Group, Bournemouth University

### Class-Oriented Metrics

- **Weighted methods per class (WMC)** -- Sum of complexities of all methods of a class
  - More methods means more potential impact on children
  - # of methods and complexity of methods indicates time and effort required to develop and maintain the class

- **Depth of the inheritance tree (DIT)** -- The maximum length from the node to the root of the tree
  - Deeper a class is in the hierarchy, more methods it is likely to inherit
  - Deeper trees constitute greater design complexity, since more methods and classes are involved
  - The deeper a particular class is in the hierarchy, the greater the potential reuse of inherited methods

Source: Chidamber and Kemerer
Class-Oriented Metrics (continued)

▲ # of children (NOC) -- # of immediate subclasses
  ● More children => the greater the reuse, since inheritance is a form of reuse
  ● More children => more chance of improper abstraction of parent class (subclassing misuse)
  ● More children, require more testing of the methods in that class

▲ Coupling between object classes (CBO) -- # of classes to which a class is coupled
  ● Two classes are coupled when methods declared in one class use methods or instance variables of another class
  ● Excessive coupling between object classes is detrimental to modular design and prevents reuse
  ● Higher coupling increases the sensitivity to changes in other parts of the design, and reducing maintainability
  ● Coupling indicates complexity of testing for various parts of a design – higher the inter-object class coupling, the more rigorous the testing required

Source: Chidamber and Kemerer

Class-Oriented Metrics (concluded)

▲ Response for a Class (RFC) -- # of methods that can be invoked in response to a message sent to an object of a class
  ● High RFC complicates class testing & debugging
  ● High RFC indicates greater class complexity
  ● Worst case value for RFC assists in appropriate allocation of testing time

▲ Lack of cohesion in methods (LCOM) -- # of class methods that reference a variable
  ● Cohesiveness of methods promotes class encapsulation
  ● LCOM implies classes should be split into more subclasses
  ● Any measure of disparateness of methods helps identify flaws in the design of classes
  ● Low cohesion increases complexity, thereby increasing the likelihood of errors during the development process

Source: Chidamber and Kemerer
Class-Oriented Metrics

- **Class size (CS):** overall size of class
  - # of operations
  - # of attributes
- **NOO -** # of operations overridden by a subclass
  - Large NOO shows abstraction violation of superclass
- **NOA -** # of operations added by a subclass
  - Increasing NOA may indicate drift in abstraction
- **Specialization index (SI):** degree of specialization
  - \( \frac{\text{NOO} \times \text{DIT Level}}{\text{Total # of methods}} \)

Source: Lorenz and Kidd [LOR94]

---

**Class-Oriented Metrics (continued)**

*The MOOD metrics set: ratios for inheritance (MIF and AIF), polymorphism (PF), message-passing (CF), encapsulation (MHF and AHF)*

- **Method Inheritance Factor (MIF):** ratio of the sum of the inherited methods in all classes of the system to the total # of available methods (locally defined plus inherited) for all classes
- **Polymorphism Factor (PF):** ratio of the actual # of possible different polymorphic situations for class \( C_i \) to the maximum # of possible distinct polymorphic situations for class \( C_i \)
- **Coupling Factor (CF):** ratio of the maximum possible # of couplings in the system to the actual # of couplings not imputable to inheritance

Class-Oriented Metrics (continued)

▲ Attribute Inheritance Factor (AIF): ratio of the sum of inherited attributes in all classes of the system to the total # of available attributes (locally defined plus inherited) for all classes

▲ Method Hiding Factor (MHF): ratio of the sum of the invisibilities of all methods defined in all classes to the total # of methods defined in the system
  • The invisibility of a method is the percentage of the total classes from which this method is not visible.

▲ Attribute Hiding Factor (AHF): ratio of the sum of the invisibilities of all attributes defined in all classes to the total # of attributes defined in the system

Operation-Oriented Metrics

▲ Average operation size ($OS_{avg}$)
  • LOC is a weak indicator
  • # of messages sent is an alternative

▲ Operation complexity (OC)
  • Traditional complexity (e.g., Cyclomatic)
  • Target low OC

▲ Average number of parameters per operation ($NP_{avg}$)
  • More parameters more complex the collaboration (Beers)
  • Target low $NP_{avg}$

Source: Lorenz and Kidd [LOR94]
Testability Metrics

▲ Encapsulation related
  • Lack of cohesion in methods (LCOM)
  • % public and protected (PAP) – indicates the % of class attributes that are public (vulnerable to side-effects)
  • Public access to data members (PAD) – indicates # of classes/methods that can access a class’ attributes (vulnerable to side-effects)

▲ Inheritance related
  • # of root classes (NOR) – # of class hierarchies in the design (high NOR => more testing)
  • Fan-in (FIN) – level of multiple inheritance
  • # of children and depth of inheritance tree – NOC and DIT

Source: Binder [BIN94]

OO Project Metrics

▲ # of scenario scripts (NSS) -- # of scenario scripts or use-cases
  • Directly proportional to # of classes, states, methods, attributes, and collaborations
  • Indicator of size

▲ # of key classes (NKC)
  • Key classes focus on problem domain
  • High NKC indicates lots of development work

▲ # of subsystems (NSUB)
  • Like NKC, high NSUB indicates more work
  • Good for effort, resource allocation, and schedule

Source: Lorenz and Kidd [LOR94]
Homework Assignment for 4/20/01

▲ Read Pressman Chapters
  ● Chapter 26 – Cleanroom Software Engineering

▲ “Dog E-Dating System” Project
  ● Continue to refine Overall Test Plan with Test Cases for System and Integration Testing
    – Pick the top 3 subsystem/packages
  ● Final Project Package Due April 20th, 2001 – Next week!

▲ Have a great week!