* Compare and contrast the 4 levels of testing
* Compare and contrast debugging vs. testing
* Describe basis-path testing
* Compare and contrast white box vs. black box testing

**Testing**: executing a program to find errors.

**Types**: functional, ‘information content’ (errors in data structures), perf, ‘thread-based’ (all classes that respond to input or event), ‘use-based testing’ (pressman invention that starts testing at the top of the object model), ‘cluster’ aka integration, regression, ‘validation’ (user input, use-cases, event flow, validating against spec, resolve deviations w/ customer; audit), ‘acceptance’ (alpha/beta)

**Small to large**; unit, integration, system. Verification (building the product right) AND validation (building the right product). Cannot substitute for quality. One does not test one’s own code - removes conflict of interest. Independent test group. Ongoing collaboration with dev team to ensure coverage.

**Unit**: bugs in individual component/module/class. Verify interface information flow, functionality, error, boundary, memory corruption. Develop stubs and drivers as needed. Focus testing on changes (regression) and hi-pri bugs (smoke tests; BVT)

**Integration**: bugs when components interact/communicate/collaborate. Requires stubs and drivers. Top-down: stub subordinates in depth-first or breadth-first; test after adding each module; regression-test as you go. Bottom-up: cluster components and remove drivers on the way up. Verify ‘interface integrity’ between modules.

**Validation**: validate requirements (functional, behavioral, and performance) established during requirements analysis.

**System**: verifies quality characteristics (error-handling/recovery, perf, security, stress, UI)

**Testing vs. debugging** (strategic/planned/systematic vs. tactical/ad-hoc/unplanned, test vs. dev, identifying the presence of a problem vs. the location of a problem). Techniques: brute-force (printf, memory dumps), back-tracking (start from the error), ‘cause elimination’ (output analysis); aided by tracing tools. Fix considerations: regression risk, defect prevention, duplicate issues in other parts of the code.

A good test: non-redundant, correct-complexity-level, best-of-breed, high probability of error-finding.

**Testability**: (James Bach). Operability (working software), observability, controllability, decomposability (modularity), simplicity, stability, understandability.

**Black box v. white box.**

**Whitebox**: all independent paths/logical decisions/conditionals executed at least once? Boundary conditions? Valid data structures?

**Basis Path**: execution path; conditionals and loops. Based on ‘**program flow graph’**. ‘**Cyclomatic complexity’** calculated from graph using formula or by counting conditional statements in PDL (procedure description language) representation; used to predict which parts of the code are most error-prone. ‘Basis set’ of linearly independent paths: taken together, causes each code block to be executed at least once. Each case executes each path in the basis set.

**Control Structure**: testing each condition in each decision statement. Covers all logical combinations of data; uses truth tables. **Data flow**: definition use chains. (function is defined in one unique spot and called from ‘n’ other unique spots). **Loop**: loop entry and exit conditions.

**Black-Box Testing**: Functional validity? System behavior and performance? Which input classes? Which input values are sensitive/boundary? MTTF/system capacity (data rates and volume)? Specific combinations of data?

**Graph-based**: the nature of the relationships (links) among program objects (nodes). Cases traverse the entire graph. **Transaction flow**: nodes are steps in a transaction; links are logical connections between steps that need to be validated. **Finite state modeling**: nodes are observable software states; links are state transitions. **Data flow modeling**: nodes are data objects; links are data object transformations. **Timing modeling**: nodes are program objects; links are sequential connections between these objects, link weights are required.

**Equivalence Partitioning**: divides input domain into equivalence classes from which test cases can be derived. Based on input. **Range**: one valid and two invalid equivalence classes. **Specific value**: one valid and two invalid. **Set member**: one valid and one invalid. **Boolean**: one valid and one invalid.

**Boundary Value Analysis**: boundaries of the input domain. Bounded by values a and b: test cases should include a, b, a + 1, a – 1, b + 1, b -1. Test at size boundaries.

**Comparison Testing**: independent implementations of redundant systems are tested for spec conformance. Equivalence class partitioning used to develop a common set of test cases for each implementation.

**Orthogonal Array Testing**: small set of test cases providing maximum test coverage. Categories of faulty logic likely to be present in the software component (without examining the code). Region faults: error category associated with faulty logic. Detect and isolate all single mode faults (single input parameter causes failure); double mode faults; multimode faults

**OO Test Case Design**: often black-box. Classes inherit methods from parents and redefine them. Scenario-based (review use-cases). **Random**. **Partition**: state-based (state change operations are tested separately from non). **Attribute**: operations grouped by those that use, modify, or do not use/modify the attribute. **Category**: operations grouped according to their function: initialization, computation, query, termination.

**Specialized**: **GUI**: developed from behavioral model of user interface, use automated tools. **Client/Sever** Architectures: 3-levels (disconnected, no network, complete application). API, server, database, transaction, network, doc/help. Real-time systems: Task: test each task independently Behavioral: equivalence partitioning of external event models created by automated tools. Intertask: synchronization and communication errors. System: interrupt-handling; cases based on state model and control specification.

**Testing Patterns**: provide useful advice, vocabulary, for testing activities. Focus attention the ‘when’ and ‘why’ of a solution. Encourage iterative thinking.

Chapter 15 – Product Metrics For Software

Overview

This chapter describes the use of product metrics in the software quality assurance process. The use of metrics for project management is discussed later in the text. Software engineers use product metrics to help them assess the quality of the design and construction of the software product being built. Product metrics provide software engineers with a basis to conduct analysis, design, coding, and testing more objectively. Qualitative criteria for assessing software quality are not always sufficient by themselves. The process of using product metrics begins by deriving the software measures and metrics that are appropriate for the software representation under consideration. Then data are collected and metrics are computed. The metrics are computed and compared to pre-established guidelines and historical data. The results of these comparisons are used to guide modifications made to work products arising from analysis, design, coding, or testing.

Software Quality Principles - a Qualitative View

1. Conformance to software requirements is the foundation from which quality is measured.

2. Specified standards define a set of development criteria that guide the manner in which software is engineered.

3. Software quality is suspect when a software product conforms to its explicitly stated requirements and fails to conform to the customer's implicit requirements (e.g., ease of use).

McCall's Quality Factors

\* Product Operation

\* Correctness

\* Efficiency

\* Integrity

\* Reliability

\* Usability

\* Product Revision

\* Flexibility

\* Maintainability

\* Testability

\* Product Transition

\* Interoperability

\* Portability

\* Reusability

McCall's Software Metrics

\* Auditability

\* Accuracy

\* Communication commonality

\* Completeness

\* Consistency

\* Data commonality

\* Error tolerance

\* Execution efficiency

\* Expandability

\* Generality

\* Hardware independence

\* Instrumentation

\* Modularity

\* Operability

\* Security

\* Self-documentation

\* Simplicity

\* Software system independence

\* Traceability

\* Training

ISO 9126 Quality Factors

\* Functionality

\* Reliability

\* Usability

\* Efficiency

\* Maintainability

\* Portability

Benefits of Product Metrics

1. Assist in the evaluation of the analysis and evaluation model

2. Provide indication of procedural design complexity and source code complexity

3. Facilitate design of more effective testing

Measurement Process Activities

\* Formulation - derivation of software measures and metrics appropriate for software representation being considered

\* Collection - mechanism used to accumulate the date used to derive the software metrics

\* Analysis - computation of metrics

\* Interpretation - evaluation of metrics that results in gaining insight into quality of the work product

\* Feedback - recommendations derived from interpretation of the metrics is transmitted to the software development team

Measurement Principles

\* The objectives of measurement should be established before collecting any data.

\* Each product metric should be defined in an unambiguous manner.

\* Metrics should be derived based on a theory that is valid for the application domain.

\* Metrics should be tailored to accommodate specific products and processes.

Metrics Characterization and Validation Principles

\* A metric should have desirable mathematical properties

\* The value of a metric should increase when positive software traits occur or decrease when undesirable software traits are encountered

\* Each metric should be validated empirically in several contexts before it is used to make decisions

Measurement Collection and Analysis Principles

1. Automate data collection and analysis whenever possible

2. Use valid statistical techniques to establish relationships between internal product attributes and external quality characteristics

3. Establish interpretive guidelines and recommendations for each metric

Goal-Oriented Software Measurement (GQM)

\* GQM emphasizes the need

1. establish explicit measurement goal specific to the process activity or product characteristic being assessed

2. define a set of questions that must be answered in order to achieve the goal

3. identify well-formulated metrics that help to answer these questions

\* A goal definition template can be used to define each measurement goal

Attributes of Effective Software Metrics

\* Simple and computable

\* Empirically and intuitively persuasive

\* Consistent and objective

\* Consistent in use of units and measures

\* Programming language independent

\* Provides an effective mechanism for quality feedback

Important Metrics Areas

\* Analysis Model Aspects

o Functionality delivered

o System size

o Specification quality

\* Design Model Attributes

o Architecture metrics

o Component-level metrics

o Specialized OO Design Metrics

\* Source Code Characteristics

o Halstead metrics

o Complexity metrics

o Length metrics

\* Testing

o Statement and branch coverage metrics

o Defect-related metrics

o Testing effectiveness

o In-process metrics

Representative Analysis Metrics

\* Function-based metrics

\* Specification quality metrics (Davis)

Representative Design Metrics

\* Architectural design metrics

o Structural complexity (based on module fanout)

o Data complexity (based on module interface inputs and outputs)

o System complexity (sum of structural and data complexity)

o Morphology (number of nodes and arcs in program graph)

o Design structure quality index (DSQI)

\* OO design metrics

o Size (population, volume, length, functionality)

o Complexity (how classes interrelate to one another)

o Coupling (physical connections between design elements)

o Sufficiency (how well design components reflect all properties of the problem domain)

o Completeness (coverage of all parts of problem domain)

o Cohesion (manner in which all operations work together)

o Primitiveness (degree to which attributes and operations are atomic)

o Similarity (degree to which two or more classes are alike)

o Volatility (likelihood a design component will change)

\* Class-Oriented Metrics

o Chidamber and Kemerer (CK) Metrics Suite

\* weighted metrics per class (WMC)

\* depth of inheritance tree (DIT)

\* number of children (NOC)

\* coupling between object classes (CBO)

\* response for a class (RFC)

\* lack of cohesion in methods (LCOM)

o Harrison, Counsel, and Nithi (MOOD) Metrics Suite

\* method inheritance factor (MIF)

\* coupling factor (CF)

\* polymorphism factor (PF)

o Lorenz and Kidd

\* class size (CS)

\* number of operations overridden by a subclass (NOO)

\* number of operations added by a subclass (NOA)

\* specialization index (SI)

\* Component-level design metrics

o Cohesion metrics (data slice, data tokens, glue tokens, superglue tokens, stickiness)

o Coupling metrics (data and control flow, global, environmental)

o Complexity metrics (e.g., cyclomatic complexity)

\* Operation-Oriented Metrics

o Average operation size (OSavg)

o Operation complexity (OC)

o Average number of parameters per operation (NPavg)

\* Interface design metrics (e.g., layout appropriateness)

Halstead's Metrics (Source Code Metrics)

\* Overall program length

\* Potential minimum algorithm volume

\* Actual algorithm volume (number of bits used to specify program)

\* Program level (software complexity)

\* Language level (constant for given language)

Testing Metrics

\* Metrics that predict the likely number of tests required during various testing phases

o Function-based metrics (e.g., function points)

o Architectural design metrics

o Cyclomatic complexity can target modules that are candidates for extensive unit testing

o Halstead effort

\* Metrics that focus on test coverage for a given component

o Cyclomatic complexity lies at the core of basis path testing

Object-Oriented Testing Metrics

\* Encapsulation

o Lack of cohesion in methods (LCOM)

o Percent public and protected (PAP)

o Public access to data members (PAD)

\* Inheritance

o Number of root classes (NOR)

o Fan in (FIN)

o Number of children (NOC)

o Depth of inheritance tree (DIT)

\* Class complexity

\* Weighted metrics per class (WMC)

\* Coupling between object classes (CBO)

\* Response for a class (RFC)

Maintenance Metrics

\* Software Maturity Index (IEEE Standard 982.1-1988)

\* SMI approaches 1.0 as product begins to stabilize