Chapter 14
Quality Concepts

Software Engineering: A Practitioner's Approach, 7/e
by Roger S. Pressman

Quality!—A Philosophical View

Robert Persig [Phil74] commented on the thing we call quality.

Quality... you know what it is, yet you don't know what it is. But that's self-contradictory. But some things are better than others, that is, they have more quality. But when you try to say what the quality is, apart from the things that have it, it all goes poof! There's nothing to talk about. But if you can't say what Quality is, how do you know what it is, or how do you know that it even exists? If no one else knows what it is, then for all practical purposes it doesn't exist at all. But for all practical purposes it really does exist. What else are the grades based on? Why else would parents buy furniture for some things and throw others in the trash pile? Obviously some things are better than others... but what's the betterness? So round and round you go, spinning mental wheels and nowhere finding anywhere to stop.

What is it?

Quality!—A Pragmatic View

Pragmatism: Concerned with practical matters "quality is a complex and multifaceted concept."

The transcendental view argues (Williamson) that quality is something that you immediately recognize, but cannot explicitly define.

The user view sees quality in terms of an end-user's specific goals. If a product meets those goals, it exhibits quality.

The manufacturer's view defines quality in terms of the original specification of the product. If the product conforms to the spec, it exhibits quality.

The product view suggests that quality can be tied to inherent characteristics (e.g., functions and features) of a product.

Finally, the value-based view measures quality based on how much a customer is willing to pay for a product. In reality, quality encompasses all of these views and more.

Software Quality

Quality?

The American Heritage Dictionary defines quality as:

"a characteristic or attribute of something."

For software, two kinds of quality may be encountered:

- Quality of design encompasses requirements, specifications, and the design of the system, (functions and features of requirements model).
- Quality of conformance is an issue focused primarily on implementation of design and system's requirements and performance goals.

User satisfaction = compliant product + good quality + delivery within budget and schedule.

I. Effective Software Process

A useful software process establishes the infrastructure that supports any effort at building a high quality software product.

The management aspects of process create the checks and balances that help avoid project pooh—a key contributor to poor quality.

Software engineering practices allow the developer to analyze the problem and design a solid solution—both critical to building high quality software.

Finally, umbrella activities such as change management and technical reviews have as much to do with quality as any other part of software engineering practice.

II. Useful Product

A useful product delivers the content, functions, and features that the end-user desires.

But as important, it delivers these assets in a reliable, error free way.

A useful product always satisfies those requirements that have been explicitly stated by stakeholders.

In addition, it satisfies a set of implicit requirements (e.g., ease of use) that are expected of all high quality software.

III. Adding Value

By adding value for both the producer and user of a software product, high quality software provides benefits for the software organization and the end-user community.

The software organization gains added value because high quality software requires less maintenance effort, fewer bug fixes, and reduced customer support.

The user community gains added value because the application provides a useful capability in a way that exceeds some business process.

The end result is:

1. Greater software product revenue,
2. Better profitably when an application supports a business process, and
3. Improved availability of information that is crucial for the business.
Garvin’s Quality Dimensions...

- David Garvin [Gar87]
  - Performance Quality. Does the software deliver all content, functions, and features that are specified as part of the requirements model in a way that provides value to the end-user?
  - Feature quality. Does the software provide features that surprise and delight first-time end-users?
  - Reliability. Does the software deliver all features and capability without failures? Is it available when it is needed? Does it deliver functionality that is error free?
  - Conformance. Does the software conform to local and external software standards that are relevant to the application? Does it conform to de-facto design and coding conventions? For example, does the user interface conform to accepted design rules for menu selection or data input?

Garvin’s Quality Dimensions

- Durability. Can the software be maintained (changed) or corrected (debugged) without the inadvertent generation of unintended side effects? What changes cause the error rate or reliability to degrade with time?
- Serviceability. Can the software be maintained (changed) or corrected (debugged) in an acceptable short time period. Can support staff acquire all information they need to make changes or correct defects?
- Aesthetics. Most of us would agree that an aesthetic entity has a certain elegance, a unique flow, and an obvious "presence" that are hard to quantify but evident nonetheless.
- Perception. In some situations, you have a set of prejudices that will influence your perception of quality.

Other Views

- McCall’s Quality Factors (SEPA, Section 14.2.2)
  - Maintainability - effort required to locate and fix an error in a program
  - Flexibility - effort required to modify an operational program
  - Testability - effort required to test a program to ensure that it performs its intended function

Product Operation - McCall’s

- Correctness - extent to which a program satisfies its specification and fulfills the customer's mission objectives
- Reliability - extent to which a program can be expected to perform its intended function with required precision
- Efficiency - amount of computing resources and code required by a program to perform its function
- Integrity - extent to which access to software or data by unauthorized persons can be controlled
- Usability - effort required to learn, operate, prepare input for, and interpret output of a program

Product Transition - McCall’s

- Portability - effort required to transfer the program from one hardware and/or software system environment to another
- Reusability - extent to which a program (or parts of a program) can be reused in other applications
- Interoperability - effort required to couple one system to another

Product Revision - McCall’s

ISO 9126 Quality Factors

- Functionality
- Reliability
- Usability
- Efficiency
- Maintainability
- Portability

Measuring Quality

Targeted Factors

- General quality dimensions and factors are not adequate for assessing the quality of an application in concrete terms
- Project teams need to develop a set of targeted questions to assess the degree to which each quality factor has been satisfied in the application
- Subjective measures of software quality may be viewed as little more than personal opinion
- Software metrics represent indirect measures of some manifestation of quality and are an attempt to quantify the assessment of software quality

The Software Quality Dilemma

- If you produce a software system that has terrible quality, you lose because no one will want to buy it.
- If on the other hand you spend infinite time, extremely large effort, and huge sums of money to build the absolutely perfect piece of software, then it’s going to take so long to complete and it will be so expensive to produce that you’ll be out of business anyway.
- Either you missed the market window, or you simply exhausted all your resources.
- So people in industry try to get to that magical middle ground where the product is good enough not to be rejected right away, such as during evaluation, but also not the object of so much perfectionism and so much work that it would take too long or cost too much to complete. [Vent03]
I. “Good Enough” Software

- Good enough software delivers high quality functions and features that end-users desire, but at the same time it delivers other more obscure or specialized functions and features that contain known bugs.
- Arguments against “good enough.”
  - It is true that “good enough” may work in some application domains, but for a few major software companies, after all it is a company has a large marketing budget and can convince enough people to buy version 1.0, it has succeeded in leaking them in.
  - If you work for a small company, you are better if this philosophy. If you deliver a “good enough” buggy product, you risk permanent damage to your company’s reputation.
  - You may need to develop version 2.0 because the buzz may cause your sales to plummet and your company to fail.
  - If you work in certain application domains (e.g., real-time embedded systems), application software that is integrated with hardware can be expensive and open your company to expensive litigation.

IV. Quality and Risk

- People bet their jobs, their company’s, their safety, their entertainment, their decisions, and their very lives on computer software. It better be right.” SEPA, Chapter 1
- Low quality software is a risk for developer and user.
- Example:
  - Throughout the month of November 2006 at a hospital in Panama, 28 patients received massive overdoses of gamma rays during treatment for a variety of cancers. In the months that followed, five of these patients died from radiation poisoning and 15 others developed serious complications. What caused this tragedy? A software package, developed by a U.S. company, was modified by hospital technicians to compute modified doses of radiation for each patient.

II. Cost of Quality

- Prevention costs include:
  - Quality planning
  - Formal technical reviews
  - Test equipment
  - Training
- Internal failure costs include:
  - Revert
  - Repair
  - Failure mode analysis
- External failure costs are:
  - Complaints resolution
  - Product return and replacement
  - Help line support
  - Warranty work

III. Quality Cost

- The relative costs to find and repair an error or defect increase dramatically as we go from prevention to detection to internal failure to external failure costs.

V. Negligence and Liability

- The story is all too common. A governmental organization or corporate entity hires a major software developer or consulting company to analyze requirements and then design and construct a software-based “system” to support some major activity.
- The system might support a major corporate function (e.g., payroll management) or some governmental function (e.g., healthcare administration or homeland security).
- Work begins with the best of intentions on both sides, but by the time the system is delivered, things have gone bad.
- The system is late, fails to deliver desired features and functions, is error-prone, and does not meet with customer approval.
- Litigation ensues.

VI. Quality and Security

- Gary McGraw comments [WIDS]:
  - “Software security makes entirely and completely to quality. You must think about security, reliability, availability, and dependability—at the beginning, in the design, architecture, test, and coding phases, all through the software life cycle [process]. Even people aware of the software security problem have focused on late-life cycle stuff. The earlier you find the software problem, the better. And there are two kinds of software problems. One is bugs, which are implementation problems. The other is software flaws—architectural problems in the design. People pay too much attention to bugs and not enough on flaws.”

Achieving Software Quality

- Critical success factors:
  - Software Engineering Methods
  - Project Management Techniques
  - Quality Control
  - Quality Assurance

Chapter 15

Review Techniques

Software Engineering: A Practitioner’s Approach, 7th edition by Roger S. Pressman

Reviews

... there is no particular reason why your friend and colleague cannot also be your sternest critic.

Jerry Weinberg
What Are Reviews?

- a meeting conducted by technical people for technical people
- a technical assessment of a work product created during the software engineering process
- a software quality assurance mechanism
- a training ground

What Reviews Are Not

- A project summary or progress assessment
- A meeting intended solely to impart information
- A mechanism for political or personal reprisal

What Do We Look For?

- Errors and defects
  - Errors—a quality problem found before the software is released to end users
  - Defects—a quality problem found only after the software has been released to end users
- We make this distinction because errors and defects have very different economic, business, psychological, and human impact
- However, the temporal distinction made between errors and defects in this book is not mainstream thinking

Defect Amplification

- Defect amplification model (DAMIT) can be used to illustrate the generation and detection of errors during the design and code generation actions of a software process.

Defect Amplification

- In the example provided in SEPA, Section 15.2,
  - a software process that does NOT include reviews:
    - yields 94 errors at the beginning of testing and
    - releases 12 latent defects to the field
  - a software process that does include reviews:
    - yields 24 errors at the beginning of testing and
    - releases 3 latent defects to the field
  - A cost analysis indicates that the process with NO reviews costs approximately 3 times more than the process with reviews, taking the cost of correcting the latent defects into account

Review Metrics and Use

- Required as part of the good quality software engineering practice.
- To understand the effectiveness of each action by defining a set of metrics that can be used to assess their efficacy.
- 1. Analyzing Metrics
- 2. Cost effectiveness of reviews

Metrics – Analysis

- The total review effort and the total number of errors discovered are defined as:
  - \[ E_{\text{total}} = E_{\text{pre}} + E_{\text{post}} \]
  - \[ E_{\text{review}} = E_{\text{total}} - E_{\text{post}} \]
  - Defect density represents the errors found per unit of work product reviewed.
    - Defect density = \[ \frac{E_{\text{total}}}{\text{WPS}} \]
    - where ...

Metrics

- Preparation effort, \( E_{\text{pre}} = \) the effort (in person-hours) required to review a work product prior to the actual review meeting
- Assessment effort, \( E_{\text{ass}} = \) the effort (in person-hours) that is expending during the actual review
- Network effort, \( E_{\text{net}} = \) the effort (in person-hours) that is dedicated to the correction of those errors uncovered during the review
- Work product size, WPS—a measure of the size of the work product that has been reviewed (e.g., the number of UML models, or the number of document pages, or the number of lines of code)
- Minor errors found, \( E_{\text{minor}} = \) the number of errors found that can be categorized as minor (requiring less than some prespecified effort to correct)
- Major errors found, \( E_{\text{major}} = \) the number of errors found that can be categorized as major (requiring more than some prespecified effort to correct)

Cost effectiveness, An Example-1

- If past history indicates that
  - the average defect density for a requirements model is 0.6 errors per page, and a new requirement model is 32 pages long
  - a rough estimate suggests that your software team will find about 19 or 20 errors (i.e., 0.6 x 32) during the review of the document
  - If you find only 6 errors, you've done an extremely good job in developing the requirements model or your review approach was not thorough enough.
Cost effectiveness. An Example-II

- The effort required to correct a minor defect error immediately after the review was found to require 6 person-weeks.
- The effort required for a major requirement error was found to be 18 person-weeks.
- Examining the review data collected, you find that minor errors occur about 6 times more frequently than major errors. Therefore, you can estimate that the average effort to find and correct a requirements error during review is about 0.6 person-weeks
- Requirements-related errors uncovered during testing require an average of 45 person-hours to fix. Using the averages noted, we get:
  - Cost savings per error: $500,000 / $45 = 10 person-hours
  - Cost savings per requirement error: 10 person-hours

Since all errors were found during the review of the requirements model, a saving of about $40,000 per error in testing effort would be achievable. And that's just for requirements-related errors.

1. Informal Reviews

- Informal reviews include:
  - A simple desk check of a software engineering work product with a colleague.
  - A casual meeting (involving more than 2 people) for the purpose of reviewing a work product, or
  - The review-oriented aspects of pair programming.
- Pair programming encourages continuous review as a work product (design or code) is created.
- The benefit is immediate discovery of errors and better work product quality as a consequence.

2. Formal Technical Reviews

- The objectives of an FTR are:
  - To uncover errors in function, logic, or implementation for any representation of the software.
  - To verify that the software under review meets its requirements.
  - To ensure that the software has been represented according to predefined standards.
  - To achieve software that is developed in a uniform manner.
  - To make projects more manageable.
- The FTR is actually a class of reviews that includes walk-throughs and inspections.

a. The Review Meeting...

- Between three and five people (typically) should be involved in the review.
- Advance preparation should occur but should require no more than two hours of work for each person.
- The duration of the review meeting should be less than two hours.
- Focus is on a work product (e.g., a portion of a requirements model, a detailed component design, source code for a component).

...The Players

- **Producer**—The individual who has developed the work product.
  - Informs the project leader that the work product is complete and that a review is required.
- **Review leader**—Evaluates the product for readiness, generates copies of the product materials, and distributes them to two or three reviewers for advance preparation.
- **Reviewer(s)**—Expected to spend between one and two hours reviewing the product, making notes, and otherwise becoming familiar with the work.
- **Recorder**—Revises who records (in writing) all important issues raised during the review.

Conducting the Review (guidelines)

- Review the product, not the producer.
- Set an agenda and maintain it.
- Limit debate and robustness.
- Enumerate problem areas, but don't attempt to solve every problem noted.
- Take written notes.
- Limit the number of participants and insist upon advance preparation.
- Develop a checklist for each product that is likely to be reviewed.
- Allocate resources and schedule time for FTRs.
- Conduct meaningful training for all reviewers.
- Review your early reviews.
Chapter 16

Software Quality Assurance

Software Engineering: A Practitioner’s Approach, 7/e
by Roger S. Pressman

Comment on Quality

Phil Crosby once said:

- The problem of quality management is not what people don’t know about it. The problem is what they think they do know. . . . In this regard, quality has much in common with sex.
- Everybody is for it. (Under certain conditions, of course.)
- Everyone feels they understand it. (Even though they wouldn’t want to explain it.)
- Everyone thinks execution is a matter of following natural inclinations. (After all, we do get along somehow.)
- And, of course, most people feel that no problems in these areas are caused by other people. (If only they would take the time to do things right.)

Role of the SQA Group-I

- Prepares an SQA plan for a project.
  - The plan identifies:
    - work items to be performed,
    - audits and reviews to be performed,
    - standards that are applicable to the project,
    - procedures for work reporting and tracking,
    - documents to be produced by the SQA group,
    - amount of leadership provided to the software project team.
- Participates in the development of the project’s software process description.
  - The SQA group reviews the process description for compliance with organizational policy, internal software standards, externally imposed standards (e.g., ISO-9001), and other parts of the software project plan.

Role of the SQA Group-II

- Reviews software engineering activities to verify compliance with the defined software process.
  - Identifies, documents, and tracks deviations from the process and verifies that corrections have been made.
- Audits designated software work products to verify compliance with those defined as part of the software process.
  - Reviews selected work products identified, documented, and tracked deviations, verifies that corrections have been made.
  - Periodically reports the results of its work to the project manager.
- Ensures that deviations in software work and work products are documented and handled according to a documented procedure.
- Records any noncompliance and reports to senior management.
  - Noncompliance items are tracked utilily until they are resolved.

SQA Goals (see Figure 16.1)

- Requirements quality. The correctness, completeness, and consistency of the requirements model will have a strong influence on the quality of all work products that follow.
- Design quality. Every element of the design model should be assessed by the software team to ensure that it exhibits high quality and that the design itself conforms to requirements.
- Code quality. Source code and related work products (e.g., other descriptive information) must conform to local coding standards and exhibit characteristics that will facilitate maintainability.
- Quality control effectiveness. A software team should apply limited resources in a way that has the highest likelihood of achieving a high quality result.
Statistical SQA

- Collect information on all defects.
- Find the causes of the defects.
- Move to provide fixes for the process.

... an understanding of how to improve quality ...

Software Reliability

- A simple measure of reliability is mean-time-between-failure (MTBF), where:
  \[ \text{MTBF} = \frac{\text{MTTF} \times \text{MTTR}}{\text{MTTF} + \text{MTTR}} \]

- The acronym MTTF and MTTR are mean-time-to-failure and mean-time-to-repair, respectively.

- Software availability is the probability that a program is operating according to requirements at a given point in time and is defined as:
  \[ \text{Availability} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}} \times 100\% \]

Software Safety

- Software safety is a software quality assurance activity that focuses on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail.
- If hazards can be identified early in the software process, software design features can be specified that will either eliminate or control potential hazards.

Six-Sigma for Software Engineering

- The term “six sigma” is derived from six standard deviations—3.4 failures (defects) per million occurrences—implying an extremely high quality standard.
- The Six Sigma methodology defines three core steps:
  - Define customer requirements and deliverables and project goals via a customer-focused methodology.
  - Measure the existing process and its output to determine current quality performance (collect defect metrics).
  - Analyze defect metrics and determine the root causes.
  - Improve the process by eliminating the root causes of defects.
  - Control the process to ensure that future work does not reintroduce the causes of defects.

ISO 9001:2000 Standard

- ISO 9001:2000 is the quality assurance standard that applies to software engineering.
- The standard contains 20 requirements that must be present for an effective quality assurance system.
- The requirements delineated by ISO 9001:2000 address topics such as:
  - Management responsibility, quality system, contract review, design control, document control, and data control, product identification and traceability, process control, inspection and testing, corrective and preventive action, control of quality records, internal quality audits, training, servicing, and statistical techniques.

Chapter 22

- Software Configuration Management

Software Engineering: A Practitioner’s Approach, 7/e
by Roger S. Pressman

The “First Law”

No matter where you are in the system life cycle, the system will change, and the desire to change it will persist throughout the life cycle.  — Bersoff, et al, 1980

What Are These Changes?
The Software Configuration

- programs
- documents
- data

The pieces

Baselines

- The IEEE (IEEE Std. No. 610.12-1990) defines a baseline as:
  - A specification or product that has been formally reviewed and agreed upon, that thereafter serves as the basis for further development, and that can be changed only through formal change control procedures.
  - A baseline is a milestone in the development of software that is marked by the delivery of one or more software configuration items and the approval of these SCIs that is obtained through a formal technical review.

Software Configuration Objects

- Object:
  - Data items:
  - Execution environment
  - Component Interface
  - Component characterization information
  - Component attributes
  - Linkage information
  - Work products (e.g., design, test, and support materials)
  - Source code

SCM Repository

- The SCM repository is the set of mechanisms and data structures that allow a software team to manage change in an effective manner.
- The repository performs or precipitates the following functions [For98]:
  - Data integrity
  - Information sharing
  - Tool integration
  - Data integration
  - Methodology enforcement
  - Document standardization

Repository Content

- Component elements—a set of tools that manages the construction of software by ensuring that the proper set of validated components (i.e., the correct version) have been assembled.
- Human elements—to implement effective SCM, the software team uses a set of tools and process features (encompassing other CM elements)

The SCM Process

Addresses the following questions:

- How does a software team identify the discrete elements of a software configuration?
- How does an organization manage the many existing versions of a program (and its documentation) in a manner that will enable change to be accommodated efficiently?
- How does an organization control changes before and after software is released to a customer?
- Who has responsibility for approving and ranking changes?
- How can we ensure that changes have been made properly?
- What mechanism is used to appraise others of changes that are made?
The SCM Process

Version Control

- Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process.
- A version control system implements or is directly integrated with four major capabilities:
  - A project database (repository) that stores all relevant configuration objects.
  - A version management capability that stores all versions of a configuration object or any version to be constructed using differences from past versions.
  - A change control capability that enables the software engineer to collect all relevant configuration objects and construct a specific version of the software.
  - An issue tracking (also called bug tracking) capability that enables the team to record and track the status of all outstanding issues associated with each configuration object.

Change Control Process—II

- Change Control Process—II
  - need for change is recognized
  - change request from user
  - developer evaluates
  - change report is generated
  - change control authority decides
    - request is queued for action
    - change request is denied
    - user is informed

Change Control Process—III

- Change Control Process—III
  - assign people to SCIs
  - check-out SCIs
  - make the change
  - review/audit the change
    - establish a "baseline" for testing
  - change control process—II

Auditing

- Auditing

Status Accounting

- Status Accounting

SCM for Web Engineering—II

- SCM for Web Engineering—II
  - Content
    - A typical WebApp contains a vast array of content—text, graphics, applets, scripts, audio/video files, forms, active page elements, tables, streaming data, and many others.
    - The challenge is to organize this sea of content into a rational set of configuration objects (Section 27.1.4) and then establish appropriate configuration control mechanisms for these objects.
  - People
    - Because a significant percentage of WebApp development continues to be conducted in an ad hoc manner, any person involved in the WebApp cannot (and often does) create content.
SCM for Web Engineering-II

- Scalability.
  - As size and complexity grow, small changes can have far-reaching and unintended effects that can be problematic. Therefore, the rigor of configuration control mechanisms should be directly proportional to application scale.
- Politics.
  - Who owns a WebApp?
  - Who assumes responsibility for the accuracy of the information on the Web site?
  - Who ensures that quality control processes have been followed before information is published to the site?
  - Who is responsible for making changes?
  - Who assumes the cost of change?

Content Management

- How can content be made available to end users?
- How can content be maintained?

Change Management for WebApps-I

- Change requests should be approved before being implemented.
- Changes should be implemented in a controlled manner.

Chapter 23

- Product Metrics
- McCall’s Triangle of Quality
- A Comment

McCall’s quality factors were proposed in the early 1970s. They are as valid today as they were in that time. It’s likely that software built to conform to these factors will exhibit high quality well into the 21st century, 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.

Goal-Oriented Software Measurement

- The Goal-Question-Metric Paradigm
  - (1) establish an explicit measurement goal that is specific to the process activity or product characteristic that is to be measured;
  - (2) define a set of questions that must be answered in order to achieve the goal,
  - (3) identify well-formed metrics that help to answer those questions.
- Goal definition template:
  - Goals: (the name of activity or attribute to be measured)
  - for the purpose of (the overall objective of the analysis)
  - with regard to (the aspect of activity or attribute to be considered)
  - from the viewpoint of (the people who have an interest in the measurement).
  - in the context of (the environment in which the measurement takes place).

Metrics Attributes

- Simple and comprehensible. It should be relatively easy to learn how to derive the metric and its computation should not demand undue effort or training.
- Empirically and intuitively persuasive. The metric should satisfy the engineer's intuitive notion about the product attribute under consideration.
- Consistent and objective. The metric should always yield results that are unambiguous.
- Consistent in its use of units and dimensions. The mathematical computation of the metric should use measures that do not lead to bizarre combinations of units.
- Programming language independent. Metrics should be based on the analysis model, the design model, or the structure of the program itself.
- Effective mechanism for quality feedback. That is, the metric should provide a software engineer with information that can lead to a higher quality end product.

Metrics for the Requirements Model

- Function-based metrics: use the function point as a normalizing factor or as a measure of the "size" of the specification.
- Specification metrics: used as an indication of quality by measuring number of requirements by type.

Function-Based Metrics

- The function point metric (FP), first proposed by Albrecht [ALB79], can be used effectively as a means for measuring the functionality delivered by a system.
- Function points are derived using an empirical relationship based on countable (direct) measures of software information domain and assessments of software complexity.
- Information domain values are defined in the following manner:
  - number of external inputs (I-E)
  - number of external outputs (O-E)
  - number of external inquiries (Q-E)
  - number of internal logical files (I-LF)
  - number of external interface links (E-IL)

Function Points

<table>
<thead>
<tr>
<th>Information Domain Value</th>
<th>Count</th>
<th>Weighting factor</th>
<th>sample_average</th>
<th>sample_total</th>
</tr>
</thead>
<tbody>
<tr>
<td>External inputs (E-I)</td>
<td>3</td>
<td>2.5</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>External outputs (E-O)</td>
<td>2</td>
<td>2.5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>External inquiries (E-Q)</td>
<td>1</td>
<td>2.5</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>Internal logical files (I-LF)</td>
<td>3</td>
<td>2.5</td>
<td>7.5</td>
<td></td>
</tr>
<tr>
<td>External interface files (E-IL)</td>
<td>1</td>
<td>2.5</td>
<td>2.5</td>
<td></td>
</tr>
</tbody>
</table>

Function Point Total: 20
Architectural Design Metrics

- Architectural design metrics
  - Structural complexity = \( g(\text{fan-out}) \)
  - Data complexity = \( f(\text{input \& output variables, \text{fan-out}}) \)
  - System complexity = \( h(\text{structural \& data complexity}) \)
  - HK metric: architectural complexity as a function of fan-in and fan-out
  - Morphology metrics: a function of the number of modules and the number of interfaces between modules

Metrics for OO Design-I

- Whitmire [Whi67] describes nine distinct and measurable characteristics of an OO design:
  - Size
    - Size is defined in terms of four classes: population, volume, length, and functionality
  - Complexity
    - Size classes of an OO design are interrelated to one another
  - Coupling
    - The physical connections between elements of the OO design
  - Size
    - 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
    - An indicator of the degree to which the abstraction or design component can be reused

Metrics for OO Design-II

- Cohesion
  - The degree to which all operations working together to achieve a single, well-defined purpose
- Primitiveness
  - Applied to both operations and classes, the degree to which an operation is atomic
- Simplicity
  - The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose
- Validity
  - Measures the likelihood that a change will occur

Distinguishing Characteristics

Berard [Ber89] 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 facets

Class-Oriented Metrics

**Proposed by Chidamber and Kemerer [Chii94]:**

- weighted methods per class
- depth of the inheritance tree
- number of children
- coupling between object classes
- response for a class
- lack of cohesion in methods

Class-Oriented Metrics

**Proposed by Lorenz and Kidd [Lor94]:**

- class size
- number of operations overridden by a subclass
- number of operations added by a subclass
- specialization index

Class-Oriented Metrics

**The MOOD Metrics Suite [Har98b]:**

- Method inheritance factor
- Coupling factor
- Polymorphism factor

Operation-Oriented Metrics

**Proposed by Lorenz and Kidd [Lor94]:**

- average operation size
- operation complexity
- average number of parameters per operation

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.

Design Metrics for WebApps

- Does the user interface promote usability?
- Are the aesthetics of the WebApp appropriate for the application domain and pleasing to the user?
- Is the content designed in a manner that imparts the most information with the least effort?
- Is navigation efficient and straightforward?
- Has the WebApp architecture been designed to accommodate the special goals and objectives of WebApp users, the structure of content and functionality, and the flow of navigation required to use the system effectively?
- Are components designed in a manner that reduces procedural complexity and enhances the correctness, reliability and performance?

Code Metrics

- Halstead's Software Science: a comprehensive collection of metrics all predicated on the number (count and occurrence) of operators and operands within a component or program.
  - It should be noted that Halstead's "laws" have generated substantial controversy, and many believe that the underlying theory has flaws. However, experimental verification for selected programming languages has been performed (e.g. [FEL89]).

Metrics for Testing

- Testing effort can also be estimated using metrics derived from Halstead measures.
- Binder [Bin94] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system.
  - 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 (DT).

Maintenance Metrics

- IEEE Std. 982.1-1988 [IEEE84] suggests a software maturity index (SMI) that provides an indication of the stability of a software product (based on changes that occur for each release of the product). The following information is determined:
  - $M_0$: the number of modules in the current release.
  - $P_1$: the number of modules in the current release that have been changed.
  - $P_2$: the number of modules in the current release that have been added.
  - $P_3$: the number of modules from the preceding release that were deleted in the current release.
  - The software maturity index is computed in the following manner:
  $$SMI = (M_0 - P_1 - P_2 + P_3)/M_0$$
  - As SMI approaches 1.0, the product begins to stabilize.