COURSE NAME

SOFTWARE ENGINEERING

CSC 3114

(UNDERGRADUATE)

CHAPTER 9

SOFTWARE DESIGN

M. MAHMUDUL HASAN
ASSISTANT PROFESSOR, CS, AIUB
http://www.dit.hua.gr/~m.hasan

LinkedIn

DESIGN

Mitch Kapor, the creator of Lotus 1-2-3, presented a "software design manifesto" in *Dr. Dobbs Journal.* He said:

- Good software design should exhibit:
  - *Firmness:* A program should not have any bugs that inhibit its function
  - *Commodity:* A program should be suitable for the purposes for which it was intended
  - *Delight:* The experience of using the program should be pleasurable one

MODULARITY

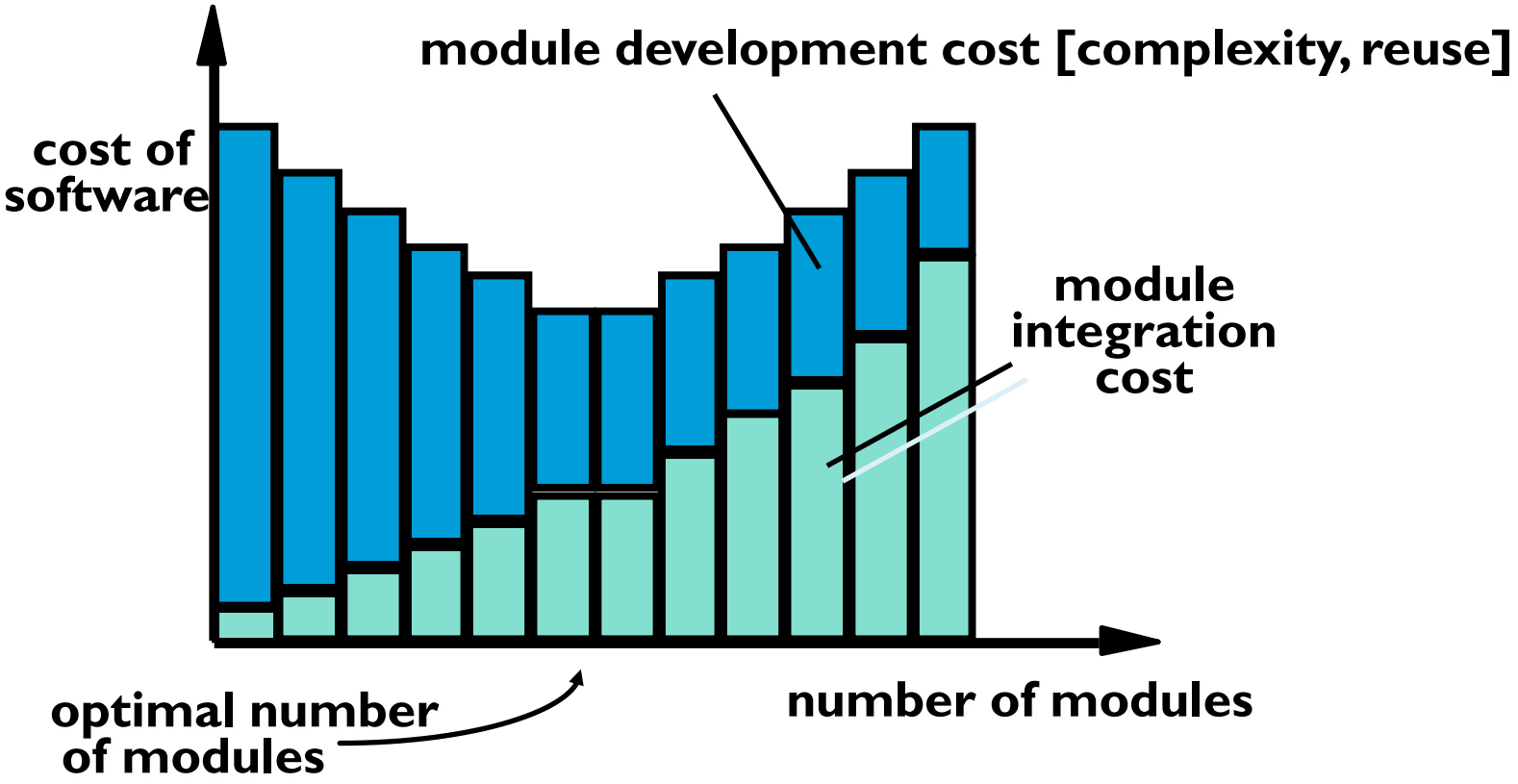
- Modularity is an attribute of software that allows a program to be intellectually manageable into distinct logical parts
- Modularity is the degree to which a system's components are logically separated into distinct parts called module and recombined again
- Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer. The number of control paths, span of reference, number of variables, and overall complexity would make understanding close to impossible.
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the complexity and cost required to build the software.

FUNCTIONAL INDEPENDENCE

- Cohesion is an indication of the relative functional strength of a module. A cohesive module performs a single task, requiring little interaction with other components in other parts of a program. Stated simply, a cohesive module should (ideally) do just one thing.
- Coupling is an indication of the relative interdependence among modules. Coupling depends on the interface complexity between modules, the point at which entry or reference is made to a module, and what data pass across the interface.
- Aspect is a representation of a cross-cutting concern. Consider two requirements, *A* and *B*. Requirement *A* *crosscuts* requirement *B* "if a software decomposition [refinement] has been chosen in which *B* cannot be satisfied without taking *A* into account.
- Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code [design] yet improves its internal structure (sort algorithm)

MODULARITY : TRADE-OFFS

What is the "right" number of modules for a specific software design?



USER INTERFACE DESIGN

- Easy to learn?
- Easy to use?
- Easy to understand?

Typical Design Errors

- **lack of consistency**
- **too much memorization**
- **no guidance / help**
- **no context sensitivity**
- **Obscure/ unfriendly**

GOLDEN RULE - PLACE THE USER IN CONTROL

- Define interaction modes in a way that does not force a user into unnecessary or undesired actions - The user should always be able to enter and exit the mode with little or no effort.
- Provide for flexible interaction (color, font, language, etc.) - Because different users have different interaction preferences, choices should be provided by using keyboard commands, mouse movements, digitizer pen or voice recognition commands.
- Allow user interaction to be interruptible and undoable - A user should be able to interrupt a sequence of actions to do something else without losing the work that has been done. The user should always be able to "undo" any action.
- Streamline interaction as skill levels advance and allow the interaction to be customized - Allow to design a macro if the user is to perform the same sequence of actions repeatedly

GOLDEN RULE - PLACE THE USER IN CONTROL

- Hide technical internals from the casual user - The user interface should move the user into the virtual world of the application. A user should never be required to type O/S commands from within application software.
- Design for direct interaction with objects that appear on the screen - The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing (progress bar).

GOLDEN RULE – REDUCE USER'S MEMORY LOAD

- Reduce demand on short-term memory (navigation) - Provide visual cues that enable a user to recognize past actions, rather than having to recall them
- Establish meaningful defaults - A user should be able to specify individual preferences; however, a reset option should be available to enable the redefinition of original default values (balance 0.00).
- Define shortcuts that are intuitive (intuitive - having the ability to understand or know something without any direct evidence or reasoning process) - "Alt-P to print"
- The visual layout of the interface should be based on a real world metaphor - Enable the user to rely on well-understood visual cues, (Print Symbol.)
- Disclose information in a progressive fashion - The interface should be organized hierarchically. The information should be presented at a high level of abstraction.

GOLDEN RULE – MAKE THE INTERFACE CONSISTENT

- Allow the user to put the current task into a meaningful context - The user should be able to determine where he has come from and what alternatives exist for a transition to a new task.
- Maintain consistency across a family of applications - "MS Office Suite"
- If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so - Once a particular interactive sequence has become a de-facto standard (Alt-S ➔ save file), the user expects this in every application she encounters.

INTERFACE ANALYSIS

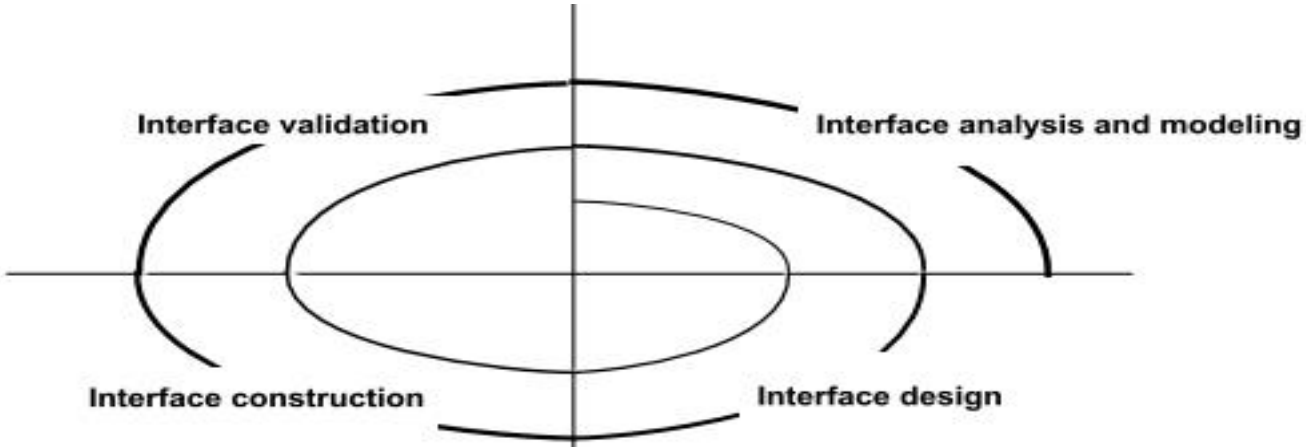
Interface analysis means understanding :

(1) the people (end-users) who will interact with the system through the interface

(2) the tasks that end-users must perform to do their work

(3) the content that is presented as part of the interface

(4) the environment in which these tasks will be conducted (e.g. embedded system)



User Interface Design Process

USER ANALYSIS

- Are users trained professionals, technician, official, or manufacturing workers?
- What level of formal education does the average user have?
- Are the users capable of learning from written materials or have they expressed a desire for classroom training?
- Are users expert typists or keyboard phobic?
- What is the gender and age range of the user community?
- How are users compensated for the work they perform? Do users work normal office hours or do they work until the job is done? (banking software)
- Is the software to be an integral part of the work users do or will it be used only occasionally?
- What is the primary spoken language among users?
- What are the consequences if a user makes a mistake using the system?
- Are users experts in the subject matter that is addressed by the system?
- Do users want to know about the technology the sits behind the interface?

TASK ANALYSIS AND MODELLING

- Answers the following questions …
  - What work will the user perform in specific circumstances?
  - What tasks and subtasks will be performed as the user does the work?
  - What specific problem domain objects will the user manipulate as work is performed?
  - What is the sequence of work tasks (hierarchy)—the workflow?
- Use-cases define basic interaction
- Object elaboration identifies interface objects (classes)
- Task elaboration refines interactive tasks
- Workflow analysis defines how a work process is completed when several people (and roles) are involved (swimlane diagram)

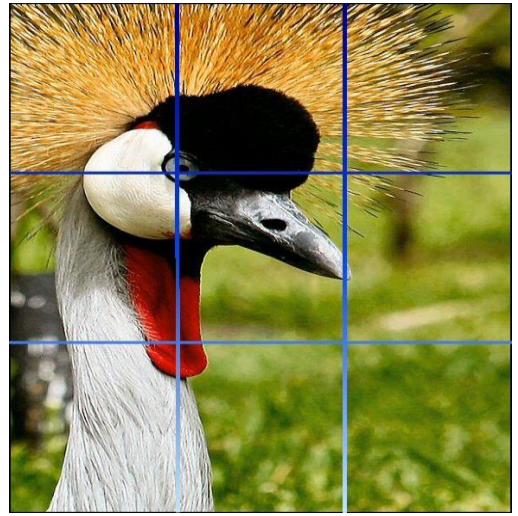
ANALYSIS OF DISPLAY CONTENT

- Are different types of data assigned to consistent geographic locations on the screen? (e.g., photos always appear in the upper right hand corner)
- If a large report is to be presented, how should it be partitioned for ease of understanding?
- Will mechanisms be available for moving directly to summary information for large collections of data?
- Will graphical output be scaled to fit within the bounds of the display device? (e.g. smart phones)
- How will color to be used to enhance understanding? (errors are in red color)
- How will error messages and warning be presented to the user? (dialogue box, or text message)

Remember that the font carry a message to!

PEACE

WAR



INTERFACE DESIGN PRINCIPLES-I

- Anticipation — WebApp should be designed so that it anticipates the use's next move
- Communication —The interface should communicate the status of any activity initiated by the user (progress bar)
- Consistency —The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, layout) consistent in each webpage.
- Controlled autonomy —The interface should facilitate user movement throughout the WebApp, but it should do so in a manner that enforces navigation conventions that have been established for the application.
- Efficiency —The design of the WebApp and its interface should optimize the user's work efficiency, not the efficiency of the Web engineer who designs and builds it or the client-server environment that executes it.

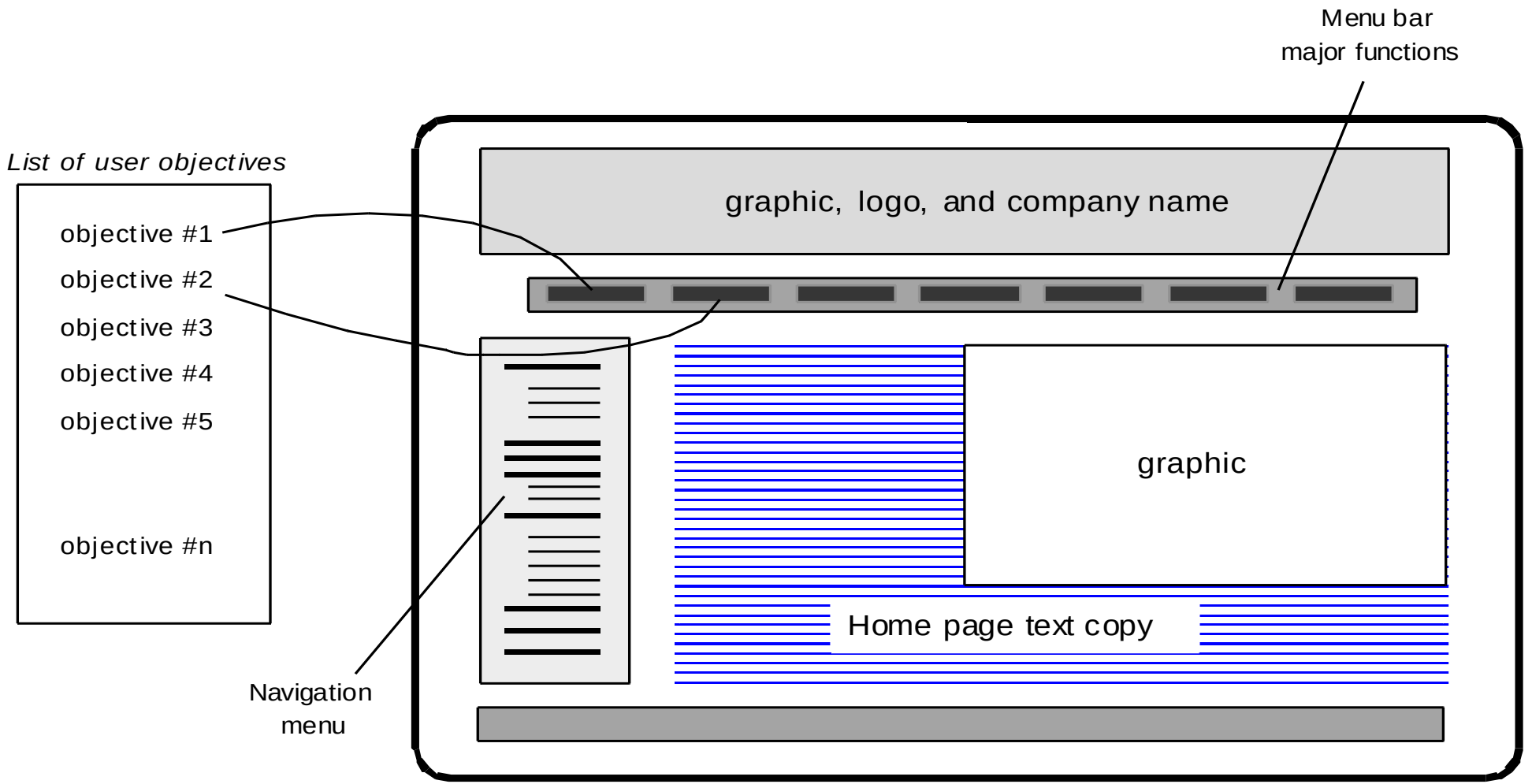
INTERFACE DESIGN PRINCIPLES-II

- Focus— WebApp interface (and the content it presents) should stay focused on the user task(s) at hand.
- Fitt's Law—"The time to acquire a target is a function of the distance to and size of the target."
- Human interface objects—A vast library of reusable human interface objects has been developed for WebApps (bootstarp).
- Latency reduction— WebApp should use multi-tasking in a way that lets the user proceed with work as if the operation has been completed.
- Learnability— WebApp interface should be designed to minimize learning time, and once learned, to minimize relearning required when the WebApp is revisited.

INTERFACE DESIGN PRINCIPLES-III

- Maintain work product integrity—A work product (e.g., a form completed by the user, a user specified list) must be automatically saved so that it will not be lost if an error occurs.
- Readability—All information presented through the interface should be readable.
- Track state—When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.
- Visible navigation—A well-designed WebApp interface provides "the illusion that users are in the same place, with the work brought to them." (rather than SCROOLLING)
- Don't be afraid of white space
- Emphasize content rather style
- Organize layout elements from top-left to bottom right

MAPPING USER OBJECTIVES (WAREFRAMMING)



REFERENCES

- R.S. Pressman & Associates, Inc. (2010). *Software Engineering: A Practitioner's Approach*.
- Kelly, J. C., Sherif, J. S., & Hops, J. (1992). An analysis of defect densities found during software inspections. *Journal of Systems and Software*, *17*(2), 111-117.
- Bhandari, I., Halliday, M. J., Chaar, J., Chillarege, R., Jones, K., Atkinson, J. S., & Yonezawa, M. (1994). In-process improvement through defect data interpretation. *IBM Systems Journal*, *33*(1), 182-214.

COURSE NAME

SOFTWARE ENGINEERING

CSC 3114

(UNDERGRADUATE)

CHAPTER 10

SOFTWARE TESTING

M. MAHMUDUL HASAN
ASSISTANT PROFESSOR, CS, AIUB
http://www.dit.hua.gr/~m.hasan

SOFTWARE TESTING

- Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user
- Software testability is simply how easily [a computer program] can be tested

Testing Shows

- Error
- Requirements Conformance
- Performance
- An indication of quality

WHO TESTS THE SOFTWARE?

Developer

- Understands the system but, will test "gently“ and, is driven by "delivery“
- Experiencing the software operation (known to the developer)

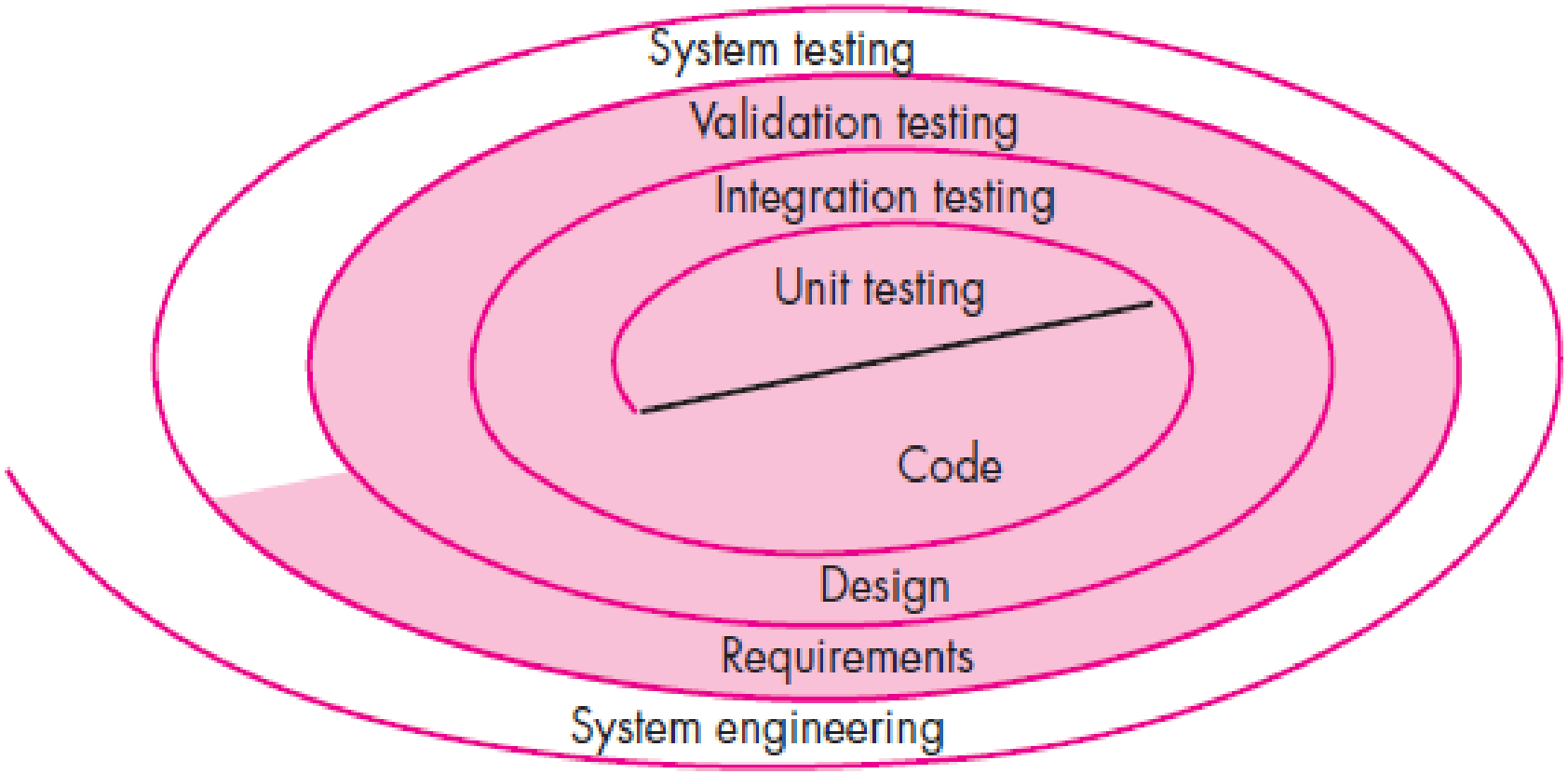
Independent tester

- Must learn about the system, but, will attempt to break it and, is driven by “quality”
- Exploring the software operation (unknown to the tester)

V & V

- *Validation* refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.
- *Verification* refers to the set of tasks that ensure that software correctly implements a specific function/process.
- Boehm states this another way:
  - *Validation:* "Are we building the right product?“
  - *Verification:* "Are we building the product right?"

TESTING STRATEGY



TESTING STRATEGY

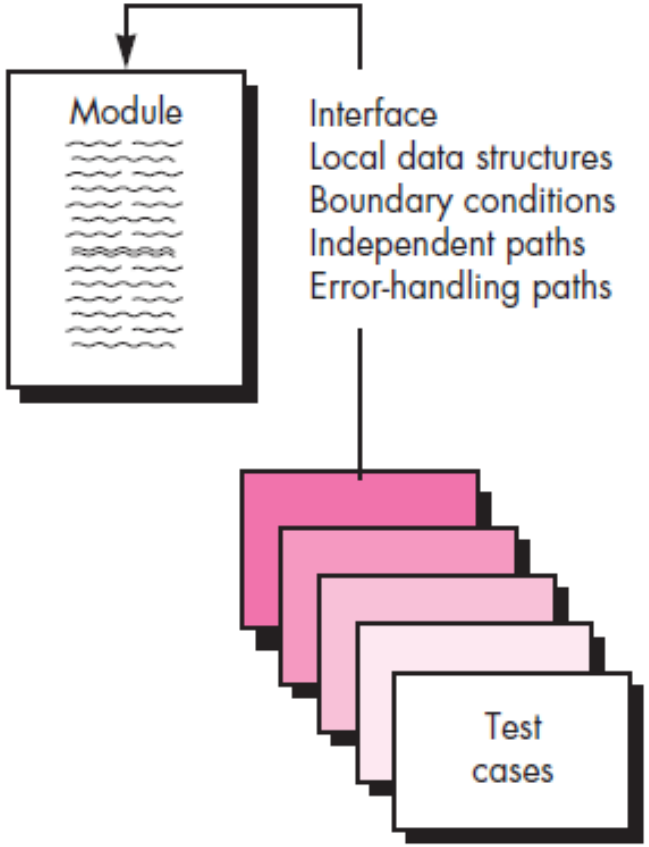
- We begin by 'testing-in-the-small' and move toward 'testing-in-the-large'
- For conventional software
  - The module (component) is our initial focus
  - Integration of modules follows
- For OO software
  - Our focus when "testing in the small" changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration

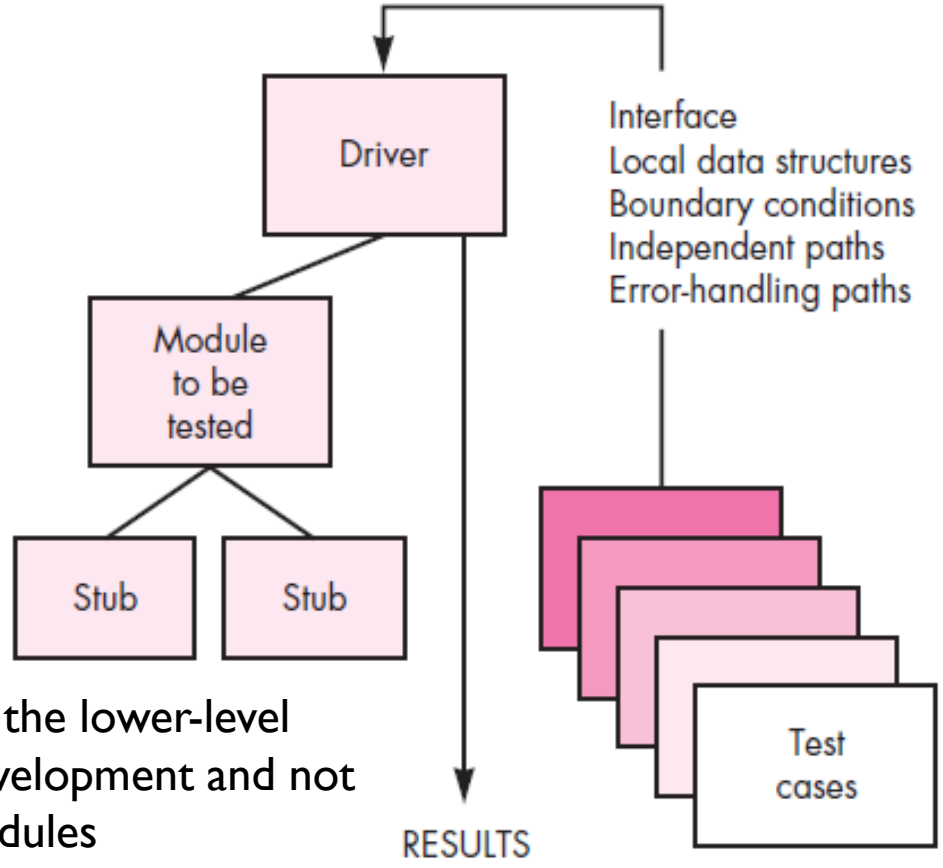
TESTING STRATEGIC ISSUES

- Specify product requirements in a quantifiable manner long before testing commences
- State testing objectives explicitly
- Understand the users of the software and develop a profile for each user category
- Develop a testing plan that emphasizes "rapid cycle testing"
- Build "robust" software that is designed to test itself
- Use effective technical reviews as a filter prior to testing; many errors will be eliminated before testing begins
- Conduct technical reviews to assess the test strategy and test cases themselves
- Develop a continuous improvement approach for the testing process

UNIT TESTING

Tests a small software unit at a time, which is typically performed by the individual programmer who implemented the unit prior to Integration testing

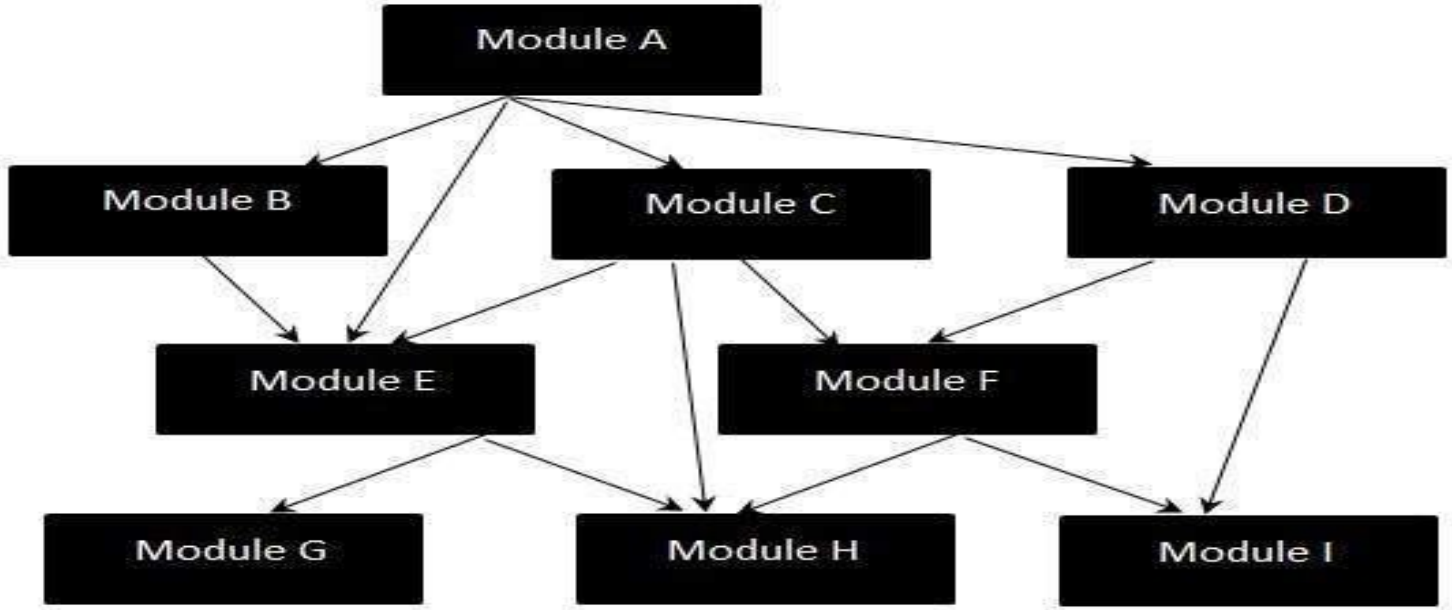




Stub: it is the behavior of the lower-level modules that are under development and not yet integrated to other modules

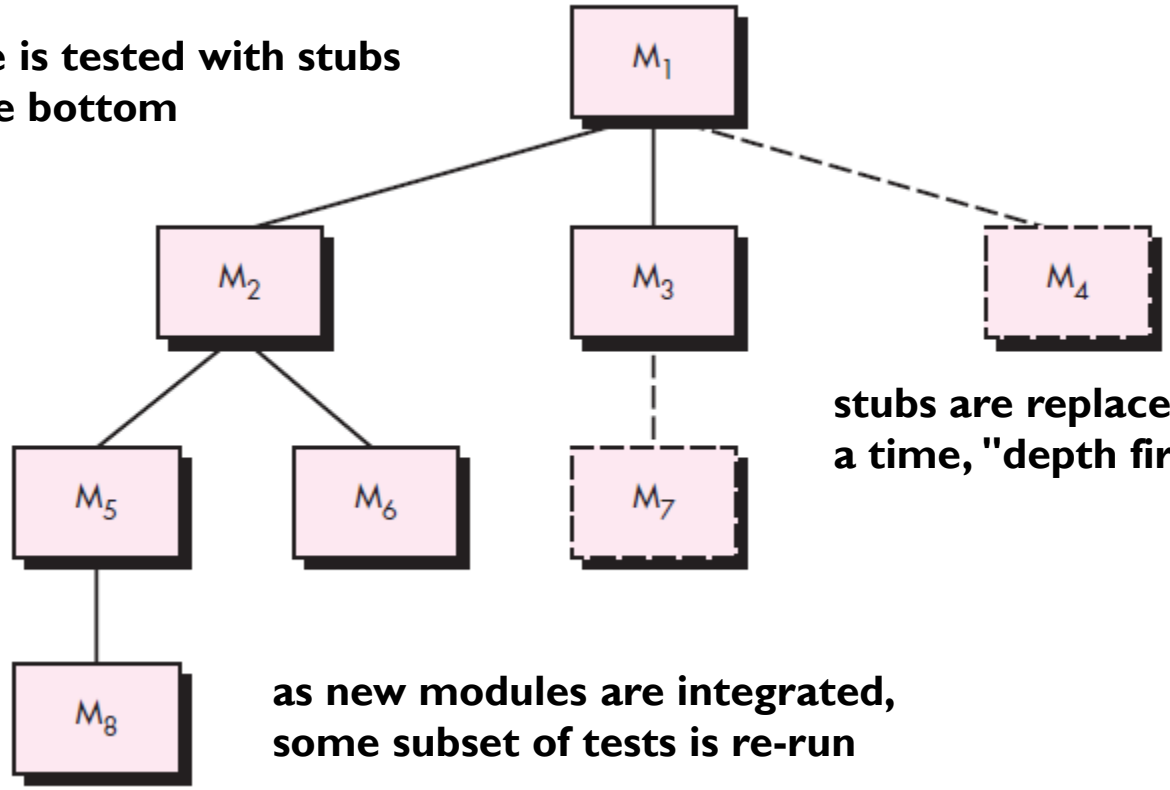
INTEGRATION TESTING STRATEGIES

- System integration testing (SIT) is a systematic technique for assembling a software system while conducting tests to uncover errors associated with interfacing the modules
- **the "big bang" approach:** Big Bang Integration Testing is an integration testing strategy where all units are linked at once, resulting in a complete system.



TOP-DOWN INTEGRATION

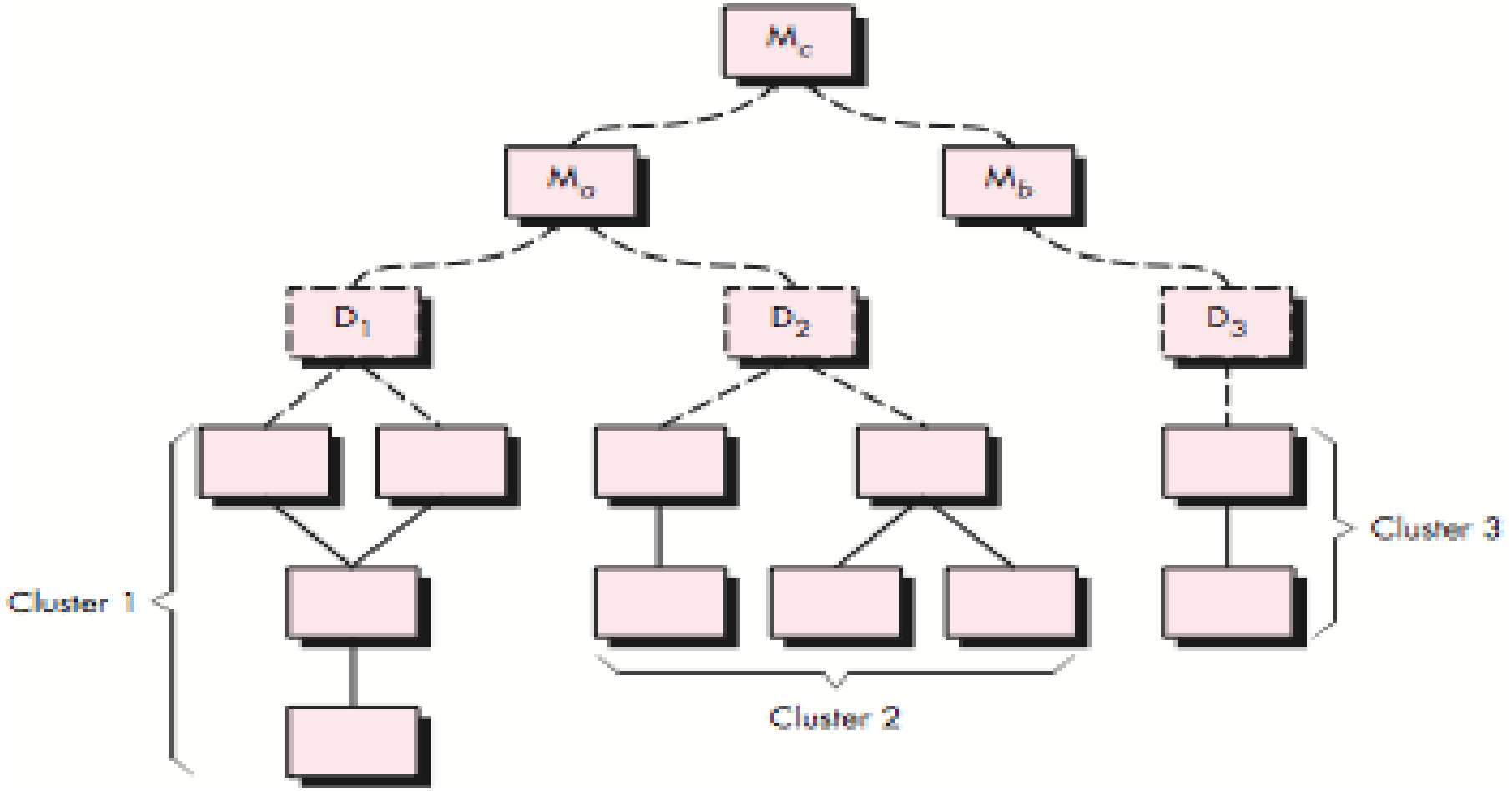
top module is tested with stubs
down to the bottom



stubs are replaced one at
a time, "depth first"

as new modules are integrated,
some subset of tests is re-run

BOTTOM-UP INTEGRATION



REGRESSION TESTING

- Regression testing is the re-execution of some subset of tests that have already been conducted to ensure that changes have not propagated unintended side effects
- Whenever software is corrected, some aspect of the software configuration (the program, its documentation, or the data that support it) is changed.
- Regression testing helps to ensure that changes (due to testing or for other reasons) do not introduce unintended behavior or additional errors.
- Regression testing may be conducted manually, by re- executing a subset of all test cases or using automated capture/playback tools.

SMOKE TESTING

Smoke testing steps:

- Software components that have been translated into code are integrated into a "**daily build**"
  - A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
- A series of tests is designed to expose errors that will keep the build from properly performing its function.
  - The intent should be to uncover "**show stopper**" errors that have the highest likelihood of throwing the software project behind schedule.
- The build is integrated with other builds and the entire product (in its current form is smoke tested daily.
  - The integration approach may be top down or bottom up.

OBJECT-ORIENTED TESTING

- Class testing is the equivalent of unit testing
  - Operations within the class are tested
  - The state behavior of the class is examined

- Integration applied three different strategies
  - Thread-based testing—integrates the set of classes required to respond one input or event
  - Use-based 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

HIGHER ORDER TESTING

- **System testing:** focus is on system integration (e.g. hardware integration, OS compatibility)
- **Alpha/Beta testing:** **Alpha testing** is simulated or actual operational **testing** by potential users or an independent **test** team at the developers' site. **Alpha testing** is often employed for off-the-shelf software as a form of internal acceptance **testing**, before the software goes to **beta testing** by users
- **Recovery testing:** forces the software to fail in a variety of ways and verifies that recovery is properly performed
- **Security testing:** verifies that protection mechanisms built into a system will, in fact, protect it from improper penetration
- **Stress testing:** executes a system in a manner that demands resources in abnormal quantity, frequency, or volume
- **Performance Testing:** test the run-time performance of software within the context of an integrated system (e.g. time required to response a request, compliance with operational constraints)

DEBUGGING

- In many cases, the non-corresponding data are a symptom of an underlying cause as yet hidden error
- The debugging process attempts to match symptom with cause, thereby leading to error correction

- symptom may disappear when another problem is fixed
- cause may be due to a combination of non-errors
- cause may be due to a system or compiler error
- cause may be due to assumptions that everyone believes

DEBUGGING TECHNIQUES

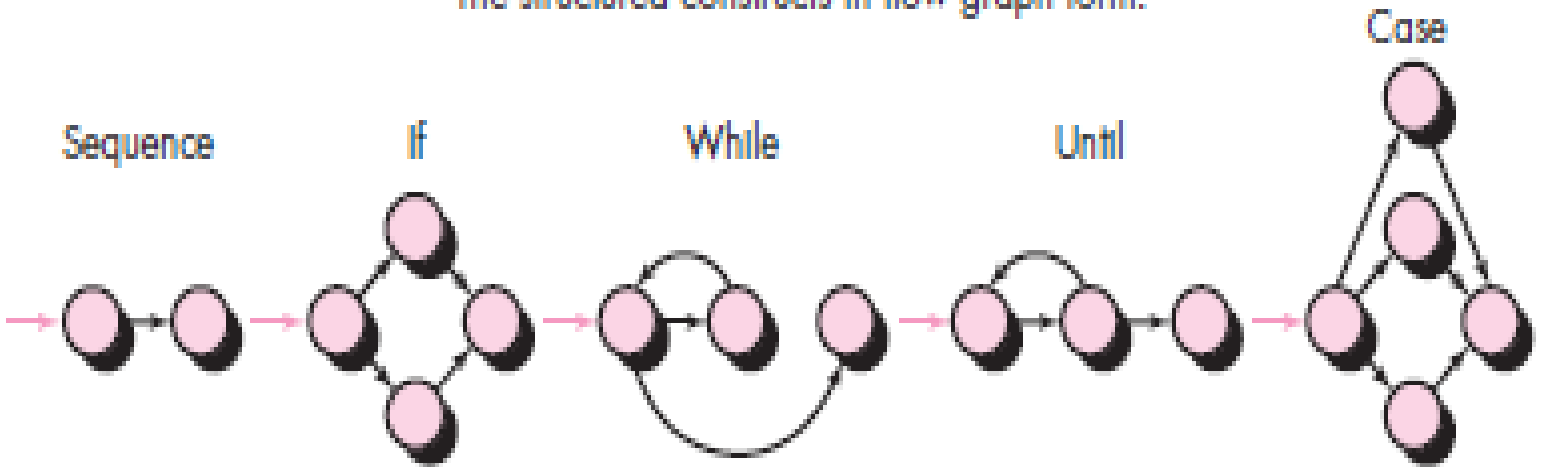
- **Brute force testing**
  - most common; but least efficient
  - memory dumps are taken, run-time traces are invoked, and the program is loaded with output statements **(Dynamic Testing)**
- **Backtracking**
  - common debugging approach that can be used successfully in small programs
  - source code is traced backward (manually) until the cause is found
- **Cause elimination**
  - a "cause hypothesis" is devised
  - if initial tests indicate that a particular cause hypothesis shows promise, data are refined in an attempt to isolate the bug (c/a-b where the possibility of a-b is zero)

BASIS-PATH TESTING

- The basis path method enables the test-case designer to derive a logical complexity measure of a procedural design and use this measure as a guide for defining a basis set of execution paths
- McCabe views a program as a directed graph in which lines of program statements are represented by nodes and the flow of control between the statements is represented by the edges

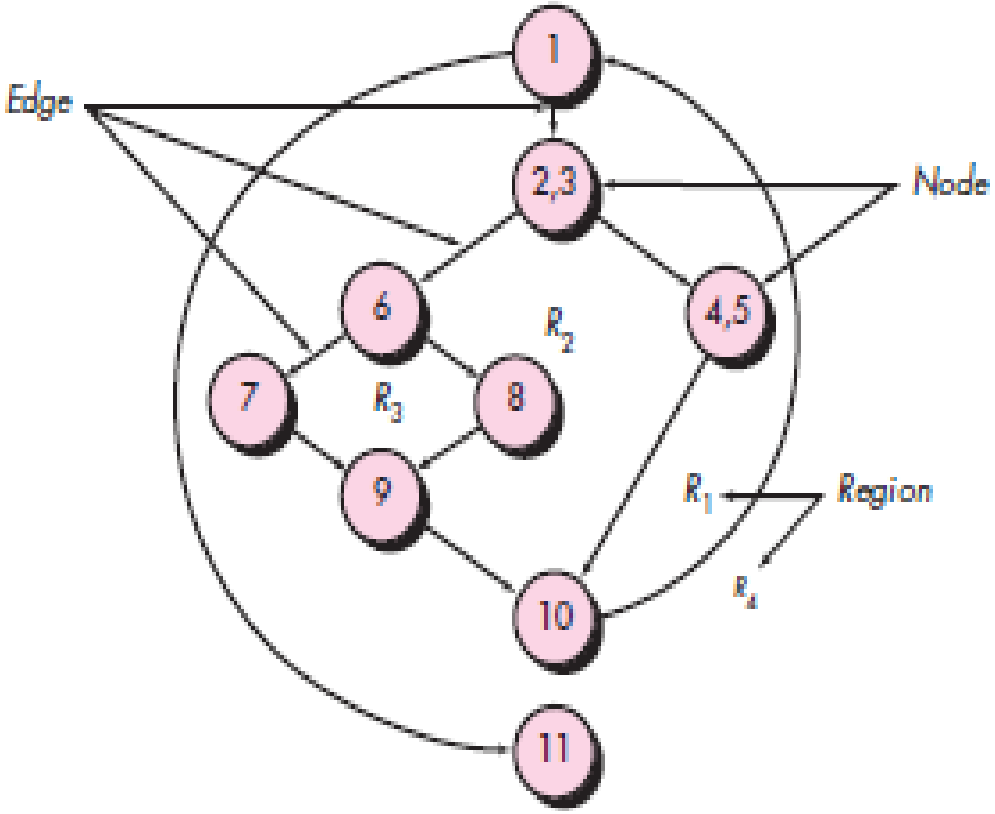
Flow Graph

The structured constructs in flow graph form:



Where each circle represents one or more nonbranching PDL or source code statements

INDEPENDENT PROGRAM PATHS



Path 1: 1-11

Path 2: 1-2-3-4-5-10-1-11

Path 3: 1-2-3-6-8-9-10-1-11

Path 4: 1-2-3-6-7-9-10-1-11

- Note that each new path introduces a new edge.
  The path 1-2-3-4-5-10-1-2-3-6-8-9-10-1-11 is not considered to be an independent path because it is simply a combination of already specified paths and does not traverse any new edges.
- How do you know how many paths to look for?
  The computation of **cyclomatic complexity** provides the answer

CYCLOMATIC COMPLEXITY

Cyclomatic complexity is a software metric that provides a quantitative measure of the logical complexity of a program. Complexity is computed in one of three ways:

1. The number of independent paths
2. The number of regions of the flow graph corresponds to the cyclomatic complexity.
   (in the previous example = 4)
3. Cyclomatic complexity *V*(*G*) for a flow graph *G* is defined as *V*(*G*) = *E* - *N* + 2
   (in the previous example 11 -9 + 2 = 4)
   - where *E* is the number of flow graph edges and *N* is the number of flow graph nodes.
4. Cyclomatic complexity *V*(*G*) for a flow graph *G* is also defined as *V*(*G*) = *P* + 1
   (in the previous example 3 + 1 = 4) [condition: 1; 2,3; 6]
   - where *P* is the number of predicate nodes (containing a condition) contained in the flow graph *G*

WHITE-BOX TESTING

Using white-box testing methods, you can derive test cases that

(1) guarantee that all independent paths within a module have been exercised at least once,

(2) exercise all logical decisions on their true and false sides,

(3) execute all loops at their boundaries and within their operational bounds, and

(4) exercise internal data structures to ensure their validity.

BLACK-BOX TESTING

- Focuses on the functional requirements of the software
- Black-box testing attempts to find errors in the following categories:

(1) incorrect or missing functions

(2) interface errors

(3) errors in external database access (accessibility)

(4) behavior or performance errors

(5) initialization and termination errors

COURSE NAME

SOFTWARE ENGINEERING

CSC 3114

(UNDERGRADUATE)

CHAPTER 11

SOFTWARE QUALITY ATTRIBUTES

M. MAHMUDUL HASAN
ASSISTANT PROFESSOR, CS, AIUB
http://www.dit.hua.gr/~m.hasan

OVERVIEW OF SOFTWARE QUALITY REQUIREMENTS

- Software success is more than just delivering the right functionality.
- Characteristics of software that fall into this category include how easy it is to use, how quickly it runs, how often it fails, and how it handles unexpected conditions.
- Such characteristics, collectively known as *software quality attributes* or *quality factors*, are part of the system's nonfunctional (also called non-behavioral) requirements.

OVERVIEW OF SOFTWARE QUALITY ATTRIBUTES

- Customers generally don't present their quality expectations explicitly.
- The trick is to **pin down** just what the users are thinking when they say the software must be user-friendly, fast, reliable, or robust.
- From a technical perspective, quality attributes drive significant architectural and design decisions.
- It's far more difficult and costly to re-architect a completed system to achieve essential quality goals than to design for them at the beginning.

AVAILABILITY

- Availability is a measure of the planned *up time* during which the system is actually available for use and fully operational.

$$\text{Availability equals} = \frac{\text{mean time to failure (MTTF) for the system}}{\text{MTTF + mean time to repair (MTTR) the system after a failure is encountered.}}$$

- Availability requirements become more complex and more important for Web sites or global applications with worldwide users.
- Example: *AV-1. The system shall be at least 99.5 percent available on weekdays between 6:00 a.m. and midnight local time, and at least 99.95 percent available on weekdays between 4:00 p.m. and 6:00 p.m. local time.*

PERFORMANCE

- Performance requirements define how well or how rapidly the system must perform specific functions.
  - speed (e.g. database response times)
  - throughput (transactions per second)
  - capacity (concurrent usage loads)
  - timing (hard real-time demands)
- Performance requirements should also address how the system's performance will degrade in an overloaded situation, such as when a 911 emergency telephone system is flooded with calls.
  - *PE-1. Every Web page shall download in 15 seconds or less over a 50 KBps modem connection.*
  - *PE-2. Authorization of an ATM withdrawal request shall not take more than 10 seconds.*

EFFICIENCY

- Efficiency is a measure of how well the system utilizes processor capacity, disk space, memory, or communication bandwidth (Davis 1993).
- Efficiency is related to performance, (response time) another class of nonfunctional requirement
- If a system consumes too much of the available resources, users will encounter degraded performance, a visible indication of inefficiency.
- Poor performance is an irritant to the user who is waiting for a database query to display results. But performance problems can also represent serious risks to safety, such as when a real-time process control system is overloaded.
- Consider **minimum hardware configurations** when defining efficiency, capacity, and performance goals.
- *Example: EF-1. At least 25 percent of the processor capacity and RAM available to the application shall be unused at the planned peak load conditions.*
- Typical users won't state efficiency requirements in such technical terms. The analyst must ask the questions that will surface user expectations regarding issues such as acceptable performance degradation, demand spikes, and anticipated growth.

FLEXIBILITY

- Flexibility measures how easy it is to add new capabilities to the product
- Also known as *extensibility*, *augmentability*, *extendability*, and *expandability*
- If developers anticipate making many enhancements, they can choose design approaches that maximize the software's flexibility. This attribute is essential for products that are developed in an incremental or iterative fashion through a series of successive releases or by evolutionary prototyping.
- *Example: FL-1. A maintenance programmer who has at least six months of experience supporting this product shall be able to make a new copy output available to the product, including code modifications and testing, with no more than one hour of labor.*

INTEGRITY

- Integrity—which encompasses security, deals with blocking unauthorized access to system functions, preventing information loss, ensuring that the software is protected from virus infection, and protecting the privacy and safety of data entered into the system. Integrity is a major issue with Internet software.
- Users of e-commerce systems want their credit card information to be secure
- Integrity requirements have no tolerance for error
- State integrity requirements in unambiguous terms: user identity verification, user privilege levels, access restrictions, or the precise data that must be protected
  - *Example: IN-1. Only users who have Auditor access privileges shall be able to view customer transaction histories.*

INTEROPERABILITY

- Interoperability indicates how easily the system can exchange data or services with other systems
- To assess interoperability, you need to know which other applications the users will employ in conjunction with your product and what data they expect to exchange.
- For example, SIM registration with NID, provide one stop service in supermarket
- *Example: I0-1. The Chemical Tracking System shall be able to import any valid chemical structure from the ChemiDraw (version 2.3 or earlier) and Chem-Struct (version 5 or earlier) tools.*

RELIABILITY

- The probability of the software executing without failure for a specific period of time is known as reliability.
- Ways to measure software reliability include the percentage of operations that are completed correctly and the average length of time the system runs before failing.
- Establish quantitative reliability requirements based on how severe the impact would be if a failure occurred and whether the cost of maximizing reliability is justifiable.
- Systems that require high reliability should also be designed for high testability to make it easier to find defects that could compromise reliability.
- Example: *RE-1. No more than five experimental runs out of 1000 can be lost because of software failures*

ROBUSTNESS

- Robustness is the degree to which a system continues to function properly when confronted with invalid inputs, defects in connected software or hardware components, or unexpected operating conditions
- Robust software recovers gracefully from problem situations and is forgiving of user mistakes
- When eliciting robustness requirements, ask users about error conditions the system might encounter and how the system should react
- Sometimes called *fault tolerance*
- *Example: RO-1. If the editor fails before the user saves the file, the editor shall be able to recover all changes made in the file being edited up to one minute prior to the failure the next time the same user starts the program.*

USABILITY

- Usability measures the effort required to prepare input for, operate, and interpret the output of the product
- Also referred to as *ease of use* and *human engineering*, usability addresses many factors that constitute what users often describe as *user-friendliness*
- Usability also encompasses how easy it is for new or infrequent users to learn to use the product. Ease-of-learning goals can be quantified and measured (e.g. language option)
- *Example: US-1. A trained user shall be able to submit a complete request for a chemical selected from a vendor catalog in an average of four and a maximum of six minutes.*

MAINTAINABILITY

- Maintainability indicates how easy it is to correct a defect or modify the software
- Maintainability depends on how easily the software can be understood, changed, and tested
- It is closely related to flexibility and testability
- High maintainability is critical for products that will undergo frequent revision and for products that are being built quickly
- Maintainability can be measured in terms of the average time required to fix a problem and the percentage of fixes that are made correctly
- *Example: MA-1. A maintenance programmer shall be able to modify existing reports to conform to revised chemical-reporting regulations from the federal government with 20 labor hours or less of development effort.*

REUSABILITY

- Reusability indicates the relative effort involved to convert a software component for use in other applications
- Developing reusable software costs considerably more than creating a component that you intend to use in just one application
- Reusable software must be modular, well documented, independent of a specific application and operating environment, and somewhat generic in capability
- Reusability goals are difficult to quantify
- *Example: RU-1. The chemical structure input functions shall be designed to be reusable at the object code level in other applications that use the international standard chemical structure representations.*

TESTABILITY

- Testability refers to the ease with which software components or the integrated product can be tested to look for defects
- Also known as *verifiability*
- Designing for testability is critical if the product has complex algorithms and logic, or if it contains indirect (ambiguous) functionality interrelationships
- Testability is also important if the product will be modified often because it will undergo frequent regression testing to determine whether the changes damaged any existing functionality
- Example: *TE-1. The maximum cyclomatic complexity* of a module shall not exceed 20.*
- **Cyclomatic complexity* is a measure of the number of logic branches in a source code module

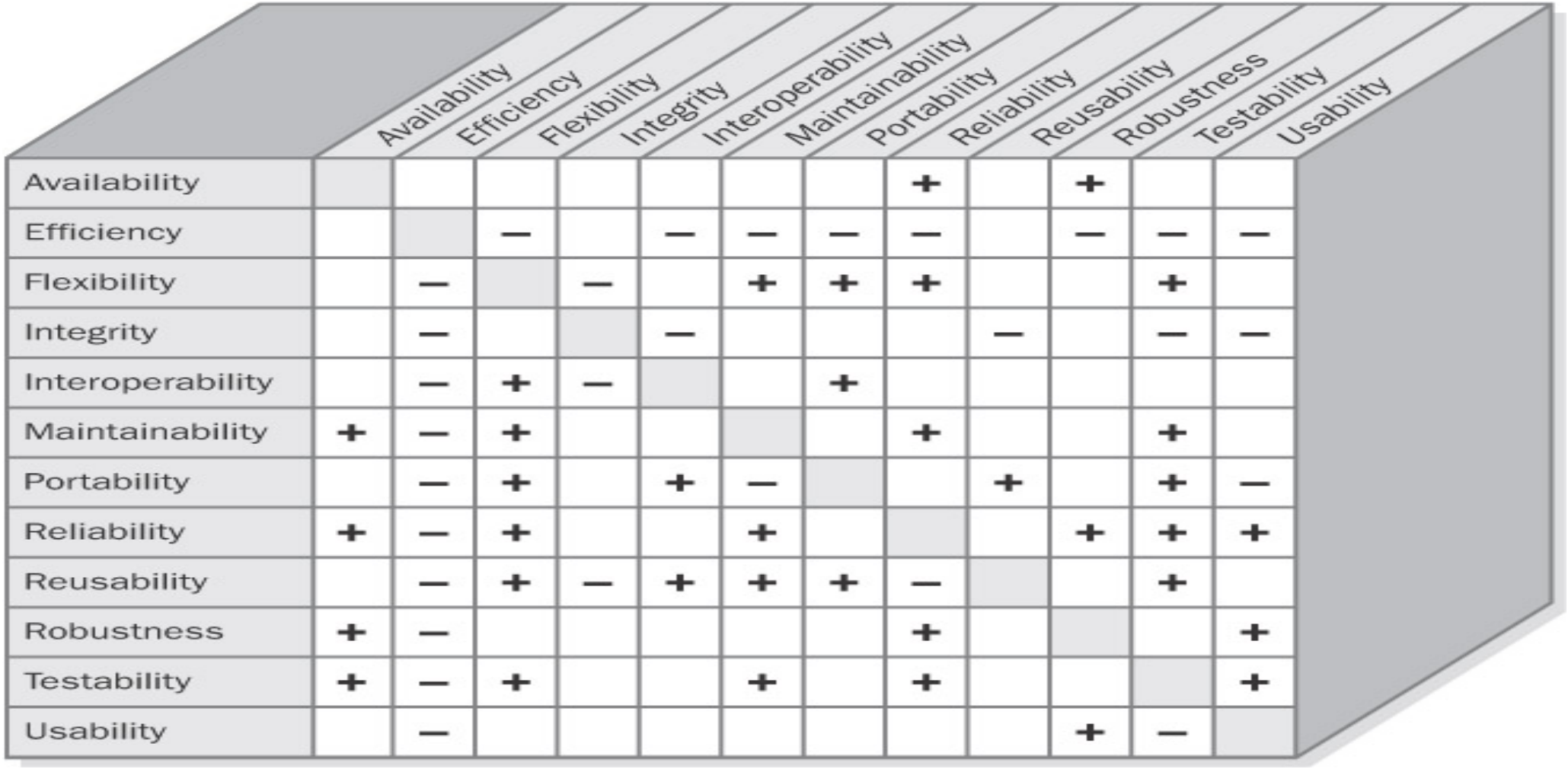
QUALITY ATTRIBUTES

- Different parts of the product need different combinations of quality attributes
- Efficiency might be critical for certain components, while usability is paramount for others
- For different types of systems different types of quality attributes might be critical:
- Embedded systems: efficiency, reliability, safety, installability, serviceability
- Internet and mainframe applications: availability, integrity, maintainability, and scalability
- Desktop systems: interoperability and usability

QUALITY ATTRIBUTES TO WHO?

Important Primarily to Users	Important Primarily to Developers
Availability	Maintainability
Efficiency	Portability (s/w runs on multiple platforms)
Flexibility	Reusability
Integrity	Testability
Interoperability	
Reliability	
Robustness	
Usability	

ATTRIBUTE TRADE-OFFS



	Availability	Efficiency	Flexibility	Integrity	Interoperability	Maintainability	Portability	Reliability	Reusability	Robustness	Testability	Usability
Availability							+		+			
Efficiency			–		–	–	–	–		–	–	–
Flexibility		–		–		+	+	+			+	
Integrity		–			–				–		–	–
Interoperability		–	+	–			+					
Maintainability	+	–	+					+			+	
Portability		–	+		+	–			+		+	–
Reliability	+	–	+			+				+	+	+
Reusability		–	+	–	+	+	+	–			+	
Robustness	+	–						+				+
Testability	+	–	+			+		+				+
Usability		–								+	–	

ATTRIBUTE TRADE-OFFS

- A plus sign in a cell indicates that increasing the attribute in the corresponding row has a positive effect on the attribute in the column
- A minus sign in a cell means that increasing the attribute in that row adversely affects the attribute in the column
- A blank cell indicates that the attribute in the row has little impact on the attribute in the column
- Design approaches that increase a software component's portability also make the software more flexible, easier to connect to other software components, easier to reuse, and easier to test
- Systems that optimize ease of use or that are designed to be flexible, reusable, and interoperable with other software or hardware components often incur a performance penalty

ATTRIBUTE MATRIX

- The matrix isn't **symmetrical** because the effect that increasing attribute A has on attribute B isn't necessarily the same as the effect that increasing B will have on A
- Example shows that designing the system to increase **efficiency** doesn't necessarily have any effect on **integrity**. However, increasing integrity likely will hurt efficiency because the system must go through more layers of user authentications, encryption, virus scanning, and data checkpointing.
- To reach the optimum balance of product characteristics, you must identify, specify, and prioritize the relevant quality attributes during requirements elicitation
- Using the matrix will avoid making commitments to conflicting goals. For example, Don't expect to maximize usability if the software must run on multiple platforms (portability)
- Defining conflicting requirements makes it impossible for the developers to fully satisfy requirements

IMPLEMENTING NON-FUNCTIONAL REQUIREMENTS

- Although quality attributes are nonfunctional requirements, they can lead to derived functional requirements, design guidelines, or other types of technical information that will produce the desired quality characteristics
- *Example: A medical device with strict availability requirements might include a backup battery power supply (architecture) and a functional requirement to visibly or audibly indicate that the product is operating on battery power.*

Quality Attribute Types	**Likely Technical Information Category**
Integrity, interoperability, robustness, usability, safety	Functional requirement
Availability, efficiency, flexibility, performance, reliability	System architecture
Interoperability, usability	Design constraint
Flexibility, maintainability, portability, reliability, reusability, testability, usability	Design guideline
Portability	Implementation constraint

MCCALL'S TRIANGLE OF QUALITY