Design principles

1. Divide and conquer
2. Increase cohesion where possible
3. Reduce coupling where possible
4. Keep the level of abstraction as high as possible
5. Increase reusability where possible
6. Reuse existing designs and code where possible
7. Design for flexibility
8. Anticipate obsolescence
9. Design for portability
10. Design for testability
11. Design defensively

Cost estimation principles

1. Divide and conquer
2. Include all activities when making estimates
3. Base your estimates on past experience combined with what you can observe of the current project
4. Be sure to account for differences when extrapolating from other projects
5. Anticipate the worst case and plan for contingencies
6. Combine multiple independent estimates
7. Revise and refine estimates as work progresses

Source: OOSE, Lethbridge & Laganiere, McGraw-Hill, 2005

Software Testing

Testing	Debugging
Defn: shows that the bug exists; establishes the existence of defects	**Defn**: find the error or misconception that led to the program's failure and to define the program changes that correct the error; locates and corrects the defects (brute force, backtracking, cause elimination)
1. testing starts with known conditions, uses predefined procedures, and has predictable outcomes.	Debugging starts from possibly unknown initial conditions and the end cannot be predicted, except statistically.
2. testing should be planned, designed, and scheduled beforehand.	The procedures for, and duration of debugging, cannot be constrained.
3. testing is a demonstration of error or apparent correctness.	Debugging is a deductive process.
4. testing proves programmer's failure.	Debugging is the programmer's vindication.
5. testing should strive to be predictable, dull, constrained, rigid, and inhuman.	Debugging demands intuitive leaps, conjectures, experimentation, intelligence, and freedom.
6. testing can be done by an outsider.	debugging must be done by an insider.
7. testing, to a large extent, can be designed and accomplished in ignorance of the design.	Debugging is impossible without detailed design knowledge.
8. test execution and design can be automated.	Automated debugging is still a dream.

Source: Boris Beizer, *Software Testing Techniques*, Van Nostrand Reinhold, 1990.

An ***oracle*** is any program, process, or body of data that specifies the expected outcome of a set of tests as applied to a tested object.

characteristics of good testing

- reusability
- operability
- controllability
- decomposability
- simplicity
- stability
- understandability

rationale

- trace to requirements
- planned
- 80-20 rule (Pareto Principle)
- small to large
- exhaustive test is not possible

testing process

1. unit testing, (module testing)
2. integration testing, (system testing)
3. acceptance testing

code (unit testing), design (integration), requirements (validation), system engineering (system test), regression test (condition--add new features) to prevent new defects resulting from fixing earlier ones

- functional (blackbox) testing -> specification
  does not take into account how the performance is derived but merely stresses the end result
- structural (whitebox) testing -> implementation
  relates to tests from which the test data are derived from examination of the internal structure (e.g., program logic); does not dwells on behavior or performance (what the system is doing), but rather on the validity of the implementation

strategies

- top-down: stubs (simple components having the same interface as the module), structural, errors in coding may be detected at an early stage and inexpensive to correct (fixing, redesign, and re-implementation costs); has a limited working system available at an early stage in the development (psychological boost); validation can be done early too; but difficult to devise complex stubs; output may be hard to observe because higher level components, in general, don't generate output/results -> may have to create artificial environment to simulate execution
- bottom-up: requires test drivers to exercise the lower level components by simulating the components' environment; critical for reuse lower level components
- combined top-down/bottom-up: all parts must be available before begin -> architectural faults will not be discovered until much of the system has been tested
- thread: system reaction to classes of event in the form of execution sequence (concurrent)
- stress: tests the failure behavior of the system (where overloading does not cause unacceptable loss of data of service); may cause defects to come to light which would not normally manifest themselves