CS 575
Software Testing and Analysis

Challenges, basic principles and terminology

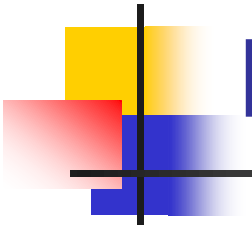
ÖZYEĞİN UNIVERSITY

(c) Slides adopted from the slides of M. Pezze & M. Young and P. Amman & J. Offut

Software Testing to Increase Reliability

- Widely utilized technique
  - Much common relative to fault prevention and fault tolerance
- Getting more important

  and challenging as well..



Need for Software Testing

- Accounts for at least half of the development budget
- Principle post-design activity in practice
- Restricting early testing usually increases cost
- Extensive hardware-software integration requires even more testing

Peculiarities of Software

- Challenging characteristics of software
  - Many different quality requirements
  - Evolving (and deteriorating) structure
  - Inherent non-linearity
  - Uneven distribution of faults

e.g., if an elevator can safely carry a load of 1000 kg, it can also safely carry any smaller load;

if a procedure correctly sorts a set of 256 elements, it may fail on a set of 255 or 53 or 12 elements, as well as on 257 or 1023.

Variety of approaches

- There are **no fixed recipes**
- Test designers must
  - choose and schedule the **right blend of techniques**
    - to reach the required level of quality
    - within cost constraints
  - design a **specific solution** that suits
    - the problem
    - the requirements
    - the development environment

When, What, How

- When does testing activities start? When are they complete?
- What particular techniques should be applied during development?
- How can we assess the readiness of a product?

When to start?
When are we complete?

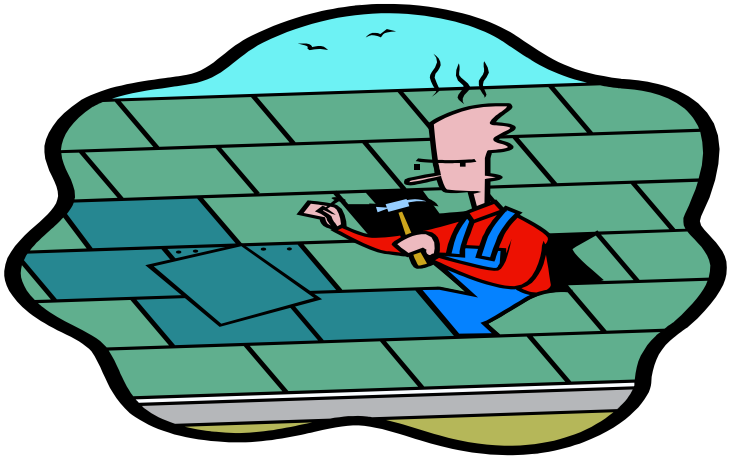
- Test is **not** a (late) phase of software development
- Execution of tests is a small part of the story
- Testing activities **start as soon as** we decide to build a software product, or even before
- Testing **last far beyond** the product delivery as long as the software is in use, to cope with evolution and adaptations to new conditions

Early start..

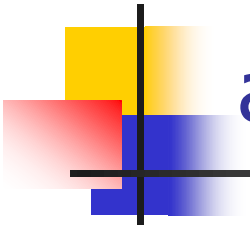
- **Written test objectives** and requirements must be documented
- What are your planned **coverage** levels?
- How much testing is **enough**?
- Common objective – **spend the budget** … **test until the ship-date** …
  - Sometimes called the "**date criterion**"

Long lasting: beyond maintenance

- analysis of changes and extensions
- generation of new test suites for the added functionalities
- re-executions of tests to check for non regression of software functionalities after changes and extensions
- fault tracking and analysis

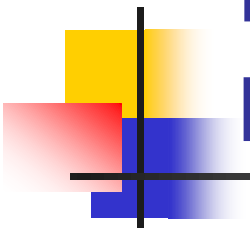


What particular techniques should be applied during development?



- No single analysis or testing technique can serve all purposes; primary reasons for combining them are:
  - Effectiveness for different **fault classes**
    e.g., analysis instead of testing for race conditions
  - Applicability at different **points in a project**
    e.g., inspection for early requirements validation
  - Differences in **purpose**
    e.g., statistical testing to measure reliability
  - Tradeoffs in **cost** and assurance
    e.g., expensive technique for key properties

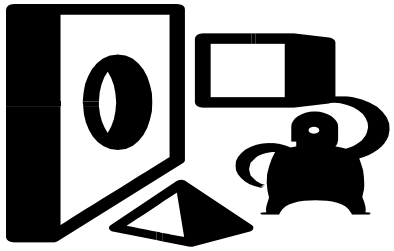
How can we assess the readiness of a product?



- We can not! :)

- Ideal case: *If the system passes an adequate suite of test cases, then it must be correct (or dependable)*
  - But that's impossible!
  - Adequacy of test suites, in the sense above, is **provably undecidable**

Practical (in)Adequacy Criteria



- Criteria that identify inadequacies in test suites
  - *e.g., no test executes a particular program statement*
- *If a test suite fails to satisfy some criterion*, the obligation that has not been satisfied may provide some useful information about improving the test suite
- *If a test suite satisfies all the obligations by all the criteria*, we do not know definitively that it is an effective test suite, but we have some evidence of its thoroughness

Analogy: Building Codes

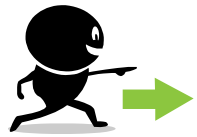
- Building codes are sets of design rules
  - Maximum span between beams in ceiling, floor, and walls; acceptable materials; wiring insulation; ...
  - Minimum standards, subject to judgment of building inspector who interprets the code
- You wouldn't buy a house just because it's "up to code"
  - It could be ugly, badly designed, inadequate for your needs
- But you might avoid a house because it isn't
  - Building codes are inadequacy criteria, like practical test "adequacy" criteria

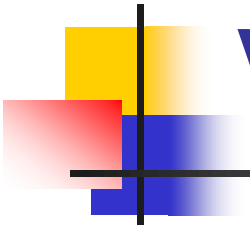
Software Testing Terms

- Like any field, software testing comes with a large number of **specialized terms** that have particular meanings in this context

- Some of the following terms are **standardized**, some are used **consistently** throughout the literature and the industry, but some **vary** by author, topic, or test organization

- Some **most commonly** used definitions follow..





Verification and validation

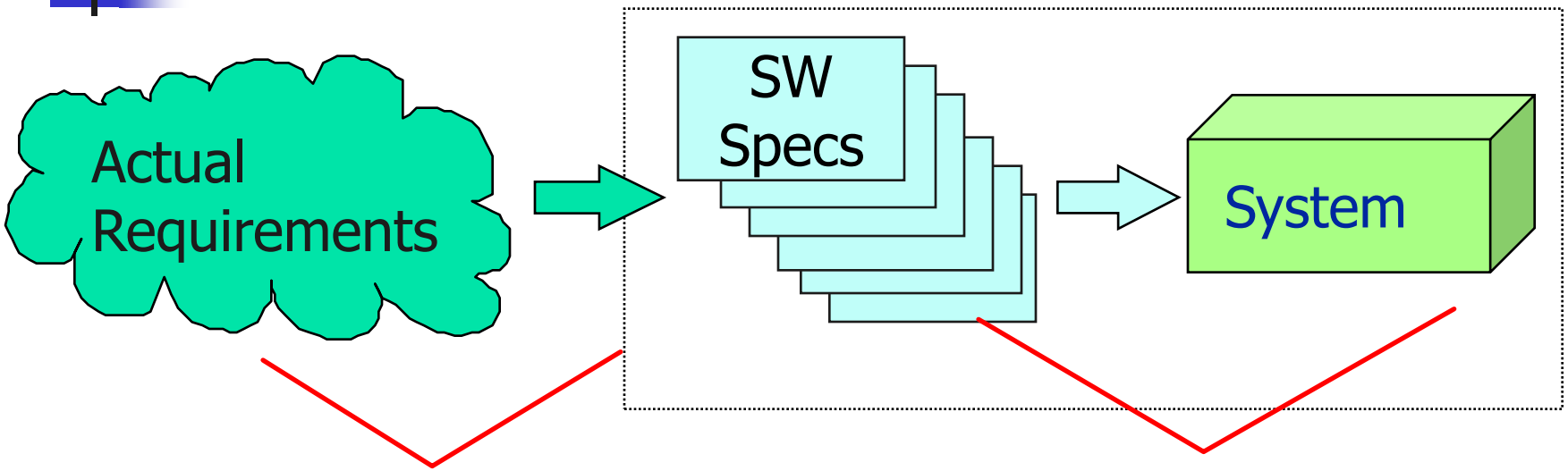
- **Validation**: does the software system meets the user's real needs?

  are we building the right software?

- **Verification**: does the software system meets the requirements specifications?

  are we building the software right?

Validation and Verification



Validation

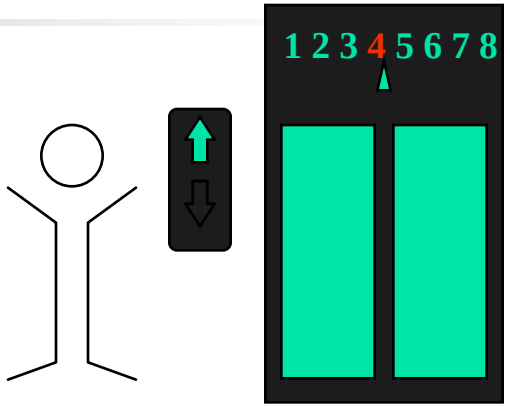
Includes usability testing, user feedback

Verification

Includes testing, inspections, static analysis

Verification or validation depends on the specification

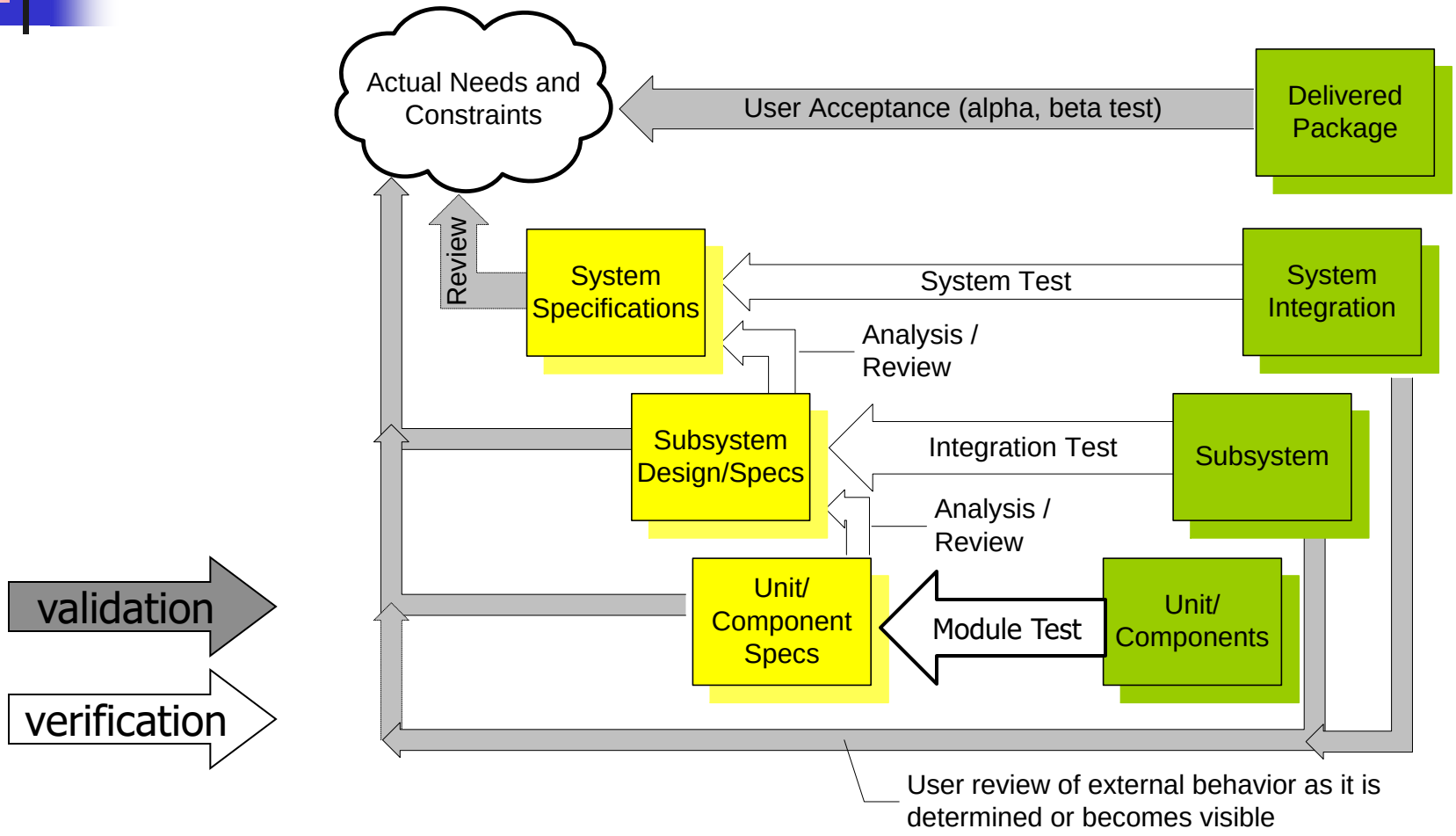
Example: elevator response



Unverifiable (but validatable) spec: ... if a user presses a request button at floor *i*, an available elevator must arrive at floor *i* **soon**...

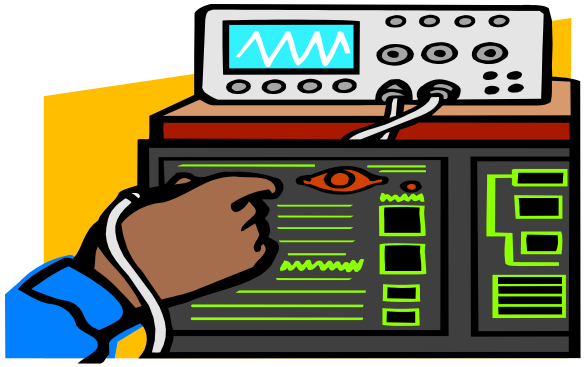
Verifiable spec: ... if a user presses a request button at floor *i*, an available elevator must arrive at floor *i* **within 30 seconds**...

Validation and Verification Activities

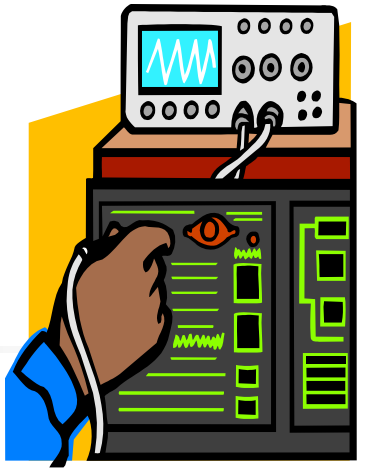


User Acceptance Tests..

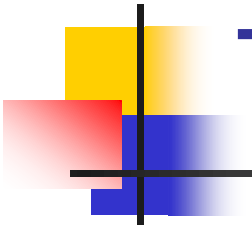
- **Alpha test**: tests performed by users in a controlled environment, observed by the development organization
- **Beta test**: tests performed by real users in their own environment, performing actual tasks without interference or close monitoring



Some useful terminology



- **Test case**: a set of inputs, execution conditions, and a pass/fail criterion
- **Test suite**: a set of test cases
- **Test case specification**: a requirement to be satisfied by one or more test cases
- **Test** or **test execution**: the activity of executing test cases and evaluating their results
- **Adequacy criterion**: a predicate that is true (satisfied) or false (not satisfied) of a ⟨program, test suite⟩ pair



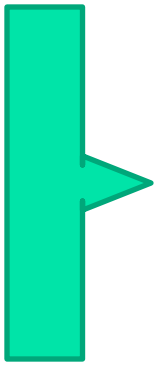
Test case, spec, suite..

Example:

- A test case specification:
  a sorted sequence of length > 2
- A corresponding test case:
  "Alpha, Beta, Chi, Omega"
- A test suite:
  { "Alpha, Beta, Chi, Omega", "Beta, Chi, Omega", "Alpha, Beta, Gama" }

What is a Test Plan?

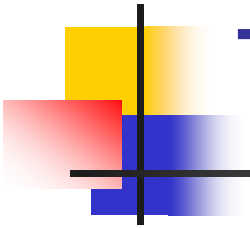
- Scope
- Approach
- Resources
- Schedule
- Test Items
- Testing Tasks
- Responsibilities
- Risks and Contingency plan



of Testing Activities

Example Test Plan Contents for System Testing

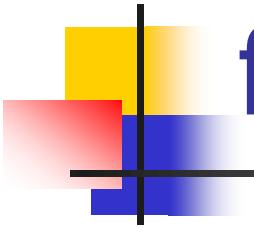
- Purpose
- Target audience and application
- Deliverables
- Information included
  - Introduction
  - Test items
  - Features tested
  - Features not tested
  - Test criteria
  - Pass / fail standards
  - Hardware and software requirements
  - Responsibilities for severity ratings
  - Staffing & training needs
  - Test schedules
  - Risks and contingencies
  - Approvals



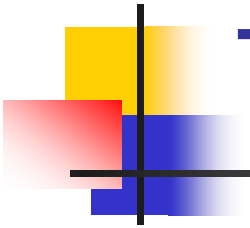
Testing vs. Debugging

- Testing: Finding inputs that cause the software to fail
- Debugging: The process of finding a fault given a failure
  - Diagnosis
  - Fault localization

Conditions for observing failures

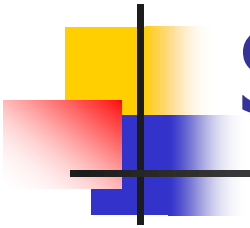


- Reachability: The location or locations in the program that contain the fault must be reached
- Infection: The state of the program must be incorrect
- Propagation: The infected state must propagate to cause some output of the program to be incorrect



Types of Testing..

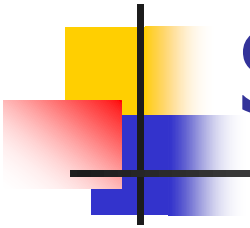
- Functional (*black box*): from software specifications
  - e.g., if spec requires robust recovery from power failure, test obligations should include simulated power failure
- Structural (*white* or *glass box*): from code
  - e.g., traverse each program loop one or more times.
- Model-based: from a model of the system
  - Models used in specification or design, or derived from code
  - e.g., exercise all transitions in communication protocol model
- Fault-based: from hypothesized faults (common bugs)
  - e.g., Check for buffer overflow handling (common vulnerability) by testing on very large inputs



Stress Testing

- "Tests that are at (out of) the limit of the software's expected input domain"

- Very **large** numeric values (or very small)
- Very **long** strings, **empty** strings
- **Null** references
- Very **large** files
- **Many users** making requests at the same time
- **Invalid** values



Some (more) terminology

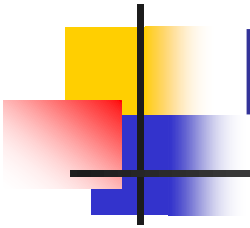
- An analysis of a program *P* with respect to a formula *F* is **sound** if the analysis returns true only when *P* does satisfy *F*

- An analysis of a program *P* with respect to a formula *F* is **complete** if the analysis always returns true when *P* actually does satisfy *F*

Sound vs. Complete for a fault detection technique F

- Sound: Fault found only when there is actually a fault
  - but it can miss some of the faults
- ~Precision
- Complete: Fault always found if there is any
  - but can it can lead to false alarms (false positives)
- ~Recall

	Property true	**Property false**
Answer true	sound, complete	complete
Answer false	sound	

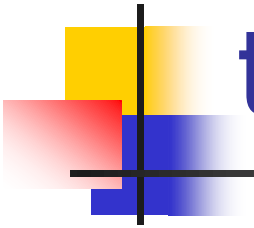


Main A&T Principles

- General engineering principles:
  - Partition: divide and conquer
  - Visibility: making information accessible
  - Feedback: tuning the development process
- Specific A&T principles:
  - Sensitivity: better to fail every time than sometimes
  - Redundancy: making intentions explicit
  - Restriction: making the problem easier

A&T: Analysis and Testing

Sensitivity: better to fail every time than sometimes



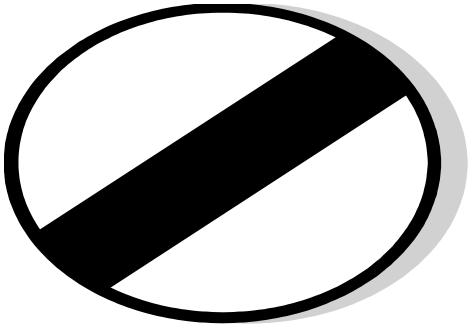
- Consistency
- Using **assert** statements
- machine independent run time deadlock analysis
  - if the program deadlocks when analyzed on one machine, it deadlocks on every machine

Redundancy: making intentions explicit

- Redundant checks can increase the capabilities of catching specific faults early or more efficiently.
  - redundant information: type, exception declarations
  - static type checking & dynamic type checking
  - validation of requirement specifications vs. validation of the final software

Restriction: making the problem easier

- Suitable restrictions can reduce hard (unsolvable) problems to simpler (solvable) problems
  - e.g., it is impossible (in general) to show that type errors do not occur at run-time in a dynamically typed language, but statically typed languages impose stronger restrictions that are easily checkable.



Restriction: example

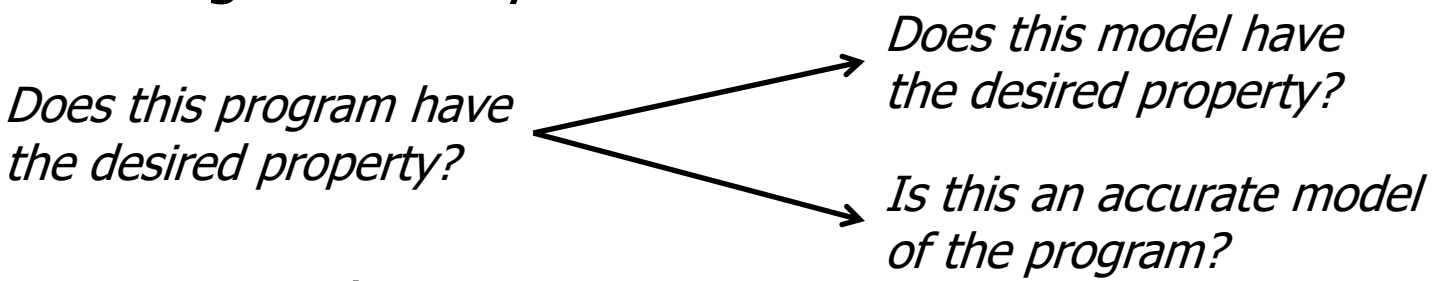
- Java rejects the method
  - initalization of k is not guaranteed

someCondition(0) = true ??

```
static void questionable() {
  int k;
    for (int i=0; i<10; ++i) {
      if(someCondition(i)) {
        k=0;
      } else {
        k += i;
      }
    }
  System.out.println(k);
}
```

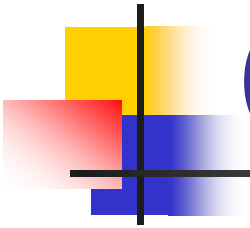
Partition: divide and conquer

- Partitioning the process:
  - Unit, integration, subsystem, system
- Modeling and analysis:



- Partitioning the input space:
  - both structural and functional test selection criteria identify suitable partitions of code or specifications

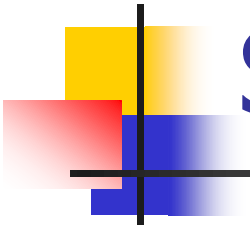
Visibility: Judging status (Observability)



- Measuring progress or status against goals
  - quality visibility = "Does quality meet our objectives?"
- Involves setting goals that can be assessed at each stage of development
- The biggest challenge is early assessment
  - e.g., assessing specifications and design with respect to product quality

Feedback: tuning the development process

- Learning from experience
- Projects provide information to improve the next
- Examples:
  - Checklists are built on the basis of errors revealed in the past
  - Error taxonomies can help in building better test selection criteria
  - Design guidelines can avoid common pitfalls

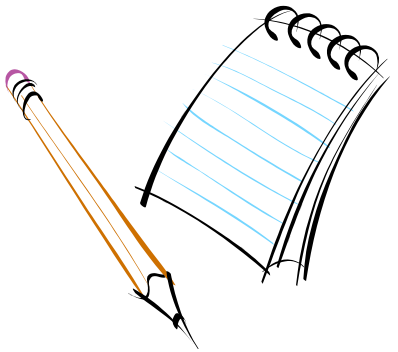


Summary: 6 main principles

- Sensitivity: better to fail every time than sometimes
- Redundancy: making intentions explicit
- Restriction: making the problem easier
- Partition: divide and conquer
- Visibility: making information accessible
- Feedback: tuning the development process

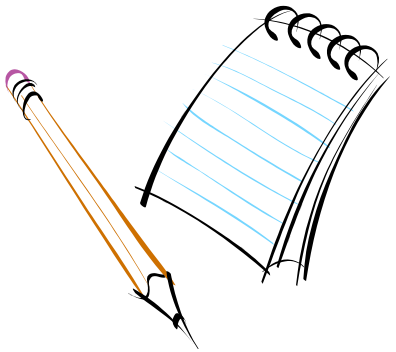
- Used for understanding andvantages and limits of different approaches and compare different techniques

Exercise



- Indicate which principles guided the following choices
  - Use an externally readible format also for internal files
  - Collect and analyze data about faults removed from the code
  - Seperate test and debugging activities
    - Design and execution of test cases to reveal failures (test)
    - Localization and removal of the corresponding faults (debugging)

Exercise



- Use an externally readible format also for internal files
  - Visibility principle
- Collect and analyze data about faults removed from the code
  - Feedback principle
- Seperate test and debugging activities
  - Partition principle