Software Testing

A survey of about 2000 researchers showed how coding has become important part of researcher's toolkit:

- 38 % spend at least one fifth of their time developing software
- But only 47 % have a good understanding of software testing

Source: Information taken from: Nature 467, 775-777 (2010) | doi:10.1038/467775a

Software testing is the process of validating and verifying that a software application/program meets the requirements that guided its design and development. It checks whether the application works as expected and satisfies the needs of the stakeholders. Software testing can be implemented at any time in the software development process. In traditional approach most of the test is performed almost finalizing requirements and implementation/coding, but in the Agile approaches testing is taken as an on-going activity during all phases of software development.

According to ANSI/IEEE 1059 standard, Testing can be defined as "*A process of analyzing a software item to detect the differences between existing and required conditions (that is defects/errors/bugs) and to evaluate the features of the software item*". Testing is a critical element of software development life cycles which is important for software quality control or software quality assurance. The basic goals of testing include validation and verification. At each stage, we need to verify that the thing we produce accurately represents its specification

Basic Notions

- Validation: are we building the right product?
- Verification: does "X" meet its specification?
  Where "X" can be code, a model, a design diagram, a requirement, etc.
- An error is a mistake made by an engineer. It is often a misunderstanding of a requirement or design specification
- A fault is a manifestation of that error in the code which is generally known as a **bug.**
- A failure is an incorrect output/behavior that is caused by executing a fault. The failure may occur immediately (e.g. crash!) or much, much later in the execution.
- Testing attempts to surface failures in our software systems. Debugging attempts to associate failures with faults so they can be removed from the system.

Difference between Testing and Debugging

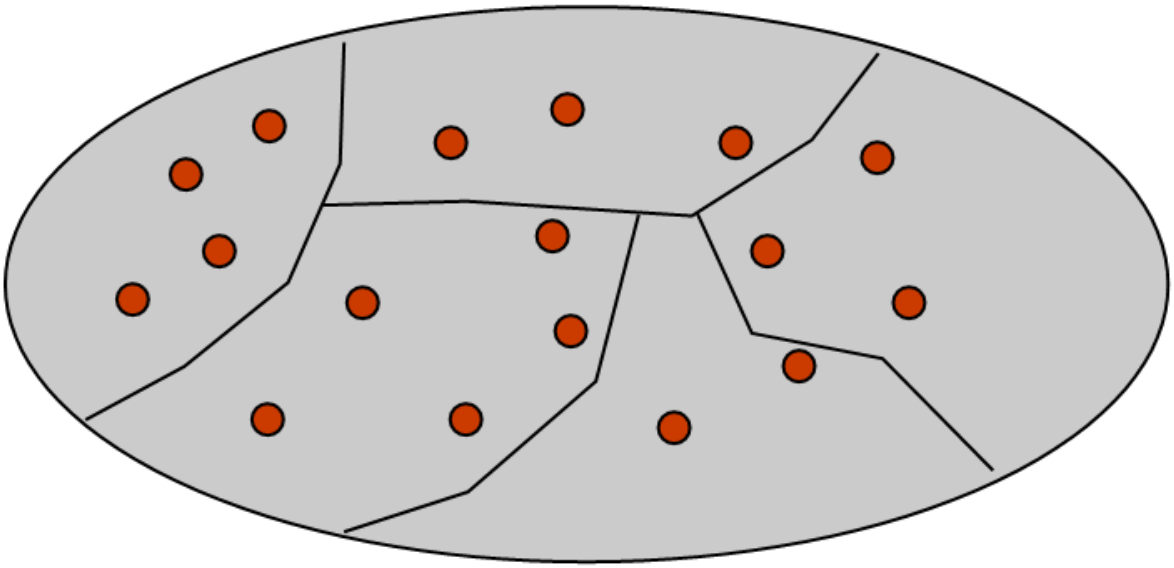
- **Testing**: It involves the identification of bug/error/defect in the software without correcting it. Normally professionals with a Quality Assurance background are involved in the identification of bugs. Testing is performed in the testing phase.
- **Debugging**: It involves identifying, isolating and fixing the problems/bug. Developers who code the software conduct debugging upon encountering an error in the code. Debugging is the part of White box or Unit Testing. Debugging can be performed in the development phase while conducting Unit Testing or in phases while fixing the reported bugs.

Verification	Validation
Are you building it right?	Are you building the right thing?
Ensure that the software system meets all the functionality.	Ensure that functionalities meet the intended behavior.
Verification takes place first and includes the checking for documentation, code etc.	Validation occurs after verification and mainly involves the checking of the overall product.
Done by developers.	Done by Testers.
Have static activities as it includes the reviews, walkthroughs, and inspections to verify that software is correct or not.	Have dynamic activities as it includes executing the software against the requirements.
It is an objective process and no subjective decision should be needed to verify the Software.	It is a subjective process and involves subjective decisions on how well the Software works.

Even if a system passes all of its tests, it cannot be guaranteed to be free of all faults.

- Faults may be hiding in portions of the code that only rarely get executed.
- "Testing can only be used to prove the existence of faults not their absence" or "Not all faults have failures"
- Sometimes faults mask each other resulting in no visible failures. This is particularly harmful.
- However, if we do a good job in creating a test set that covers all functional capabilities of a system and covers all code using a metric such as "branch coverage", then, having all tests pass increases our confidence that our system has high quality and can be deployed in the real production environment.

It is a known fact that **completely** testing a system is impossible. Hence, the testing literature advocates folding the space into equivalent behaviors and then sampling each partition. So, we need to apply some **heuristics**. We should attempt to fold the input space into different functional categories. Then create tests that sample the behavior/output for each functional partition.



An example: the greatest common denominator function: *int gcd(int x, int y)*

- This function seems to have an infinite number of test cases.
- But, folding the space:
  - x=6 y=9, returns 3, tests common case
  - x=2 y=4, returns 2, tests when x is the GCD
  - x=3 y=5, returns 1, tests two primes
  - x=9 y=0, returns ?, tests zero
  - x=-3 y=9, returns ?, tests negative

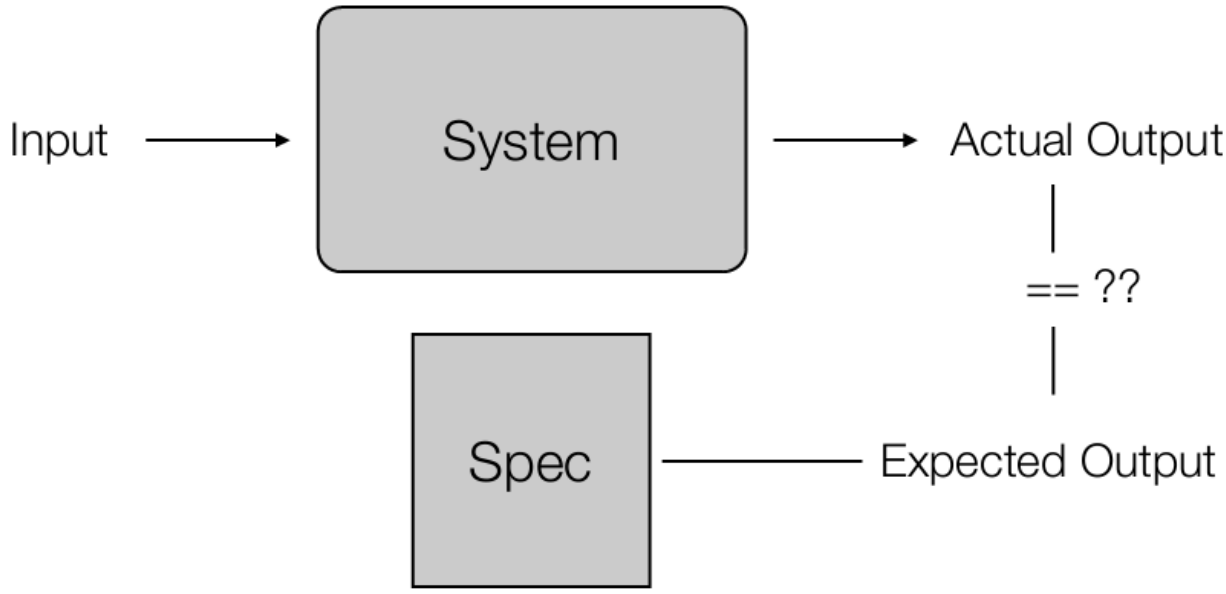
Testing is a continuous process that should be performed at every stage of a software development process. For example, during requirements collection phase, we must continually query the user, "Did we get this right?". At the end of each iteration we should check our results to see if what we built is meeting our requirements (specification).

Testing the System involves unit testing, integrating testing, system testing and acceptance testing.

- Unit Tests: Tests that cover low-level aspects of a system. For each module, does each operation perform as expected? For method foo(), we would see another method testFoo()
- Integration Tests: Tests that check that modules work together in combination. Most projects are on schedule until they reach integration. All sorts of hidden assumptions are surfaced when code written by different developers are used in tandem. Lack of integration testing has led to great failures (e.g. Mars Polar Lander).
  - *The Mars Polar Lander, also referred to as the Mars Surveyor '98 Lander, was a 290-kilogram robotic spacecraft lander, launched by NASA on January 3, 1999, to study the soil and climate of Planum Australe, a region near the south pole on Mars, as part of the Mars Surveyor '98 mission. However, on December 3, 1999, after the descent phase was expected to be complete, the lander failed to reestablish communication with Earth. It was determined that the most likely cause of the mishap was an improperly terminated engine firing prior to the lander touching the surface, causing the lander to impact at a high velocity*
- System Tests: Tests performed by the developer to ensure that all major functionality has been implemented. Have all user stories been implemented and function correctly?
- Acceptance Tests: Tests performed by the user to check that the delivered system meets their needs. In large, custom projects, developers will be on-site to install system and then respond to problems as they arise.

After finishing coding, we can perform three types of tests:

- **Black Box Testing:**
  Does the system behave as predicted by its specification.
  A black box test passes input to a system, records the actual output and compares it to the expected output.



Results:

if actual output == expected output
TEST PASSED
else
TEST FAILED

• Write at least one test case per functional capability
• Iterate on code until all tests pass
• Need to automate this process as much as possible

Black Box Testing Categories:

• Functionality
• User input validation (based off specification)
• Output results
• State transitions
• are there clear states in the system in which the system is supposed to behave differently based on the state?
• Boundary cases and off-by-one errors

- **Grey Box Testing**:
  Having a bit of insight into the architecture of the system, does it behave as predicted by its specification?
  • Use knowledge of system's architecture to create a more complete set of black box tests
  • Verifying auditing and logging information. Logging typically means the recording of implementation level events that happen as the program is running (e.g. methods get called, objects are created, etc.). Auditing is about recording domain-level events: a transaction is created; a user is performing an action, etc.
  • For each function, is the system really updating all internal state correctly?
  • Data destined for other systems
  • System-added information (timestamps, checksums, etc.)
  • "Looking for Scraps"
  Is the system correctly cleaning up? E.g. Temporary files, memory leaks, data

duplication/deletion

- **White Box Testing**:
  Since, we have access to most of the code; let's make sure we are covering all aspects of the code: statements, branches, etc.
  • Writing test cases with complete knowledge of code
  • Format: input, expected output, actual output
  • White box testing involves:
    - code coverage (more on this in a minute)
    - proper error handling
    - Working as documented (is method "foo" thread safe?)
    - Proper handling of resources: How does the software behave when resources become constrained?

  • **Code coverage:** A criteria for knowing white box testing is "complete". It includes:
    - Statement coverage: write tests until all statements have been executed
    - Branch coverage (edge coverage): write tests until each edge in a program's control flow graph has been executed at least once (covers true/false conditions)
    - Condition coverage: like branch coverage but with more attention paid to the conditionals (if compound conditional, ensure that all combinations have been covered)
    - Path coverage: write tests until all paths in a program's control flow graph have been executed multiple times as dictated by heuristics, e.g., for each loop, write a test case that executes the loop:
      - zero times (skips the loop)
      - exactly one time
      - more than once (exact number depends on context)

	Black Box Testing	Grey Box Testing	White Box Testing
1.	The Internal Workings of an application are not required to be known	Somewhat knowledge of the internal workings are known	Tester has full knowledge of the Internal workings of the application
2.	Also known as closed box testing, data driven testing and functional testing	Another term for grey box testing is translucent testing as the tester has limited knowledge of the insides of the application	Also known as clear box testing, structural testing or code based testing
3.	Performed by end users and also by testers and developers	Performed by end users and also by testers and developers	Normally done by testers and developers
4.	-Testing is based on external expectations -Internal behavior of the application is unknown	Testing is done on the basis of high level database diagrams and data flow diagrams	Internal workings are fully known and the tester can design test data accordingly
5.	This is the least time consuming and exhaustive	Partly time consuming and exhaustive	The most exhaustive and time consuming type of testing
6.	Not suited to algorithm testing	Not suited to algorithm testing	Suited for algorithm testing
7.	This can only be done by trial and error method	Data domains and Internal boundaries can be tested, if known	Data domains and Internal boundaries can be better tested

References:

http://en.wikipedia.org/wiki/Software_testing
http://www.tutorialspoint.com
http://en.wikipedia.org/wiki/Mars_Polar_Lander