

Black Box Testing

Sources:

Code Complete, 2nd Ed., Steve McConnell

Software Engineering, 5th Ed., Roger Pressman

Testing Computer Software, 2nd Ed., Cem Kaner, et. Al.

Black Box Testing

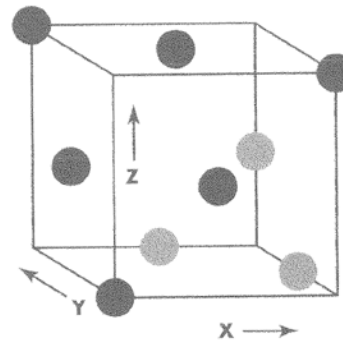
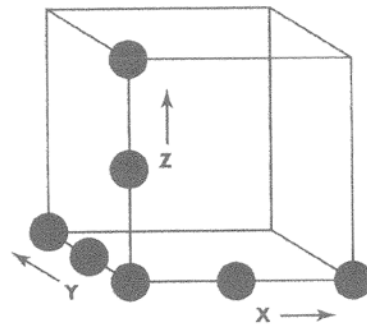
- Testing software against a specification of its external behavior without knowledge of internal implementation details
 - Can be applied to software “units” (e.g., classes) or to entire programs
 - External behavior is defined in API docs, Functional specs, Requirements specs, etc.
- Because black box testing purposely disregards the program's control structure, attention is focused primarily on the information domain (i.e., data that goes in, data that comes out)
- The Goal: Derive sets of input conditions (test cases) that fully exercise the external functionality

Black Box Testing

- Black box testing tends to find different kinds of errors than white box testing
 - Missing functions
 - Usability problems
 - Performance problems
 - Concurrency and timing errors
 - Initialization and termination errors
 - Etc.
- Unlike white box testing, black box testing tends to be applied later in the development process
- [Black Box Testing Example #1](#)

The Information Domain: inputs and outputs

- Inputs
 - Individual input values
 - Try many different values for each individual input
 - Combinations of inputs
 - Individual inputs are not independent from each other
 - Programs process multiple input values together, not just one at a time
 - Try many different combinations of inputs in order to achieve good coverage of the input domain
 - Vary more than one input at a time to more completely cover the input domain



The Information Domain: inputs and outputs

- Inputs (continued)
 - Ordering and Timing of inputs
 - In addition to the particular combination of input values chosen, the ordering and timing of the inputs can also make a difference
- Outputs
 - In addition to covering the input domain, make sure your tests thoroughly cover the output domain
 - What are the legal output values?
 - Is it possible to select inputs that produce invalid outputs?

The Information Domain: inputs and outputs

- Defining the input domain
 - Boolean value
 - T or F
 - Numeric value in a particular range
 - $99 \leq N \leq 99$
 - Integer, Floating point
 - One of a fixed set of enumerated values
 - {Jan, Feb, Mar, ...}
 - {Visa, MasterCard, Discover, ...}
 - Formatted strings
 - Phone numbers
 - File names
 - URLs
 - Credit card numbers
 - Regular expressions

Equivalence Partitioning

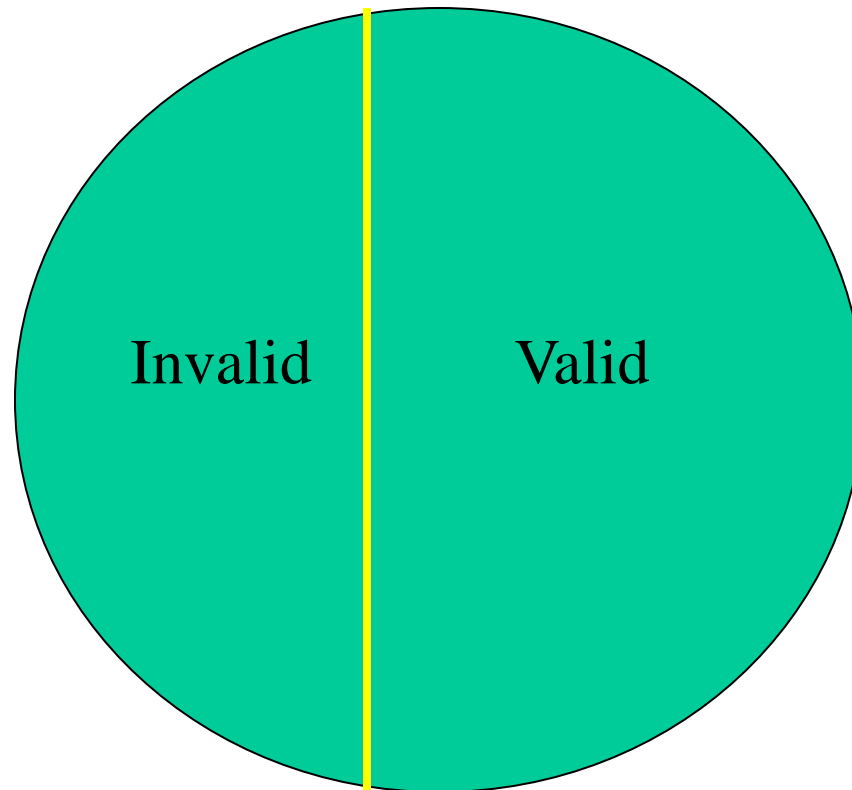
- Consider all of the possible values for a single input (i.e., the input's domain)
- Frequently the domain is so large that you can't test all possible values
- You have to select a relatively small number of values to actually test
- Which values should you choose?
- Equivalence partitioning helps answer this question

Equivalence Partitioning

- Partition the input's domain into "equivalence classes"
- Each equivalence class contains a set of "equivalent" values
- Two values are considered to be equivalent if we expect the program to process them both in the same way
- If you expect the program to process two values in the same way, only test one of them, thus reducing the number of test cases you have to run

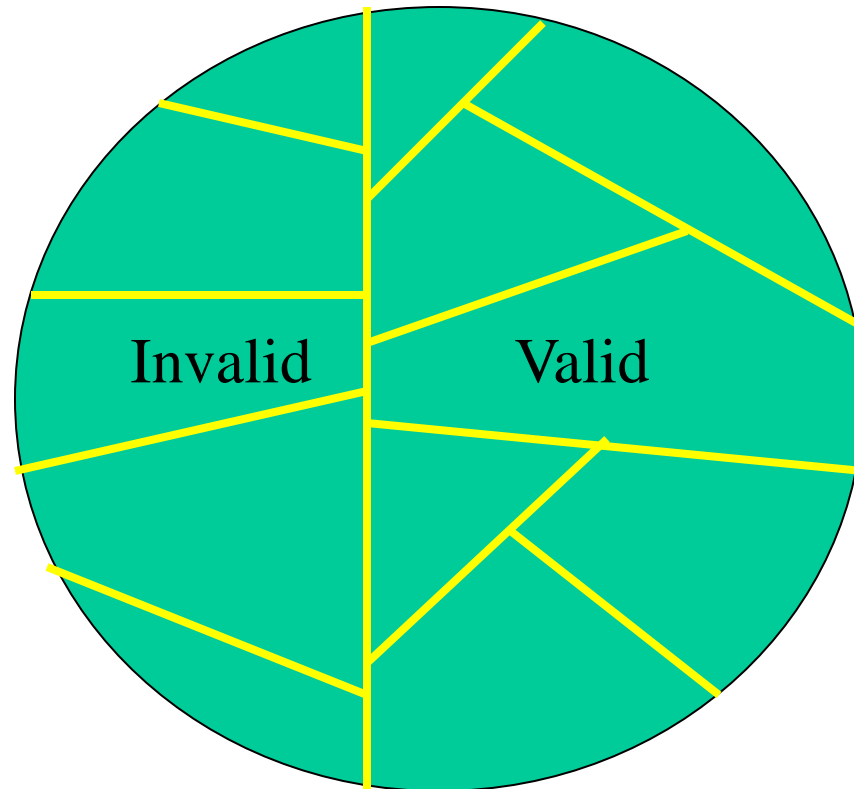
Equivalence Partitioning

- First-level partitioning: Valid vs. Invalid values



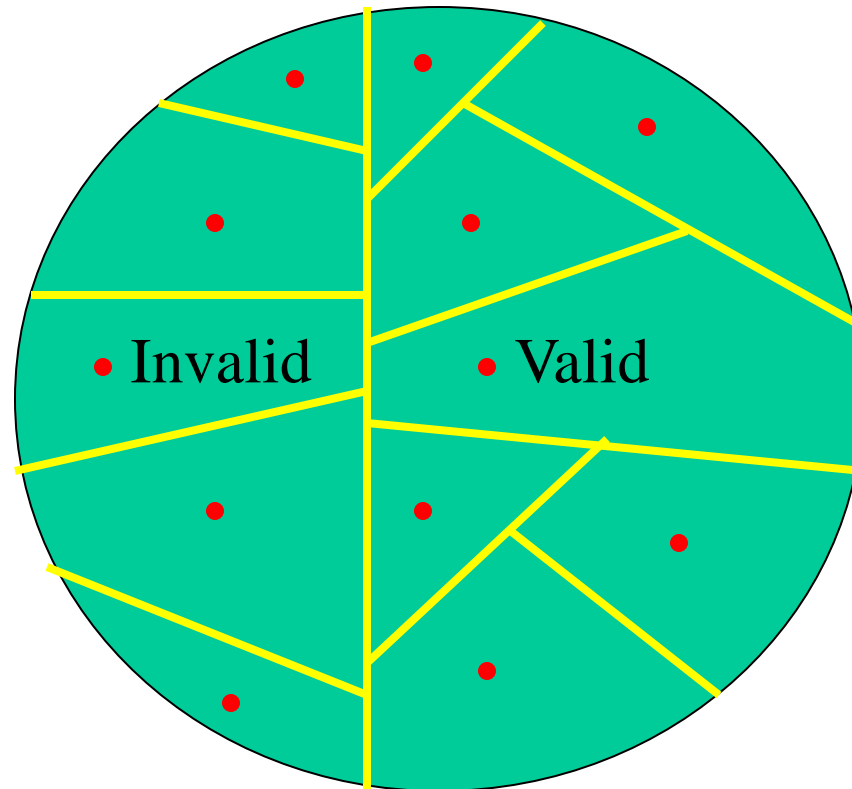
Equivalence Partitioning

- Partition valid and invalid values into equivalence classes



Equivalence Partitioning

- Create a test case for at least one value from each equivalence class



Equivalence Partitioning

- Equivalence classes can overlap
- If an oracle is available, the test values in each equivalence class can be randomly generated. This is more useful than always testing the same static values.
 - Oracle: something that can tell you whether a test passed or failed
- Test multiple values in each equivalence class. Often you're not sure if you have defined the equivalence classes correctly or completely, and testing multiple values in each class is more thorough than relying on a single value.

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: -99 <= N <= 99	?	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?


Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	[-99, -10] [-9, -1] 0 [1, 9] [10, 99]	< -99 > 99 Malformed numbers {12-, 1-2-3, ...} Non-numeric strings {junk, 1E2, \$13} Empty value
Phone Number Area code: [200, 999] Prefix: (200, 999) Suffix: Any 4 digits	?	?

Equivalence Partitioning - examples

Input	Valid Equivalence Classes	Invalid Equivalence Classes
A integer N such that: $-99 \leq N \leq 99$	$[-99, -10]$ $[-9, -1]$ 0 $[1, 9]$ $[10, 99]$	< -99 > 99 Malformed numbers $\{12-, 1-2-3, \dots\}$ Non-numeric strings $\{\text{junk}, 1E2, \$13\}$ Empty value
Phone Number Area code: $[200, 999]$ Prefix: $(200, 999]$ Suffix: Any 4 digits	555-5555 (555)555-5555 555-555-5555 $200 \leq \text{Area code} \leq 999$ $200 < \text{Prefix} \leq 999$	

Equivalence Partitioning - examples

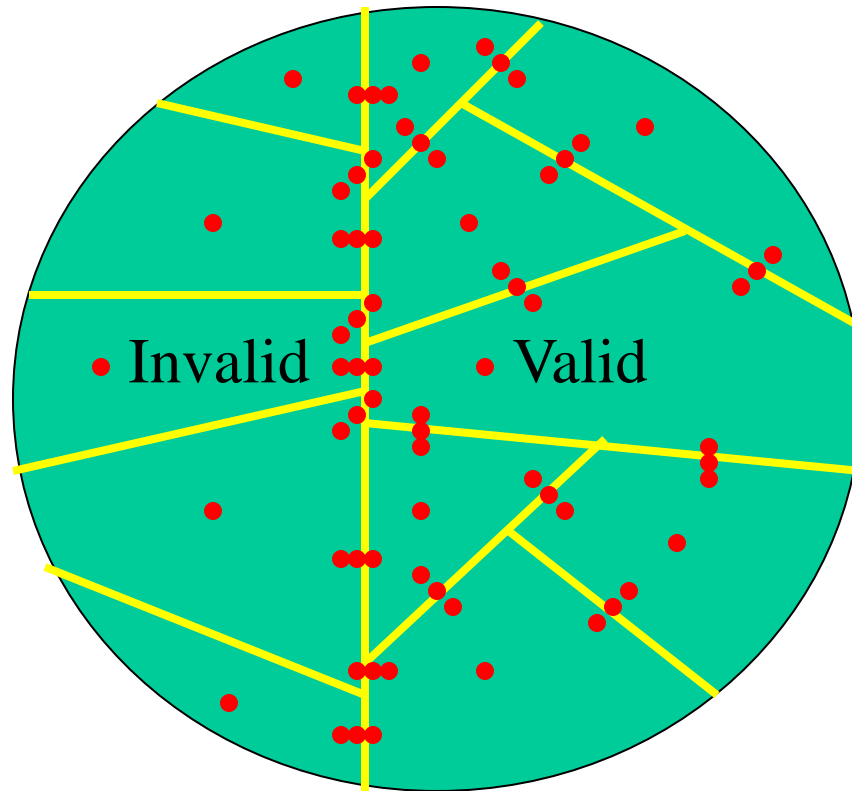
Input	Valid Equivalence Classes	Invalid Equivalence Classes
<p>A integer N such that: $-99 \leq N \leq 99$</p>	<p>$[-99, -10]$ $[-9, -1]$ 0 $[1, 9]$ $[10, 99]$</p>	<p>< -99 > 99 Malformed numbers $\{12-, 1-2-3, \dots\}$ Non-numeric strings $\{\text{junk}, 1E2, \\$13\}$ Empty value</p>
<p>Phone Number Area code: $[200, 999]$ Prefix: $(200, 999]$ Suffix: Any 4 digits</p>	<p>555-5555 $(555)555-5555$ 555-555-5555 $200 \leq \text{Area code} \leq 999$ $200 < \text{Prefix} \leq 999$</p>	<p>Invalid format 5555555, $(555)(555)5555$, etc. Area code < 200 or > 999 Area code with non-numeric characters <i>Similar for Prefix and Suffix</i></p>

Boundary Value Analysis

- When choosing values from an equivalence class to test, use the values that are most likely to cause the program to fail
- Errors tend to occur at the boundaries of equivalence classes rather than at the "center"
 - If `(200 < areaCode && areaCode < 999) { // valid area code }`
 - Wrong!
 - If `(200 <= areaCode && areaCode <= 999) { // valid area code }`
 - Testing area codes 200 and 999 would catch this error, but a center value like 770 would not
- In addition to testing center values, we should also test boundary values
 - Right on a boundary
 - Very close to a boundary on either side

Boundary Value Analysis

- Create test cases to test boundaries of equivalence classes



Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$?
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	?

Boundary Value Analysis - examples

Input	Boundary Cases
A number N such that: $-99 \leq N \leq 99$	-100, -99, -98 -10, -9 -1, 0, 1 9, 10 98, 99, 100
Phone Number Area code: [200, 999] Prefix: (200, 999] Suffix: Any 4 digits	Area code: 199, 200, 201 Area code: 998, 999, 1000 Prefix: 200, 199, 198 Prefix: 998, 999, 1000 Suffix: 3 digits, 5 digits

Boundary Value Analysis - examples

- Numeric values are often entered as strings which are then converted to numbers internally [int x = atoi(str);]
- This conversion requires the program to distinguish between digits and non-digits
- A boundary case to consider: Will the program accept / and : as digits?

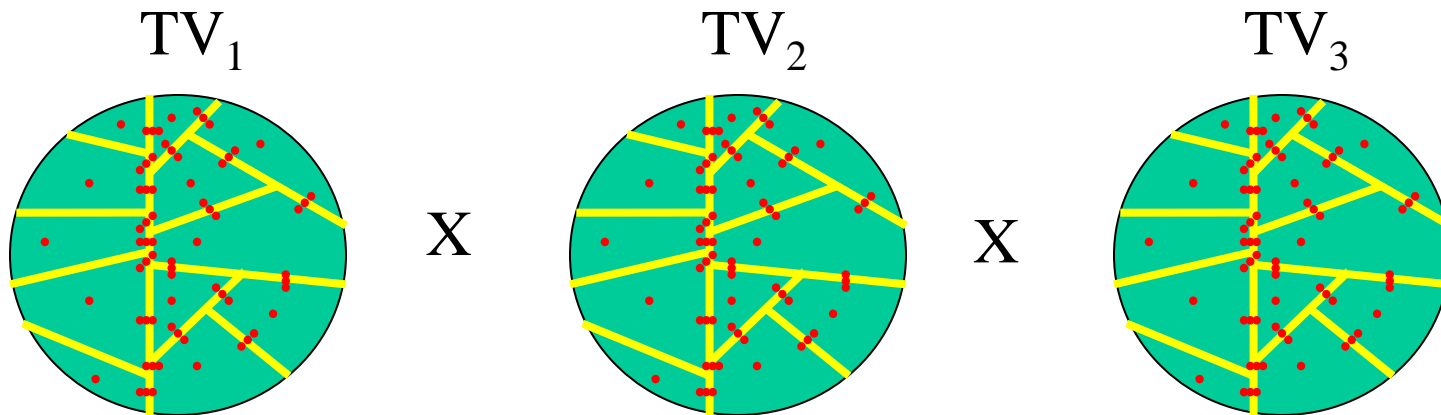
Char	/	0	1	2	3	4	5	6	7	8	9	:
ASCII	47	48	49	50	51	52	53	54	55	56	57	58

Testing combinations of inputs

- Equivalence Partitioning and Boundary Value Analysis are performed on each individual input, resulting in a set of test values for each input
 - TV_1 = set of test values for Input 1
 - TV_2 = set of test values for Input 2
 - Etc.
- Beyond testing individual inputs, we must also consider input combinations

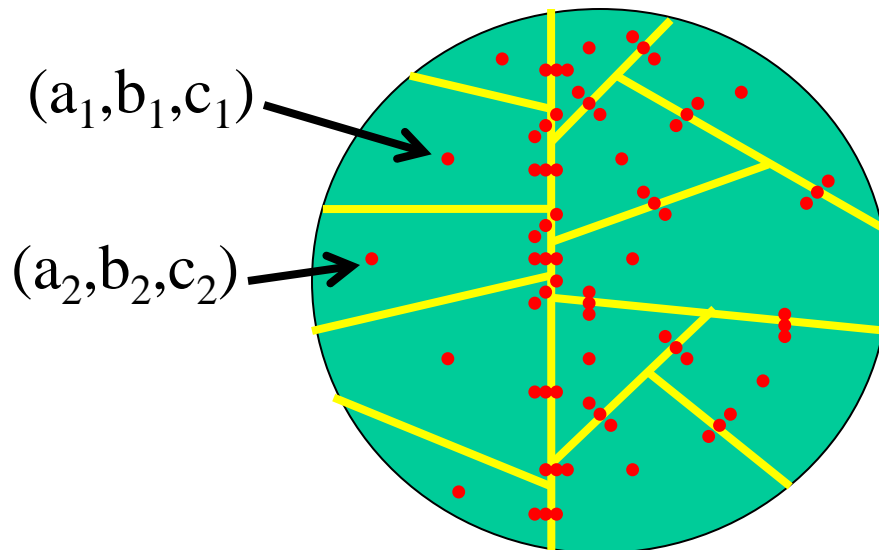
Testing combinations of inputs

- Suppose there are 3 inputs
- An input combination is a 3-tuple (tv_1, tv_2, tv_3)
 - where $tv_1 \in TV_1, tv_2 \in TV_2, tv_3 \in TV_3$
- Test as many different combinations as possible
- Avoid testing redundant combinations (ones that follow the same path thru the code)
- Vary multiple inputs at once to better cover the input domain



Testing combinations of inputs

- Rather than analyzing each input individually, it often makes sense to do EP and BVA on multiple inputs together
- If two or more inputs interact heavily with each other, effective testing often requires their values to be selected together
- In this case, each “test value” is actually a tuple of values
- Example: [Substring Search \(solution\)](#)



Mainstream usage testing

- Don't get so wrapped up in testing boundary cases that you neglect to test "normal" input values
 - Values that users would typically enter during mainstream usage

More black box testing examples

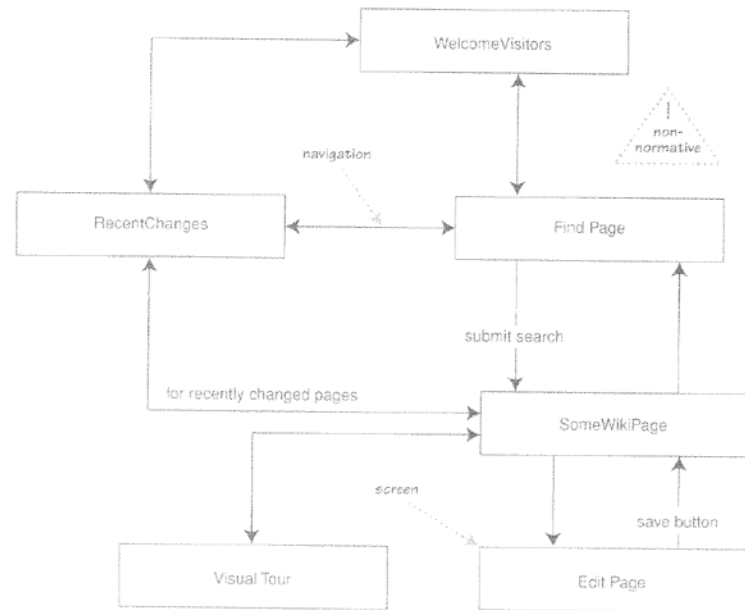
- [Black Box Example #2](#)
- [Black Box Example #3](#)

Error Guessing

- Based on intuition, guess what kinds of inputs might cause the program to fail
- Create some test cases based on your guesses
- Intuition will often lead you toward boundary cases, but not always
- Some special cases aren't boundary values, but are mishandled by many programs
 - Try exiting the program while it's still starting up
 - Try loading a corrupted file
 - Try strange but legal URLs: `hTtP://Www.bYu.EDU/`

State Transition testing

- Every interactive program has user observable states
 - What screen the user is on
 - What information is displayed to the user
 - What actions the user is allowed to perform
- User observable states are often modeled using a screen flow diagram that shows how users can move from screen to screen



State Transition testing

- Ideally, you will test all of the different paths that a user may follow to reach each screen, and ensure that the program is always in the correct state
 - Example: Test all of the different paths for reaching the “Order Confirmation” screen on an e-commerce web site. Can you trick the software into letting you submit an order without entering payment information?
- It's not only where you are, but also how you got there

Comparison Testing

- Also called Back-to-Back testing
- If you have multiple implementations of the same functionality, you can run test inputs through both implementations, and compare the results for equality
- Why would you have access to multiple implementations?
 - Safety-critical systems sometimes use multiple, independent implementations of critical modules to ensure the accuracy of results
 - You might use a competitor's product, or an earlier version of your own, as the second implementation
 - You might write a software simulation of a new chip that serves as the specification to the hardware designers. After building the chip, you could compare the results computed by the chip hardware with the results computed by the software simulator
- Inputs may be randomly generated or designed manually

Testing for race conditions and other timing dependencies

- Many systems perform multiple concurrent activities
 - Operating systems manage concurrent programs, interrupts, etc.
 - Servers service many clients simultaneously
 - Applications let users perform multiple concurrent actions
- Test a variety of different concurrency scenarios, focusing on activities that are likely to share resources (and therefore conflict)
- "Race conditions" are bugs that occur only when concurrent activities interleave in particular ways, thus making them difficult to reproduce
- Test on hardware of various speeds to ensure that your system works well on both slower and faster machines

Performance Testing

- Measure the system's performance
 - Running times of various tasks
 - Memory usage, including memory leaks
 - Network usage (Does it consume too much bandwidth? Does it open too many connections?)
 - Disk usage (Is the disk footprint reasonable? Does it clean up temporary files properly?)
 - Process/thread priorities (Does it play well with other applications, or does it hog the whole machine?)

Limit Testing

- Test the system at the limits of normal use
- Test every limit on the program's behavior defined in the requirements
 - Maximum number of concurrent users or connections
 - Maximum number of open files
 - Maximum request size
 - Maximum file size
 - Etc.
- What happens when you go slightly beyond the specified limits?
 - Does the system's performance degrade dramatically, or gracefully?

Stress Testing

- Test the system under extreme conditions (i.e., beyond the limits of normal use)
- Create test cases that demand resources in abnormal quantity, frequency, or volume
 - Low memory conditions
 - Disk faults (read/write failures, full disk, file corruption, etc.)
 - Network faults
 - Unusually high number of requests
 - Unusually large requests or files
 - Unusually high data rates (what happens if the network suddenly becomes ten times faster?)
- Even if the system doesn't need to work in such extreme conditions, stress testing is an excellent way to find bugs

Random Testing

- Randomly generate test inputs
 - Could be based on some statistical model
- How do you tell if the test case succeeded?
 - Where do the expected results come from?
 - Some type of “oracle” is needed
- Expected results could be calculated manually
 - Possible, but lots of work
- Automated oracles can often be created to measure characteristics of the system
 - Performance (memory usage, bandwidth, running times, etc.)
 - Did the system crash?
 - Maximum and average user response time under simulated user load

Security Testing

- Any system that manages sensitive information or performs sensitive functions may become a target for intrusion (i.e., hackers)
- How feasible is it to break into the system?
- Learn the techniques used by hackers
- Try whatever attacks you can think of
- Hire a security expert to break into the system
- If somebody broke in, what damage could they do?
- If an authorized user became disgruntled, what damage could they do?

Usability Testing

- Is the user interface intuitive, easy to use, organized, logical?
- Does it frustrate users?
- Are common tasks simple to do?
- Does it conform to platform-specific conventions?

- Get real users to sit down and use the software to perform some tasks
- Watch them performing the tasks, noting things that seem to give them trouble
- Get their feedback on the user interface and any suggested improvements

- Report bugs for any problems encountered

Recovery Testing

- Try turning the power off or otherwise crashing the program at arbitrary points during its execution
 - Does the program come back up correctly when you restart it?
 - Was the program's persistent data corrupted (files, databases, etc.)?
 - Was the extent of user data loss within acceptable limits?
- Can the program recover if its configuration files have been corrupted or deleted?
- What about hardware failures? Does the system need to keep working when its hardware fails? If so, verify that it does so.

Configuration Testing

- Test on all required hardware configurations
 - CPU, memory, disk, graphics card, network card, etc.
- Test on all required operating systems and versions thereof
 - Virtualization technologies such as VMWare and Virtual PC are very helpful for this
- Test as many Hardware/OS combinations as you can
- Test installation programs and procedures on all relevant configurations

Compatibility Testing

- Test to make sure the program is compatible with other programs it is supposed to work with
- Ex: Can Word 12.0 load files created with Word 11.0?
- Ex: "Save As... Word, Word Perfect, PDF, HTML, Plain Text"
- Ex: "This program is compatible with Internet Explorer and Firefox"
- Test all compatibility requirements

Documentation Testing

- Test all instructions given in the documentation to ensure their completeness and accuracy
- For example, “How To ...” instructions are sometimes not updated to reflect changes in the user interface
- Test user documentation on real users to ensure it is clear and complete