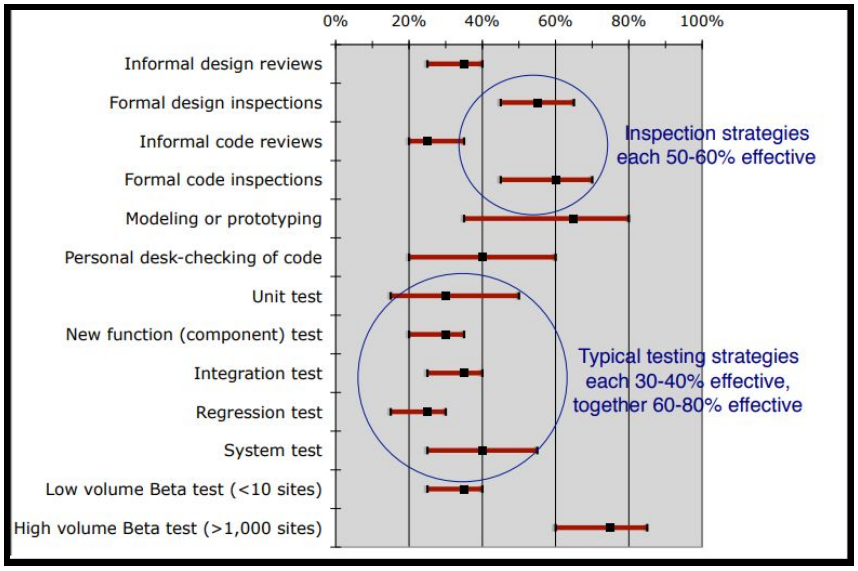
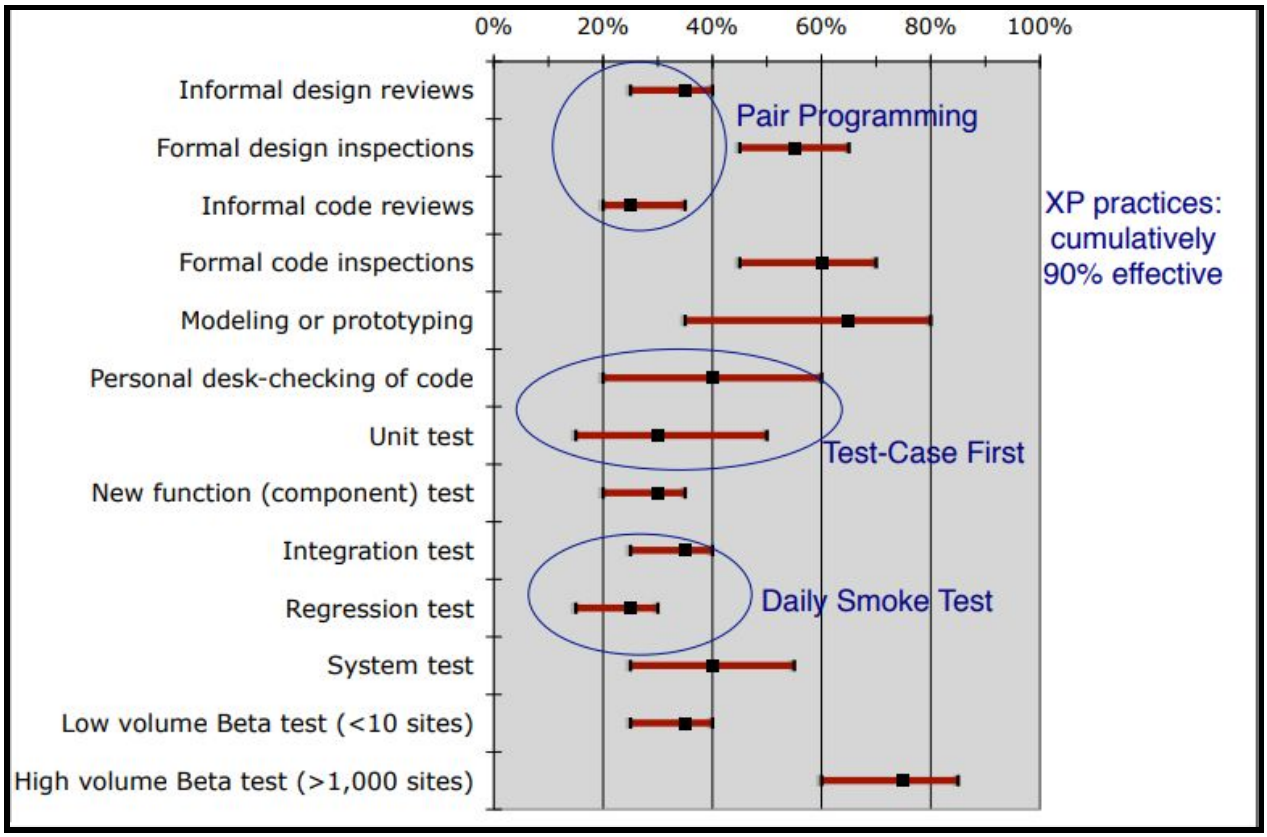
Introduction to Testing:

- **Defects vs. Failures:**
- Many causes of defects in software include:
  - Missing requirement
  - Specification wrong
  - Requirements that were infeasible
  - Faulty system design
  - Wrong algorithms
  - Faulty implementation
- Defects may lead to failures but the failure may show up somewhere else. Tracking the failure back to a defect can be hard.
- Examples of program defects are:
  - **Syntax Faults:** Incorrect use of programming constructs.
  - **Algorithmic Faults:**
    - Branching too soon or too late.
    - Testing for the wrong condition.
    - Failure to initialize correctly.
    - Failure to test for exceptions. E.g. divide by 0
    - Type mismatch
  - **Precision Faults:**
    - Mixed precision.
    - Faulty/incorrect floating point conversion.
  - **Documentation Faults:** Design docs or user manual is wrong.
  - **Stress Faults:**
    - Overflowing buffers.
    - Lack of bounds checking.
  - **Timing Faults:**
    - Processes fail to synchronize.
    - Events happen in the wrong order (Race Condition).
  - **Throughput Faults:** Performance is lower than required.
  - **Recovery Faults:** Incorrect recovery after another failure.
  - **Hardware Faults:** Hardware doesn't perform as expected.
- **Effectiveness of defect detection strategies:**
- Defect Detection Effectiveness



- XP Practices

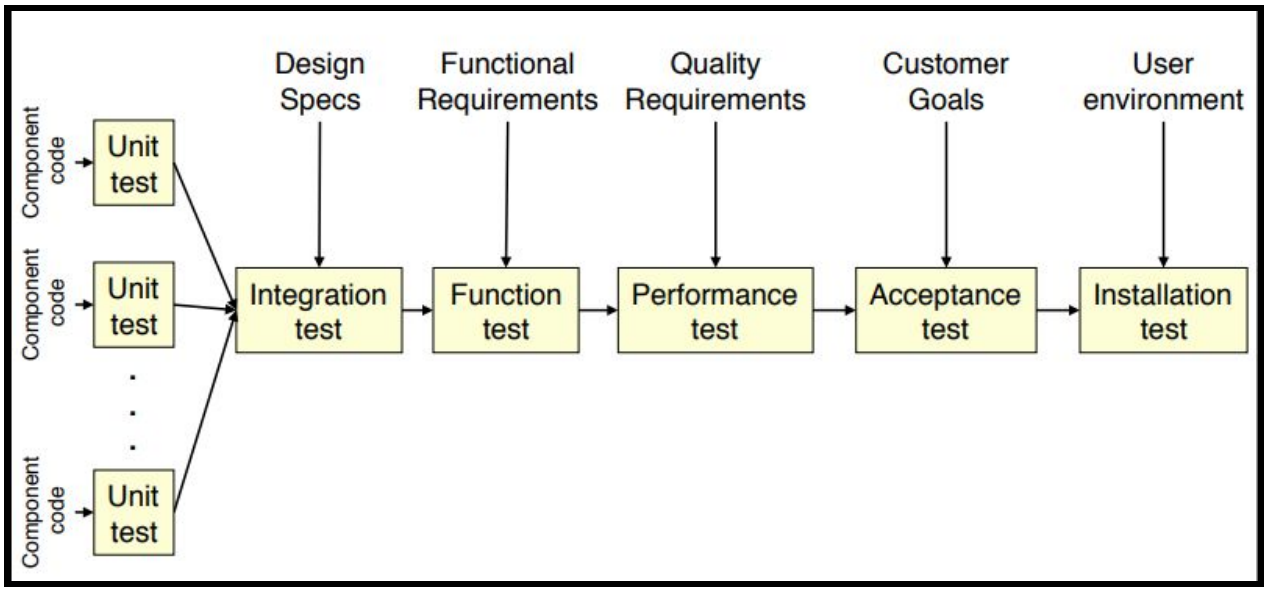


- Use a combination of techniques because:
  - Different techniques will find different defects.
  - Different people will find different defects.
  - Testing alone is only 60-80% effective.
  - The best organisations achieve 95% defect-removal.
  - Inspection, modeling, prototyping, and system tests are all important.
- Costs vary (This is from an IBM study):
  - On average, 3.5 hours are spent on each defect for inspection.
  - On average, 15-25 hours are spent on each defect for testing.
- Costs of fixing defects also vary:
  - It is 100 times more expensive to remove a defect after implementation than in design.
  - 1-step methods (inspection) are cheaper than 2-step methods (test+debug).
- Cost of Rework:
  - The industry average is 10-50 lines of delivered code per day per person.
  - Debugging + retesting is 50% of the effort in traditional software engineering.
- Removing defects early saves money. Testing is easier if the defects are removed first and high quality software will be delivered sooner at a lower cost.
- How not to improve quality: “Trying to improve quality by doing more testing is like trying to diet by weighing yourself more often.”
- **Basics of Testing:**
- Benefits of testing:
  - Find important defects to get them fixed.
  - Assess the quality of the product.
  - Help managers make release decisions.

    - Block premature product releases.
    - Help predict and control product support costs.
    - Check interoperability with other products.
    - Find safe scenarios for use of the product.
    - Assess conformance to specifications.
    - Certify that the product meets a particular standard.
    - Ensure that the testing process meets accountability standards.
    - Minimize the risk of safety-related lawsuits.
    - Measure reliability.
- Testing is more effective if you removed the bugs first.
- The goal of testing is unachievable. We cannot ever prove absence of errors. Furthermore, finding no errors probably means your tests are ineffective.
- The goal of testing is also counter-intuitive. It is the only goal in software engineering whose aim is to find errors/break the software. All other development activities aim to avoid errors/breaking the software.
- Testing does not improve software quality. Test results measure the quality of the existing code, but doesn't improve it. Test-debug cycles are the least effective way to improve quality of code.
- Testing requires you to assume your code is buggy. If you assume otherwise, you probably won't find them.
- Testing must be done appropriately based on the context and the requirements/specification.
- Good tests have:
    - **Power:** When a problem exists, the test will find it.
    - **Validity:** The problems found are genuine problems.
    - **Value:** Each test reveals things the clients will want to know.
    - **Credibility:** Each test is a likely operational scenario.
    - **Non-redundancy:** Each test provides new information.
    - **Repeatability:** Each test is easy and inexpensive to re-run.
    - **Maintainability:** Tests can be revised as the requirements/specifications/products are revised.
    - **Coverage:** The test cases exercise the product in a way not already tested for. This is similar to non-redundancy.
    - **Ease of evaluation:** The test results are easy to interpret.
    - **Diagnostic power:** The tests help pinpoint the cause of problems.
    - **Accountability:** You can explain, justify and prove you ran the test cases.
    - **Low cost:** The time and effort to develop and execute the test cases are low.
    - **Low opportunity cost:** Creating and running a test is a better use of your time than other things you could be doing.
- Types of testing:
    - **Unit test:** Testing an individual software component or module.
      I.e. Each unit is tested separately to check it meets its specification.
    - **Integration test:** Testing performed to expose defects in the interfaces and in the interactions between integrated components or systems.
      I.e. Units are tested together to check they work together.
      Integration testing is hard because it is much harder to identify equivalence classes, problems of scale may occur and it tends to reveal specification errors rather than integration errors.
    - **Function test:** Testing that validates the software system against the functional requirements/specifications.

- **Performance test:** Testing the speed, responsiveness and stability of a computer, network, software program or device under a workload.
- **Acceptance test:** An acceptance test verifies whether the end to end flow of the system is as per the business requirements and if it is as per the needs of the end-user. The client accepts the software only when all the features and functionalities work as expected. It is the last phase of the testing, after which the software goes into production. This is also called User Acceptance Testing (UAT).
- **Installation test:** Testing to check if the software has been correctly installed with all the inherent features and that the product is working as per expectations.

I.e.



- Systematic testing depends on partitioning. We need to partition the set of possible behaviours of the system and choose representative samples from each partition and make sure we covered all partitions. Identifying suitable partitions is what testing is all about. We can use different test strategies and methods to do so.
- Coverage 1: Structural

```
boolean equal (int x, y) {
/* effects: returns true if
  x=y, false otherwise
*/
if (x == y)
  return(TRUE)
else
  return(FALSE)
}
```

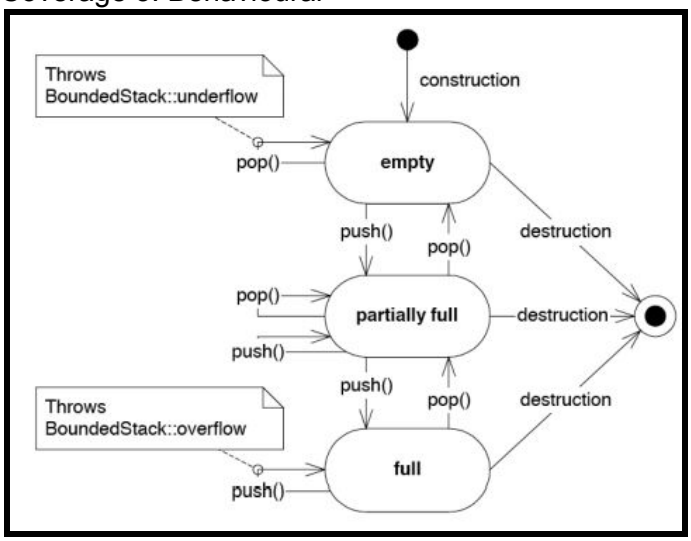
A naïve testing strategy is to pick random values for x and y and test ‘equals’ on them. However, we might never test the first branch of the ‘if’ statement, so we need enough test cases to cover every branch in the code.

- Coverage 2: Functional

```
int maximum (list a)
/* requires: a is a list of
   integers
   effects: returns the maximum
   element in the list
*/
```

A naïve testing strategy is to generate lots of lists and test maximum on them. However, we haven't tested off-nominal cases such as empty lists, non-integers, negative integers, etc, so we need enough test cases to cover every kind of input the program might have to handle.

- Coverage 3: Behavioural



A naïve testing strategy is to push and pop things off the stack and check it all works. However, we might miss full and empty stack exceptions, so we need enough tests to exercise every event that can occur in each state that the program can be in.

- Other types of tests:
  - **Facility testing:** Does the system provide all the functions required?
  - **Volume testing:** Can the system cope with large data volumes?
  - **Stress testing:** Can the system cope with heavy loads?
  - **Endurance testing:** Will the system continue to work for long periods?
  - **Usability testing:** Can the users use the system easily?
  - **Security testing:** Can the system withstand attacks?
  - **Performance testing:** How good is the response time?
  - **Storage testing:** Are there any unexpected data storage issues?
  - **Configuration testing:** Does the system work on all target hardware?
  - **Installation testing:** Can we install the system successfully?
  - **Reliability testing:** How reliable is the system over time?
  - **Recovery testing:** How well does the system recover from failure?
  - **Serviceability testing:** How maintainable is the system?
  - **Documentation testing:** Is the documentation accurate, usable, etc.
  - **Operations testing:** Are the operators' instructions right?
  - **Regression testing:** Repeat all the testing every time we modify the system.

Testing Strategies:

- **Good Practices:**
- Write the test cases first. This forces you to think carefully about the requirements first and exposes requirements problems early.
- **Structural Coverage Strategies (White box testing):**
- **White box testing** is a software testing method in which the internal structure/design/implementation of the item being tested is known to the tester.
- Types of white box testing include:
  - **Structured Basis Testing:**
    - Structured basis testing gives you the minimum number of test cases you need to exercise every path.
    - Start with 1 test case for the straight path. Add 1 test case for each of these keywords: if, while, repeat, for, and, or add 1 test case for each branch of a case statement.
    - E.g.

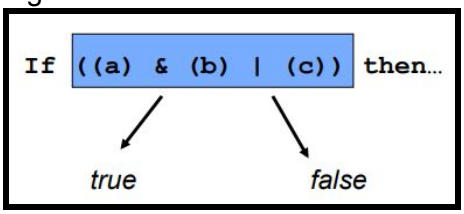
```
int midval (int x, y, z) {
/* effects: returns median
  value of the three inputs
*/
if (x > y) {
  if (x > z) return x
  else return z }
else {
  if (y > z) return y
  else return z } }
```

We will need 4 test cases.

  - **Statement Coverage:**
    - Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code.

$$\textit{Statement Coverage} = \frac{\textit{Number of executed statements}}{\textit{Total number of statements}} \times 100$$

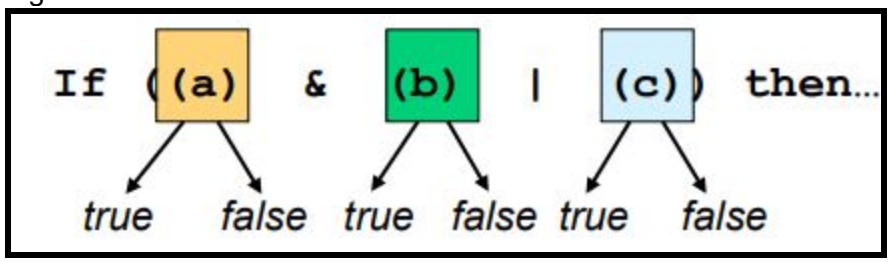
  - **Branch Coverage:**
    - Branch coverage is a white box testing method in which every outcome from a code module is tested. The purpose of branch coverage is to ensure that each decision condition from every branch is executed at least once.
    - E.g.



$$\textit{Branch Coverage} = \frac{\textit{Number of Executed Branches}}{\textit{Total Number of Branches}}$$

  -

- **Condition/Decision Coverage:**
  - Condition coverage is a testing method used to test and evaluate the variables or sub-expressions in the conditional statement. The goal of condition coverage is to check individual outcomes for each logical condition. Condition coverage offers better sensitivity to the control flow than decision coverage. In this coverage, expressions with logical operands are only considered.
  - Condition coverage does not give a guarantee about full decision coverage.
  - E.g.



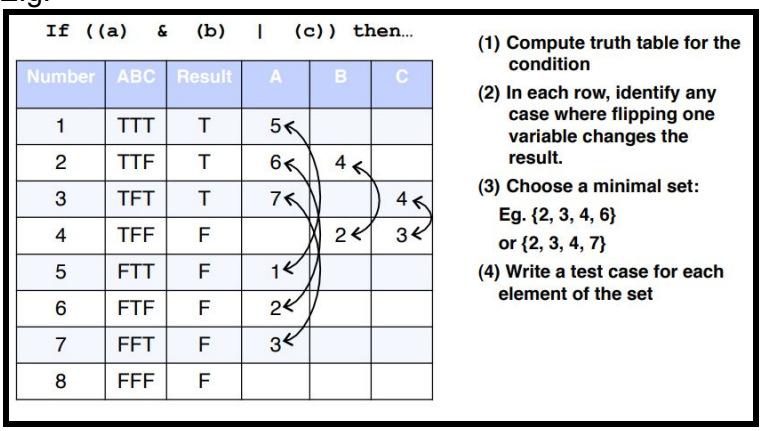
  - E.g.
    Suppose we have the code below.

```
if ((A || B) && C)
{
  << Few Statements >>
}
else
{
   << Few Statements >>
}
```

    The 3 following tests would be sufficient for 100% condition coverage testing.
    A = true | B = not eval | C = false
    A = false | B = true | C = true
    A = false | B = false | C = not eval

$$Condition\ Coverage = \frac{Number\ of\ Executed\ Operands}{Total\ Number\ of\ Operands}$$

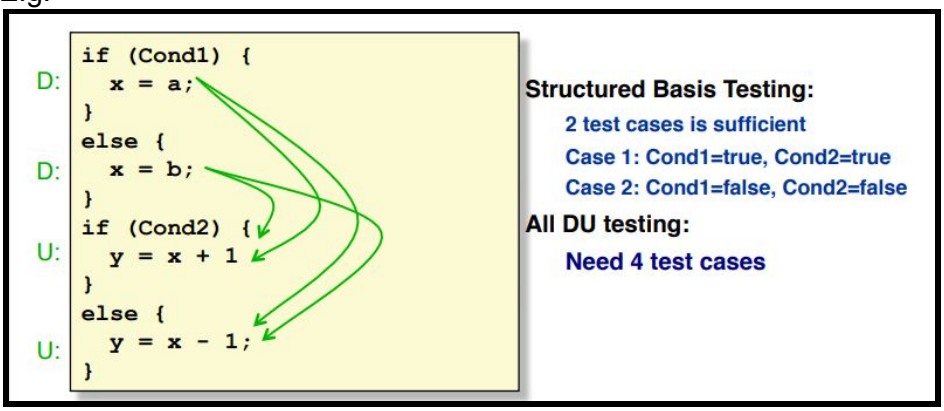
- **Modified Condition/Decision (MC/DC) Coverage:**
  - The modified condition/decision coverage (MC/DC) coverage is like condition coverage, but every condition in a decision must be tested independently to reach full coverage. This means that each condition must be executed twice, with the results true and false, but with no difference in the truth values of all other conditions in the decision. In addition, it needs to be shown that each condition independently affects the decision.
  - The Modified Condition/Decision Coverage enhances the condition/decision coverage criteria by requiring that each condition be shown to independently affect the outcome of the decision. This kind of testing is performed on mission critical application which might lead to death, injury or monetary loss.
  - E.g.



Number	ABC	Result	A	B	C
1	TTT	T	5		
2	TTF	T	6	4	
3	TFT	T	7		4
4	TFF	F		2	3
5	FTT	F	1		
6	FTF	F	2		
7	FFT	F	3		
8	FFF	F			

  - Advantages:
    - Linear growth in the number of conditions.
    - Ensures coverage of the object code.
    - Discovers dead code (operands that have no effect).
  - It is mandated by the US Federal Aviation Administration. In avionics, complex boolean expressions are common, and MC/DC coverage has been shown to uncover important errors not detected by other test approaches.
  - It's expensive. The total cost of aircraft development for Boeing 777 is $5.5 billion while the cost of testing to MC/DC criteria is approximately $1.5 billion.
- **Data Flow Coverage:**
  - Things that happen to data:
    - **D**efined - Data is initialized but not yet used.
    - **U**sed - Data is used in a computation.
    - **K**illed - Space is released.
    - **En**tered - Working copy created on entry to a method.
    - **Ex**ited - Working copy removed on exit from a method.
  - Normal life: Defined once, Used a number of times, then Killed
  - Potential Defects:
    - D-D: A variable is defined twice.
    - D-Ex, D-K: A variable is defined but not used.
    - En-K: Destroying a local variable that wasn't defined.
    - En-U: Using a local variable before it's initialized.

    - K-K: Unnecessary killing a variable. This can hang the machine.
    - K-U: Using data after it has been destroyed.
    - U-D: Redefining a variable after it has been used.
- Data flow testing helps us to pinpoint any of the following issues:
    - A variable that is declared but never used within the program.
    - A variable that is used but never declared.
    - A variable that is defined multiple times before it is used.
    - Deallocating a variable before it is used.
- To get the minimal set of tests to cover every D-U path, we need 1 test for each path from each definition to each use of the variable.
- E.g.



- **Boundary Checking:**
    - Every boundary needs 3 tests.
    - E.g. Suppose we have this line:
      if (x < 3)
      We need 3 test cases:
      One to test when x < 3.
      One to test when x == 3.
      One to test when x > 3.

- **Function Coverage Strategies (Black box testing):**
- **Black box testing** is a software testing method in which the internal structure/design/implementation of the item being tested is not known to the tester.
- Generating Tests from Use Cases:
    1. Test the Basic Flow.
    2. Test the Alternate Flows.
    3. Test the Postconditions.
    4. Break the Preconditions.
    5. Identify options for each input choice.
- Classes of Bad Data:
    - Too little data or no data.
    - Too much data.
    - The wrong kind of data (invalid data).
    - The wrong size of data.
    - Uninitialized data.
- Classes of Good Data:
    - Nominal cases - middle of the road, expected values.
    - Minimum normal configuration.
    - Maximum normal configuration.
    - Compatibility with old data.

- Classes of input variables:
  - **Values that trigger alternative flows:**
    E.g. Invalid credit card information.
  - **Trigger different error messages:**
    E.g. The text is too long for the field.
    E.g. An email address with no "@".
  - **Inputs that cause changes in the appearance of the UI:**
    E.g. A prompt for additional information.
  - **Inputs that cause different options in dropdown menus:**
    E.g. US/Canada triggers a menu of states/ provinces.
  - **Cases in a business rule:**
    E.g. No next day delivery after 6pm.
  - **Border conditions:**
    E.g. If the password must be min 6 characters, test passwords of 5,6,7 characters.
  - **Check the default values:**
    E.g. When the cardholder's name is filled automatically.
  - **Override the default values:**
    E.g. When the user enters a different name.
  - **Enter data in different formats:**
    E.g. phone numbers: (416) 555 1234 vs 416-555-1234 vs 416 555 1234.
  - **Test country-specific assumptions:**
    E.g. date order: 3/15/12 vs 15/3/12.
- Limits of Use Cases as Test Cases:
  - **Use case tests are good for:**
    User acceptance testing.
    "Business as usual" functional testing.
    Manual black-box tests.
    Recording automated scripts for common scenarios.
  - **Limitations of use case tests:**
    Likely to be incomplete.
    Use cases don't describe enough detail of use.
    Gaps and inconsistencies may occur between use cases.
    Use cases might be out of date.
    Use cases might be ambiguous.
  - **Defects you won't discover:**
    System errors (e.g. memory leaks).
    Things that corrupt persistent data.
    Performance problems.
    Software compatibility problems.
    Hardware compatibility problems.
- **Stress Testing:**
- Stress testing is a type of software testing that verifies the stability and reliability of the software application. The goal of stress testing is measuring software on its robustness and error handling capabilities under extremely heavy load conditions and ensuring that software doesn't crash under crunch situations. It even tests beyond normal operating points and evaluates how software works under extreme conditions.
- **QuickTests:**
  - QuickTests are tests that don't cost much to design, are based on some estimated idea for how the system could fail and don't take much prior knowledge in order to apply.

- **Explore the input domain:**
    1. Inputs that force all the error messages to appear.
    2. Inputs that force the software to establish default values.
    3. Explore allowable character sets and data types.
    4. Overflow the input buffers.
    5. Find inputs that may interact, and test combinations of their values.
    6. Repeat the same input numerous times.
- **Explore the outputs:**
    7. Force different outputs to be generated for each input.
    8. Force invalid outputs to be generated.
    9. Force properties of an output to change.
    10. Force the screen to refresh.
- **Explore stored data constraints:**
    11. Force a data structure to store too many or too few values.
    12. Find ways to violate internal data constraints.
- **Explore feature interactions:**
    13. Experiment with invalid operator/operand combinations.
    14. Make a function call itself recursively.
    15. Force computation results to be too big or too small.
    16. Find features that share data.
- **Vary file system conditions:**
    17. File system full to capacity.
    18. Disk is busy or unavailable.
    19. Disk is damaged.
    20. Invalid file name.
    21. Vary file permissions.
    22. Vary or corrupt file contents.

- **Interference Testing:**
    - Examples include:
        - Generate Interrupts
        - Change the context
        - Cancel a task
        - Pause the task
        - Swap out the task
        - Compete for resources
- **A radical alternative - Exploratory Testing:**
- **Exploratory testing** is a style of software testing that emphasizes personal freedom and responsibility of the tester to continually optimize the value of their work by treating test-related learning, test design, and test execution as mutually supportive activities that run in parallel throughout the project.
- Exploratory testing is a type of software testing where test cases are not created in advance but testers check the system on the fly. They may note down ideas about what to test before test execution. The focus of exploratory testing is more on testing as a "thinking" activity.
- Exploratory testing is widely used in Agile models and is all about discovery, investigation, and learning. It emphasizes personal freedom and responsibility of the individual tester.
- Under scripted testing, you design test cases first and later proceed with test execution. On the contrary, exploratory testing is a simultaneous process of test design and test execution all done at the same time. It is unscripted.