Software Testing （II)

Jianyong Wang(王建勇)
Department of Computer Science and Technology
Tsinghua University, Beijing, China

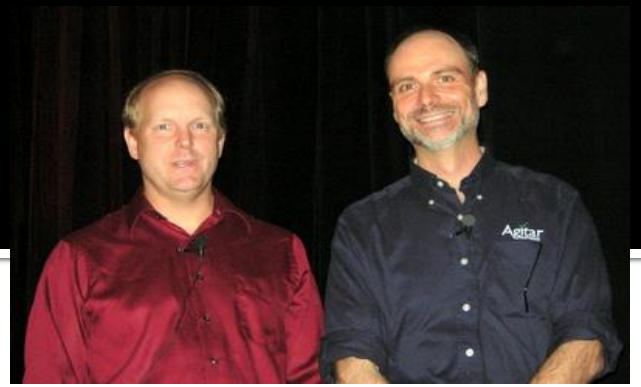
Outline

- Software Quality Assurance
- Basic Testing Concepts
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Other types of Testing

xUnit: : A family of Unit Test Frameworks

- The xUnit family of test automation frameworks is designed for use in ***automating programmer tests***.
  - Make it easy for developers to write tests without needing to learn a new programming language.
  - Make it easy to test individual functions/methods, classes, and objects without needing to have the rest of the application available.
    - Test software from the inside
  - Make it easy to run one test or many tests with a single action.
  - Minimize the cost of running the tests so programmers aren't discouraged from running the existing test.

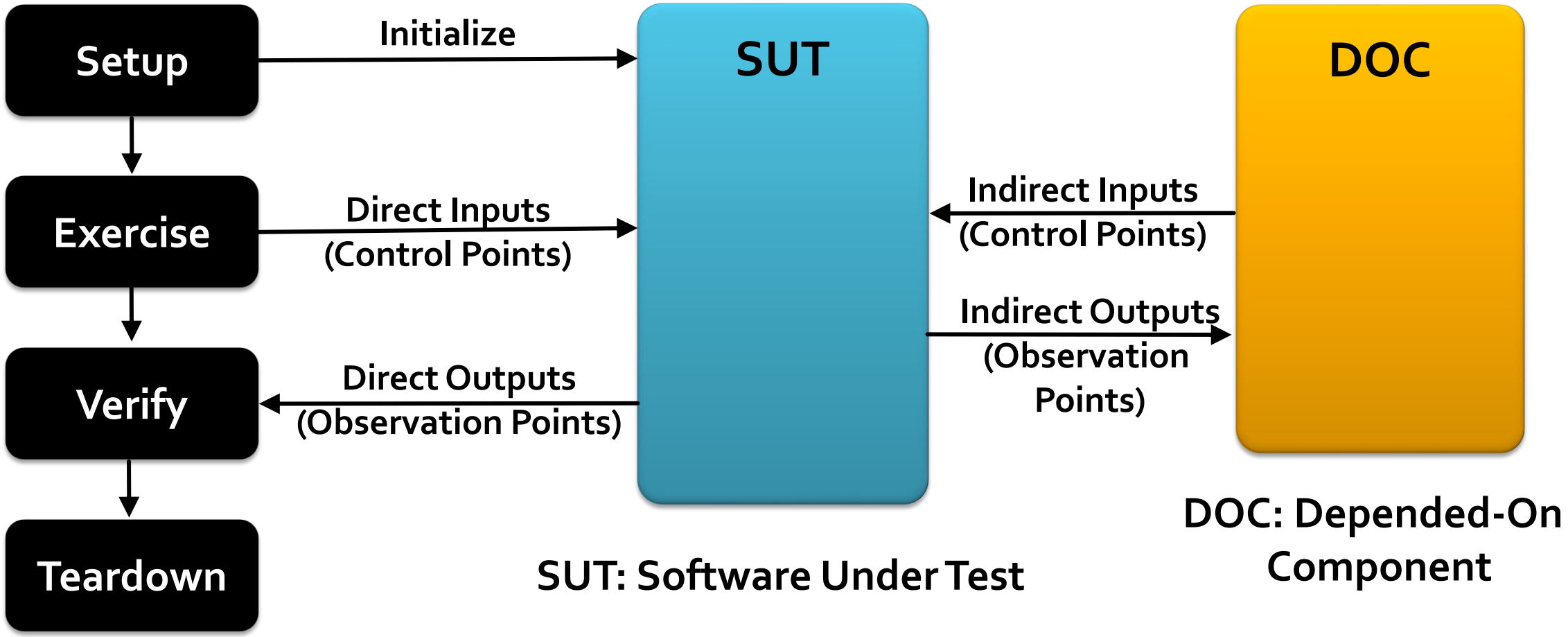
xUnit History



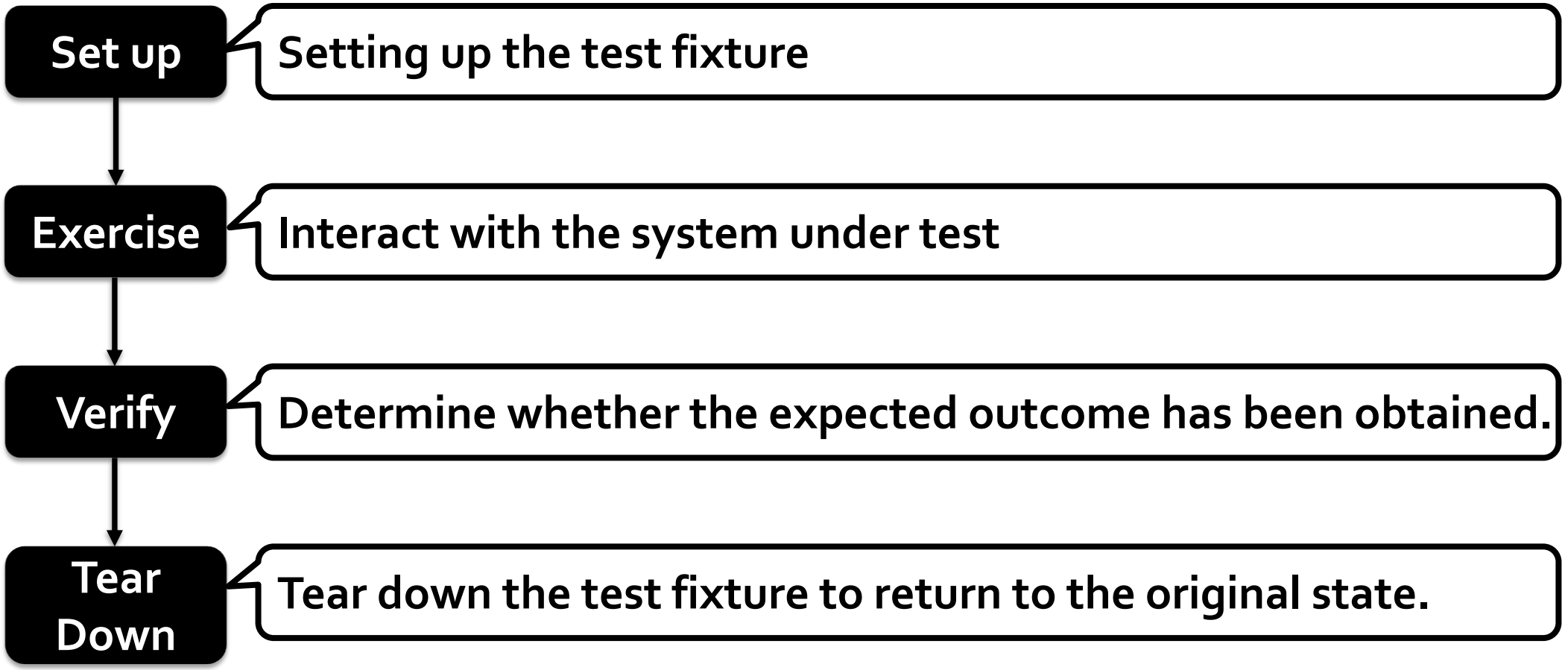
Kent Beck (left) and Erich Gamma(right)

- "Each object should be able to test itself"
- **SUnit** for Smalltalk, highly structured object-oriented style, written by Kent Beck 1988 (published in 1989).
- **JUnit** for Java: SUnit was ported in 1997 to Java by Kent Beck and Erich Gamma, and gained wide popularity.
- Since then unit testing frameworks have been developed for a broad range of computer languages. Those that are variation on SUnit are collectively called **xUnit.**
  - A list of test frameworks http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

A Holistic View of xUnit



Test Phases


Set up
Setting up the test fixture
Exercise
Interact with the system under test
Verify
Determine whether the expected outcome has been obtained.
Tear Down
Tear down the test fixture to return to the original state.

Suppose we want to test the following calculator python program calc.py:

```
def add(x, y):
        """Add Function"""
        return x + y

def subtract (x, y):
        """Subtract Function"""
        return x - y

def multiply (x, y):
        """multiply Function"""
        return x * y

def divide (x, y):
        """divide Function"""
        if y == 0:
            raise ValueError('Can not divide by zero!')
        return x / y
```

A PyUnit (unittest) test for calc.py

```
0  import unittest
   import calc

1  class TestCalc(unittest.TestCase):

2      def setUp(self):
           print ('setUp. I can setup a database here')

       def tearDown(self):
           print ('tearDown\n')

4      def test_add(self):
           print ('test_add')                    5
           self.assertEqual(calc.add(10,5), 15)
6          self.assertEqual(calc.add(-1,1), 0)
           self.assertEqual(calc.add(-1,-1), -2)

4      def test_divide(self):
         print ('test_divide')                   5
6        self.assertEqual(calc.divide(10,5), 2)
         with self.assertRaises(ValueError):
              calc.divide(10, 0)

3  if __name__ == '__main__':
       unittest.main()
```

0 Link to the software under test, and the test library

1 Create a new test case by subclassing unittest.TestCase

2 setup () is executed prior to test, and teardown () is invoked after test.

3 A command-line interface to test script.

4 Define the test scripts, naming convention test_***()

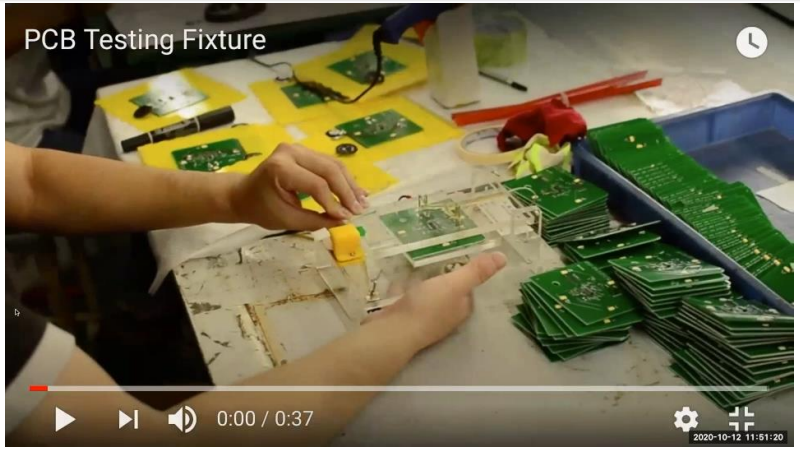
5 Exercise on the method under test

6 Evaluate test results:
assertEqual (): check for an expected result.
assertRaises (): verify that an expected exception gets raised.

Test Fixture

Test Fixture in General

- Electronics
  - Printed Circuit Board testing fixture



  - In testing electronic equipment such as circuit boards, a test fixture is a device or setup designed to hold the device under test in place and allow it to be tested by being subjected to controlled electronic test signals.

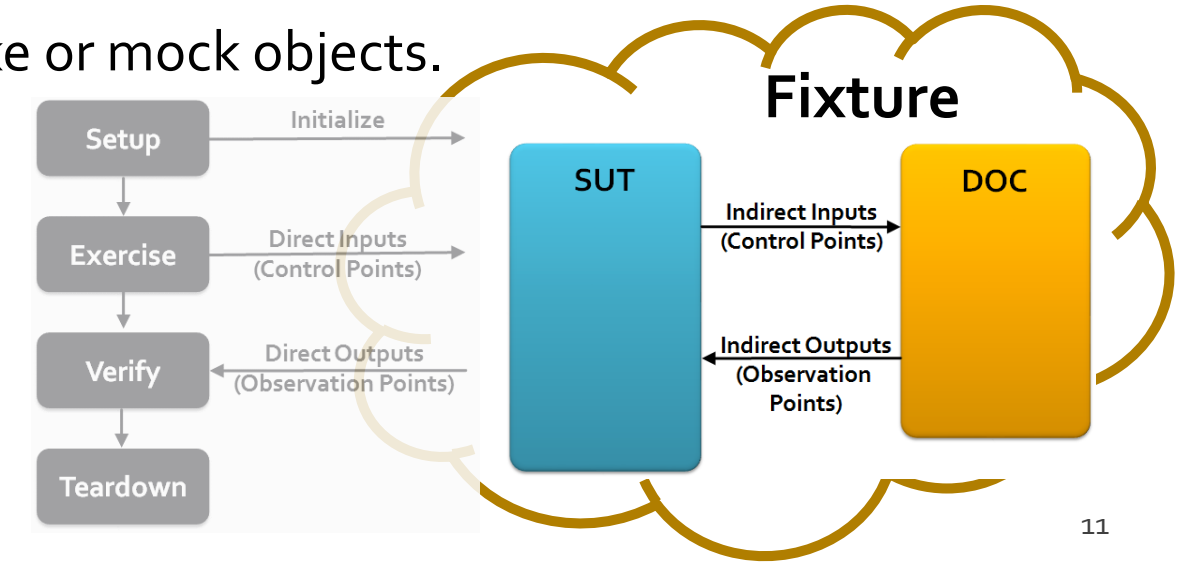
A functional test fixture is a complex device to interface the DUT (Device Under Test) to the ATE (Automatic Test Equipment).

- Physical testing
  - A physical fixture is a device or apparatus to hold or support the test specimen during the test. The influence of test fixture on test results is important.

Test structure fixtures on large seismic table for earthquake tests.

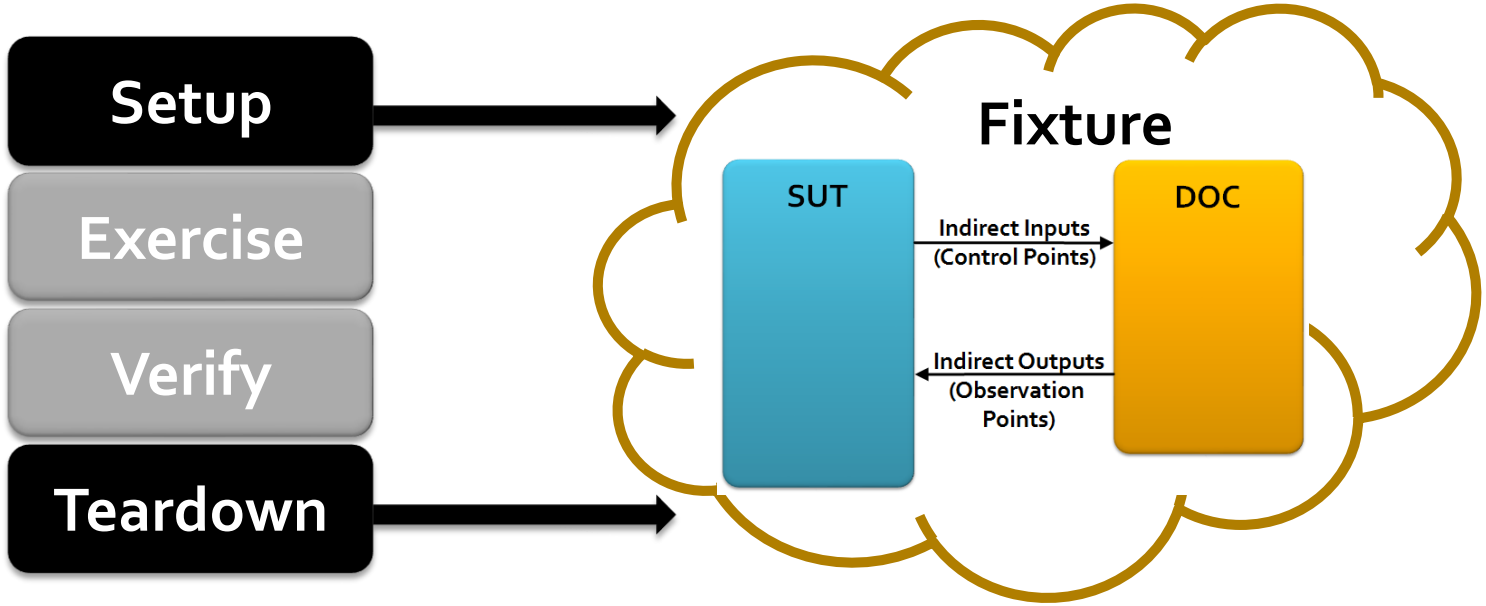
Software Test Fixture

- Everything we need to have in place to exercise the SUT.
- A fixed state of the software under test used as a baseline for running tests. It may also refer to the actions performed in order to bring the system into such state. For examples
  - Loading a database with a specific, known set of data
  - Erasing a hard disk and installing a known clean operating system installation
  - Copying a specific known set of files
  - Preparation of input data and set-up/create of fake or mock objects.
- In xUnit
  - **Set up**: to create the expected state for the test
  - **Tear down**: to clean up what had been set up.

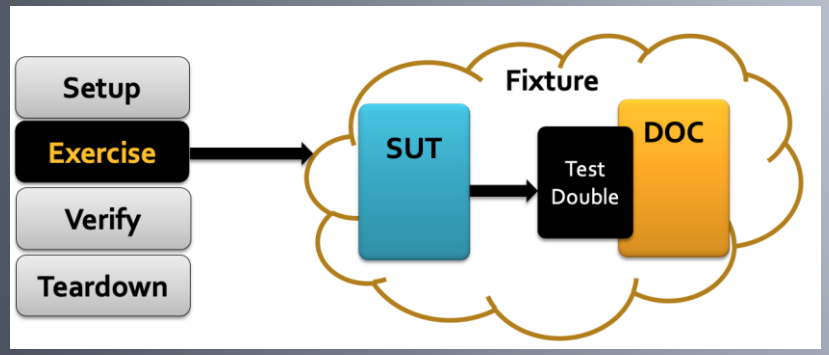


Fresh Fixture Strategy

A large part of making tests repeatable and robust is ensuring that the test fixture is torn down after each test and a new one created for the next test run.



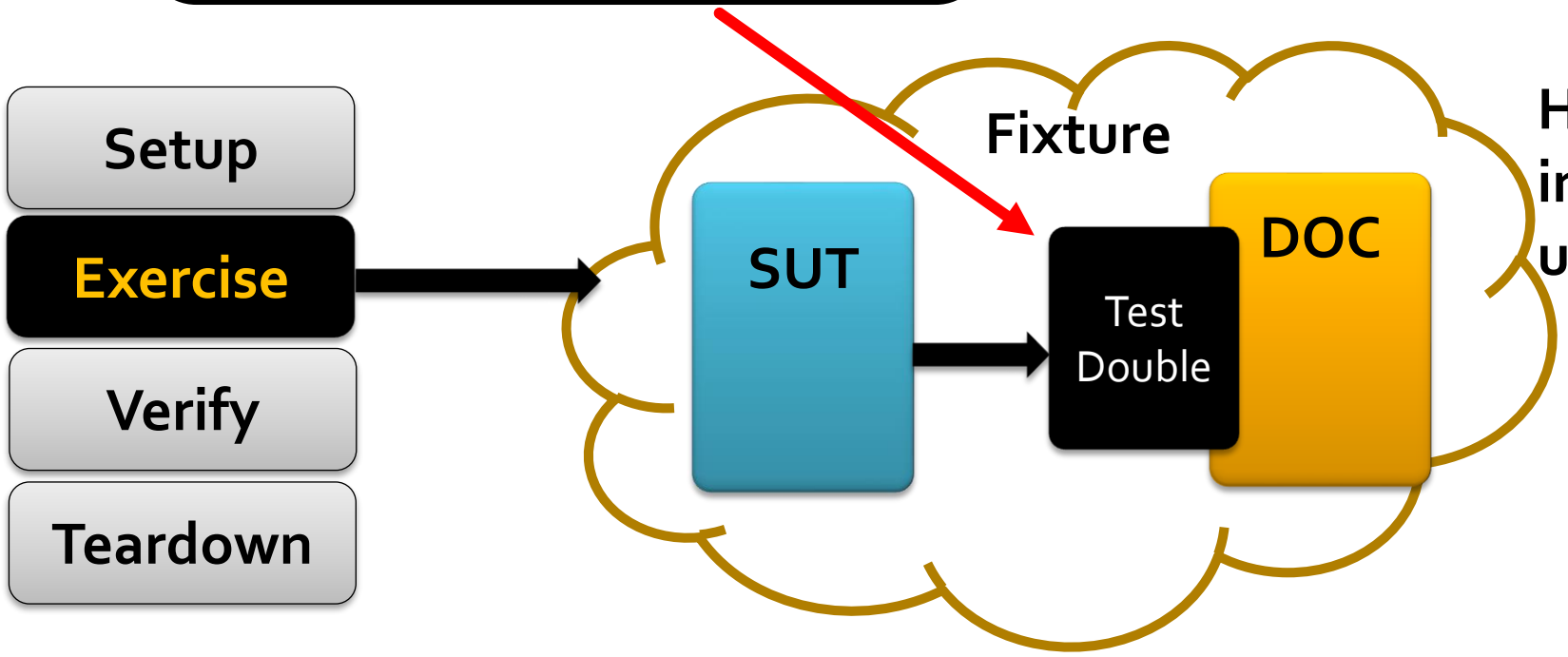
Test Doubles



Test Double

We replace a component on which the SUT depends with a "test-specific" equivalent.

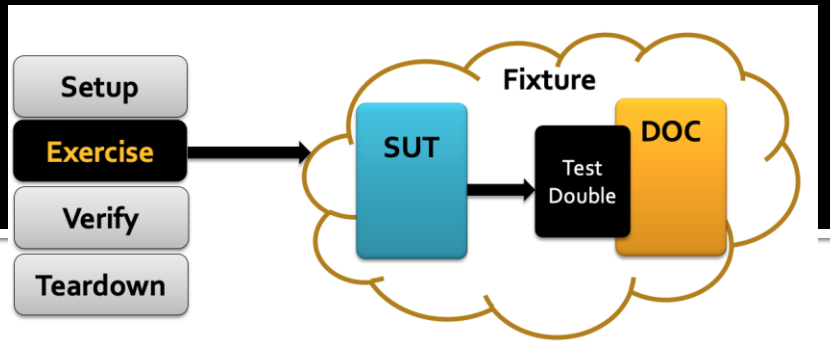
The *Test Double* doesn't have to behave exactly like the real DOC; it merely has to provide the same API as the real DOC so that the SUT thinks it is the real one!



How can we verify logic independently when the DOC is unusable?

- DOC unavailable yet
- Can not return the results needed for the test yet
- Executing them would have undesirable side effects

Implementation Alternatives



Test Double

Dummy Object

passed around but never actually used. Usually they are just used to fill parameter lists

Test Stub

To provide indirect inputs

Test Spy

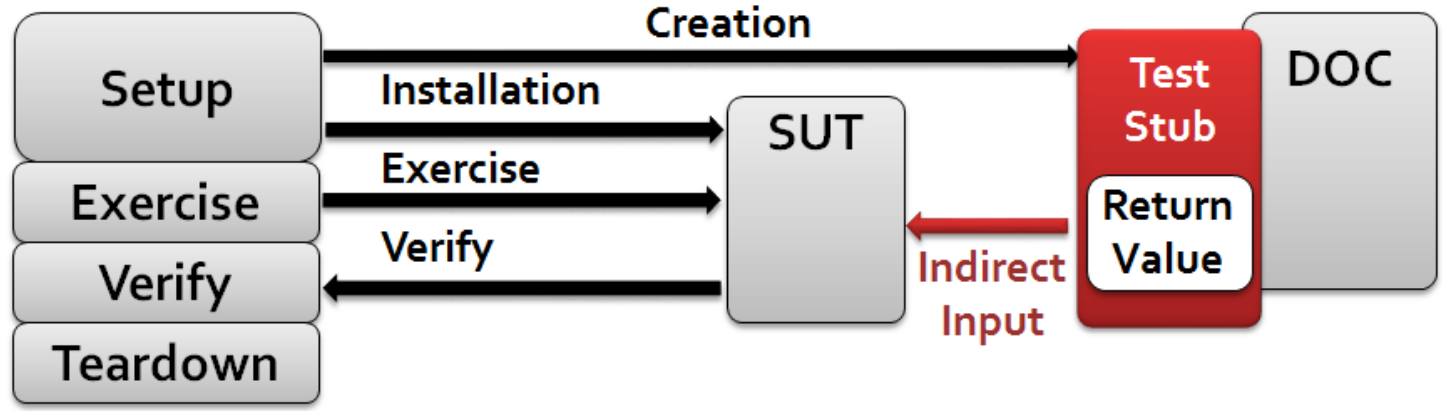
Mock Object

To verify indirect outputs

Fake Object

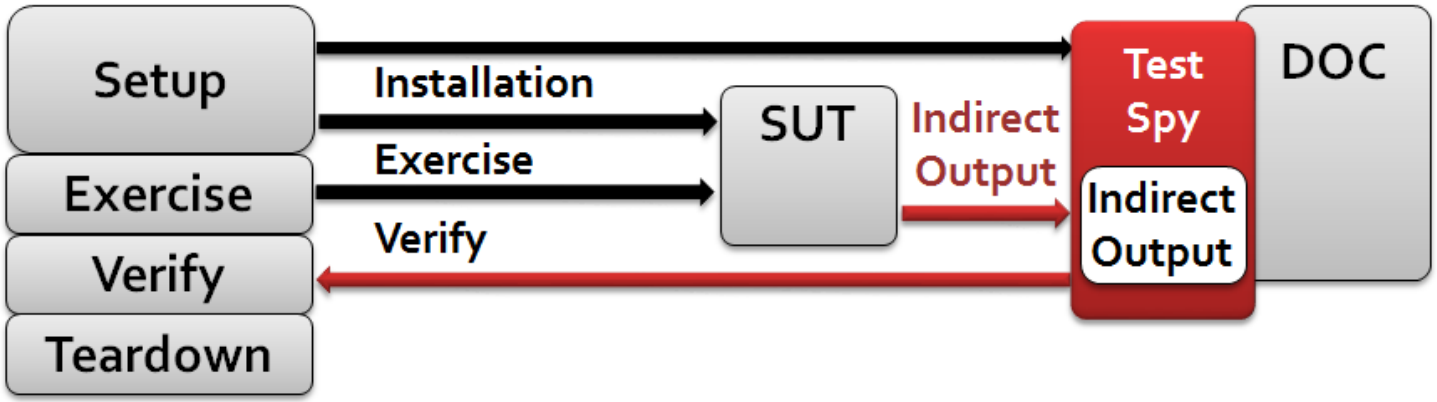
actually have working implementations, but usually take some shortcut which makes them not suitable for production (an **in memory hash table** is a good example)

Test Stub



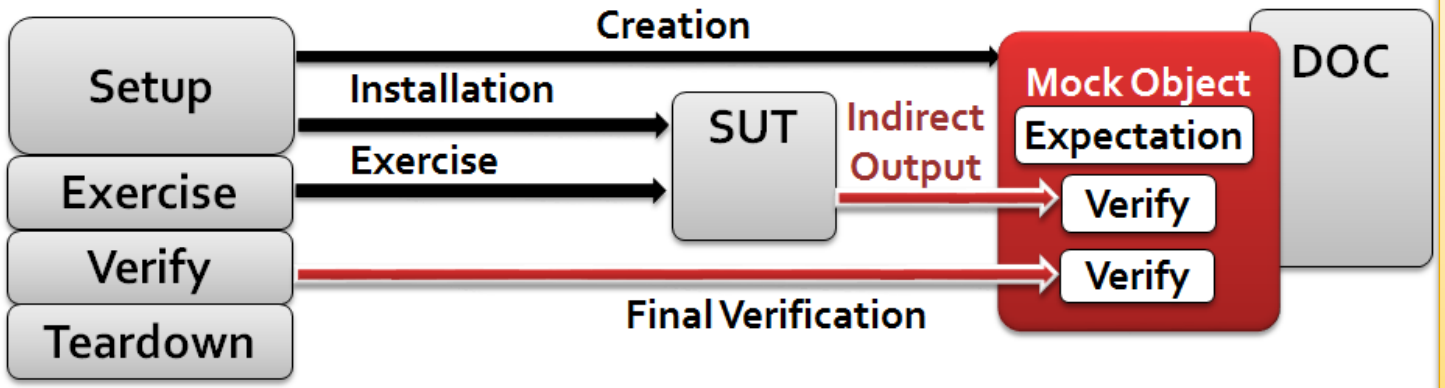
A test stub is used to replace DOC on which the SUT depends so that the test has a control point for the indirect inputs of the SUT

Test Spy



To capture the indirect output calls made to another component by the SUT for later verification by the test.

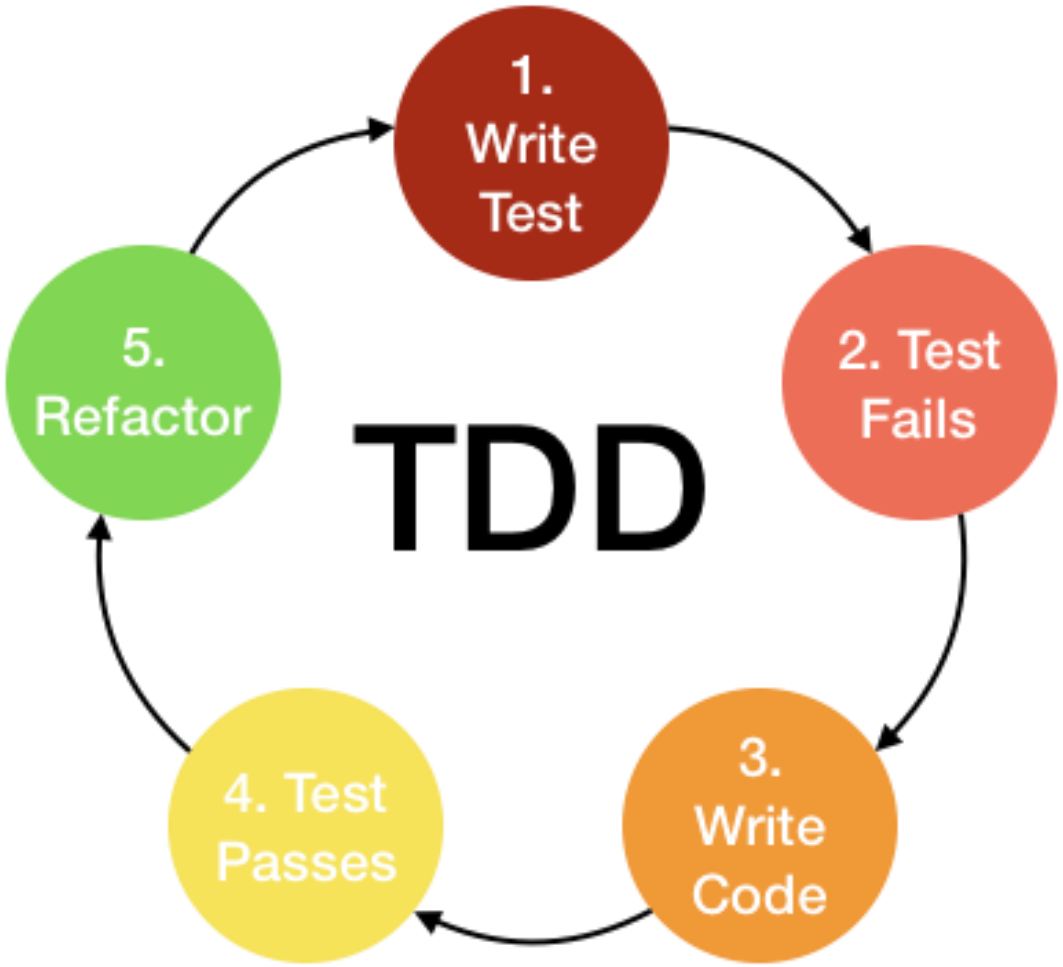
Mock Object



To replace an object on which the SUT depends on with a test-specific object that verifies it is being used correctly by the SUT , which form a specification of the calls they are expected to receive.

Test Driven Development (TDD)

Test Driven Development (TDD) Cycle



- Each new feature begins with writing a test.
  - It makes the developer focus on the requirements ***before*** writing the code.
- Refactor code after new code passes the test
  - Move code from where it was convenient for passing the test to where it logically belongs.
  - Remove duplications

The Three Laws of TDD [Uncle Bob]

- You are not allowed to write any production code unless it is to make a failing unit test pass.
- You are not allowed to write any more of a unit test than is sufficient to fail; and compilation failures are failures.
- You are not allowed to write any more production code than is sufficient to pass the one failing unit test.

Comments on TDD

- “TDD is hard at first, but it gets easier”
- “Was great for quickly noticing bugs [regression] and made them easy to fix”
- “Helped me organize my thoughts as well as my code” [The code you wish you had]
- “We didn’t always test before pushing, and it caused a lot of pain”
- “Wish we had committed to it earlier & more aggressively”

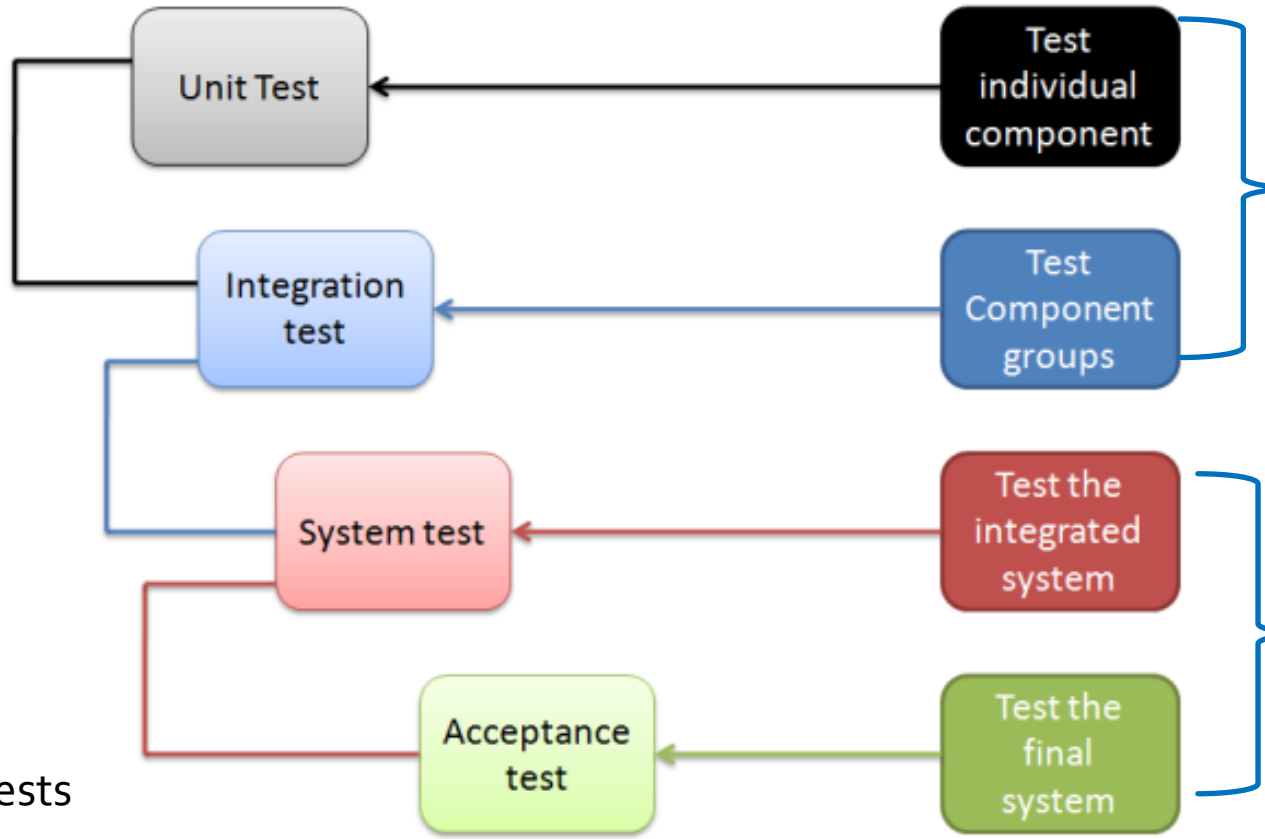
Outline

- Software Quality Assurance
- Basic Testing Concepts
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Other types of Testing

Levels of Software Testing

(Each Test)
shorter run time, fewer cost, less dependency

More Tests



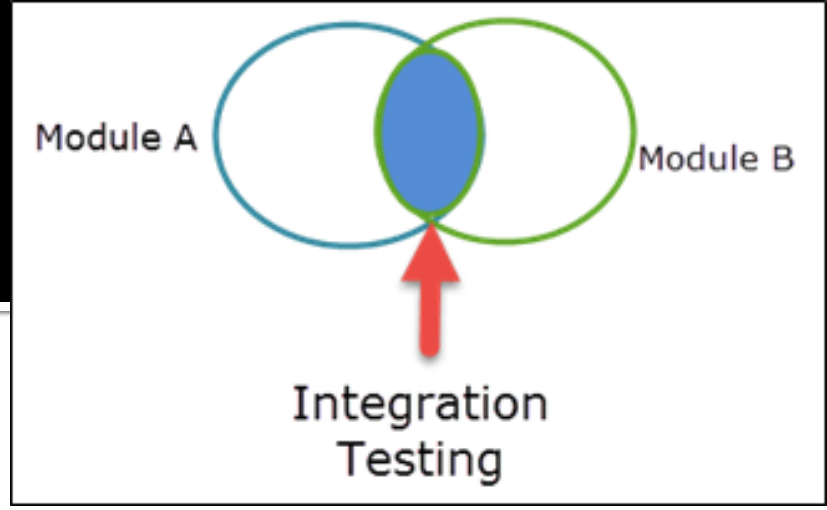
White-Box Testing (code-based) or Black-Box Testing (interface/specification-based)

Black-Box Testing (interface/specification-based)

(Each Test)
longer run time, more cost, more dependency

Less Tests

Integration Testing



- The problems with "Interfacing"
  - One module can have an inadvertent error, the adverse affect on another
  - Sub-functions, when combined, may not produce the desired major function
  - Individually acceptable imprecision may be magnified to unacceptable levels
  - Global data structures can present problems
  - Data can be lost across an interface
  - .....

"If they all work individually, why do you doubt that they'll work when we put them together?"

Integration Testing

"NASA lost a $125 million Mars orbiter because a Lockheed Martin engineering team used English units of measurement while the agency's team used the more conventional metric system for a key spacecraft operation."

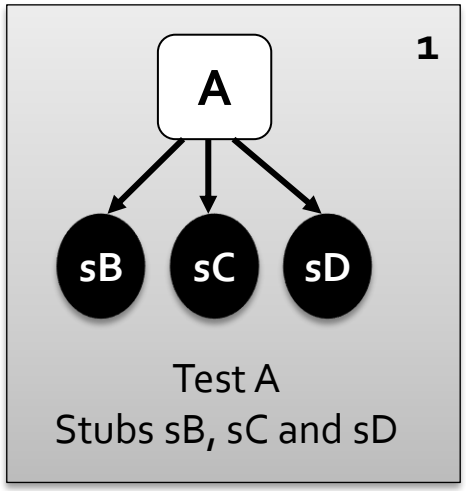
NASA's Climate Orbiter
was lost September 23, 1999

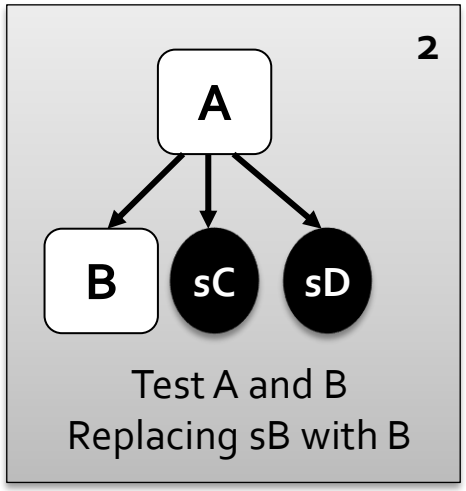
"Metric mishap caused loss of NASA orbiter",
CNN news, Sep. 30, 1999

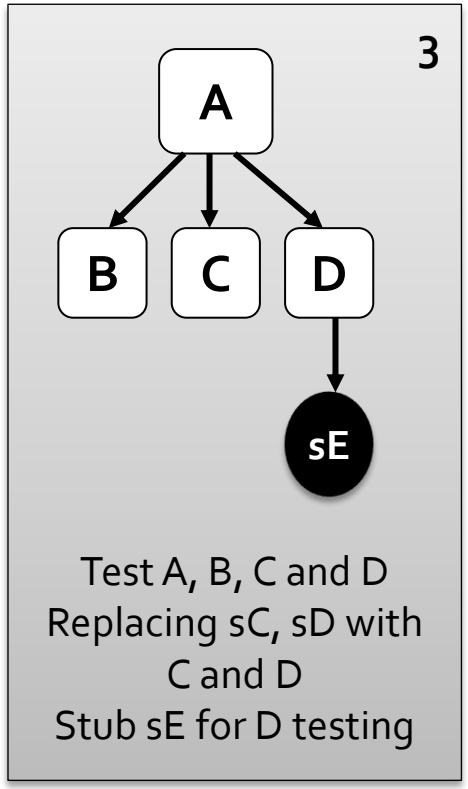
Integration Testing Approaches

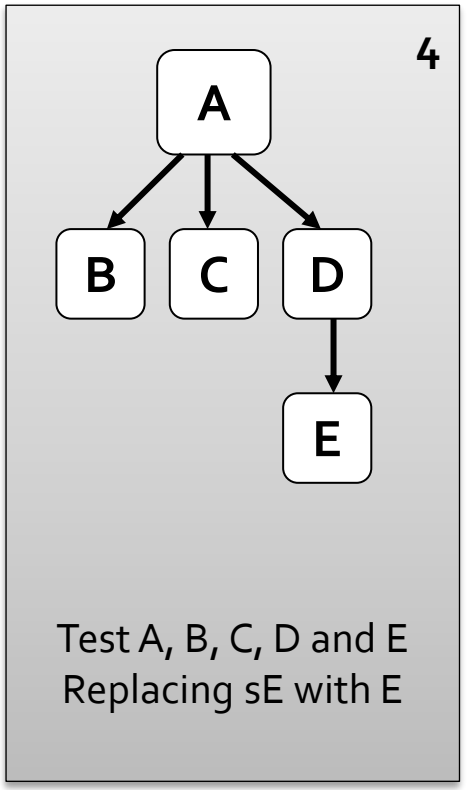
- The "Big Bang"
  - All components are combined and tested as a whole.
  - May encounter multiple failure which are hard to isolate and to identify the defects.
- Incremental integration
  - Program is constructed and tested in small increments
    - Based on functional decomposition
    - Based on call diagram
    - Based on usage scenario
  - Functional decomposition
    - Top down
    - Bottom up
    - Sandwich

Top-Down Integration









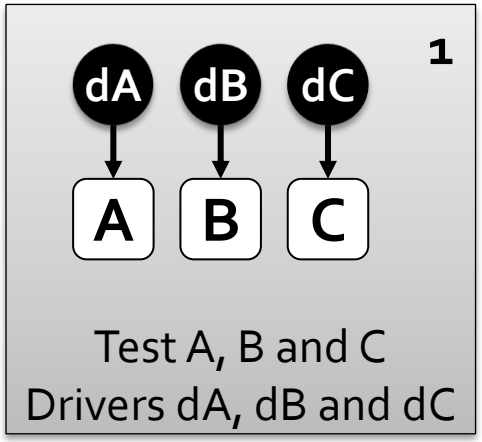
- Starts with the high-level components and works down through control hierarchy
- Test Stubs
  - Represent the interfaces of subordinate programs
- Two ways to expand the structure
  - Depth-first: integrate all components on a major control path.
  - Width first: incorporates all components directly subordinate at each level.

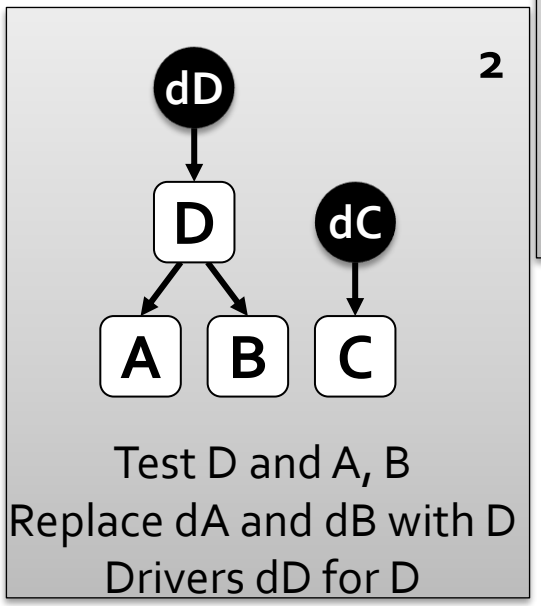
Top-Down Integration Pros and Cons

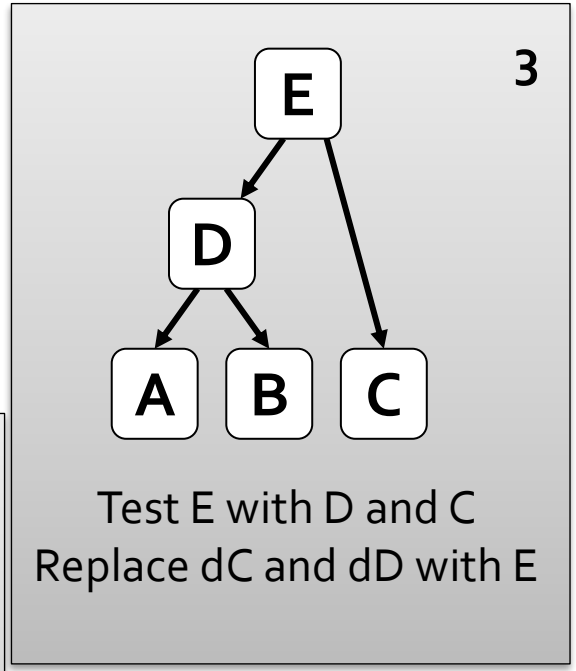
- Pros
  - Verifies major control or decision early in the test process
  - Allows early recognition of major problems

- Cons
  - Writing stubs is difficult: Stubs must allow all possible conditions to be tested.
  - Large number of stubs may be required, especially if the lowest level of the system contains many methods.
  - Some interfaces are not tested separately.

Bottom-Up Integration







- Starts from the low-level module testing and progress up the hierarchy
- Test drivers
  - A control program for testing to coordinate test cases input and output.

Bottom-Up Integration Pros and Cons

- Pros
  - No stubs needed
  - Operational modules tested thoroughly
  - Useful for integration testing of the following systems
    - Real-time systems
    - Systems with strict performance requirements.
- Cons:
  - Tests the important subsystems last, verifying control logic late
  - Drivers needed

Outline

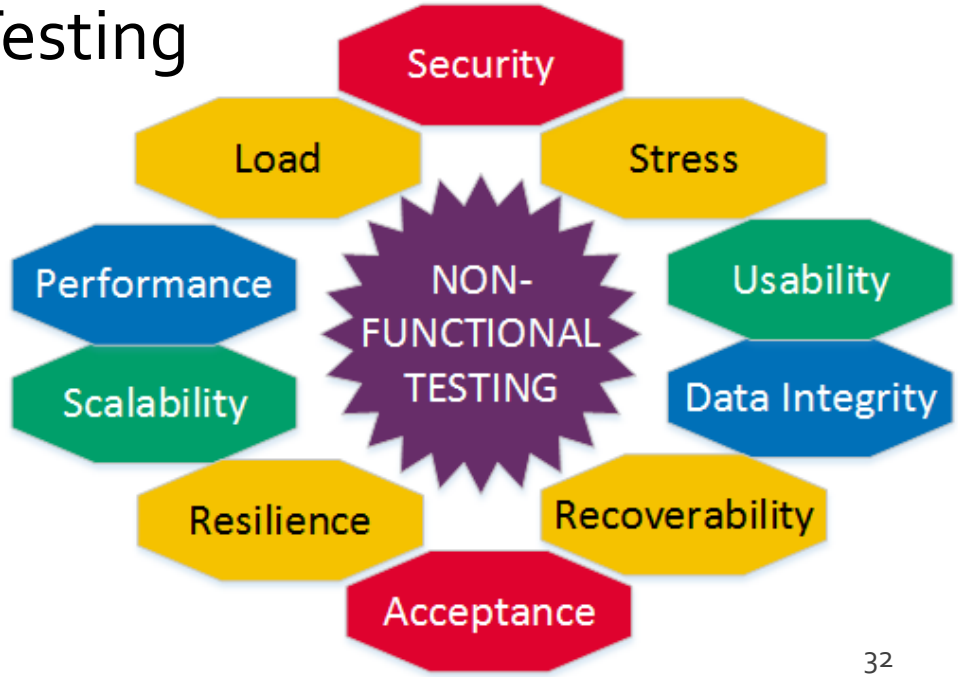
- Software Quality Assurance
- Basic Testing Concepts
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Other types of Testing

System Testing

System Testing

- A systematic process to fully test the computer-based system
  - Software is only one element of a larger computer-based systems.
  - Software is ultimately incorporated with other parts which fall outside the scope of the software process and are not conducted solely by software engineers.
  - Verify and validate the system from various perspectives.
- Functional Testing vs Property (non-functional) Testing

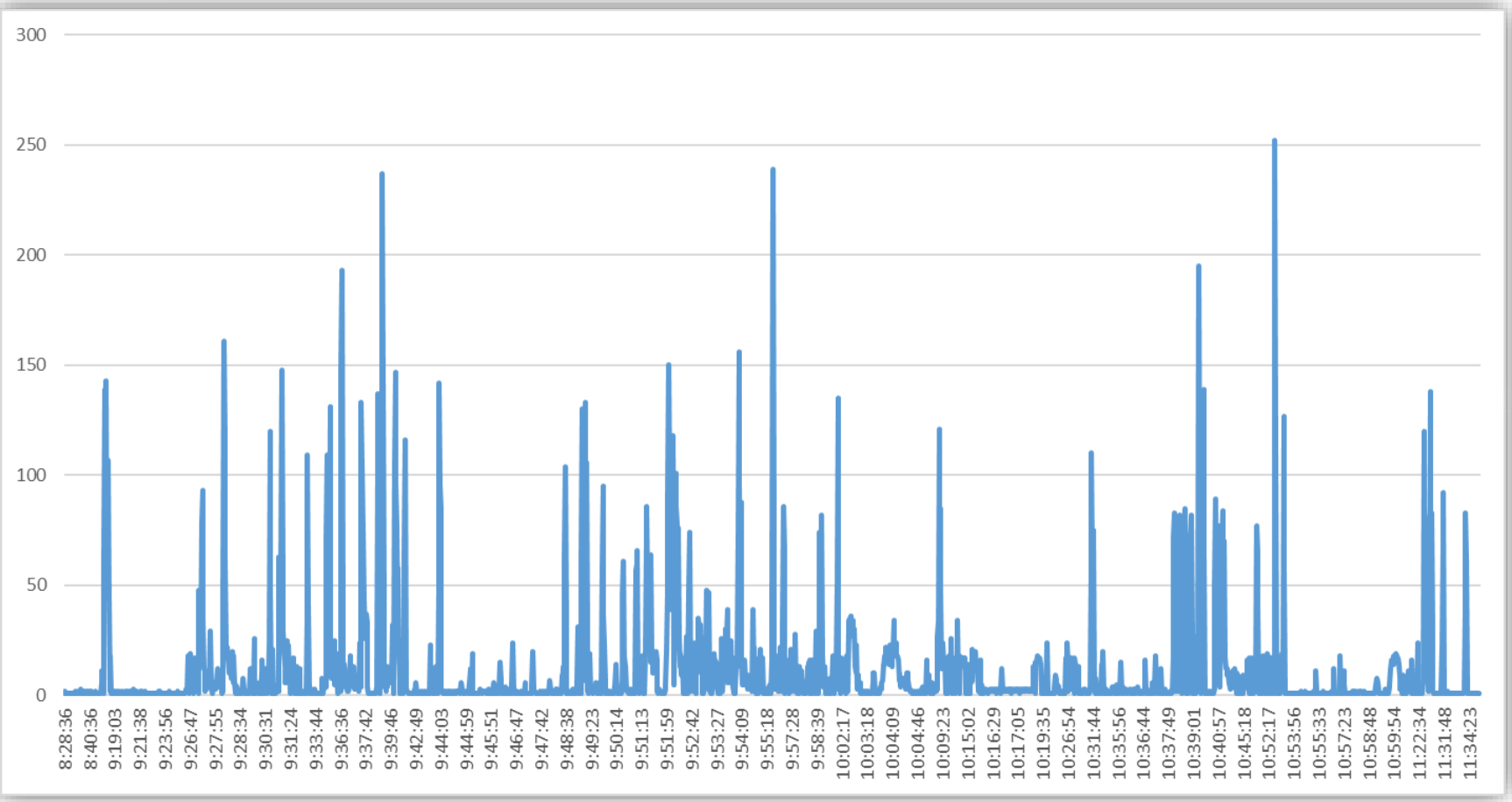
FUNCTIONAL	NONFUNCTIONAL
WHAT a system does	HOW *well* the system does it within design & resource constraints



Performance Testing

General Dashboard - Overall Analyzed Requests

Total Requests	24066	Unique Visitors	114	Referrers	59	Log ...	6.56 MiB
Failed Reques...	0	Unique Files	874	Unique 404	45	Ban...	434.89 MiB
Generation Ti...	1	Static Files	197	access.log			

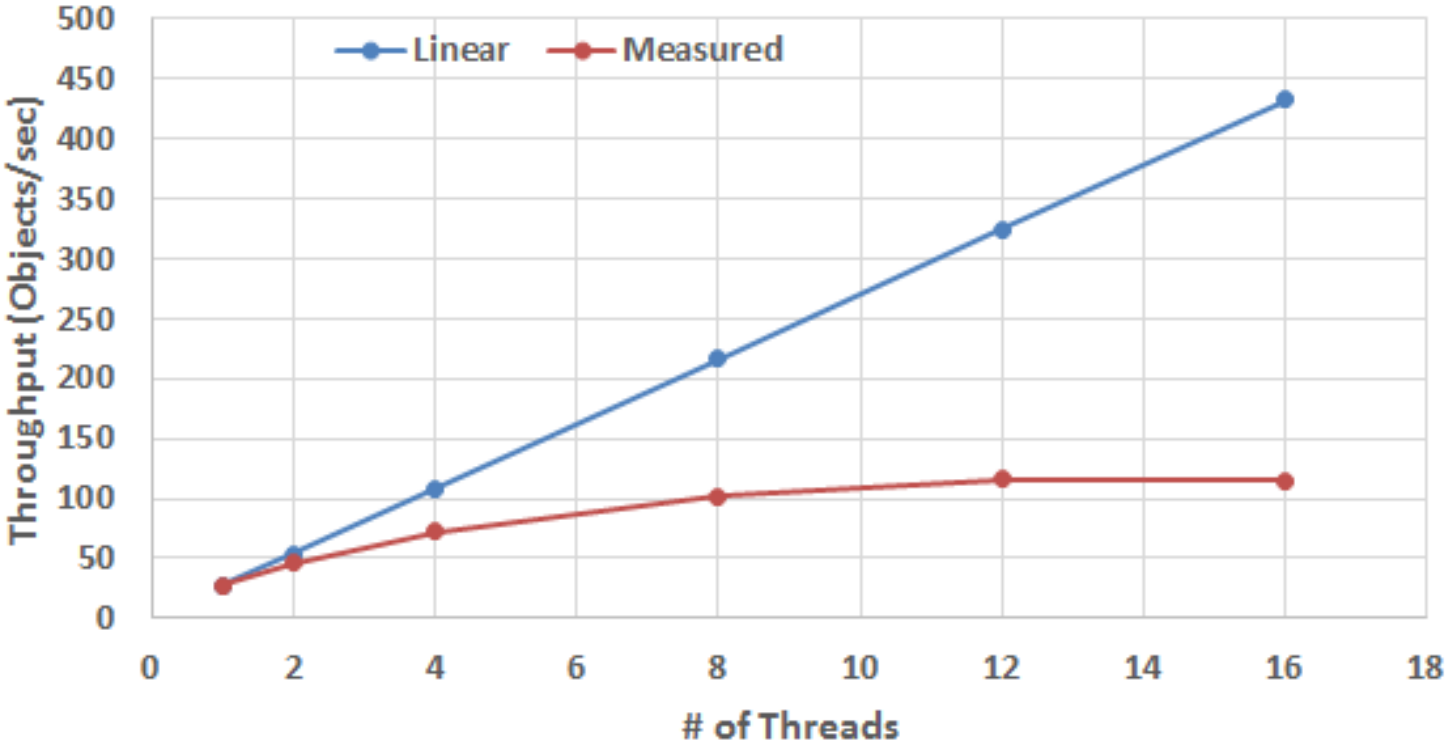


System and Software Performance

- Factors that have the most impacts on system performance
  - The raw capabilities of the underlying **hardware** platform
  - The maturity of the underlying **software** platform
  - The **application** design and implementation
  - 3rd party APIs
- Even well-designed software may not perform and scale optimally on fast hardware without going through a full cycle of optimized and tuning for the best possible performance and scalability.
- Understanding the performance of a software system quantitatively requires a good understanding of how a software program is developed **inside out**.

Performance impacted by Application Design: e.g., Multithreading

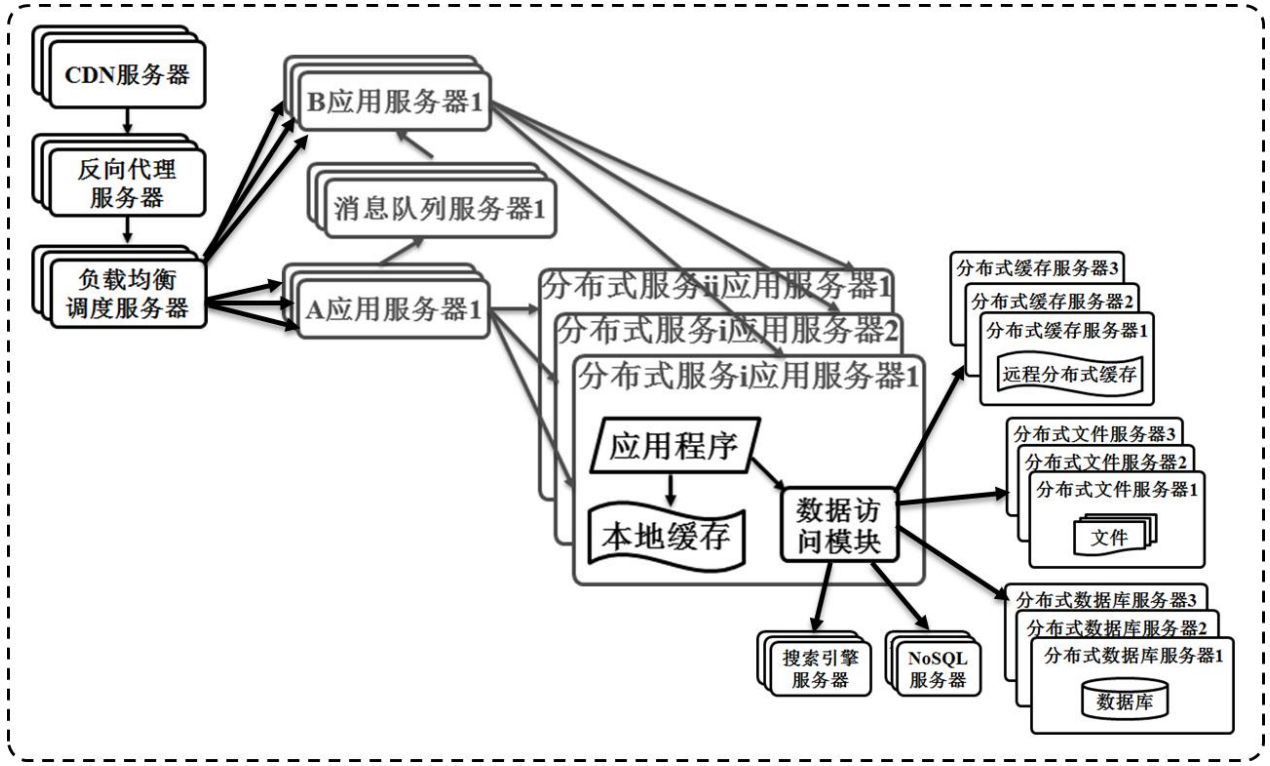
- To allow multitasks to be executed simultaneously or in parallel on computer hardware with multiple CPUs.

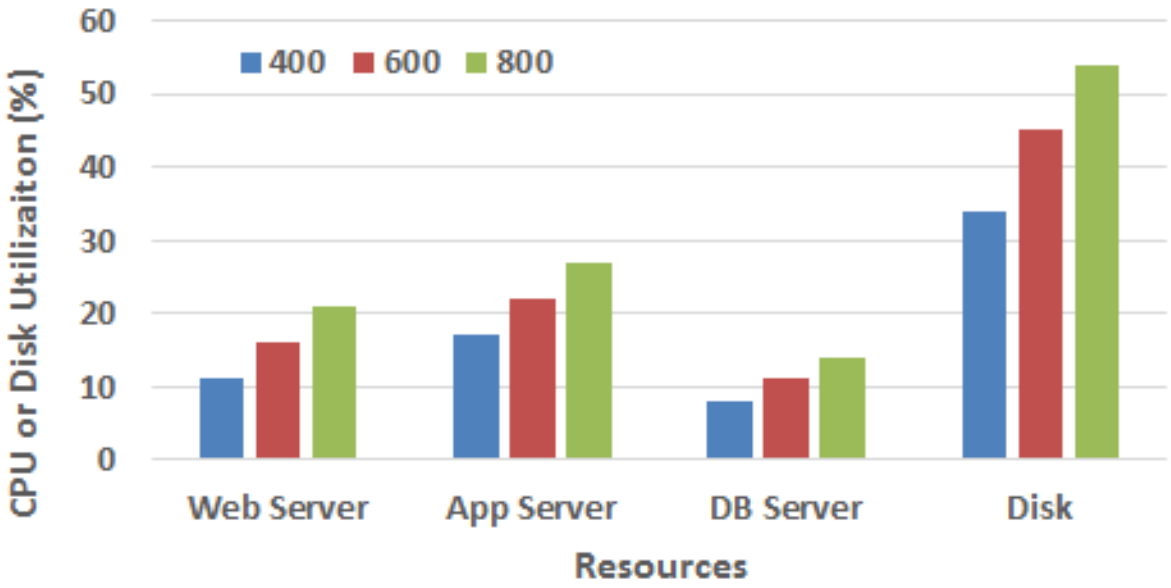


- The measured throughput does not scale up linearly with the number of threads
- No substantial gain in throughput was observed beyond 8 threads
- The 8-threaded throughput is about a factor of 4 improvement compared with single threaded throughput.

The # of CPU cores

Why performance testing?
Enterprise Software Architecture

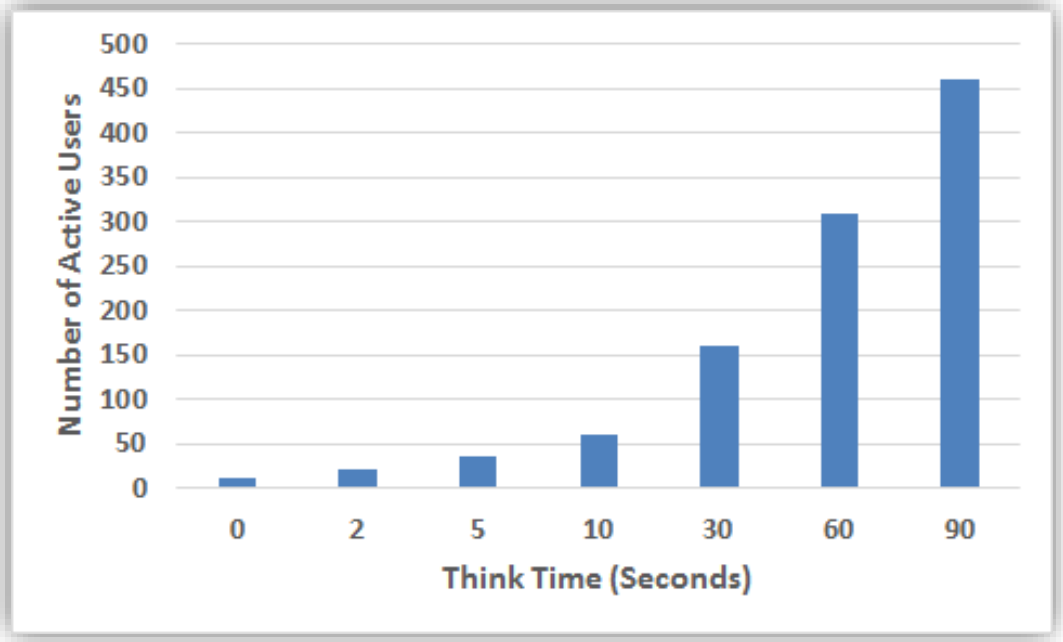




Resource Utilization with an increasing number of users.

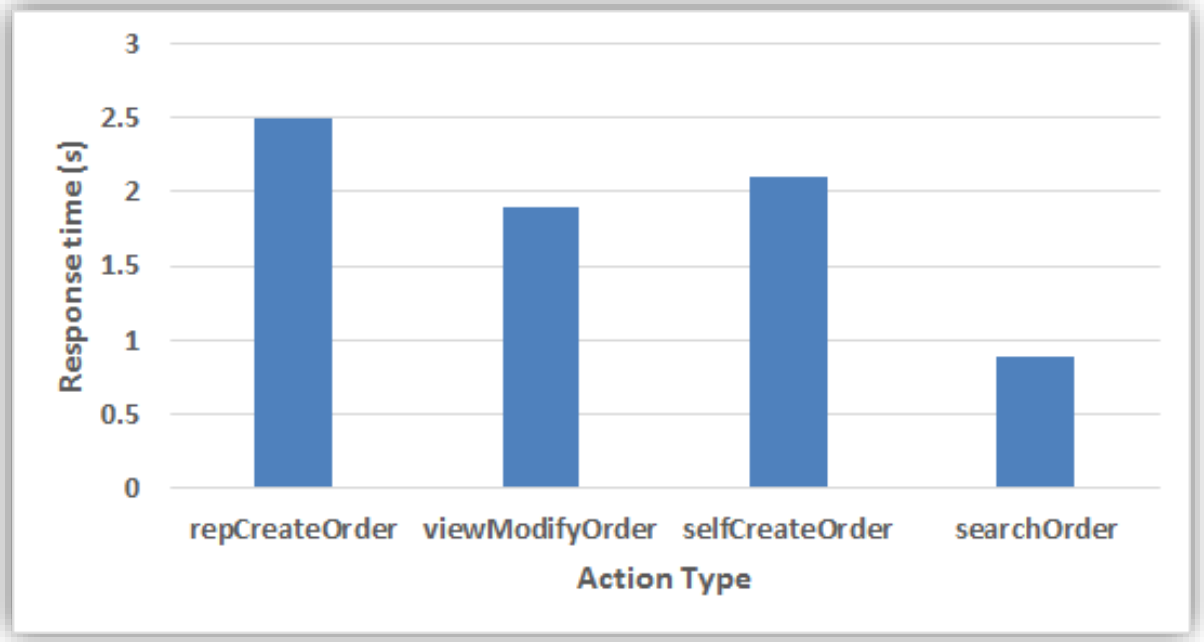
Why performance testing? Enterprise Software Architecture

- OLTP (Online-Transaction Processing) Performance



Active User versus think time

Number of active users that an OLTP system can support with think times varying from 0 to 90 seconds.



Per User Action Type

Average response times for each user action.

Performance Testing

- Load Testing
  - To understand the behavior of the system under a specific **expected** load.
    - The expected concurrent number of users on the application performing a specific number of transactions within the set duration.
- Stress Testing
  - To understand the upper limits of capacity within the system.
  - To determine the system's robustness in terms of **extreme** load.
- Soak Testing (Endurance Testing)
  - To determine if the system can **sustain** the continuous expected load.
    - Memory leaks, performance degradation

Performance Testing

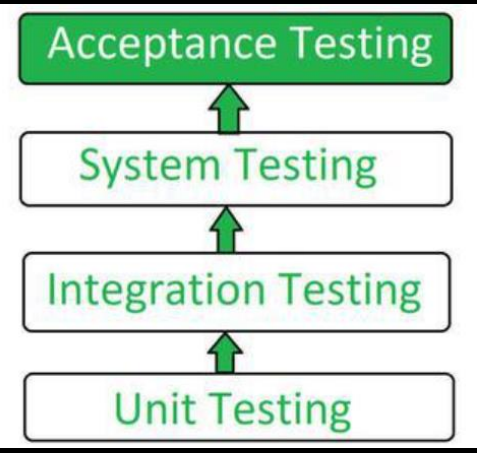
- Spike Testing
  - Spike testing is done by suddenly increasing the load generated by a very large number of users, and observing the behavior of the system.
  - The goal is to determine whether performance will suffer, the system will fail, or it will be able to handle **dramatic changes** in load.
- Configuration Testing
  - To determine the effects of configuration changes to the system's components on the system's performance and behavior.
    - e.g. load-balancing strategy

Outline

- Software Quality Assurance
- Basic Testing Concepts
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Other types of Testing

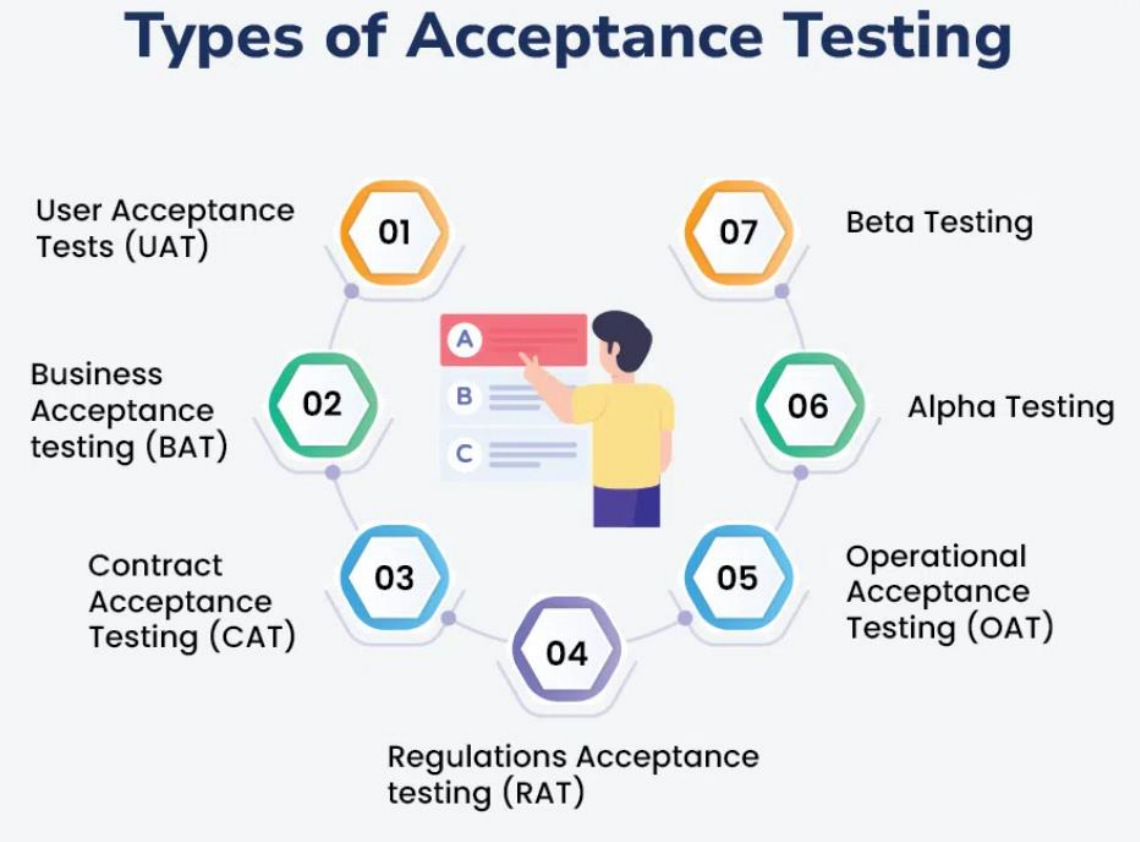
Acceptance Testing

- Formal testing according to user needs, requirements, and business processes conducted to determine whether a system satisfies the acceptance criteria and to enable the user, customers or other authorized entity to determine whether to accept the system.
  - — *Standard Glossary of Terms used in Software Testing*
  - — Acceptance Testing is the last phase of software testing performed after System Testing and before making the system available for actual use.



Acceptance Testing

- **Functional acceptance testing** concentrates on verifying that the software system satisfies the prescribed functional requirements
- **Non-functional acceptance testing** focuses on validating the non-functional aspects related to the quality of the software system



Note: In some references, alpha, beta, operational, contract, regulation, and business acceptance testing make up User Acceptance Test

User acceptance testing (UAT)

- User acceptance testing (UAT) consists of a process of verifying that a solution works for the user. It is not system testing (ensuring software does not crash and meets documented requirements) but rather ensures that the solution will work for the user (i.e., tests that the user accepts the solution)
  - User acceptance testing is used to determine whether the product is working for the user correctly.
  - Specific requirements which are quite often used by the customers are primarily picked for testing purposes. This is also termed as **End-User Testing**.

Alpha/Beta testing

- Alpha testing is used to determine the product in the development testing environment by a specialized testers team usually called alpha testers
  - **Alpha Testing** is a type of acceptance testing; performed **to identify all possible issues and bugs before releasing the final product to the end users**. **Alpha testing is carried out by the testers (to simulate real users) who are internal employees of the organization**. The main goal is to identify the tasks that a typical user might perform and test them.
- Beta testing is used to assess the product by exposing it to the real end-users, typically called beta testers in their environment
  - Feedback is collected from the users and the defects are fixed. Also, this helps in enhancing the product to give a rich user experience.

Acceptance Testing vs. System Testing

Aspect	System Testing	Acceptance Testing
Purpose	Checks if the software or product meets specified requirements.	Checks if the software meets customer requirements.
Executed by	Only developers and testers.	Testers, stakeholders, and costumers.
Type of Testing	Functional and non-functional testing.	(Primarily) functional and (sometimes) non-functional testing.
Order of execution	Performed before Acceptance Testing.	Performed after System Testing.
Test Cases	Includes both positive and negative test cases.	Generally includes only positive test cases.
Input Types	demo input values that are selected by the testing team.	Actual real-time input values provided by the user.
Bug fixes	The defects found in system testing are considered to be fixed	The defects found in acceptance testing are considered as product failure.

Outline

- Software Quality Assurance
- Basic Testing Concepts
- Unit Testing
- Integration Testing
- System Testing
- Acceptance Testing
- Other types of Testing

Chaos Engineering (混沌工程)

Simian Army Projects

- Chaos Monkey
- Chaos Gorilla
- Chaos Kong
- Janitor Monkey
- Doctor Monkey
- Compliance Monkey
- Latency Monkey
- Security Monkey

The Netflix Simian Army

Chaos Monkey randomly disables our production instances to make sure we can survive this common type of failure without any customer impact. The name comes from the idea of unleashing a wild monkey with a weapon in your data center (or cloud region) to randomly shoot down instances and chew through cables – all the while we continue serving our customers without interruption. By running Chaos Monkey in the middle of a business day, in a carefully monitored environment with engineers standing by to address any problems, we can still learn the lessons about the weaknesses of our system, and build automatic recovery mechanisms to deal with them. So next time an instance fails at 3 am on a Sunday, we won't even notice.

Latency Monkey induces artificial delays in our RESTful client-server communication layer to simulate service degradation and measures if upstream services respond appropriately. In addition, by making very large delays, we can simulate a node or even an entire service downtime (and test our ability to survive it) without physically bringing these instances down. This can be particularly useful when testing the fault-tolerance of a new service by simulating the failure of its dependencies, without making these dependencies unavailable to the rest of the system.

Conformity Monkey finds instances that don't adhere to best-practices and shuts them down. For example, we know that if we find instances that don't belong to an auto-scaling group, that's trouble waiting to happen. We shut them down to give the service owner the opportunity to re-launch them properly.

Doctor Monkey taps into health checks that run on each instance as well as monitors other external signs of health (e.g. CPU load) to detect unhealthy instances. Once unhealthy instances are detected, they are removed from service and after giving the service owners time to root-cause the problem, are eventually terminated.

Janitor Monkey ensures that our cloud environment is running free of clutter and waste. It searches for unused resources and disposes of them.

Security Monkey is an extension of Conformity Monkey. It finds security violations or vulnerabilities, such as improperly configured AWS security groups, and terminates the offending instances. It also ensures that all our SSL and DRM certificates are valid and are not coming up for renewal.

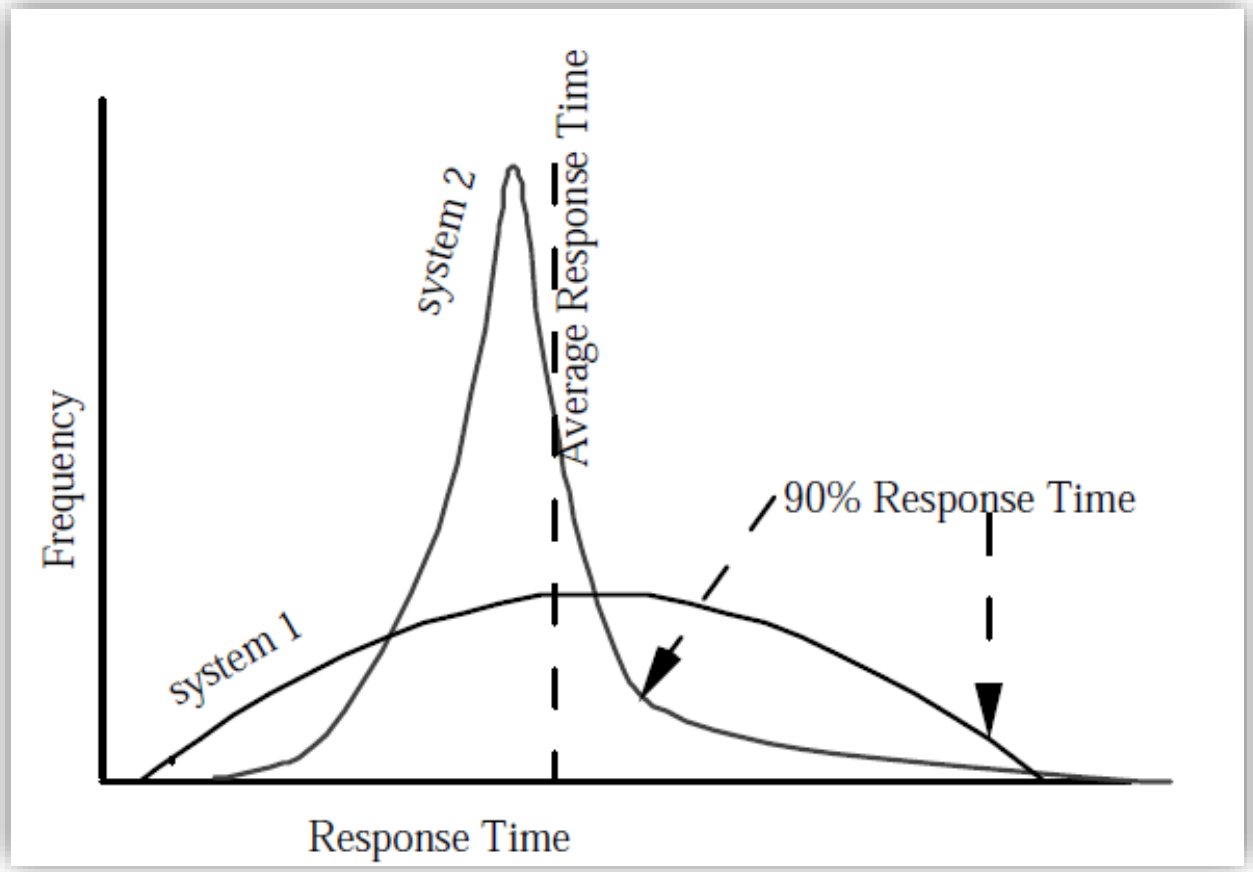
10-18 Monkey (short for Localization-Internationalization, or l10n-i18n) detects configuration and run time problems in instances serving customers in multiple geographic regions, using different languages and character sets.

Chaos Gorilla is similar to Chaos Monkey, but simulates an outage of an entire Amazon availability zone. We want to verify that our services automatically re-balance to the functional availability zones without user-visible impact or manual intervention.

Benchmarking

"In computing, a benchmark is the act of running a computer program, a set of programs, or other operations, in order **to assess the relative performance of an object, normally by running a number of standard tests** and trials against it.

The term 'benchmark' is also mostly utilized for the purposes of elaborately designed benchmarking programs themselves." (wikipedia)



Benchmarking

- SPEC (Standard Performance Evaluation Corporation)
  - Founded in 1988, to evaluate the performance of computer systems.
  - An umbrella organization encompassing four diverse groups
    - Graphics and Workstation Performance Group (GWPG)
    - The High Performance Group (HPG)
    - The Open System Group (OSG)
    - The Research Group (RG)

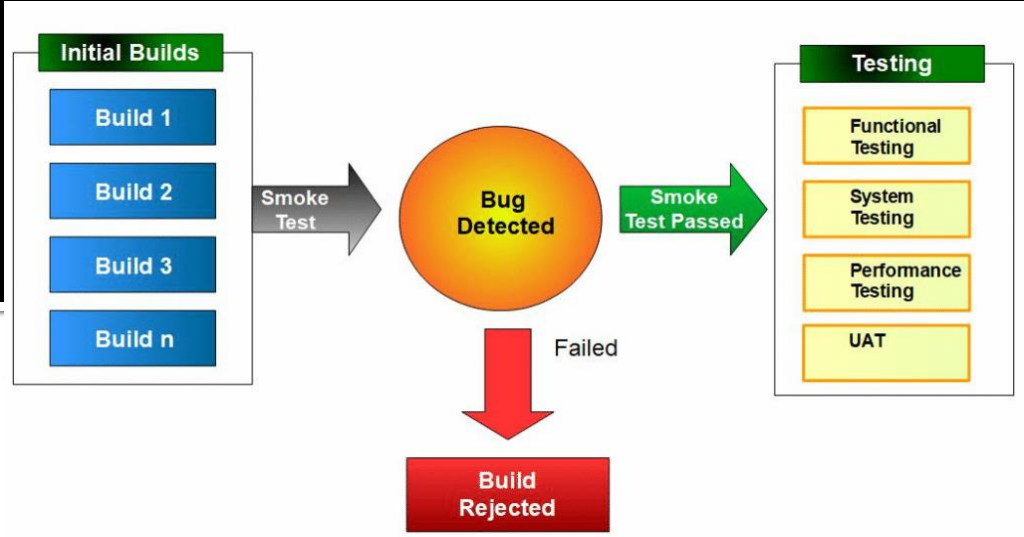
Performance

Price-Performance

Energy-Performance

- Transaction Processing Performance Council (TPC)
  - Founded in 1988, to define transaction processing and database benchmarks.

Smoke Testing (Build Verification Testing)



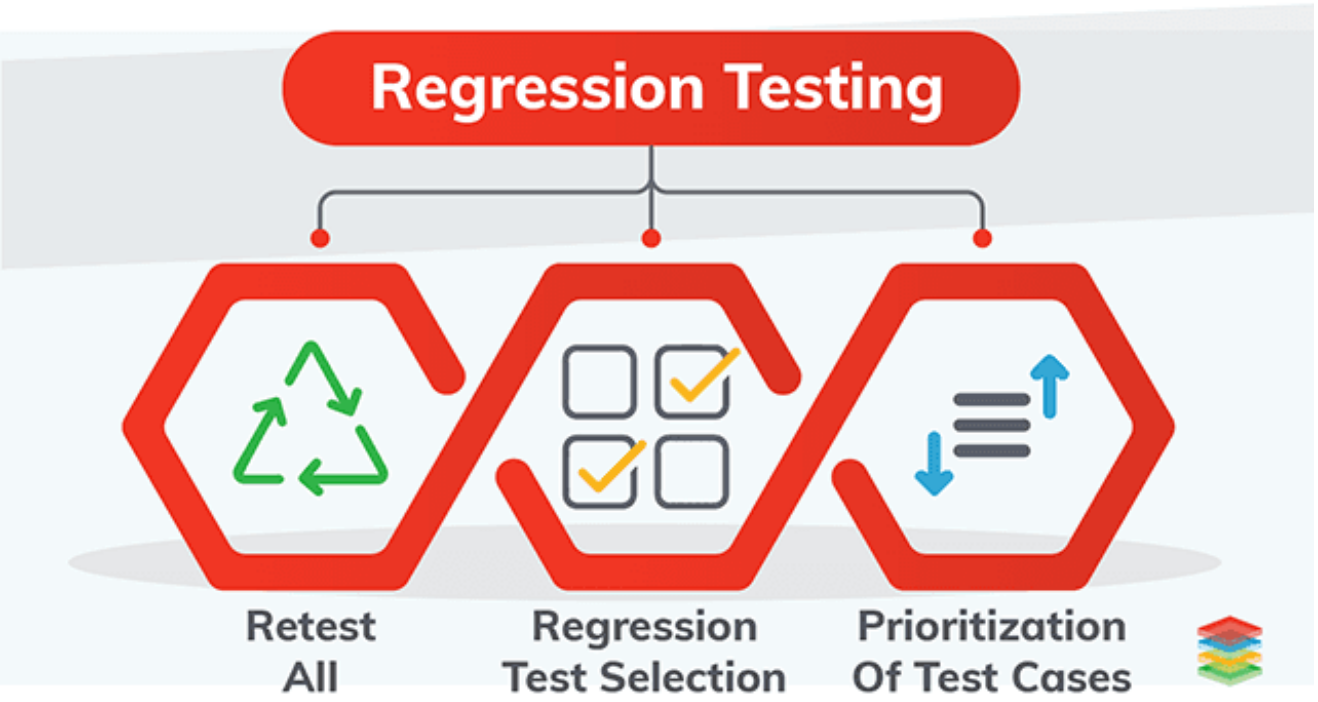
- **preliminary testing** to **reveal simple failures severe enough** to, for example, reject a prospective software release. Smoke tests are a subset of test cases that cover the **most important functionality** of a component or system, used to aid assessment **of whether main functions of the software appear to work correctly**.

- address basic questions like "**does the program run**?", "**does the user interface open**?", or "**does clicking the main button do anything**?" The process of smoke testing aims **to determine whether the application is so badly broken as to make further immediate testing unnecessary**.

- A daily build and smoke test is **among industry best practices**. Microsoft claims that after code reviews, "*smoke testing* is the **most cost-effective method** for identifying and fixing defects in software".

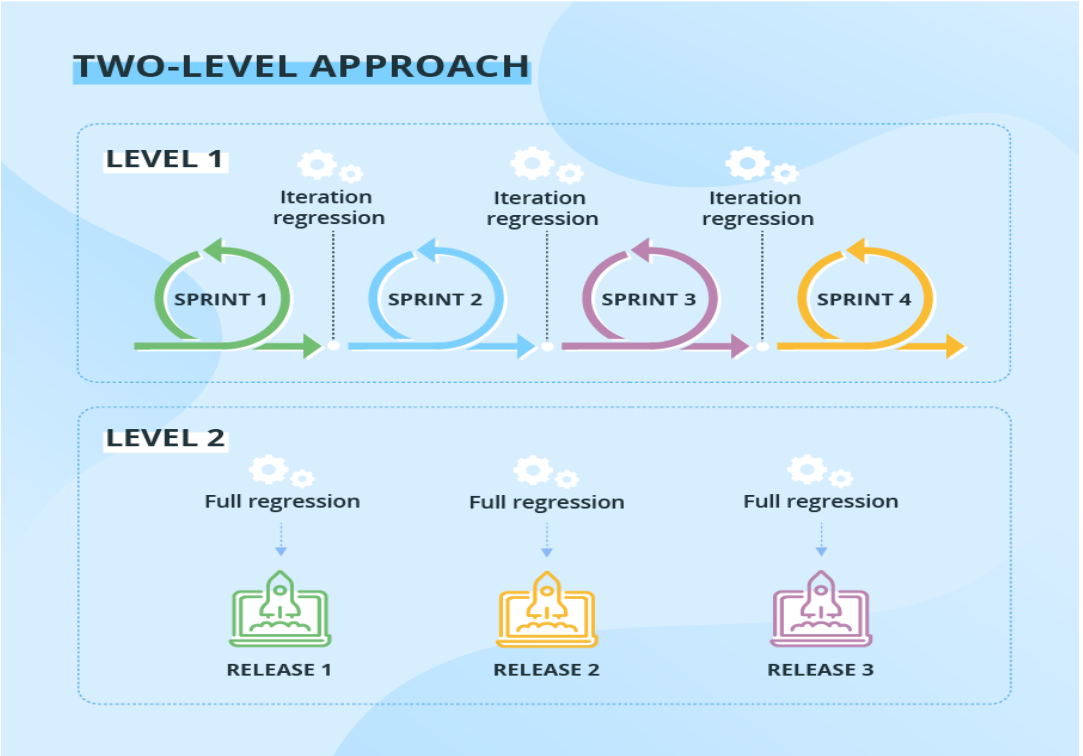
Regression Testing

Regression Testing is a process to make sure that a newly modified software can run successfully against existing test cases.

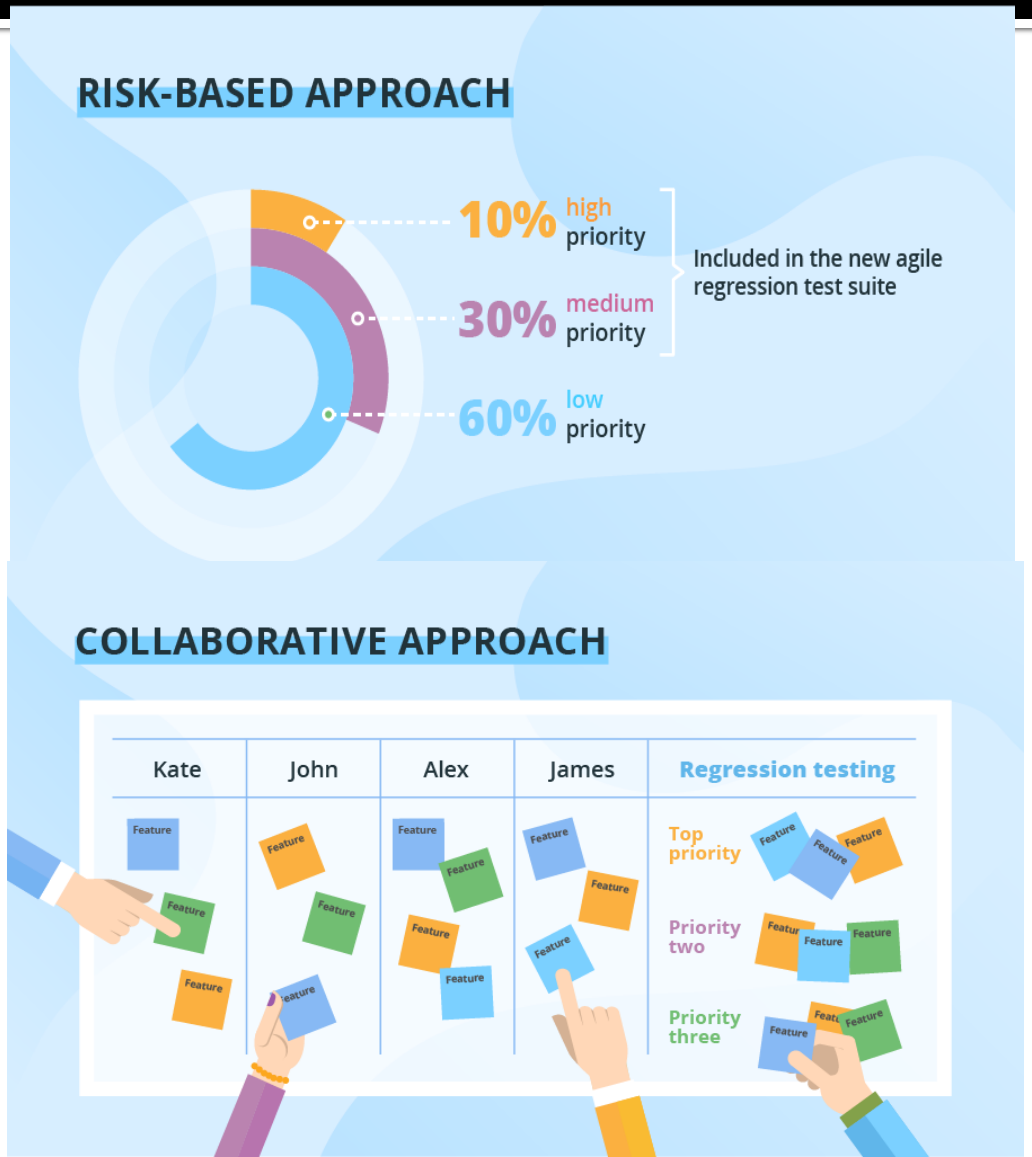


Regressing Testing matters because:

1. **New Functionality**
2. **Functionality**
3. **Bug Fixing**
4. **Performance Improvement**
5. **Integration**



Regression Test Prioritization



High priority. They cover the critical functions of the software, software areas highly visible to users, defect-prone areas, and areas that underwent many changes.

Medium priority. These regression test cases describe exceptional conditions (negative test cases, boundary value test cases, etc.). This priority mostly applies to the test cases that repeatedly detected bugs in previous product releases.

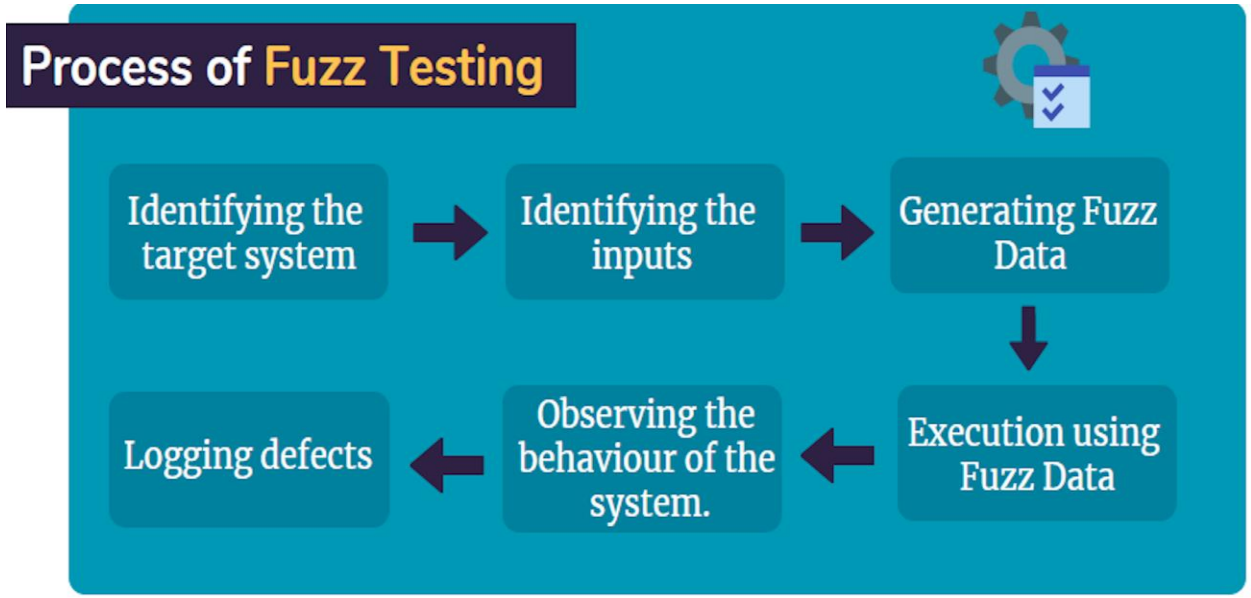
Low priority. These test cases cover the rest of functionality. Test engineers use them in regression testing before a major release to ensure full test coverage.

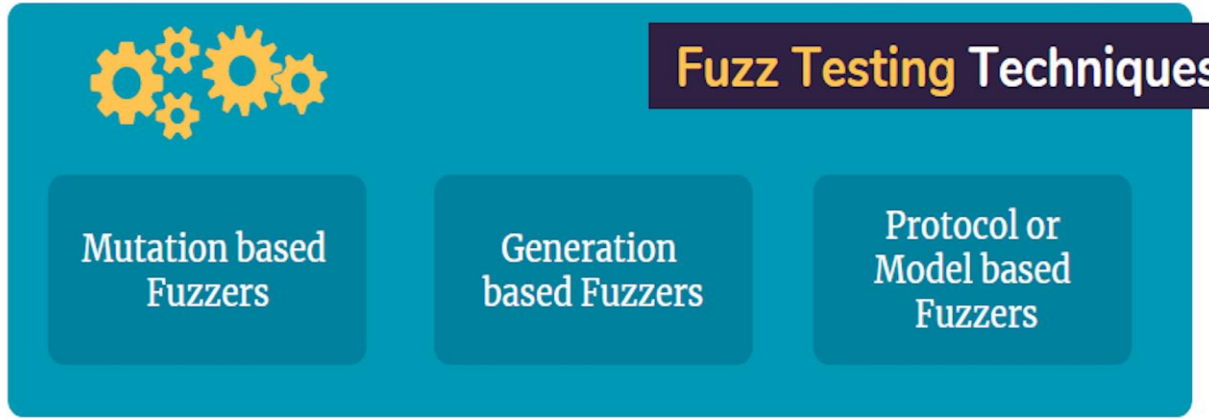
Fuzzing (Fuzz Testing, 模糊测试)

Fuzzing or **fuzz testing** is an automated black-box software testing technique that involves providing invalid, unexpected, or random data as inputs to a computer program. The program is then monitored for exceptions such as crashes, failing built-in code assertions, or potential memory leaks.

Barton Miller, a professor at the University of Wisconsin, introduced the "fuzz" notion in 1988 in a class project.

Sometimes treated the same as "Monkey testing". Some argue that fuzzing has more emphasis on random data inputs while monkey has more emphasis on random actions.





Fuzzing (Fuzz Testing, 模糊测试)

An effective fuzzer generates semi-valid inputs that are **"valid enough"** in that they are not directly rejected by the parser, but do create unexpected behaviors deeper in the program and are **"invalid enough"** to expose corner cases that have not been properly dealt with.

Types of defects explored by Fuzzy testing

- **Assertion Failure or Memory Leaks:** Bugs or defects, which are responsible for hampering the safety of the memory.
- **Invalid Input:** Defects arises from the invalid inputs, and are being responsible for the "error handling" feature of the software product.
- **Correctness Bug:** It may include corrupted database, poor search results, etc.

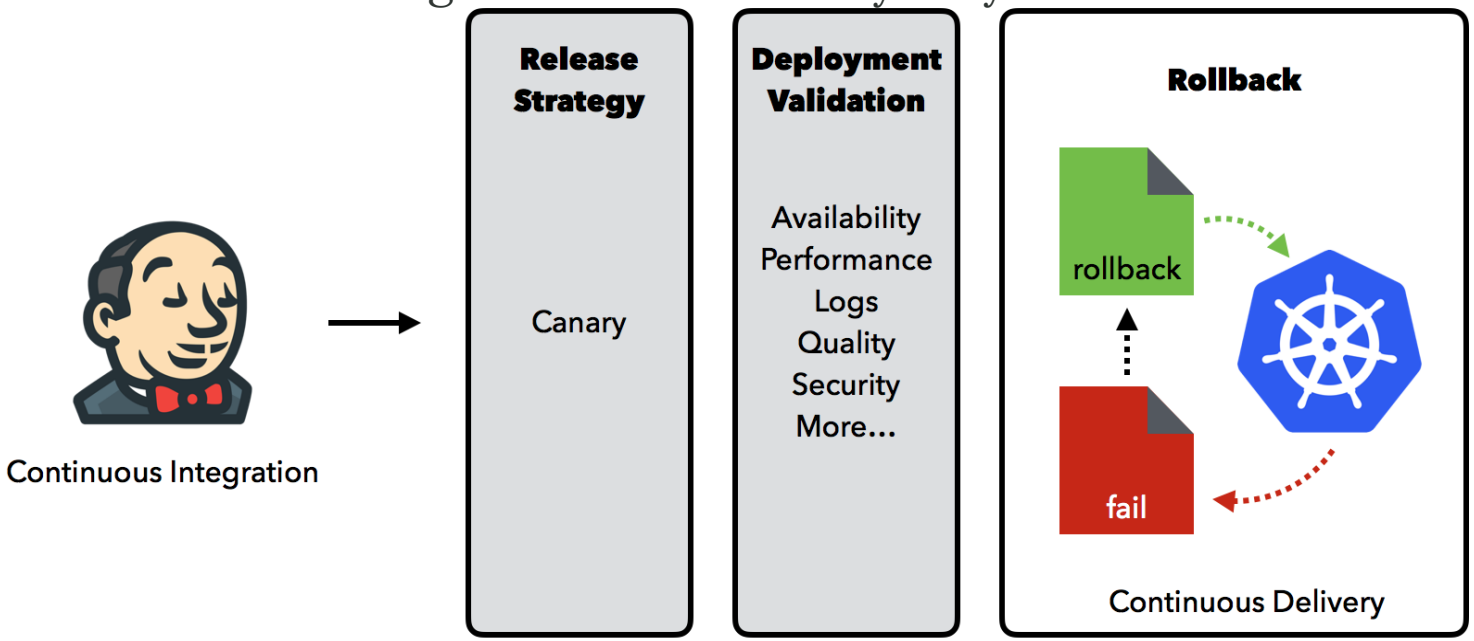
Advantages of Fuzz Testing

- Enhances the job of **security testing.**
- Explores severe defects, which are left invisible and are not detected, even by the test cases designed and prepared by an expert tester.
- Ensures the coverage of all possible negative scenarios for the software product.
- Detects race conditions & deadlocks and checks control flow integrity.

Canary Testing

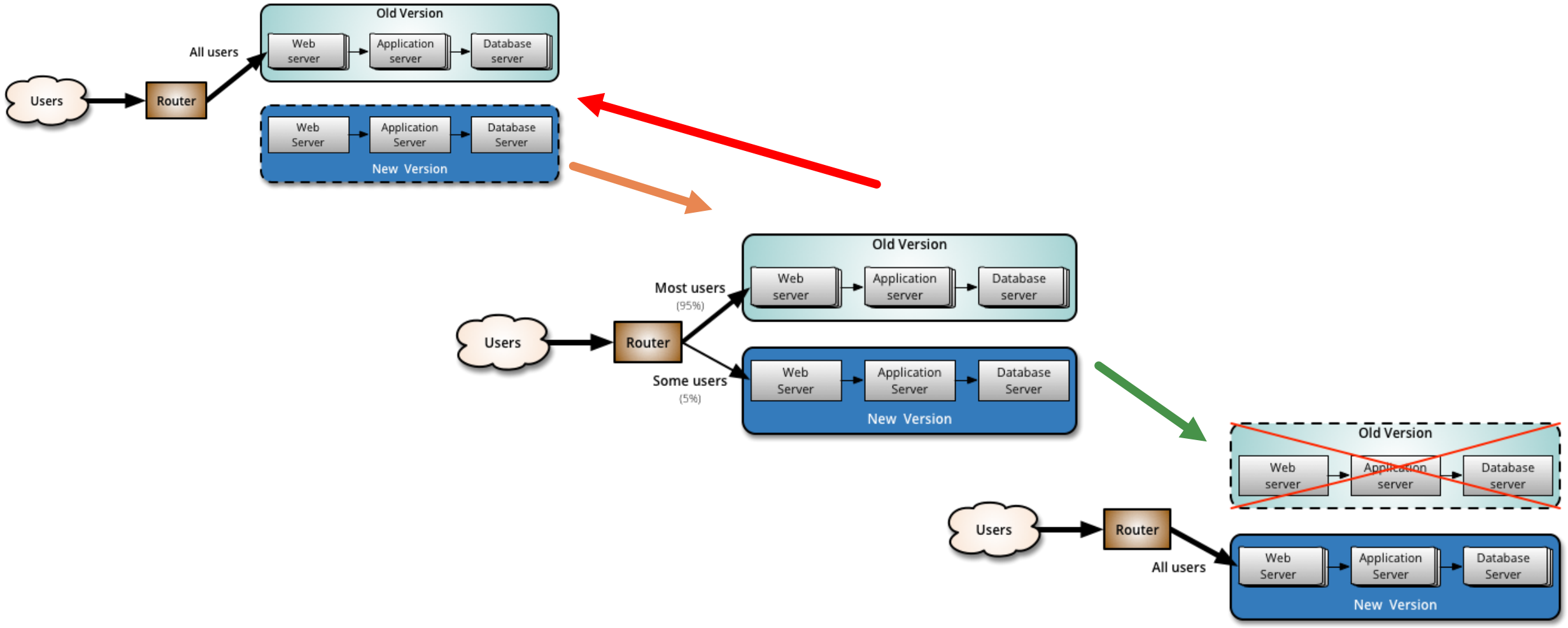
Coal miners brought canaries into coal mines as an early-warning signal for toxic gases, primarily carbon monoxide. The birds, being more sensitive, would become sick before the miners, who would then have a chance to escape or put on protective respirators.

Canary release is a technique to reduce the risk of introducing a new software version in production by slowly rolling out the change to a small subset of users before rolling it out to the entire infrastructure and making it available to everybody.

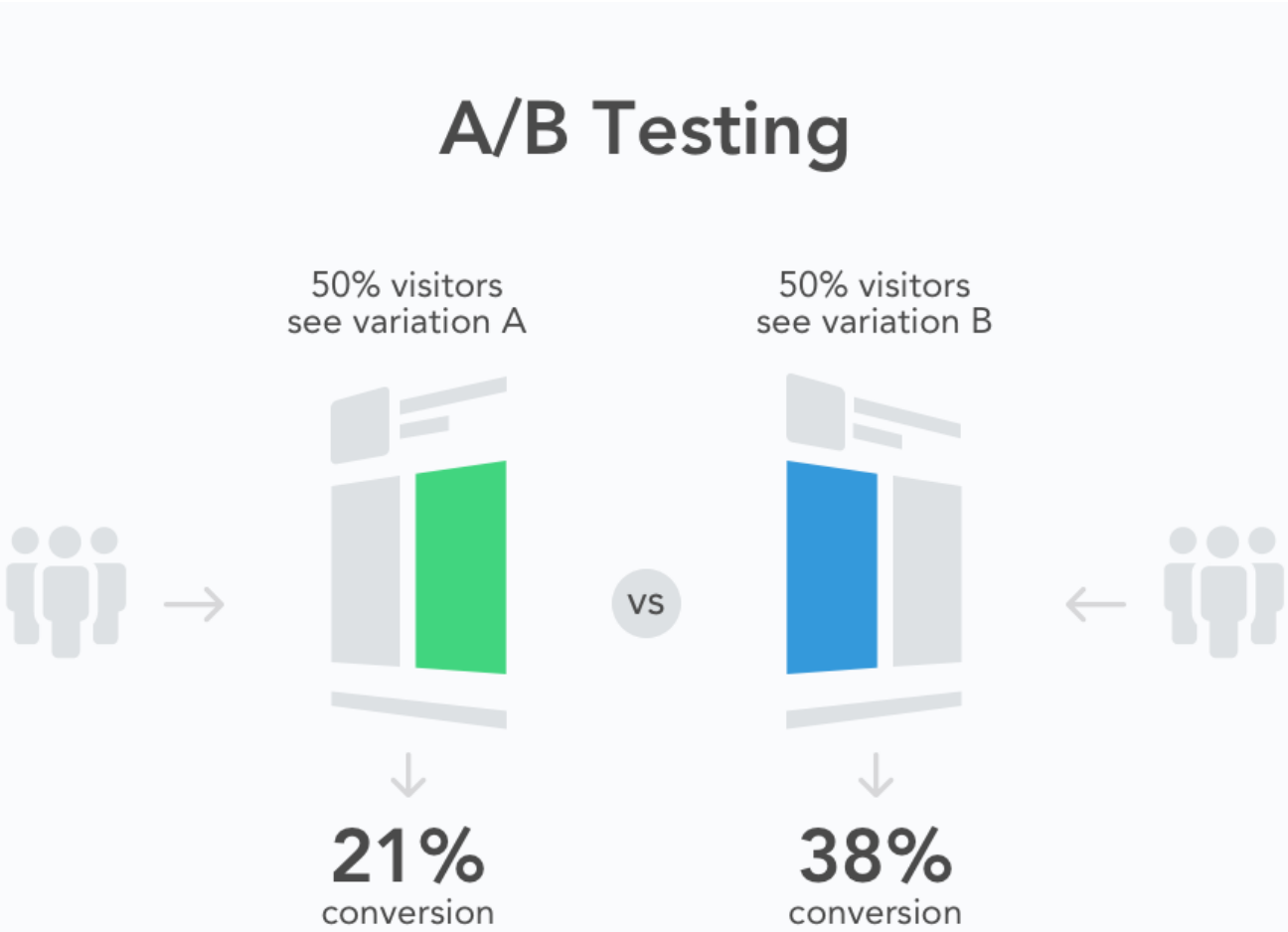


Auto Rollback Strategy

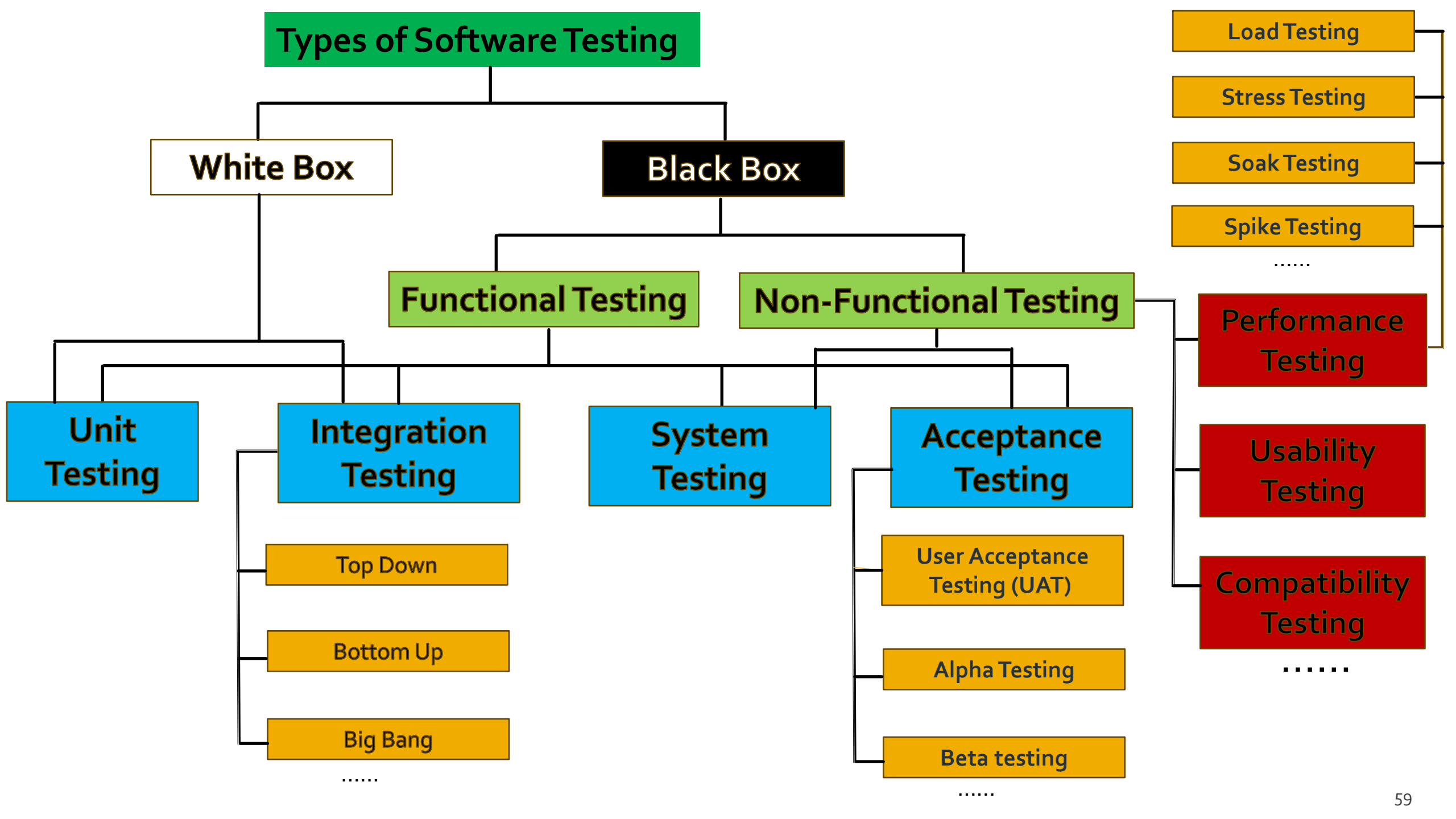
Canary Testing



A/B Testing



- A/B testing is a method of running a controlled experiment to determine if a proposed change is more effective than the current approach
- Two versions (A and B) of a single variable are compared, which are identical except for one variation that might affect a user's behavior.
- A/B tests are useful for understanding user engagement and satisfaction of online features.


Types of Software Testing
White Box
Black Box
Functional Testing
Non-Functional Testing
Unit Testing
Integration Testing
System Testing
Acceptance Testing
Top Down
Bottom Up
Big Bang
.....
User Acceptance Testing (UAT)
Alpha Testing
Beta testing
.....
Load Testing
Stress Testing
Soak Testing
Spike Testing
.....
Performance Testing
Usability Testing
Compatibility Testing
.....

Thank you!