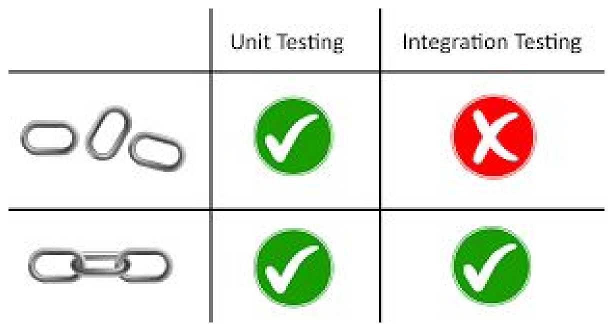
Lecture Notes
November 10, 2020

Working as a team: what comes after unit testing?



	Unit Testing	Integration Testing
	✓	✗
	✓	✓

The goal of *integration testing* is to find errors at the interfaces **between** units and assemblies consisting of multiple units

Integration tests are often automated via the same testing tools as unit testing, with constructed inputs, assertions that check outputs, setup, teardown, etc.

“Inputs” may come from stubs, mocks and real resources during test execution, not just from test setup. “Outputs” to be checked may include side-effects.

A common cause of integration bugs is inconsistencies between the viewpoints of different units. The units work fine in isolation but do not work together.

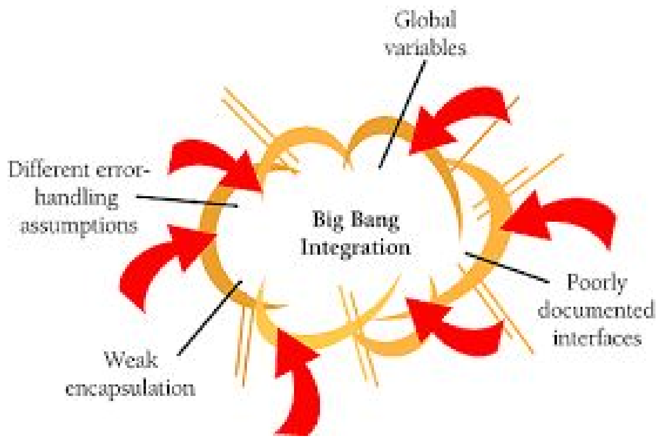
This is why integration testing should begin within the narrow scope of testing each unit with a *faithful* test double for each of the other components it interacts with or otherwise depends on. This checks for errors in the interface from your code to the other component.

Then more broadly test the components together. This also checks for errors in the interface between the other component and your code.

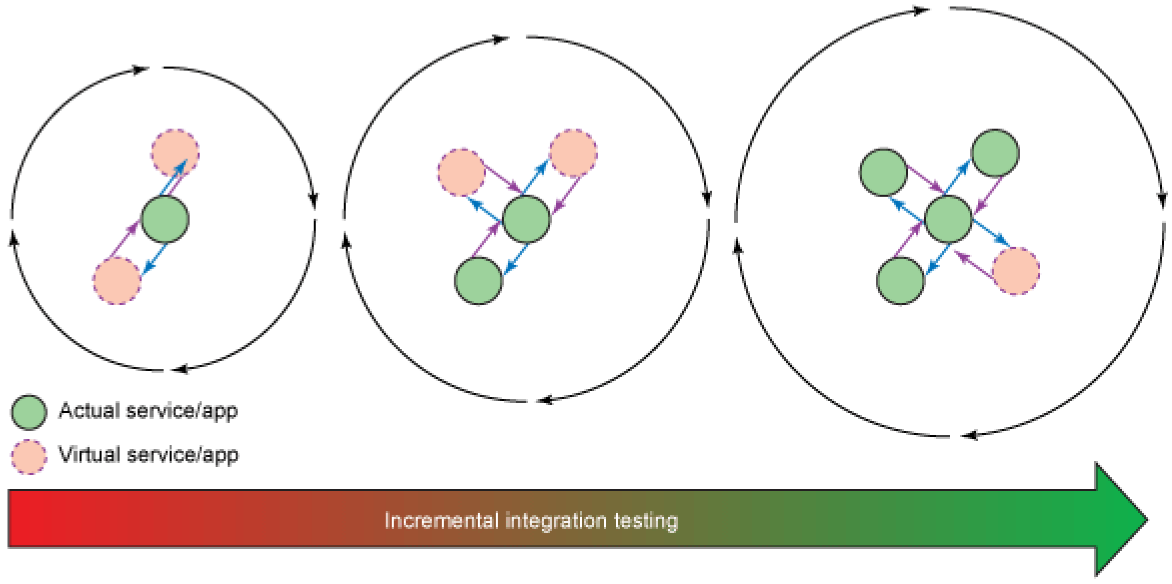
More about narrow vs. broad integration testing here.

"Big bang" approach to integration testing: Unit test all independently developed units in isolation (usually classes or modules, not literally testing only individual methods), then test the system. All done!

Not so fast: Bugs detected during "big bang" could be anywhere!


Global variables
Different error-handling assumptions
Big Bang Integration
Poorly documented interfaces
Weak encapsulation

Incremental integration makes it easier to localize bugs: Pick two units that have both already been unit-tested, one unit and an assembly of multiple units (that have already been unit-tested individually and then integrated), or two assemblies.



Since we integrate only a small number of units and assemblies at a time, that limits where we need to look to find the “root cause” (the underlying coding mistake or interface mismatch) and fix the bug.

Replace all test doubles from one or both units that are implemented by the other unit, and re-run unit test cases - same equivalence classes and boundaries.

But some equivalence classes/boundaries feasible in isolation may not be feasible in a particular combination (or in any combination within the current version of the program).

Why not?

```
foo(int a) {
    if (a<0) { do something }
    else { do something else }
}
```

Equivalence classes are a<0 and a>=0

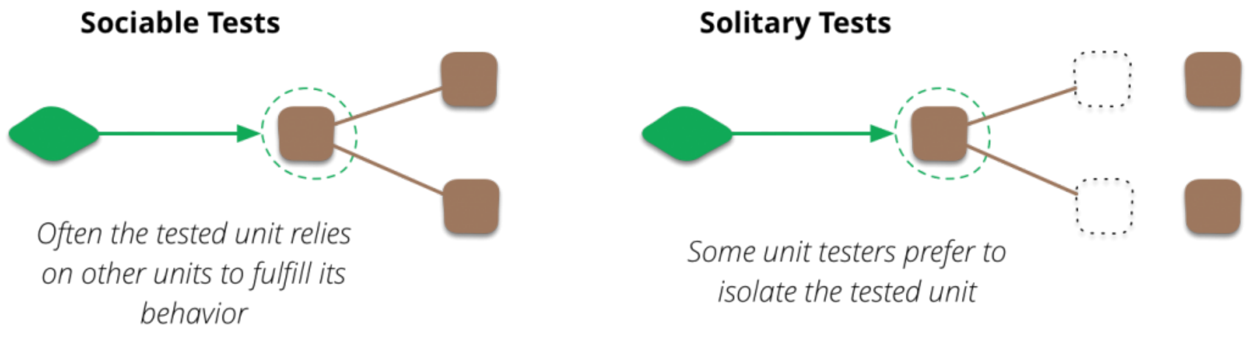
```
bar(int b) {
    if (b>=0) { foo(b) }
    else { don’t call foo }
}

foo(int a) {
    if (a<0) { can’t get here }
    else { do something else }
}
```

So why should we bother to test foo with a<0? “Defense programming”

Integration tests also need to cover other kinds of dependencies between the units, e.g., neither unit directly invokes the other, but one unit reads data written by the other or they both affect the same application state.

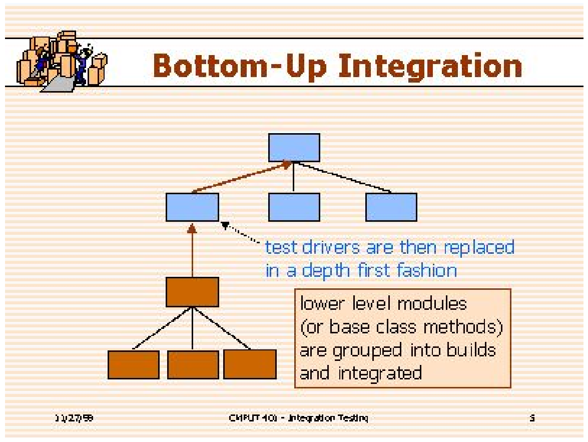
For example, push and pop do not call each other but they change the same stack data structure.

Integration testing may seem not much different from "sociable unit testing". Unit testing in isolation is "solitary unit testing"


Sociable Tests
Often the tested unit relies on other units to fulfill its behavior
Solitary Tests
Some unit testers prefer to isolate the tested unit

So why do we need integration testing if we could do "sociable" unit testing?

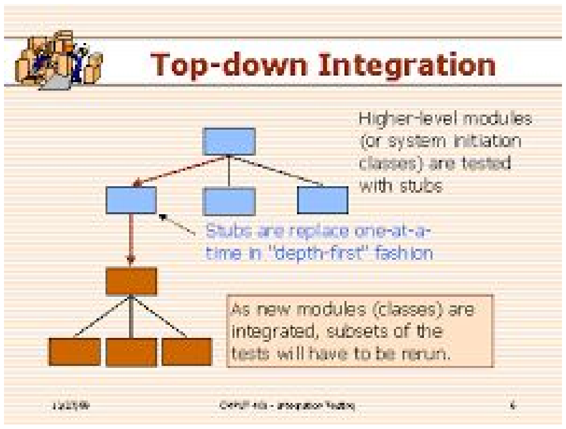
Traditional integration testing orders:


Bottom-Up Integration

test drivers are then replaced in a depth first fashion

lower level modules (or base class methods) are grouped into builds and integrated

11/27/99 CMPUT 401 - Integration Testing 5


Top-down Integration

Higher-level modules (or system initiation classes) are tested with stubs

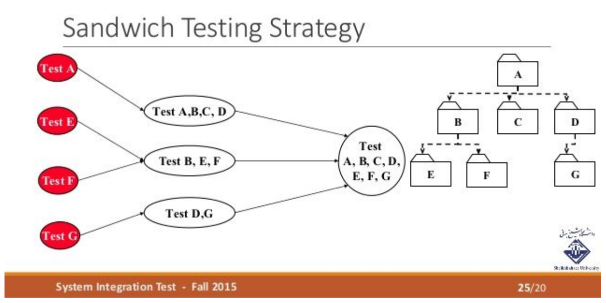
Stubs are replace one-at-a-time in "depth-first" fashion

As new modules (classes) are integrated, subsets of the tests will have to be rerun.

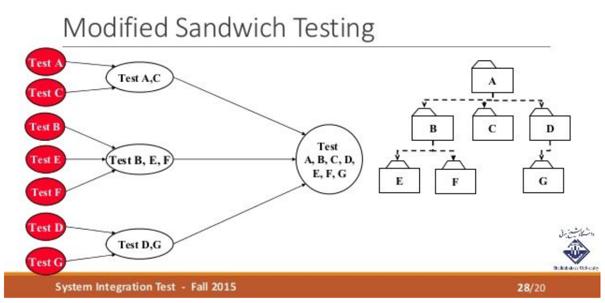
11/27/99 CMPUT 401 - Integration Testing 6

Bottom up starts integration process from lowest layer, top down starts integration process from top

Sandwich (or hybrid) integration testing integrates “natural” assemblies or layers, meets in middle - e.g., integrate front end and back end separately

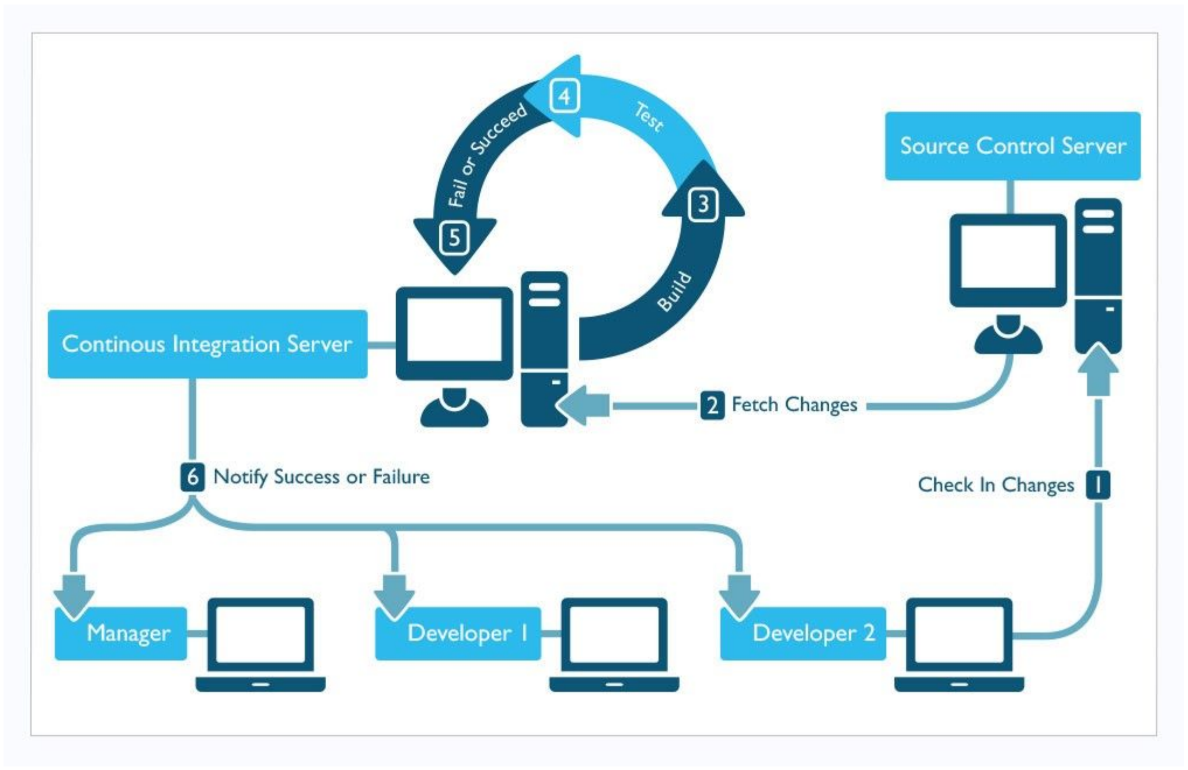

Sandwich Testing Strategy

System Integration Test - Fall 2015 25/20


Modified Sandwich Testing

System Integration Test - Fall 2015 28/20

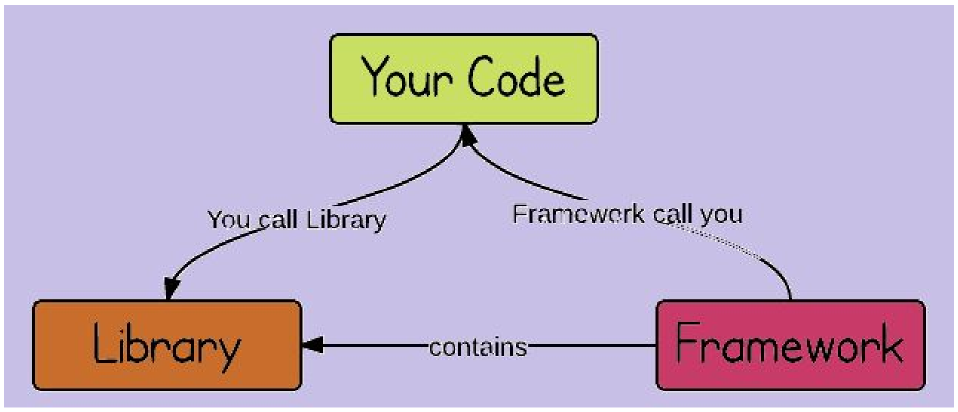
Continuous integration (CI) tools like Jenkins and Travis CI hook a version control system (like github) with build and testing tools that run after every commit to the shared repository, or nightly for very large codebases and test suites, so errors can be detected quickly

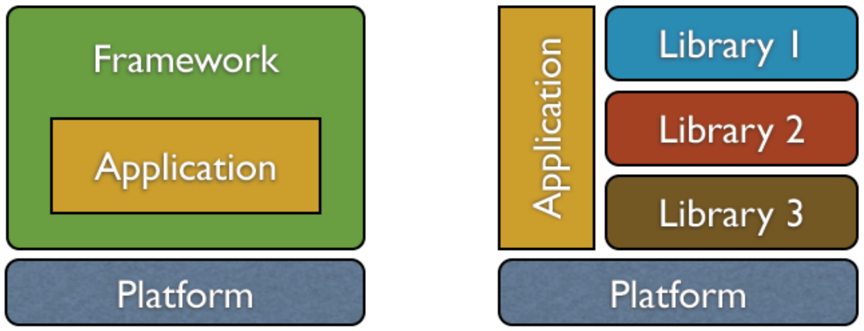


Where is the integration in continuous integration?

More about CI coming soon…

Testing how your code uses third-party frameworks, libraries, services, APIs (or how other code could use *your* API) is also referred to as integration testing





Testing needs to cover synchronous call/response and asynchronous events, errors/exceptions visible at API interface, and API functions that read/write the internals of shared data or affect shared resources

API testing seeks to answer these questions (validating the documentation, if any):

- What orders can the API functions be called in?
- What happens if called in the wrong order?
- Does caller code handle all possible errors and exceptions that can be raised by API functions?
- What happens when caller code omits a mandatory parameter, a parameter is the wrong type, or parameters are out of order?
- What parameters should the caller code provide to an API function to force a particular error response?
- Does the caller need to supply callback code for API events?
- What if callback code does not fully implement the interface expected by the API?
- Can race conditions arise due to multi-threaded access to shared data?
- More...

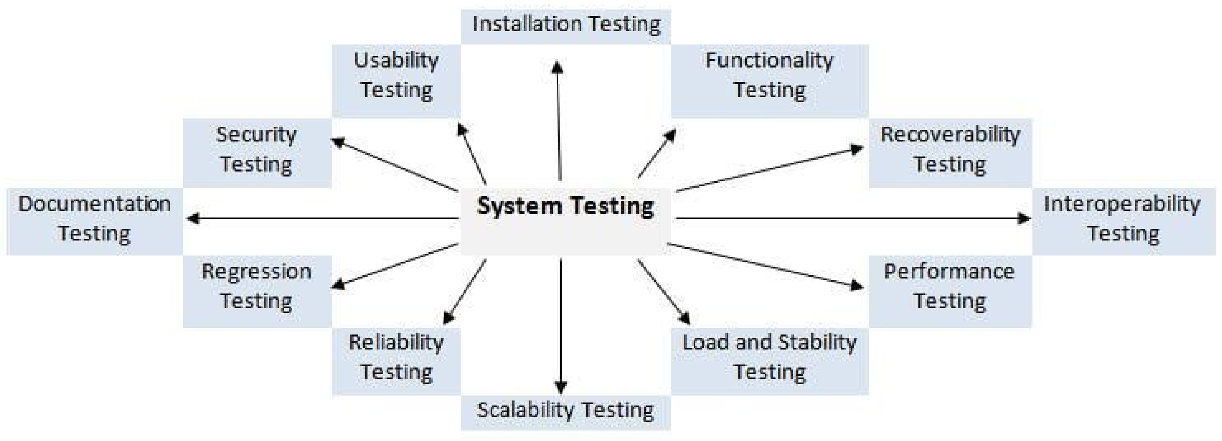
There are many alternatives to Postman for testing APIs

Integration testing eventually approaches system testing:

	Consider the steering wheel… it is round, it has turn signal and windshield wiper controls, everything we need in a steering wheel. Passes all unit tests. But can a typical driver use it to steer the car at 65 mph?

Let’s say we have tested each different car part in isolation, and they all fulfill the specifications defined when we designed the new car model. Do we still need system testing of the whole car?

But system testing != integration testing with all the units.



System Testing - © www.SoftwareTestingHelp.com

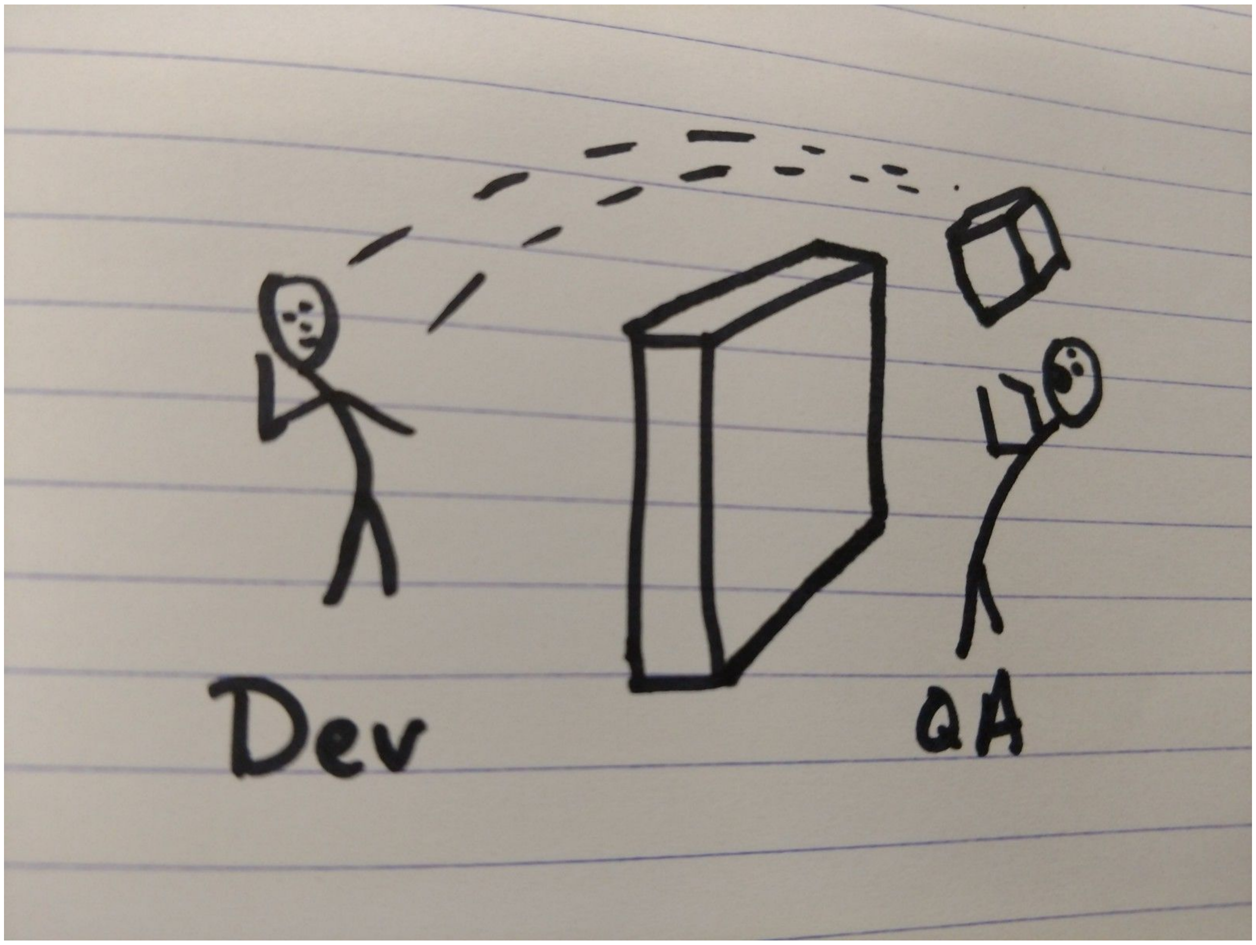
System testing looks for many different kinds of problems from many perspectives.

So far, we have mostly considered *functionality testing*. System level functionality testing is similar to unit testing, but applied to the whole system - equivalence classes and boundary analysis consider the inputs to the whole application, via GUI, command line, network I/O, sensors

But there is an even stronger emphasis on **invalid** inputs and input validation. Why?

Consider where your input comes from:

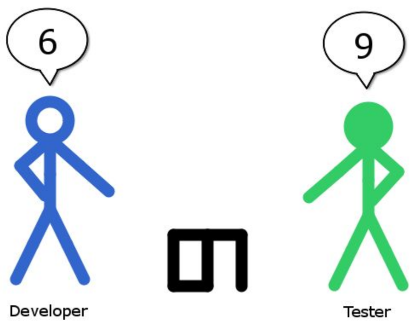
System testing is often conducted by separate testing teams not involved in development. In contrast, unit testing is almost always done by the development team and is part of their continuous integration process.

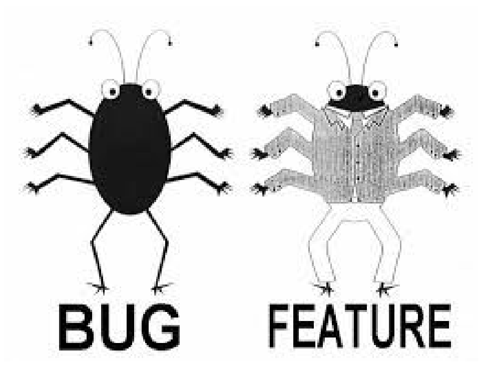


Is separation of development and testing (aka QA = quality assurance) teams a good idea? Contrast to (intended) non-separation of DevOps.

For functional testing, independent testers typically mimic real-world production use, when possible with real data

Testers think about what the system *is supposed to do*, or what the documentation says it does, not what the system (as implemented by developers) actually does





The testers' goal is to find discrepancies that would be visible to users and external systems. As with all testing, a "successful" test is one that exposes a bug (fault,flaw,error,failure).

Two children are playing with a competitor's toy water gun, and both got soaked.

Now the same two children are playing with your company’s toy water gun (a simple smoke test). Ethan shot his water gun at Emily, but Emily did not get wet.

Thinking like a software tester: What are some possible reasons that Emily did not get wet? You are trying to find bugs in the toy.

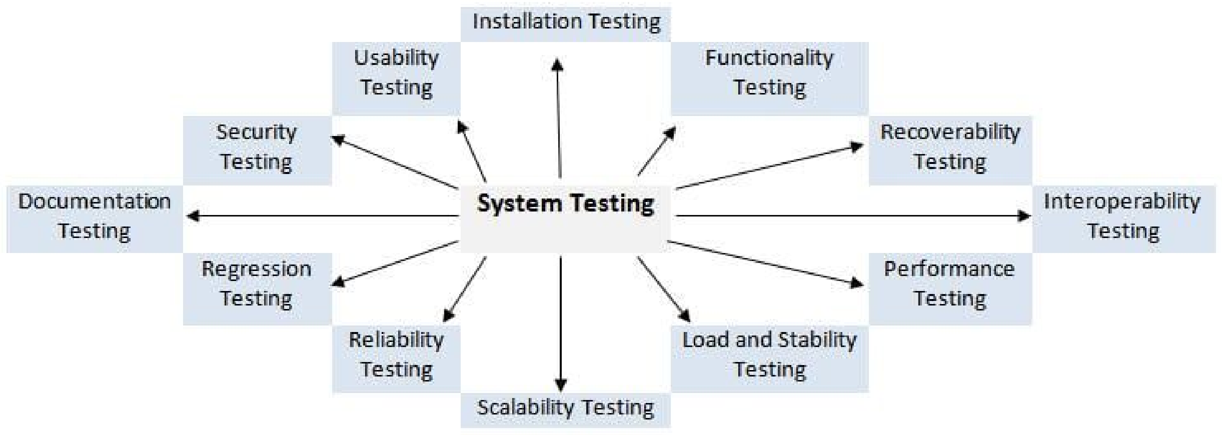
Thinking like a software developer: What are some possible reasons that Emily did not get wet? You do not believe there are any bugs in your toy.

Breakout room exercise, automatic room assignments

Odd breakout rooms (1, 3, 5, etc.): You are the leader of an independent consulting team hired to find any problems with the toy water gun before it is marketed to customers. The water gun manufacturer is paying you to find bugs *before* the customers do, so they can be fixed before the toys are shipped to stores. What would you do to find bugs in the water gun? You have a group of several children to help with testing.

Even breakout rooms (2, 4, 6, etc.): You are the leader of the team employed by the manufacturer to design the toy water gun, and you *know* it works! Your employer is paying you to design toys that *do not have any bugs*. What would you do to demonstrate that your water gun does not have any bugs? You have a group of several children to help with testing.

Please come back to the main room to report when breakout rooms close. Do not leave, the lecture is not over yet.



System Testing - © www.SoftwareTestingHelp.com

You now hire the Avengers (or their stunt doubles) to test the water gun. What other testing should you do now?

What would you do if you could use automated equipment to test one thousand toy water guns at the same time?

We will consider other kinds of system testing, beyond functional testing, in more depth later on.

“Assignment T3: First Iteration” due November 17.
https://courseworks2.columbia.edu/courses/104335/assignments/490808

Implement your MVP (minimum viable product). Tell us what user stories you really implemented and what acceptance tests you used to check that the user stories really work. You must use build/package, unit testing, style checking, and static analysis bug finding.

“Assignment T4: Initial Demo” due November 23. Note this is the Monday before Thanksgiving.
https://courseworks2.columbia.edu/courses/104335/assignments/491014

There will be very short team meetings in class on Thursday.