Lecture Notes
October 30, 2018

First Iteration due tonight, 11:59pm

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

Sometimes system testing is distinguished from end to end testing, but we will usually treat
system testing = end-to-end 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 whatever interfaces are provided (GUI, command line, network I/O, sensors)

Security testing also considers equivalence partitions/boundaries, but with strongest emphasis on **invalid** inputs, particularly from untrusted end-users or otherwise outside the control of the hosting organization - because that's often how hackers find and exploit security vulnerabilities

We have already conducted unit tests for both valid and invalid equivalence classes, and their boundaries, on every unit in the system. Do we still need system testing using equivalence classes and boundaries?

Consider a new model of car:

Let's say we have tested each different kind of part in isolation, and they all fulfill the specifications defined when we designed the new car model

Do we need system testing of the whole car?

	Consider the steering wheel… its 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?

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) teams a good idea?

For functional testing, independent testers 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

Their goal is to find discrepancies that would be visible to end-users and external systems. As with all testing, a "successful" test is one that exposes a fault

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

Two children are playing with your company's toy water gun. Ethan shot his water gun at Emma, but Emma did not get wet

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

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

How does system testing fit into agile iterations?

One approach is close interaction between development and testing teams within each iteration

Pros of QA members as part of the module team:

- Better integration of QA resources in sprint planning as QA is an inherent part of the team
- High visibility on what should be tested for a planned release
- Better planning and execution since QA is aware of day-to-day development updates
- QA efforts are well thought out during sprint planning and execution.

Cons of QA members as part of the module team:

- QA is not a separate team, so its efficiency can be negatively impacted by delays in development, changes to product features, etc.
- Since QAs are tightly bound with a module, the verification of that module can become monotonous. This might impact performance of long-term projects.
- It may present challenges when the Dev and QA teams are not in the same time zone as it demands close intra-team communication.

Another approach is parallel iterations across development team and testing team

Pros of separating QA and Dev teams:

- Less dependency on each development sprint as it runs on its own schedule.
- Team assignments are flexible, which allows test engineers to move across modules.
- It works better if Dev and QA are located at different geographical locations or operate in different time zones

Cons of separating the QA and Dev teams:

- As QA is a separate team managing its own sprints, it must be aware of the ongoing development and modules to be released for production to ensure accuracy and efficiency. It also must work collaboratively with the AppDev team to receive the latest release updates to understand the test scenarios and coverage.
- Each release takes additional time compared to the one team scenario.
- Dev and QA processes must be monitored to ensure good communication and collaboration.
- Danger of falling into "scrummerfall"

Bugs from a previous iteration or a previous release may be reported (by testers or by customers) in the midst of a development iteration

Showstopper bugs need to be fixed RIGHT NOW, so create new bug-fix user story and push other user stories planned for current iteration to overflow

Version control branching enables bug fixes and release patches to "old" versions

Otherwise new bug-fix user stories are time-estimated and prioritized along with feature user stories for upcoming iterations

A bug-fix user story usually has a different structure than a feature user story (or use case):

Summary: Identify what the problem is, the area it occurs and what it's impacting. If possible, outline the bug from the perspective of the user.

Scenario: Roadmap for recreating the bug, step-by-step account of exactly what was done to find it. So can find it, fix it and test it efficiently.

Expected Result: An 'expected' result is what **SHOULD** happen when the steps in the scenario are followed.

Actual Result: An 'actual' result is what **IS** happening after the steps outlined in the scenario are followed.

Note these elements need to appear in the initial bug report in order to be included in the bug-fix user story

Description

Users should be able to select (and apply) the ‘disabled’ filter on the ‘Agents’ page without being redirected to another page.

Scenario

- Log in and navigate to 'Agents' section.
- Select the ‘disabled’ filter button.

Expected result

The agents table should only show ‘disabled’ agents.

Actual result

The user is redirected to another page — this is an undefined ‘Agent’ page.

System testing != integration testing

Goal of *integration testing* is to find errors at the interfaces **between** units or assemblies (consisting of multiple units)

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

A common cause of integration bugs is inconsistencies between the viewpoints of different units

"Big bang" approach to integration testing: Unit test all components (modules, classes, etc.), then test system

Bugs detected by "big bang" could be anywhere!

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 (mocks) from one or both units that are implemented by the other unit, and re-run unit test cases - same equivalence classes and boundaries, **except** some equivalence classes/boundaries feasible in isolation may not be feasible in combination

Why not?

Integration tests also need to cover other kinds of interactions 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 - e.g., push and pop do not call each other but they change the same stack data structure

Agile testing via CI combines big bang with incremental

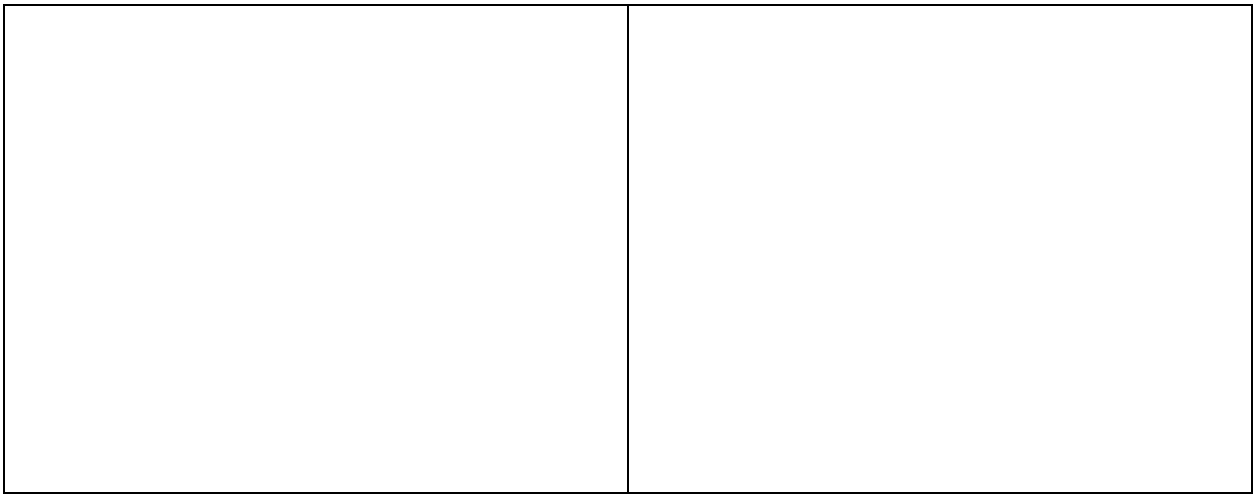
- Nightly build runs both unit and big-bang integration test suites - "always" ready for production

- Drill down to incremental assemblies to localize interface bugs

Integration testing seems not much different from sociable unit testing

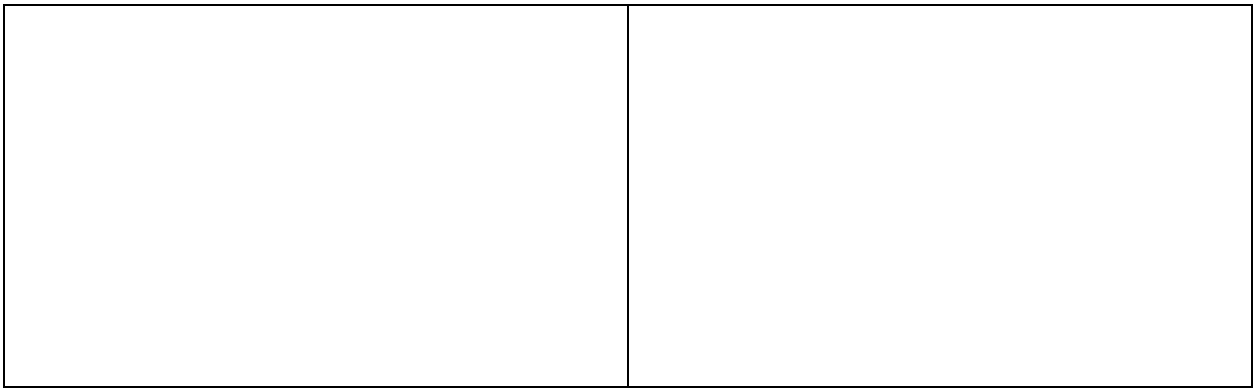
So why do we need integration testing if we have do "sociable" unit testing?

Traditional integration testing orders:



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



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 or 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:

- What orders can the API functions be called in?
- What happens if called in wrong order?
- Does caller code handle all possible errors and exceptions that can be raised by API function?
- What happens when caller code omits mandatory parameter, a parameter is wrong type, or parameters out of order?
- What parameters should caller code provide to API function to force particular error response?
- Does 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-threading access to shared data?
- More...

Often viewed as part of *gray box* testing, particularly when services accessed over network - consistency, fault tolerance, scalability, and other distributed system issues need to be tested, e.g., using API virtualization sandbox (specialized mock) or command line/script tool like curl

Reading for Thursday: How to Debug Any Problem and Simplifying and Isolating Failure-Inducing Input

Next team assignment First Iteration due Thursday, October 30, 11:59pm

Follow-up team assignment First Iteration Demo due Thursday, November 8, 11:59pm

Next pair assignment Bug Hunt 2 due Tuesday, November 20, 10:10am