Lecture Notes
November 14, 2017

Second Iteration Plan due tonight

Second Iteration Development and Code Inspection due November 30 (preliminary demo for second iteration)

Panel with professional software engineers Friday November 17, 11:45-12:45, Davis Auditorium

Integration Testing vs. System Testing

System testing considers equivalence classes/boundaries from the viewpoint of the *whole* application, via whatever user interfaces are provided (GUI, command line, network I/O)

> Strong emphasis on *invalid* inputs from untrusted end-users, because that's how hackers (including insiders) find and exploit security vulnerabilities

Often conducted by separate testing teams not involved in development

> Testers mimic real-world production use end-to-end 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

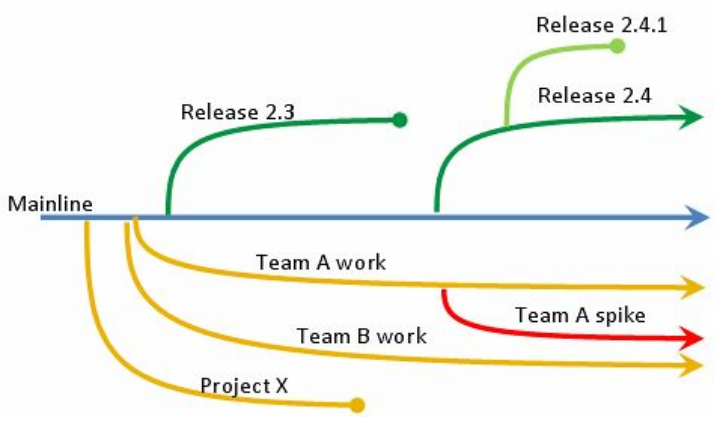
Parallel iterations between testing team and development team

> Bugs from a previous iteration or a previous release may be reported in the midst of a development iteration

> May need to be fixed RIGHT NOW, pushing other tasks planned for iteration to overflow

> Otherwise prioritized along with other tasks for upcoming iterations

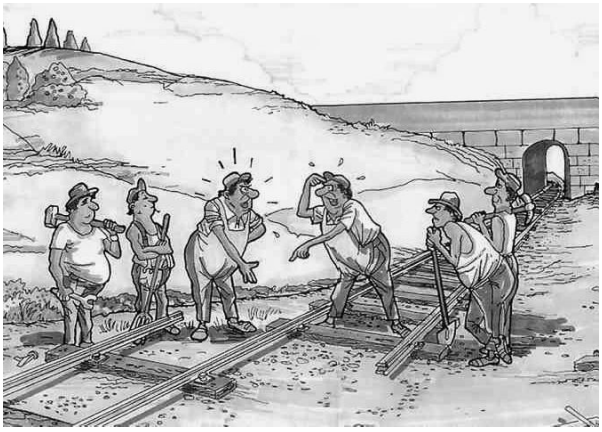
Good reason for version control branching/merging (most other reasons are bad)



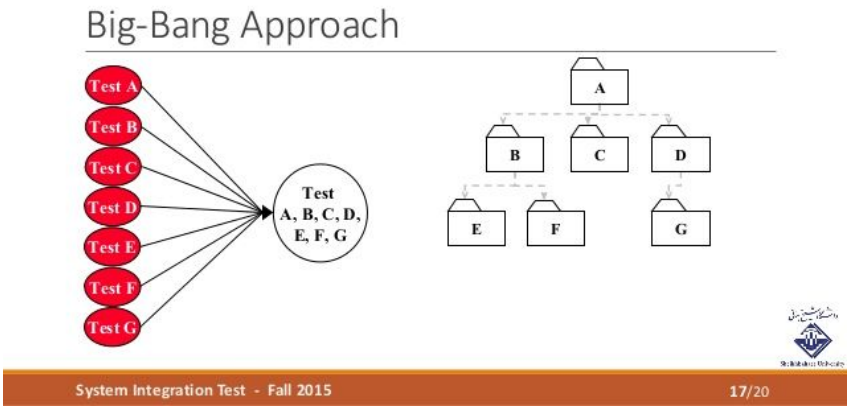
The goal of *integration testing* is to find errors at the *interfaces* between units or between larger components consisting of multiple units, often via same automated testing tool as unit testing

Force execution paths through multiple units that may be hard to reach from the UI

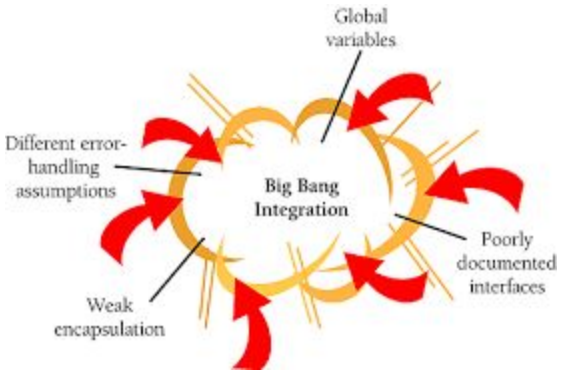
Check consistency between the viewpoints of the different units and components



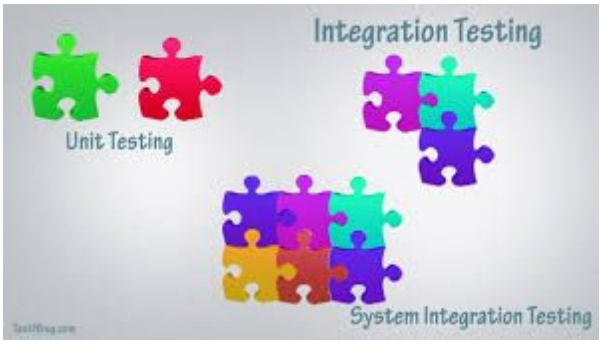
“Big bang” (system testing without prior integration testing) vs. Incremental integration:



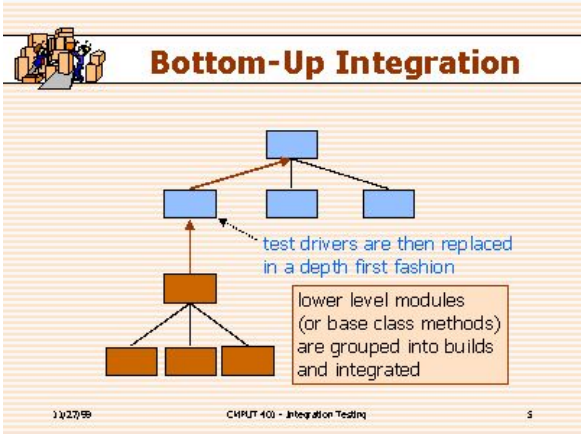
Bugs detected by “big bang” approach could be anywhere!



Incremental integration makes it easier to find 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

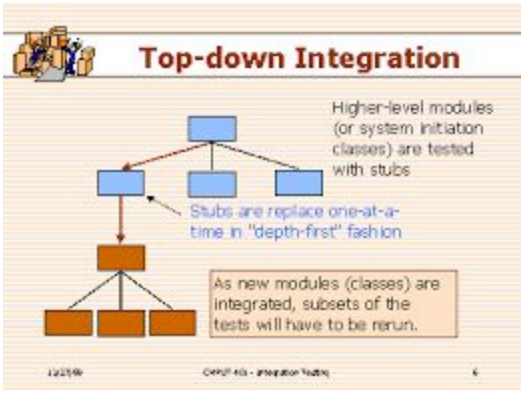


Bottom up - ideally start from lowest level units that do not use any other units, if there are any such units, if not start from two units that only use each other, if any, etc.



> “Use” here means unit calls a method in the other unit OR unit accesses data also accessed by another unit (usually focus on read/write not read/read)
>
> Replace all mocks, stubs, etc. from both units that are implemented by the other unit, and re-run same test cases as unit tests - same equivalence classes and boundaries, **except** some equivalence classes/boundaries feasible in isolation may not occur in combination
>
> Move upwards toward higher levels of system

Top down - start from main() or equivalent



> Again replace mocks/stubs moving down towards lower levels of system

Sandwich

> Both/either, typically based on “natural” assemblies (e.g., front end vs. back end)

Besides integration *within* the application, also need to integrate with *external* libraries, frameworks and systems, replacing mocks/stubs from unit testing with the real externals

> This corresponds to gray box testing
>
> But in practice, initial black box unit testing may already be conducted with certain externals that “shape” the application code, such as development frameworks

Besides repeating inputs from unit testing, specially craft interactions by working backwards: what inputs are needed to produce those outputs that *force* full exercise of interfaces

> Equivalence classes and boundary conditions for outputs rather than inputs - have all status codes, error messages, exceptions been produced?
>
> Parameters, global variables, files, etc. that component C provides as inputs to component D are treated as outputs from component C

Gray Box Testing

Recall this is primarily for data destined for other systems or subsystems, e.g., database, but also applies to internal “side-effect” data not part of the main computation, such as audit trails

Grey box testing can be static or dynamic

What does static mean here? Read “specification” (requirements and/or design)

- Determine all communication paths and data flows to external systems and between internal subsystems
- Check communication protocols and formats, both control and data
- Do specified inputs and outputs “match”? Can find bugs before writing any code

What does dynamic mean here? Running the code with sample inputs and checking the results

- Intuition that internal data could be “wrong” for long periods before user visible
- Often generated in the course of system-level or integration-level black box testing, but need to look below surface
- Can also supply inputs specific to the internal subsystem inputs that would normally be received from other subparts of the system, including “inputs” that are actually responses from system/library/API calls and native methods

Logs and audit trails - correct format and contents, file permissions, recoverable?

System-added info - Checksums, timestamps (right time zone?)

Data destined for other systems - check formats for incoming and outgoing data, check responses (acks) and timeouts

Example: web browser - http headers, cookies, cache expiration, security vulnerabilities

Scraps left lying around, such as file handles, temporary files, database / network connections

Resource leak, security risk - delete what should be deleted, clean uninstall

Also consider thread-safe code for multi-threading, use of security roles, what code will do when needed resources are not available, what happens when user turns on “accessibility” or machine suddenly loses power, multiple screen sizes (mobile), future proof for later browser versions and mobile devices