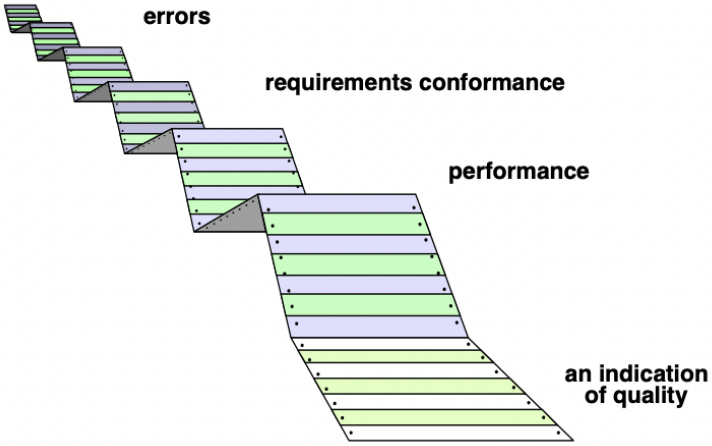
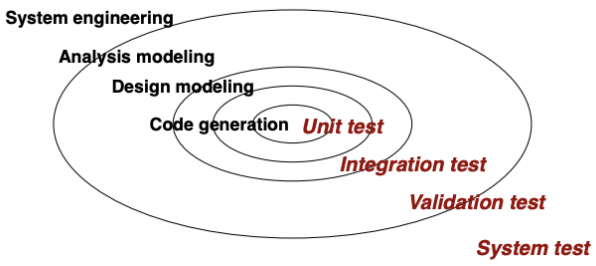
CAS CS 411
Lec 13

Testing Strategies

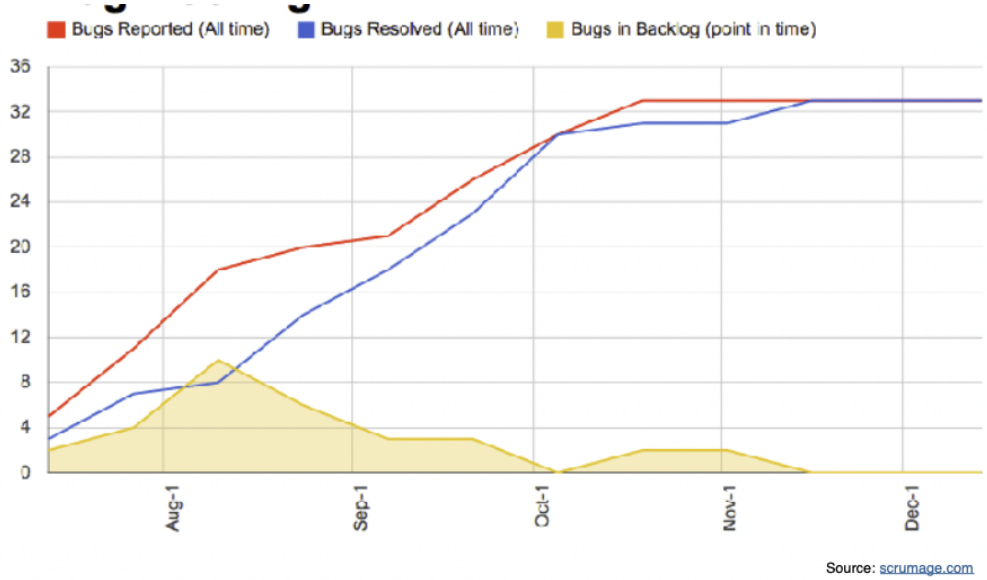
1. Software Testing
    a. Testing is the process of exercising a program with the specific intent of finding errors prior to delivery to the end user.
2. What Testing Shows



    a.
3. Strategic Approach
    a. To perform effective testing, you should conduct effective technical reviews. By doing this, many errors will be eliminated before testing commences.
    b. Testing begins at the component level and works "outward" toward the integration of the entire computer-based system.
    c. Different testing techniques are appropriate for different software engineering approaches and at different points in time.
    d. Testing is conducted by the developer of the software and (for large projects) an independent test group.
    e. Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.
4. Testing Strategy



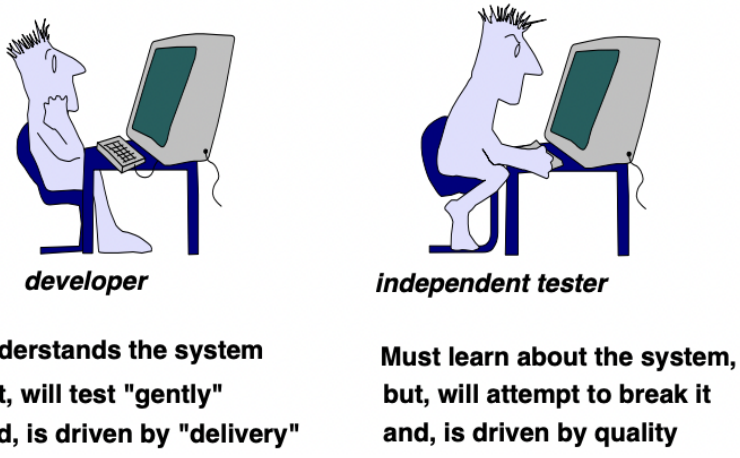
    a.

5. Strategic Issues
    a. Specify product requirements in a quantifiable manner long before testing commences.
    b. State testing objectives explicitly.
    c. Understand the users of the software and develop a profile for each user category.
    d. Develop a testing plan that emphasizes “rapid cycle testing.”
    e. Build “robust” software that is designed to test itself
    f. Use effective technical reviews as a filter prior to testing
    g. Conduct technical reviews to assess the test strategy and test cases themselves.
    h. Develop a continuous improvement approach for the testing process.
6. Bug Lists and Triage
    a. When bugs are found at any level of testing, they are logged
    b. Periodically the team comes together to evaluate and prioritize these bugs
    c. This prioritization is similar to what hospital emergency rooms do — it’s called triage
    d. Triage typically divides the bug list up into levels of severity, such as showstopper, high, medium, and low
    e. Sometimes also by function, such as security, logging, workflow, and so on
    f. Progress on bug fixes (and introductions) are closely tracked
7. Bug Tracking
    a. 

8. V & V
    a. Verification refers to the set of tasks that ensure that software correctly implements a specific function.
    b. Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements. Boehm [Boe81] states this another way:

i. Verification: "Are we building the product right?"
ii. Validation: "Are we building the right product?"

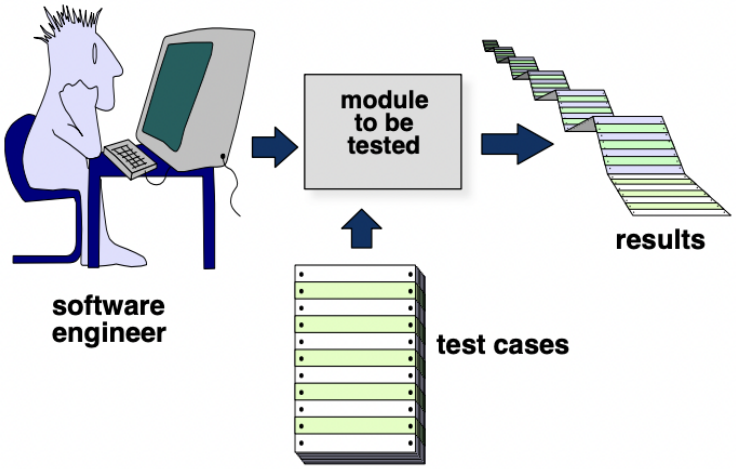
9. Who Tests the Software?
    a.



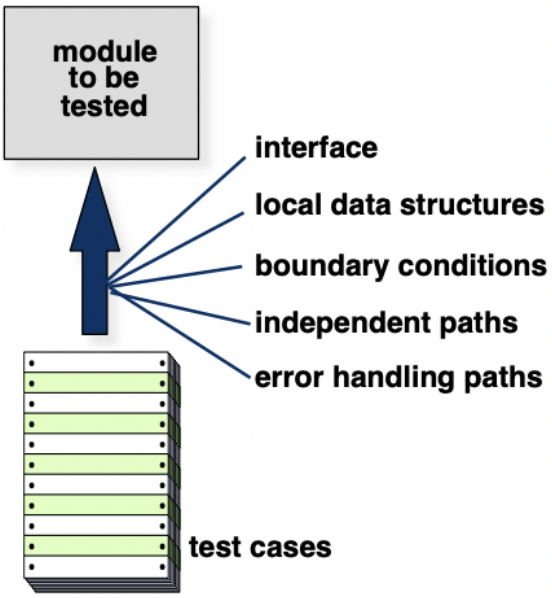
10. Testing Strategy
    a. We begin by ‘testing-in-the-small’ and move toward ‘testing-in-the-large’
    b. For conventional software
        i. The module (component) is our initial focus
        ii. Integration of modules follows
    c. For OO software
        i. our focus when “testing in the small” changes from an individual module (the conventional view) to an OO class that encompasses attributes and operations and implies communication and collaboration
11. Smoke Testing
    a. A common approach for creating “daily builds” for product software
    b. Smoke testing steps:
        i. Software components that have been translated into code are integrated into a “build.”
            1. A build includes all data files, libraries, reusable modules, and engineered components that are required to implement one or more product functions.
        ii. A series of tests is designed to expose errors that will keep the build from properly performing its function.
            1. The intent should be to uncover “show stopper” errors that have the highest likelihood of throwing the software project behind schedule.
        iii. The build is integrated with other builds and the entire product (in its current form) is smoke tested daily.
            1. The integration approach may be top down or bottom up.

12. Unit Testing

a.

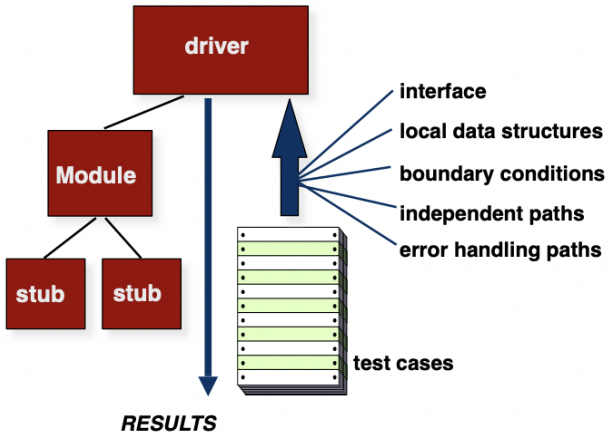


b.



13. Unit Test Environment

a.



14. Test-Driven Development (TDD)
    a. The TDD cycle:
        i. Design a compilable unit / module and document its functionality fully
        ii. Create a set of tests for both correct and incorrect functionality…boundary cases, out-of-bounds cases… and provide data to illustrate each
        iii. Run the test suite (everything will fail the first time)
        iv. Write the minimum amount of code necessary to correct the first error identified
        v. Run the test suite again
        vi. Rinse and repeat
15. What Should be Tested?
    a. All paths of execution in a module / object
    b. We want to see 100% code coverage in test
    c. Test all normal input values and ranges
    d. Test as many non-normal input values as possible
        i. Example: Interface is looking for an string of no more than 10 chars
        ii. Test 10, 1, 0, 11, 999, 1024 chars etc
    e. Why would it be so important to test 1 and 11 chars in that example?
16. TDD Frameworks
    a. Just about all development languages have a testing framework
        i. Java: JUnit
        ii. PHP: phpunit
        iii. C++: Google Test, boost
        iv. Ruby: RSpec
        v. Python: unittest
        vi. Javascript: Mocha, Sinon, Chai
        vii. All: asserts
    b. The frameworks allow you to write tests in an easily readable fashion
    c. Most will automatically run after code changes
    d. Example of Mocha test for JS:

```
describe("Tags", function(){
   describe("#parse()", function(){
       it("should parse long formed tags", function(){
           var args = ["--depth=4", "--hello=world"];
           var results = tags.parse(args);

           expect(results).to.have.a.property("depth", 4);
           expect(results).to.have.a.property("hello", "world");
       });
   });
}); //source: tutsplus.com
```

        i.

17. Simple Assertions
    a. Most languages provide a statement of the form

```
assert(pointerRef!=NULL);
```

    b. Typically the assertion, if triggered, will halt the program
    c. We don't enable these for production code…why?

```
#ifdef DEBUG
   assert(pointerRef!=NULL);
#endif
```

18. Stubs
    a. During development, we don't have all the code written
    b. We still want to test
    c. Stubbing interfaces allows us to test an incomplete system

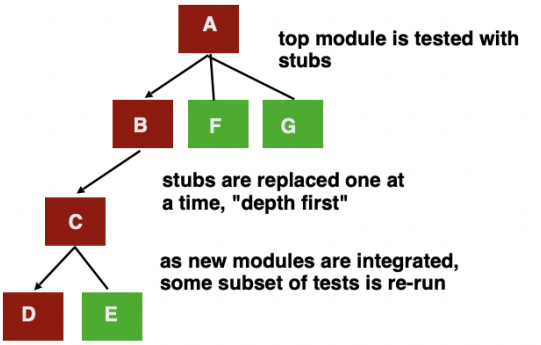
```
function someFunc (err, data) {
   return { data: "lahlahlah";}
}
```

    d. These are sometimes called reflectors, especially when the stub represents a subsystem (rather than just a unit)
19. Integration Testing
    a. Integration testing combines units and subsystems…it integrates them into one testable thing
    b. In a typical shop, unit testing is done while development is in progress, and an automated integration test is fired when code is checked in (committed)
    c. We assume that checked-in code has been unit-tested, so integration testing tends to focus on behaviors rather than data
20. Behavior-Driven Development (BDD)
    a. As more units come together to for larger subsystems, we start to focus on the behaviors of the software
    b. We can think of this in terms of a use case
        i. The use case describes the goal of the software:
            1. "As a logged-in user I want to edit my profile page"
        ii. We write unit code to provide the functionality - there might be many units required
        iii. Once the units are integrated, we can test the behavior
    c. TDD and BDD have very vocal proponents and opponents
    d. TDD critics say that it requires developers to spend too much time in the details of testing
    e. BDD critics say that it takes too much time to come up with useful cases
    f. Really BDD and TDD are two different approaches to the same problem

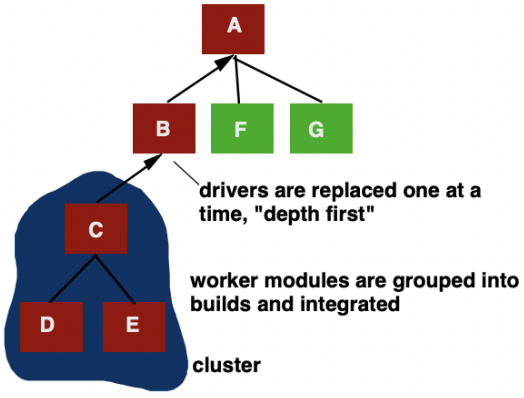
g. Usually the “best” solution is a mix of TDD/BDD, with the flexibility to feed BDD results back into dev

21. Styles of Integration Test
    a. Generally speaking there are two
        i. Top-down
        ii. Bottom-up
    b. Both will normally require stubs
    c. The difference is philosophy…which code is written first?
    d. A hybrid approach is to do both (sandwich testing)

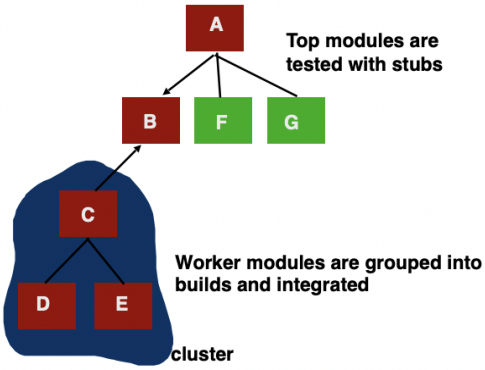
22. Top Down Integration
    a.



23. Bottom-Up Integration
    a.



24. Sandwich Testing
    a.



25. Regression Testing
    a. Once we have a large portion of the software written and integrated, we have to test it in its target environment
    b. Software doesn't run in a vacuum (with the exception of Dysons)
    c. Normally we load software into an environment where there are several large components
        i. Databases
        ii. Other software packages
        iii. Authentication
    d. It's crucial to make sure that the new software doesn't break the functionality of the old software
    e. Testing is done in an environment that is as close as possible to production - same software versions, hardware, and so on
    f. We also use scrubbed live data to simulate the production environment
26. Integration Test Tools
    a. The goal of integration testing is to simulate the production environment
    b. That means we have to simulate interaction
    c. For batch and other data-driven architectures this is relatively easy, since we can generate test data
    d. For interactive systems such as web or mobile applications, we have to simulate user interactions - much more difficult!
    e. One approach: Capture logs and replay them

```
124.182.195.187 - - [23/Mar/2015:05:01:11 -0400] "GET /story/my-speech-student-
council-please-please-please-read-d-ill-do-anything-1917122655 HTTP/1.1" 200 9439
"https://www.google.com.au/" "Mozilla/5.0 (iPad; CPU OS 8_0_2 like Mac OS X)
AppleWebKit/600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12A405 Safari/600.1.4"
124.182.195.187 - - [23/Mar/2015:05:01:12 -0400] "GET /files/css/
ebc5d341e633b965705ff0db99ed2e2e.css HTTP/1.1" 200 8855 "http://www.kidpub.com/
story/my-speech-student-council-please-please-please-read-d-ill-do-
anything-1917122655" "Mozilla/5.0 (iPad; CPU OS 8_0_2 like Mac OS X) AppleWebKit/
600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12A405 Safari/600.1.4"
124.182.195.187 - - [23/Mar/2015:05:01:12 -0400] "GET /themes/sky/js/sky.js HTTP/
1.1" 200 359 "http://www.kidpub.com/story/my-speech-student-council-please-please-
please-read-d-ill-do-anything-1917122655" "Mozilla/5.0 (iPad; CPU OS 8_0_2 like
Mac OS X) AppleWebKit/600.1.4 (KHTML, like Gecko) Version/8.0 Mobile/12A405
Safari/600.1.4"
124.182.195.187 - - [23/Mar/2015:05:01:12 -0400] "GET /sites/all/modules/
jquery_update/compat.js HTTP/1.1" 200 1465 "http://www.kidpub.com/story/my-speech-
student-council-please-please-p
```

    f. Also many commercial
        i. HP suites (used to be Mercury)
        ii. Loadrunner
    g. and FOSS tools
        i. AutoIt
        ii. Selenium (very popular, named so because selenium removes mercury…)
    h. Most allow you to record live interactions for playback and will autogenerate data and interactions given a template

27. End-to-end Testing
    a. Moving up from BDD at the integration / regression level, we must test software in the context of the business process
    b. For example, we might have written a new shopping cart module for our web site
    c. It needs to be tested from the point that a user logs in all the way to the point where a box is shipped off the dock
    d. All of the modules and packages involved in the business process are tested
28. General Testing Criteria
    a. Interface integrity – internal and external module interfaces are tested as each module or cluster is added to the software
    b. Functional validity – test to uncover functional defects in the software
    c. Information content – test for errors in local or global data structures
    d. Performance – verify specified performance bounds are tested
29. Bottom Line
    a. Testing goals vary depending on when and by whom the test is being done
    b. Our strategy is to try and uncover problems as early in the process as possible
    c. One of the most difficult aspects of testing is creating appropriate data
    d. Automation is key; we want to be consistent and thorough in our testing