Introduction to Software Testing

Practical Considerations

Paul Ammann & Jeff Offutt
Modified

www.introsoftwaretesting.com

Beizer's Testing Levels

1. No difference between testing and debugging
2. The purpose of testing is to show that the software works
3. The purpose of the software is to show that the software doesn't work.
4. The purpose of testing is not to prove anything specific, but to reduce risk.
5. Testing is a mental discipline that helps all IT professionals develop high quality software.

Testing Levels based on Software Activity

- Acceptance Testing – assess software with respect to requirements.
- System Testing – assess software w.r.t. To architectural design.
- Integration Testing – assess software w.r.t. subsystem design.
- Module Testing – assess software w.r.t. Detailed design.
- Unit Testing – assess software w.r.t. implementation.

The Toolbox

- Previous lectures give a "toolbox" with useful criteria for testing software
- To move to level 3 (reducing risk) or level 4 (mental discipline of quality), testing must be integrated into the development process
- Most importantly :
  - In any activity, knowing the tools is only the first step
  - The key is utilizing the tools in effective ways
- Topics :
  - Integrating software components and testing
  - Integrating testing with development
  - Test plans
  - Checking the output

Outline

1. Regression Testing
2. Integration and Testing
3. Test Process
4. Test Plans

Regression Testing

Definition

The process of re-testing software that has been modified

- Most software today has very little new development
  - Correcting, perfecting, adapting, or preventing problems with existing software
  - Composing new programs from existing components
  - Applying existing software to new situations
- Because of the deep interconnections among software components, changes in one method can cause problems in methods that seem to be unrelated
- Not surprisingly, most of our testing effort is regression testing
- Large regression test suites accumulate as programs (and software components) age

Automation and Tool Support

Regression tests must be automated

- Too many tests to be run by hand
- Tests must be run and evaluated quickly
  - often overnight, or more frequently for web applications
- Testers do not have time to view the results by inspection
- Types of tools :
  - Capture / Replay – *Capture* values entered into a GUI and *replay* those values on new versions
  - Version control – Keeps track of collections of *tests*, expected *results*, where the tests *came from*, the *criterion* used, and their past *effectiveness*
  - Scripting software – Manages the process of obtaining test *inputs*, *executing* the software, obtaining the *outputs*, *comparing* the results, and generating *test reports*
- Tools are plentiful and inexpensive (often free)

Managing Tests in a Regression Suite

- Test suites accumulate new tests over time
- Test suites are usually run in a fixed, short, period of time
  - Often overnight, sometimes more frequently, sometimes less
- At some point, the number of tests can become unmanageable
  - We cannot finish running the tests in the time allotted
- We can always add more computer hardware
- But is it worth it?
- How many of these tests really need to be run ?

Policies for Updating Test Suites

- Which tests to keep can be based on several policies
  - Add a new test for every problem report
  - Ensure that a coverage criterion is always satisfied

- Sometimes harder to choose tests to remove
  - Remove tests that do not contribute to satisfying coverage
  - Remove tests that have never found a fault (risky !)
  - Remove tests that have found the same fault as other tests (also risky !)

- Reordering strategies
  - If a suite of *N* tests satisfies a coverage criterion, the tests can often be reordered so that the first *N-x* tests satisfies the criterion – so the remaining tests can be removed

When a Regression Test Fails

- Regression tests are evaluated based on whether the result on the new program *P* is equivalent to the result on the previous version *P-1*
  - If they differ, the test is considered to have failed
- Regression test failures represent three possibilities :
  - The software has a fault – *Must fix the fix*
  - The test values are no longer valid on the new version – *Must delete or modify the test*
  - The expected output is no longer valid – *Must update the test*
- Sometimes hard to decide which !!

Evolving Tests Over Time

- Changes to external interfaces can sometimes cause all tests to fail
  - Modern capture / replay tools will not be fooled by trivial changes like color, format, and placement
  - Automated scripts can be changed automatically via global changes in an editor or by another script

- Adding one test does not cost much – but over time the cost of these small additions start to pile up

Choosing Which Regression Tests to Run

Change Impact Analysis

How does a change impact the rest of the software ?

- When a small change is made in the software, what portions of the software can are affected by by that change ?
- More directly, which tests need to be re-run ?
  - Conservative approach : Run all tests
  - Cheap approach : Run only tests whose test requirements relate to the statements that were changed
  - Realistic approach : Consider how the changes propagate through the software
- Clearly, tests that never reach the modified statements do not need to be run
- Lots of clever algorithms to perform CIA have been invented
  - Few if any available in commercial tools

Rationales for Selecting Tests to Re-Run

- Inclusive : A selection technique is *inclusive* if it includes tests that are "*modification revealing*"
  - Unsafe techniques have less than 100% inclusiveness
- Precise : A selection technique is *precise* if it omits regression tests that are not modification revealing
- Efficient : A selection technique is *efficient* if deciding what tests to omit is cheaper than running the omitted tests
  - This can depend on how much automation is available
- General : A selection technique is *general* if it applies to most practical situations

Summary of Regression Testing

- We spend far more time on regression testing than on testing new software
- If tests are based on covering criteria, all problems are much simpler
  - We know why each test was created
  - We can make rationale decisions about whether to run each test
  - We know when to delete the test
  - We know when to modify the test
- Automating regression testing will save much more than it will cost

Outline

1. Regression Testing
2. Integration and Testing
3. Test Process
4. Test Plans

Integration and Testing

Big Bang Integration

Throw all the classes together, compile the whole program, and system test it

- The polite word for this is risky
  - Less polite words also exist …
- The usual method is to start small, with a few classes that have been tested thoroughly
  - Add a small number of new classes
  - Test the connections between the new classes and pre-integrated classes
- Integration testing : testing interfaces between classes
  - Should have already been tested in isolation (unit testing)

Methods, Classes, Packages

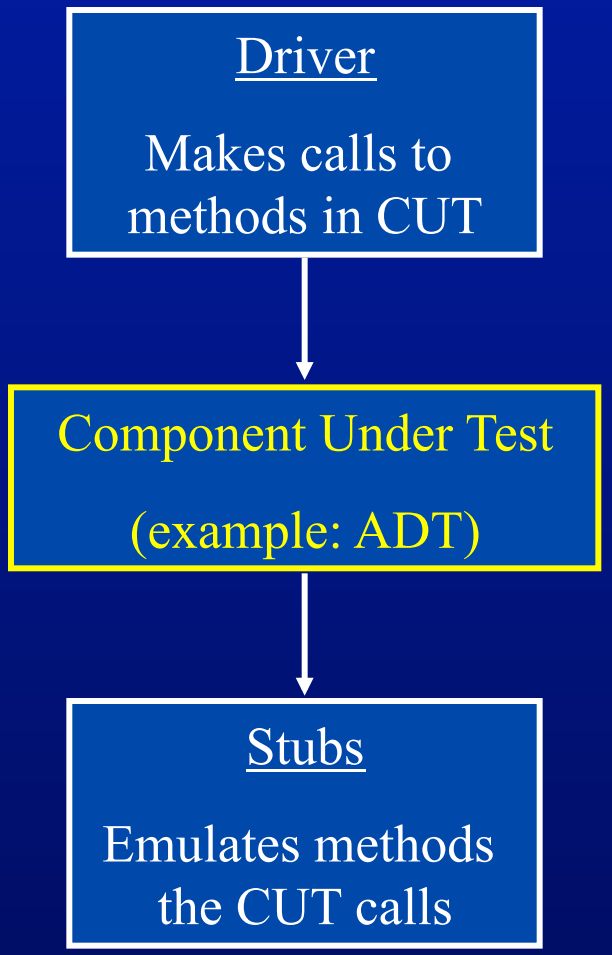
- Integration can be done at the method level, the class level, package level, or at higher levels of abstraction
- Rather than trying to use all the words in every slide …
- Or not using any specific word …
- We use the word component in a generic sense

- *A component is a piece of a program that can be tested independently*

- Integration testing is done in several ways
  - Evaluating two specific components
  - Testing integration aspects of the full system
  - Putting the system together "piece by piece"

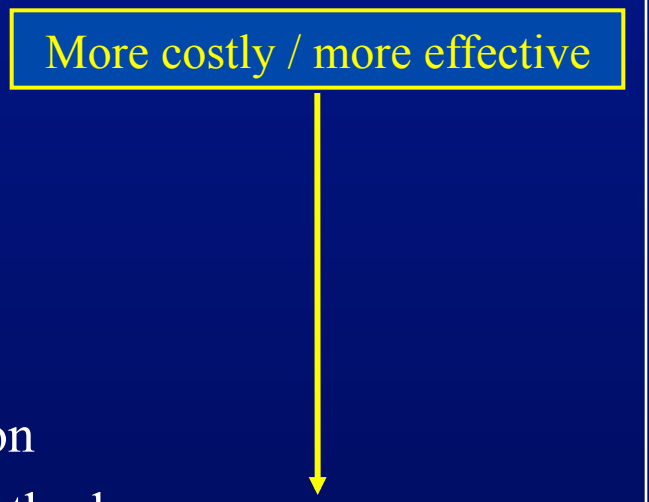
Software Scaffolding

- Scaffolding is extra software components that are created to support integration and testing
- A stub emulates the results of a call to a method that has not been implemented or integrated yet
- A driver emulates a method that makes calls to a component that is being tested



Stubs

- The first responsibility of a stub is to allow the CUT to be compiled and linked without error
  - The signature must match
- What if the called method needs to return values ?
- These values will not be the same the full method would return
- It may be important for testing that they satisfy certain limited constraints
- Approaches:
  - Return constant values from the stub
  - Return random values
  - Return values from a table lookup
  - Return values entered by the tester during execution
  - Processing formal specifications of the stubbed method



Drivers

- Many good programmers add drivers to every class as a matter of habit
  - Instantiate objects and carry out simple testing
  - Criteria from previous chapters can be implemented in drivers
- Test drivers can easily be created automatically
- Values can be hard-coded or read from files

Class Integration and Test Order (CITO)

- Old programs tended to be very hierarchical
- Which order to integrate was pretty easy:
  - Test the "leaves" of the call tree
  - Integrate up to the root
  - Goal is to minimize the number of stubs needed
- OO programs make this more complicated
  - Lots of kinds of dependencies (call, inheritance, use, aggregation)
  - Circular dependencies : A inherits from B, B uses C, C aggregates A
- CITO : *Which order should we integrate and test* ?
  - Must "break cycles"
  - Common goal : least stubbing
- Designs often have few cycles, but cycles creep in during implementation

Outline

1. Regression Testing
2. Integration and Testing
3. Test Process
4. Test Plans

Test Process

We know what to do … but now …
how can we do it?

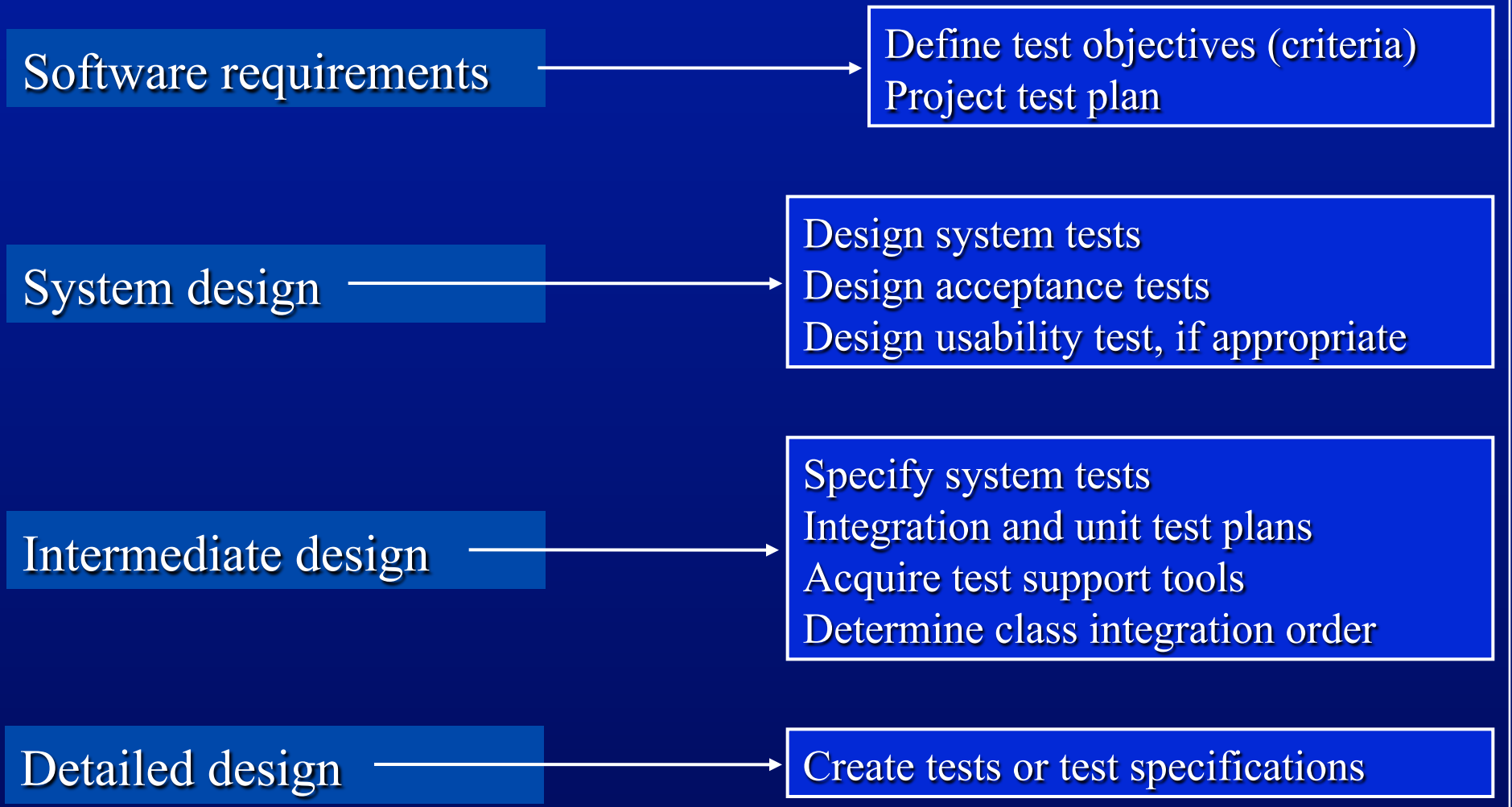
Testing by Programmers

- The important issue is about quality
- Quality cannot be "tested in"!
- If it is not tested it's broken.

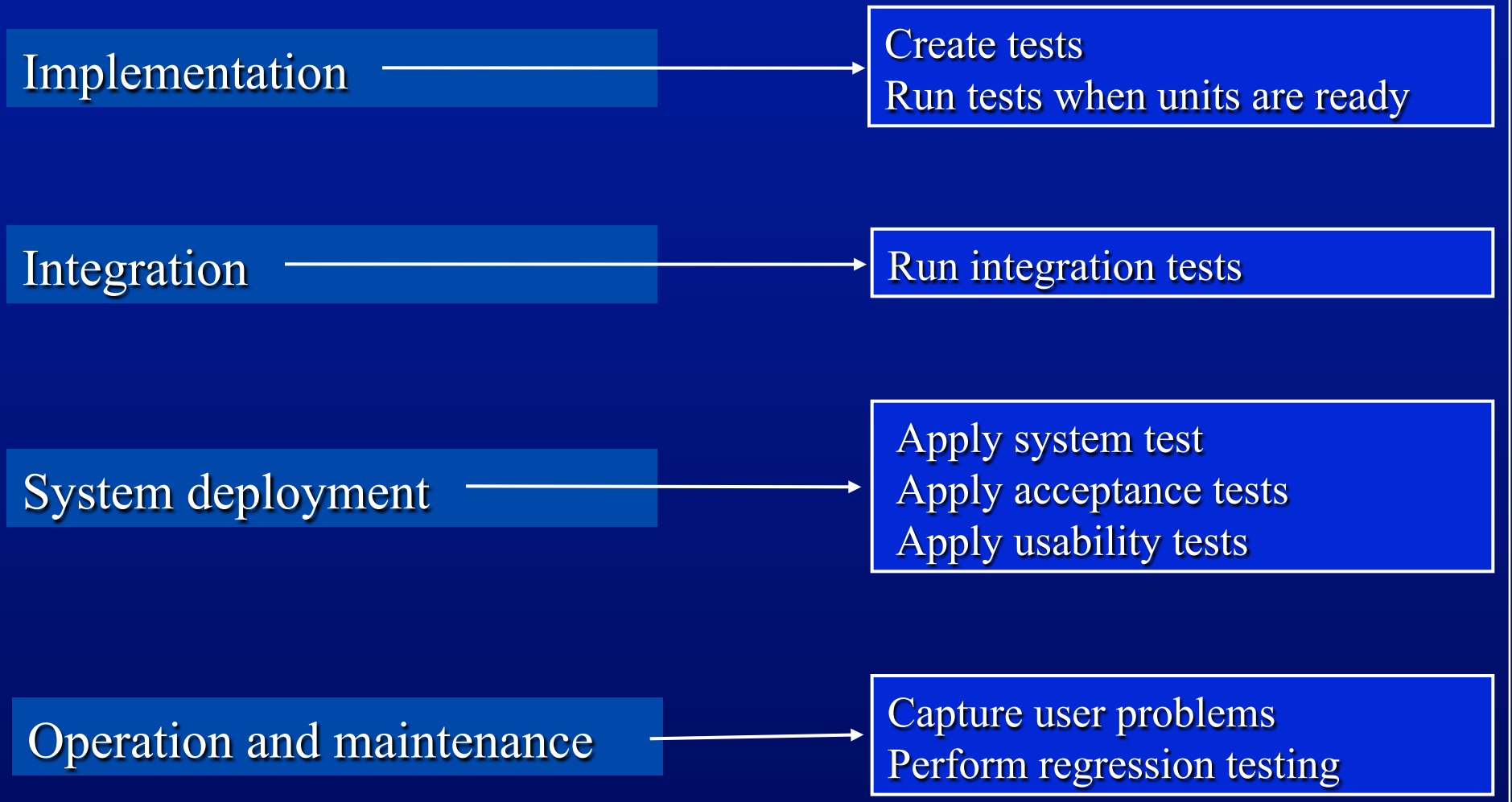
Changes in Software Production

- Teamwork has changed
  - *1970*: we built log cabins
  - *1980*: we built small buildings
  - *1990*: we built skyscrapers
  - *200X*: we are building integrated communities of buildings
- We do more maintenance than construction
  - Our knowledge base is mostly about testing new software
- We are reusing code in many ways
- Quality vs efficiency is a constant source of stress
- Level 4 thinking requires the recognition that quality is usually more crucial than efficiency
  - Requires management buy-in !
  - Requires that programmers respect testers

Test Activities



Test Activities (2)



Managing Test Artifacts

- Don't fail because of lack of organization
- Keep track of :
  - Test design documents
  - Tests
  - Test results
  - Automated support
- Use configuration control
- Keep track of source of tests – when the source changes, the tests must also change

Professional Ethics

- Put quality first : Even if you lose the argument, you will gain respect
- If you can't test it, don't build it
- Begin test activities early
- Decouple
  - Designs should be independent of language
  - Programs should be independent of environment
  - Couplings are weaknesses in the software!
- Don't take shortcuts
  - If you lose the argument you will gain respect
  - Document your objections
  - Vote with your feet
  - Don't be afraid to be right!

Outline

1. Regression Testing
2. Integration and Testing
3. Test Process
4. Test Plans

Test Plans

- The most common question about testing is

" How do I write a test plan? "

- This question usually comes up when the focus is on the document, not the contents
- It's the contents that are important, not the structure
  - Good testing is more important than proper documentation
  - However – documentation of testing can be very helpful
- Most organizations have a list of topics, outlines, or templates

Standard Test Plan

- ANSI / IEEE Standard 829-1983 is ancient but still used

> **Test Plan**
>
> A document describing the scope, approach, resources, and schedule of intended testing activities. It identifies test items, the features to be tested, the testing tasks, who will do each task, and any risks requiring contingency planning.

- Many organizations are required to adhere to this standard
- Unfortunately, this standard emphasizes documentation, not actual testing – often resulting in a well documented vacuum

Types of Test Plans

- Mission plan – tells "why"
  - Usually one mission plan per organization or group
  - Least detailed type of test plan
- Strategic plan – tells "what" and "when"
  - Usually one per organization, or perhaps for each type of project
  - General requirements for coverage criteria to use
- Tactical plan – tells "how" and "who"
  - One per product
  - More detailed
  - Living document, containing test requirements, tools, results and issues such as integration order

Test Plan Contents – System Testing

- Purpose
- Target audience and application
- Deliverables
- Information included
  - Introduction
  - Test items
  - Features tested
  - Features not tested
  - Test criteria
  - Pass / fail standards
  - Criteria for starting testing
  - Criteria for suspending testing
  - Requirements for testing restart
  - Hardware and software requirements
  - Responsibilities for severity ratings
  - Staffing & training needs
  - Test schedules
  - Risks and contingencies
  - Approvals

Test Plan Contents – Tactical Testing

- Purpose
- Outline
- Test-plan ID
- Introduction
- Test reference items
- Features that will be tested
- Features that will not be tested
- Approach to testing (criteria)
- Criteria for pass / fail
- Criteria for suspending testing
- Criteria for restarting testing
- Test deliverables
- Testing tasks
- Environmental needs
- Responsibilities
- Staffing & training needs
- Schedule
- Risks and contingencies
- Approvals

Outline

1. Regression Testing
2. Integration and Testing
3. Test Process
4. Test Plans

Summary – Practical Considerations

- A major obstacle to the adoption of advanced test criteria is that they affect the process
  - It is very hard to change a process
  - Changing process is required to move to level 3 or level 4 thinking
- Most testing is actually regression testing
- Test criteria make regression testing much easier to automate
- OOP has changed the way in which we integrate and test software components
- To be successful, testing has to be integrated throughout the process
- Identifying correct outputs is almost as hard as writing the program

What should you have learned in the course?

- Testing is impossible. There is mathematically no way that you can find a complete set of tests that prove that the software is correct. Except in very specific cases (finite automata).
- You still have to test. There are always bugs in code, any software engineering discipline that reduces bugs is worth using.
- As an Engineer you have to make compromises. So we have the various coverage criteria. None of these guarantee that we have a complete set of test cases, but you know that there is something wrong if your test cases do not cover all statements for example.
- Coverage criteria often give too many test cases.

What should you have learned in the course?

- Many of the coverage criteria will give too many test cases (even though we know that they are not enough).
- So again you have to make trade-offs. That is why there are so many different criteria
- Your job as a test engineer is to make an informed decision about what it means for your test set to be adequate for the job at hand.
- Different levels of testing can be unified by thinking about how you derive coverage criteria for different levels of abstraction of you code.

What we haven't done in this course.

- What we've talked about a lots is how to prove a set of test cases actually satisfy some coverage criteria.
- We haven't done much on how you actually come up with test cases.
- The reason that we've not done this, is that it is hard. For simple programs you trace through the control flow graph finding values of input variables that will give you certain execution paths. There are tools, but they are not complete.
- Coming up with the test cases can be the creative part of testing.

Does anybody actually draw these control flow graphs?

- **Short answer: Probably not, but some of the tools will actually do some of the work.**
- **Long answer: Like many bits of theory in software engineering, you need the theory in the back of your head when you derive test cases and make statements about their completeness. Any amount of testing is a bonus.**

Exam

- Be prepared to give definitions from the book.
- Be prepared to draw some of the control flow graphs from the book.
- You might be given pieces of code and test cases and asked about coverage criteria.
- You might be asked to derive very simple test cases.
- You will be asked some high-level philosophical questions about testing.