CALIFORNIA STATE UNIVERSITY
FULLERTON™

COLLEGE OF ENGINEERING
AND COMPUTER SCIENCE

Advanced Software Process

Part III: The Defined Process

11. Software Testing

Dave Garcia-Gomez
Faculty / Lecturer
Department of Computer Science

Course Roadmap

Part I: Software Process Maturity
- 1. A Software Maturity Framework
- 2. The Principles of Software Process Change
- 3. Software Process Assessment
- 4. The Initial Process

Part II: The Repeatable Process
- 5. Managing Software Organizations
- 6. The Project Plan
- 7. Software Configuration Management (Part I)
- 8. Software Quality Assurance

Part III: Defined Process
- 9. Software Standards
- 10. Software Inspections
- 11. Software Testing
- 12. Software Configuration Management (continued)
- 13. Defining the Software Process
- 14. The Software Engineering Process Group

Part IV: The Managed Process
- 15. Data Gathering and Analysis
- 16. Managing Software Quality

Part V: The Optimizing Process
- 17. Defect Prevention
- 18. Automating the Software Process
- *19. Contracting for Software*
- *20. Conclusion*

Software Testing

- Software Testing Definitions
- Software Testing Principles
- Types of Software Tests
- Test Planning
- Test Development
- Test Execution and Reporting
- Test Tools and Methods
- Real-Time Testing
- The Test Organization

Software Testing

- Software testing is defined as the execution of a program to find its faults.
- While more time typically is spent on testing than in any other phase of software development, there is considerable confusion about its purpose.
- Many software professionals, for example, believe that tests are run to show that the program works rather than to learn about its faults.

Software Testing

- Dijkstra, 1969
  - "Program testing can be used to show the presence of bugs, but never their absence!"
- In fact, testing is an inefficient way to find and remove many types of bugs.
- Software organizations can often improve product quality while at the same time reducing the amount of time they spend on testing.
  - Software inspections
- In spite of its limitations, however, testing is a critical part of the software process.

Definitions [Myers 1976]

- Testing
  - The process of executing a program (or part of a program) with the intention of finding errors
- Verification
  - An attempt to find errors by executing a program in a test or simulated environment; It is now preferable to view verification as the process of proving the program's correctness.
- Validation
  - An attempt to find errors by executing a program in a real environment
- Debugging
  - Diagnosing the precise nature of a known error and then correcting it; Debugging is a correction and not a testing activity

Definitions [Pressman 2005]

- Verification
  - Verification refers to the set of activities that ensure that software correctly implements a specific function.
  - Are we building the product right?
- Validation
  - Validation refers to a different set of activities that ensure that the software that has been built is traceable to customer requirements.
  - Are we building the right product?

Definitions [Pressman 2005]

Verification and validation encompasses a wide array of SQA activities that include formal technical reviews, quality and configuration audits, performance monitoring, simulation, feasibility study, documentation review, DB review, algorithm analysis, development testing, usability testing, qualification testing, and installation testing.

The Seven Types of Software Tests

- Unit or module tests
- Integration tests
- External function tests
- Regression tests
- System tests
- Acceptance tests
- Installation tests

Testing Methods

Two basic ways of constructing tests:

- White box tests
  - Examine the internal design of the program and require that the tester has detailed knowledge of its structure
- Black box tests
  - Are designed without knowledge of the program's internals and are generally based on the functional requirements

Software Testing Principles

- Software testing presents a problem in economics.
- With large systems it is almost always true that more tests will find more bugs.
- The question is not whether all the bugs have been found but whether the program is sufficiently good to stop testing.
- This trade-off should consider the probability of finding more bugs in test, the marginal cost of doing so, the probability of the users encountering the remaining bugs, and the resulting impact of these bugs on the users.

The Axioms of Testing

- A good test case is one that has a high probability of detecting a previously undiscovered defect, not one that shows that the program works correctly.
- One of the most difficult problems in testing is knowing when to stop.
- It is impossible to test your own program.
- A necessary part of every test case is a description of the expected output.
- Avoid non-reproducible or on-the-fly testing

The Axioms of Testing

- Write test cases for invalid as well as valid input conditions
- Thoroughly inspect the result of each test
- As the number of detected defects in a piece of software increases, the probability of the existence of more undetected defects also increases.
- Assign your best programmers to testing
- Ensure that testability is a key objective in your software design

The Axioms of Testing

- The design of a system should be such that each module is integrated into the system only once.
- Never alter the program to make testing easier (unless it is a permanent change)
- Testing, like almost every other activity, must start with objectives.

The Proper Role of Testing

- An examination of even relatively simple programs demonstrate that exhaustive testing is generally impossible.
- Test design thus reduces to the judicious selection of a small subset of conditions that will reveal the characteristics of the program.

Types of Software Tests

- When unit tests are done on a white box basis, they are essentially path tests.
- The idea is to focus on a relatively small segment of code and aim to exercise a high percentage of the internal paths.
- A path is an instruction sequence that threads through the program from initial entry to final exit.
  - The simplest approach is to ensure that every statement is exercised at least once.
  - A more stringent criterion is to require coverage of every path within a program.

Unit Testing (White Box)

- One disadvantage of white box testing is that the tester may be biased by previous experience.
- The tests are often designed by the programmers who produced the code since they may be the only ones who understand it.
- Unfortunately, those who created the program's faults are least likely to recognize them.

Unit Testing (White Box)

- Another limitation of white box testing is coverage.
- Even if each instruction is tested and every branch traversed in all directories, many other possible combinations may still be overlooked.
  - All possible combinations of every loop
  - All possible values for every parameter
  - All allowed and un-allowed but possible values for all variables

Unit Testing (White Box)

- While its disadvantages are significant, white box testing generally has the highest error yield of all testing techniques.
- In fact, in a retrospective study, Thayer and Lipow (1978) state that comprehensive path and parameter testing could potentially have caught 72.9 percent of all the problems encountered by the users of one large program.

Integration Testing

- The proper approach to integration depends on both the kind of system being built and the nature of the development project.
- With a new system, for example, there is no foundation on which to assemble and to run newly developed program fragments.
- The initial problem is thus to establish a test framework on which to run these various elements.

Integration Testing Approaches

- On very large systems it is often wise to do integration testing in several steps.
- Such systems generally have several relatively large components that can be built and integrated separately before combination into a full system.
- Table 11.1 Top-down and Bottom-up Integration [Myers 1976]

Integration Testing Approaches

- In bottom-up testing the modules are individually tested, using specially developed drivers that provide the needed system functions.
- As more modules are integrated, these drivers are replaced by the modules that perform those functions.
- The disadvantage of bottom-up testing are the need for drivers and the amount of testing required before functional testing is possible.

Integration Testing Approaches

- Top-down testing is essentially a prototyping philosophy.
- The initial tests establish a basic system skeleton from the top and each new module adds capability.
- The problem is that the functions of the lower-level modules that are not initially present must be simulated by program stubs.
  - Often requires very sophisticated stubs
- With top-down testing it may be difficult or impossible to test certain logical conditions such as error handling or special checking until most of the system has been integrated.

System Build

- Since integration is a process of incrementally building a system, there is often a need to have special groups do this work.
  - In building large software system, build experts often integrate the components in the system builds (spins), maintain configuration management control, and distribute the builds back to development for module and component test.
  - These experts work with development to establish an integration plan and then build the drivers and integrate the system.

Function Testing (Black Box)

- Functional (or black box) tests are designed to exercise the program to its external specifications.
- The testers are typically not biased by knowledge of the program's design and thus will likely provide tests that resemble the user's environment.
  - The two most typical problems with black box testing are the need for explicitly stated requirements and the ability of such tests to cover only a small portion of the possible test conditions.

Regression Testing

- Testing is both progressive and regressive.
- The progressive phase introduces and tests new functions, uncovering problems in the newly added or modified modules and in their interface with the previously integrated modules.
- The regressive phase concerns the effects of the newly introduced changes on all the previously integrated code.

Regression Testing

- Problems arise when errors made in incorporating new functions affect previously tested functions.
- In large software systems these regression problems are common.

Regression Testing

- Regression testing is particularly important in software maintenance.
- The basic regression testing approach is to incorporate selected test cases into a regression test bucket that is run periodically in an attempt to detect regression problems.
  - In many software organizations regression testing consists of re-running all the functional tests every few months.
  - This delays the discovery of regression problems and generally results in significant rework after every regression run.

System Test

- The purpose of system tests is to find those cases in which the system does not work as intended, regardless of the specifications.
- Regardless of what the contracts says, if the system does not meet the user's real needs, every one loses.
  - Some areas that need special system test attention are operational errors and intentional misuse.
  - With the increasingly pervasive use of sophisticated computing systems in modern society, we must also be more concerned about the problems of intentional misuses by vandals, criminals, and even terrorists.

System Test

- The program objectives should specify what is to be done for the end users.
- When these objectives are not specific, neither the testing nor the system design and implementation can be accurately planned.
- When systems are built without good objectives, they must often be rebuilt before they can be used.

System Test

- System test planning can be done by a special organization with a reasonably direct link to the end users, possibly through a users group or by close contact with selected key users.
- System test planning will often uncover problems that have been undiscovered throughout the entire development process.
- Categories of System Tests [Table 11.2]

System Test Categories

- Load/stress
- Volume
- Configuration
- Compatibility
- Security
- Performance
- Installability
- Reliability/Availability
- Recovery
- Serviceability
- Human factors

Acceptance and Installation Tests

- After all development testing is completed, it is often advisable to try the system in a real user's environment.
- Even though such field tests generally require special support, the results are invariably worth far more than the added costs.
- Often when software is developed under contract, some form of acceptance testing is required in a real or simulated user environment.

Acceptance and Installation Tests

- When this is not the case, the developers can often arrange with selected users to act as special "beta test" sites in return for special installation support and early program availability.
- If end user testing is not practical, it may be possible to try the system in an internal application environment.

Test Planning

- Test planning starts with an overall development plan that defines the functions, roles, and methods for all test phases.
- Test Plan Review Checklist [Table 11.3]

The Test Files

- It is helpful to establish a development file system to retain this information both for the test plan in general and for each test and test case.
- These files should be defined and provisions made to retain them as part of the configuration management plan.
- Such files provide a useful way to relate all the material on each test and help to establish a test DB.

The Test Files

- This file should also contain the specifications, design, documentation, review history, test history, test instructions, anticipated results, and success criteria for each test case.

Test Success Criteria

- Establishing the test success criteria is probably the most difficult part of test planning, due to both the developer's attitudes and to their general lack of quantitative experience.
- The developer's viewpoint is that a successful test is one that executes completely without encountering a problem.
- The testers measure success, however, by bug found.

Test Success Criteria

- Each test should be planned against yield criterion (bugs per test run) that is determined from prior experience.
- The testers' goal should then be to increase test case yield, while the developers' is to produce code that will reduce test yield.

Test Success Criteria

- When the testers acknowledge that they cannot meet their yield objectives, their work should be technically reviewed to see if it is adequate.
- The testers should determine why they did not catch any bug that is later found and how to improve their tests so they would catch similar problems in the future.

Test Development

- After developing the test plan, the next job is to produce the test cases.
- The decision on what test cases to develop is complicated by two factors.
  - Full test coverage is generally impossible.
  - There is no proven comprehensive method for selecting the highest yielding test conditions.

Test Development

- From an experiment, Myers (1976) found that, even with very experienced programmers, the likelihood of their producing tests that covered a reasonably complete set of all test combinations was quite small.
- Test Results for Triangle Program [Table 11.4]
  - The Triangle Program read three integers representing the sides of a triangle and printed out whether the triangle was scalene, isosceles, or equilateral.
  - The percentage of programmers (test design) who checked for specific bugs

Test Development

- Because even experienced software professionals make errors and overlook conditions, it is essential to hold walkthroughs of the test plans and inspections of the test cases.
- Unit Test Review Checklist [Table 11.5]

Test Coverage Techniques

- One approach to the test coverage problem is to view the program as a graph with nodes and to ensure that these nodes and the paths between them are adequately covered by the tests.
- The adequacy of test coverage can be roughly assessed by comparing the number of test paths executed with the complexity of the program graph.

Path Selection In Unit Test

- While there are no simple rules for selecting testing paths to achieve adequate test coverage, some general guidelines can help:
  - Pick defined (in the requirements) functional paths that simply connect from entrance to exit
  - Pick additional paths as small variations, favoring short paths over long ones, simple paths over complicated ones, and logically reasonable paths over illogical ones.
  - Next, to provide coverage, pick additional paths that have no obvious functional meaning

Path Coverage in Functional Testing

- For functional or black box testing, one technique for selecting tests is based on functional paths.
  - Here the program functions are considered as a set of transaction flows that form functional paths through the program.
  - These transaction flows can be reduced to graphs of nodes and paths much as in white box testing, and the tests are again designed to achieve coverage of these nodes and paths.
  - These then form the paths to be covered by the functional tests in much the way as with unit tests.

Path Selection

- With a defined set of functional paths it is necessary to decide which paths to cover.
- While there are no guidelines for functional path selection, it is generally a good idea to start by considering the data handled by the program.
- Functional Testing Data Selection Guidelines [Table 11.6]

Path Selection

- While there is no magic way to select a sufficient set of practical tests, the objective is to test reasonably completely all valid classes for normal operations and to exhaustively test behavior under unusual and illegal conditions.
- One way to look at the adequacy of a test is to determine the degree to which all the requisite conditions have been covered.

A Suggested Path Selection Technique

- Prather (1987) and Myer (1979) also have suggested techniques for handling path selection.
- There general approach is to establish equivalence classes based on the various parameters and conditions for the program.
- For each parameter or condition, an equivalence class would cover all valid values.
- An equivalence class would also be established for each invalid class, such as above range or below range.

Test Case Design Guidelines

- Since the objective of test case design is to catch errors rather than to prove the program works, it is wise to consider the common types of errors.
- Major errors [Goodenough and Gerhart 1975]
  - Missing control flow paths
  - Inappropriate path selection
  - Inappropriate or missing action

Test Case Design Guidelines

- Even though there are no simple rules for test case design, some general guidelines can be helpful:
  - Do it wrong
  - Use wrong combinations
  - Don’t do enough
  - Do nothing
  - Do too much

Test Case Design Guidelines

- In addition, some more overall considerations are:
  - Don't forget the normal cases!
  - Don't go overboard with cases.
  - Don't ignore structure.
  - Remember that there is more than one kind of test.

Test Reports

- At the conclusion of each test, a test report should be produced.
- The amount of detail included depends on the purpose of the test and the audience for the report.
- In the case of acceptance tests, a considerable amount of detail is generally required, including the qualification of the test observers, detailed data on test results, and the plans to resolve the trouble reported.

Test Reports

- Sample Test Incident Form [Table 11.7]
- Sample Bug Report Form [Table 11.8]

Bug Classification

- As part of the test reporting process, it is desirable to have standard definitions for bug severity.
- This allows the follow-up actions to be prioritized and the critical items to be tracked.
- Bug Severity Classification – Military System [Table 11.9]
- Bug Severity Classification [Beizer 1983] [Table 11.10]

Test Analysis

- Since tests are intended to gather data on programs, the resulting information should be analyzed and used to make decisions.
- In general the evidence for jungle-like conditions can be found by examining the test results and the characteristic of the problem found.

Test Tools and Methods

- Test Tools [Table 11.11]
  - A list of available testing tools in a document TRW prepared for the Air Force on “Software Testing and Evaluation

Real-Time Testing

- Real-time testing can be particularly difficult because the development work is done on a host system and then compiled for execution on a target system.
- Typically a reasonable set of test and debug facilities is available for the host environment but the target system is generally much more sparsely equipped.

The Test Organization

- Bug guilt [Beizer 1983]
- Programmers are inherently incapable of effectively testing their own programs.
- Not only do they feel guilty about the existence of bugs, but they are biased by the creative work they have done.
- One remedy is to separate testing from program design and implementation.

The Test Organization

- Such special test groups often assume test responsibility after unit testing.
  - This generally makes sense because white box unit tests generally require an intimate knowledge of the program's internal structure.
  - Unit test standards and walkthroughs of unit test plans can help the programmers do a reasonably effective job of unit testing their own code.

The Test Organization

- Following unit test it is generally advisable to transfer test responsibility to a dedicated test group.
  - This group's role is to find as many bugs as possible.
  - As they gain experience, such groups can become extraordinarily effective at finding those odd pathological cases that will only come up once in a billion executions.

References

Humphrey, Watts S., *Managing the Software Process*, The SEI Series in Software Engineering, Addison-Wesley, 1989. (29th Printing, May 2003) (ISBN 0-201-18095-2)