Lecture 9
Software Testing: Part 2

Dr Peter T. Popov
Centre for Software Reliability

29th November 2018

Goals and topics covered

- Goal: awareness of
  - Different forms of testing
    - Unit testing
    - Integration testing
    - System testing
- The need for system testing
  - Reliability assessment
  - Performance evaluation
- Role of operational profile
  - Ways of constructing a realistic operational profile for testing
- Will illustrate the concepts with modern testing tools

2

Types of software testing and their use in the software lifecycle

3

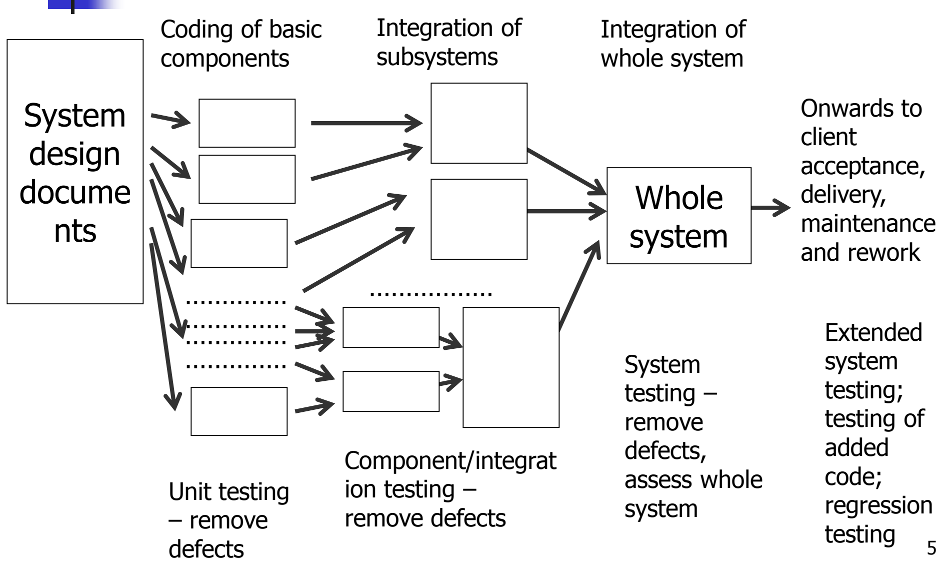
Recall: classifications of software testing

We can describe testing activities from various viewpoint:

- scope of testing
  - Unit testing
  - Integration testing
  - System testing
- use or not of knowledge of the internal structure of the software to help the testing
  - Black-box – the internals of the tested software (what exactly the code is) are not known; just its specification
  - "White-box" – the tester uses knowledge of the code
- goal of testing
  - Assessment (reliability, performance, resilience)
  - Debugging (fault finding)
- By the phase of software lifecycle, the parties involved, the rules applied to decide that we have "tested enough"
  - e.g., regression testing, acceptance testing, user testing, beta testing, independent testing, coverage testing, mutation testing ...

4

Testing in the big picture of development



5

Scope of testing: Unit Testing

- Unit testing tests small units of software in isolation
  - to detect faults in these units
- Units may be:
  - Individual operations/methods within a class
  - The entire class with its attributes and methods
  - Composite components with defined interfaces used to access their functionality
- The tests should show that when used as expected, the unit that you are testing does what it is supposed to do.
  - test cases should ideally be such that if there are defects in the component, they are revealed by test cases.
- 2 types of unit test cases are needed:
  - *normal* operation of a program: inputs and should show that the component works as expected.
  - *abnormal inputs*: are these processed as specified?
- full (100%) coverage targets are *often feasible* at the unit level

6

Unit testing: testing a class, an object

- "Complete" test coverage of a class involves:
  - testing all operations of a class
  - setting and interrogating all attributes
  - exercising the object in all possible states defined by the object's state machine
- Inheritance makes it more difficult to design object/class tests
  - a method of a subclass may differ from the same of the parent class
- Automation: When possible, unit testing *should be automated*: tests are run and checked without manual intervention.
  - You then use a test automation framework (such as Junit for Java) to write and run your unit tests.

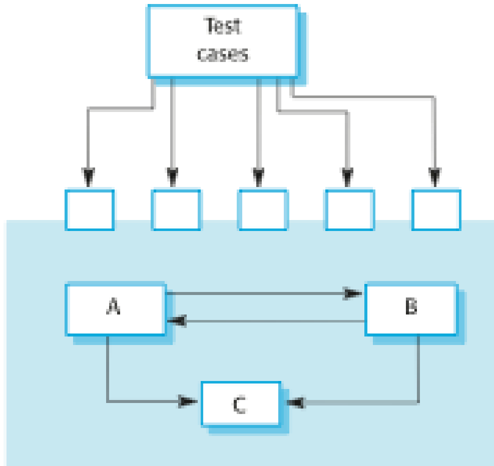
7

Component testing

- Software components are made up of interacting objects
- Components or integrated sets of components are tested by applying test cases (requests, calls, messages) to their provided ***interface***(s)
- Testing an individual component checks that the ***component interface behaves according to its specification.***

8

Interface testing



The test cases cause each one of A, B and C to perform its function, calling on the functions of the other components

9

What bugs do we find by this testing?

- If unit tests on the individual objects within the component were successful, one expects now to detect, for instance:
  - Interface misuse
    - A calling component calls another component and makes an error in its use of its interface, e.g. parameters in the wrong order.
  - Interface misunderstanding
    - A calling component makes wrong assumptions about the behaviour of the called component
  - Timing errors
    - The called and the calling component operate at different speeds and out-of-date information is accessed.
- These are examples of the components being apparently "correct" according to their own specifications, but these component specifications being:
  - mismatched (wrong design of how components should interact)
  - or misunderstood by programmers (wrong implementation)
    - e.g. should calls with certain unintended parameter values be never made by the calling components, or be treated by the called component?
- Another possibility is that the interface testing reveals bugs in the units' internal operation that we had *missed in unit testing*

10

System Testing

- System testing ***during development*** involves integrating components to create a version of the whole system and then testing the entire integrated system.
- it checks, again, for correct ***interactions between components***:
  - that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.
- System testing tests the *emergent behaviour* of a system: what you cannot see from testing its parts
- Only in system testing is each system part certain to receive inputs that it will receive in real operation

11

System Testing

- System testing requires integrating:
  - reusable components that have been *separately developed* or acquired *off-the-shelf*, to be integrated with newly developed components
  - components developed by different team members or sub-teams, integrated at this stage. System testing is a *collective rather than an individual process*.
- In some companies, system testing may involve a ***separate testing team*** with no involvement from designers and programmers
  - such *independent testing* may be a contractual requirement

examples of techniques suitable for system testing:

- use case testing,
- requirement coverage testing,
- statistical (operational) testing

12

Testing policies

- Exhaustive system testing is impossible, so ***testing policies*** which define the required system test coverage (known as the testing "*stopping rule*")
- ***Examples*** of testing policies (coverage criteria):
  - All system functions that are accessed through menus must be tested.
  - Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  - where incorrect user inputs are possible, all functions must be tested with both correct and incorrect input.
- *A risk of these policies*: allowing testing with short sequences of operations:
  - *but many bugs reveal themselves after a long period of operation*
- *A solution*: testing through realistically long operation, e.g. long user sessions, while checking whether the sequence of user commands together satisfy the test policy.
  - "Memory leaks" phenomenon

13

Regression testing

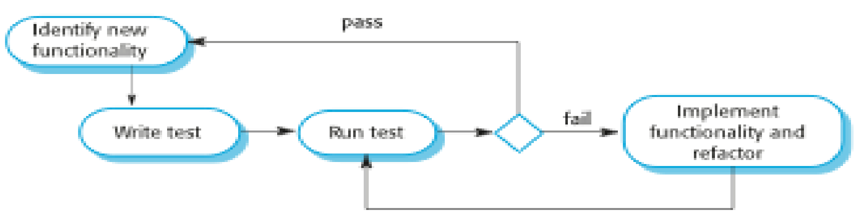
- (Usually) *system* testing *after changes* to check that the changes have not 'broken' previously working functionality.
  - major changes often create new defects
  - bug fixes sometimes create new bugs:
    after checking that the failure caused by the old bug no longer happens (re-testing with the specific test case[s]
- In a manual testing process, regression testing is expensive but, with automated testing, it is simple and straightforward.
  - ***All tests*** are rerun every time a change is made to the program.
- All tests must be passed before the change is committed.

Does not *guarantee* that the new version has no new bugs, but with a large set of test cases demonstrates it is *unlikely* to have serious *new* bugs

14

Test-driven development (TDD)

- an approach to program development in which you inter-leave testing and code development and...
- ... *tests are written before code* and 'passing' the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don't move on to the next increment until the new code passes its test.
- introduced as part of *agile methods* such as Extreme Programming. However, it can also be used in plan-driven development processes.



15

TDD (2)

- TDD process activities:
  - Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
  - Write a test for this functionality and implement this as an automated test.
  - Run the test, along with all other tests that have been implemented before.
    - Initially, you have not implemented the functionality so the *new* test will fail.
    - Implement the functionality and re-run the test.
    - Once all tests run successfully, you move on to implementing the next chunk of functionality.

16

TDD (3)

- Benefits of test-driven development
  - Code coverage
    - Every code segment has at least one associated test so all code written has at least one test.
  - Regression testing
    - A regression test suite is developed incrementally as a program is developed.
  - Simplified debugging
    - When a test fails, it should be obvious where the problem lies.
  - System documentation
    - The tests themselves are a form of documentation of what the code does.
    - *NB if the test cases give less than full coverage, they only document parts of the possible behaviour*

17

Release testing

- *System testing* of a particular release of a system that is intended for use outside of the development team.
- Primary goal: convince the *supplier* of the system that it is good enough for use.
  - Release testing has to show that the system delivers its specified functionality (does not fail during normal use) satisfies its non-functional requirements (e.g. performance and dependability).
- Usually black-box (tests are only derived from the system specification).

18

User Testing

- User, or customer, testing is testing, in which users or customers provide input and advice on system testing
- User testing is essential, even when comprehensive system and release testing have been carried out.
- Reason: influences from the users' working environment (details of how they use the system) have a major effect on the reliability, performance, usability and robustness of a system
    - In the developer's testing environment, we won't be sure whether the user's environment is reproduced correctly
    - Even very thoroughly tested systems have failed often and with severe effects when put into real operation.

19

User Testing (2)

- Forms of user testing
    - Alpha testing
        - Users of the software work with the development team to test the software at the developer's site.
    - Beta testing
        - A release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.
    - Acceptance testing
        - Customers test a system to decide whether or not it is ready to be accepted from the developers and deployed in the customer environment. Primarily used for ***bespoke systems***.

20

Statistical Testing
(also called *operational* testing)

- Involves exercising the software
  - with test cases that are ***statistically representative*** of real use
  - to check whether it has reached the required level of quality:
    - Reliability - the system will be reliable enough in the intended operational environment
    - Performance – the system is acceptably fast in the intended operational environment
- The goal is not to find defects
  - Test cases for defect testing are (usually) chosen to be
    - likely to reveal typical defects
    - so, atypical of actual usage
    - ... and useless for assessing actual reliability/performance
  - note: for software without 'typical' bugs, operational testing can efficiently reveal those bugs that cause failures ***most frequently***
- The measurement targets the anticipated operational environment
  - test cases are chosen to replicates system usage in ***that*** environment
    - e.g. that of a specific client organisation (bespoke software)
    - e.g. specific types of users, and their combination in the user base.

21

Statistical testing activities

- Establish the ***"operational profile"*** for the system.
  - that is, probabilities of the various ways the users of the software are expected to use it
  - test cases will be chosen ***"randomly"*** according to the ***"profile"***.
    - choosing "randomly" does not mean "all demands have the same probability"
- Construct ***test suites*** (sets of test cases with their data) reflecting the operational profile
  - the various use cases, operations, ranges of parameter values and internal states have ***the same frequencies*** as in real operation
- Test the system (run the test cases from the suite) and record the observations
  - test 'oracle' discriminates between test cases that are processed successfully and those on which the software fails.

22

Statistical testing activities (2)

- After sufficiently long testing, the measures of interest are computed using an appropriate *statistical procedure*
  - Performance measures:
    - Number of demands served per second
    - Mean response time for a demand (performance evaluation)
    - Frequency of response times longer than a given required bound
  - Reliability measures
    - *failure rate* (failures/hour), or *mean time between failures*
    - Probability of failure free operation for a given period of operation (*time* or *number of demands*).
    - e.g. when used in this organisation,
      - this file server's estimated mean time between failures is 3500 hours"
      - this database server software exhibits a failure every 2 million transactions (on average)"

23

Operational profile (OP)

- A way of describing an OP is to identify
  - *types of users* with different usage habits
    - If we take a use case model this is already given (actors)
  - for each type of user, its contribution to overall use
    - What are the use cases available to each user (actor)
  - for users of each type, the relative *frequencies* of the various *operations* they request from the system
  - for each operation, the frequencies of the various *values of inputs*

24

Operational profile - example

- Consider software for a telephone switch (connects phone calls, adds them to bills, etc.)
- Consider a fixed line phone user who can make two types of calls:
  - A local call (i.e. within the same town)
  - A long distance call
- The OP for this user (the way this user uses the software) is defined by two (conditional) probabilities:
  - The probability P_L that a call, when made, is a local call
  - The probability P_D that a call, when made, is a long distance call
  - The sum of probabilities must be equal to 1
- Now consider two user types with different profiles:
  - $User_1$: $P_L(User_1) = 0.3$, $P_D(User_1) = 0.7$.
  - $User_2$: $P_L(User_2) = 0.5$, $P_D(User_2) = 0.5$.

25

Operational profile (2)

- What would be the OP for the ***switch*** if used by the two user types:
  - what is the probability of a call, by either of the two user types, being of a ***particular type*** (local vs. long distance)?
  - ***Case 1:*** suppose that they make the same number of calls per unit of time, i.e.:
    - $P(User_1) = 0.5$ and $P(User_2) = 0.5$.
  - From the formula of total probability we derive:
    - $P_L(User_1 \text{ or } User_2) =$
      $P(User_1) * P_L(User_1) + P(User_2) * P_L(User_2) =$
      $0.5 * 0.3 + 0.5 * 0.5 = 0.4$.
    - $P_D(User_1 \text{ or } User_2) = 0.5*0.7 + 0.5*0.5 = 0.6$.

26

Operational profile (3)

- ***Case 2:*** Suppose that for every 100 calls by user type $User_1$, users of type $User_2$ make 300 calls.
  - If you pick a phone call at random, the probabilities of it being by one user type or the other are respectively:
  - $P(User_1) = 0.25$ and $P(User_2) = 0.75$

- Now let us compute the profile for the telecom switch:
  - $P_L(User_1 \text{ or } User_2) = 0.25*0.3 + 0.75*0.5 = 0.45$.
  - $P_D(User_1 \text{ or } User_2)= 0.25*0.7 + 0.75*0.5 = 0.55$.

- Clearly the profile generated in case 2 is different from the profile generated in case 1.

27

Operational profile (4)

- Constructing an OP usually requires defining an ***automatic procedure*** (possibly writing *specialised software*) that generates a large number of test cases (a ***test suite***) for use in statistical testing

- possible methods include:
  - sampling from *logs ("traces")* of use of this system or of similar systems
  - samples of the data that will be processed
  - beta testing (if there are enough testers and time)
  - *simulators* (of human users, of physical systems) which randomly generate sequences of inputs following patterns observed/expected in reality

28

Operational Profile (5): the role of data

- Each test case includes data (recall the details needed in use case testing)

- The data (e.g. the *phone numbers* that are called in our example) are also part of the operational profile.
  - E.g. We may provide a list of all *valid numbers*
    - we may identify the relative frequency of numbers from various classes that the switch must process differently
      - e.g. billing class – not just local/long distance, but normal, toll free, premium; numbers with different numbers of digits; ...
    - how to specify frequencies? e.g. use logs of operation of existing telephone switches

29

Operational Profile: An example

- Now let us try to *generate a test suite*. Consider this (simplistic) example:
  - users make calls of two types (as described earlier)
  - we need a test suite of 10,000 test cases (i.e. calls) for testing a phone switch with the two users described earlier:
- How will we populate the test cases?
- Hint:
  - a test case will be a sequence of calls, each to a randomly selected number among those *that could be called* (even by mistake) , either local or long distance
  - The numbers are selected at random from all available numbers:
    - all equally likely ? Unrealistic – some well-known numbers are more likely to be called
    - instead make some numbers more popular (non-uniform distribution)
    - also, testing must mimic how in real use there will be periods in which specific numbers are called *more frequently than usual*.

30

OP: Further reading

- on Moodle you find:
  - works by the late John Musa who popularised the concept the 70s and 80s while working for AT&T in the USA.
  - A thorough report produced for the European Space Agency, "Guidelines for Statistical Testing“ (by Littlewood and Strigini):
    - section 3 to 8 useful background
    - section 7 about ways of recreating the "operational profile"
    - also there: examples of how one designs "test oracles"
  - also: chapter 5 of the Handbook of Software Reliability Engineering http://www.cse.cuhk.edu.hk/~lyu/book/reliability: especially up to 5.3.

31

Performance testing

- This may be simpler/cheaper than testing for reliability
  - instead of an oracle, requires measuring performance (e.g. throughput, response time)
  - no need to wait for (rare) failures
- The operational profile is often ***known*** (or, often, given by a ***performance benchmark***, to compare products)
- however, we may vary the load, e.g.
  - expected load (e.g. given expected load on a shared system, what response time do users experience?)
    - predict likely performance
  - steadily increased load until performance becomes unacceptable:
    - assesses how well the software will cope with unusual/unforeseen usage
  - Stress testing: a form of performance testing where the system is deliberately *overloaded* to test its failure behaviour.

    (does it fail? How often? Is failure "graceful" or "catastrophic"? E.g., a server refusing some jobs vs. crashing of serving no jobs)

32

Performance testing: Example with benchmarks

- The Transaction Processing Council's benchmark TPC-C suite (www.tpc.org) for database servers:
  - An on-line transaction processing (OLTP) benchmark for a warehouse application.
  - The benchmark uses a database with 9 tables
- TPC-C defines a warehouse application with 5 types of transactions. Each transaction type consists of a number of statements either reads only or reads and writes.
- The operational profile is ***given*** by the TPC-C standard:
  - New-Order 45% , enters a new order from a customer
  - Payment – 43%, updates customer balance to reflect a payment
  - Order-Status – 4%, retrieves status of customer's most recent order
  - Delivery – 4%, delivers orders (done as a batch transaction)
  - Stock-Level – 4%, monitors warehouse inventory
- A measure of interest:
  - Number of "New-Order" transactions per second.

33

Exercise

- Suppose I have the following TPC-C scores of 2 RDBMS :
  - Product A: 300 New-Order transactions per second
  - Product B: 200 New-Order transactions per second
- I would like to deploy my application with the faster product. I know that my application uses the same database (9 tables) as TPC-C. However, I also know that my application will use the database with a *different operational profile*.
- Can I base my decision on the TPC-C scores and choose Server A?
  - Yes – why?
  - No – why not? In this case, what further measurement should I undertake before deciding which server to use?

34

Statistical Testing for Reliability vs for Performance Evaluation

- Essential differences
  - statistical testing for ***reliability requires an automated test oracle***; ***performance evaluation does NOT require an oracle***
    - Oracle is responsible for discriminating between test cases being processed by the software ***correctly*** and those processed ***incorrectly***.
  - typically a large number of tests are processed automatically (thousands or even millions tests, or months of simulated continuous usage)
    - failures are rare: estimating whether they are rare enough requires a lot of operational testing
    - performance testing usually requires much less testing
  - for some applications, constructing an accurate automatic oracle may be ***difficult***, as it requires
    - high ***failure coverage***, otherwise the reliability assessment will be over-optimistic
    - ***low*** enough rate of ***false alarms*** not to overwhelm human testers who need to review them

35

Tools for statistical testing

- JMeter (http://jmeter.apache.org/) has been a popular choice for performance measuring under variable load.
- Selenium (http://seleniumhq.org/) automates browsers. Its primary purpose is to automate web applications for testing purposes.
  - e.g. a City student used Selenium recently to test statistically his on-line game.

36

A live demo with jMeter

37

Summary

- We have reviewed
  - the basic needs for software testing
    - Test case generation, inputs/stimuli/demands as test cases,
    - The essential role of test oracles
    - Different forms of testing during development
  - The needs of statistical testing *for performance and reliability assessment*
    - test cases must be a realistic, fair sample of the possible demands, according to the "operational profile" (i.e. users' environment of use)
    - discussed how one can identify the correct operational *profile* for testing
- reference readings:
  - Sommerville *Software Engineering,* "Software testing" chapter
  - Pressman *Software Engineering: A Practitioner's Approach, 7th edition*

38