Lecture 8
Software Testing: Part 1

Dr Peter T. Popov
Centre for Software Reliability

22nd November 2018

Goals and topics covered

- Goal: awareness of
  - the need for testing
  - the various uses of software testing
  - the various approaches to testing that fit its different uses
  - and in particular
    - ways of generating test cases, checking results, setting policies for "sufficient testing".. with their pros and cons
- In particular, some detail of techniques
  - use case based testing
  - extreme value testing
  - path coverage policy
  - statistical testing for assessing reliability, performance
- with exercises on specifying testing

2

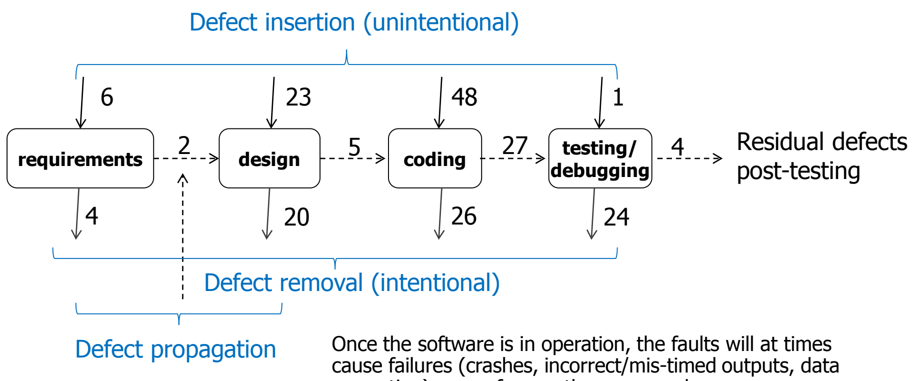
"Testing" means...

- running a program to see whether it performs correctly or not (*fails*)
  - usually many times
- we give it inputs and look at what outputs it produces in return:
  - do they match what is required? Good
  - do they not? We usually need to find the *defect* (also called "*fault*", "*bug*") causing this *failure,* and correct it
  - or sometimes we are interested in assessing *non-functional requirements*: performance, reliability, usability
    - over many tests in order to be *confident* in the assessment results
- testing is part of the ***verification and validation*** process
  - which also includes *static* techniques (clever ways of examining the code to find any defects)

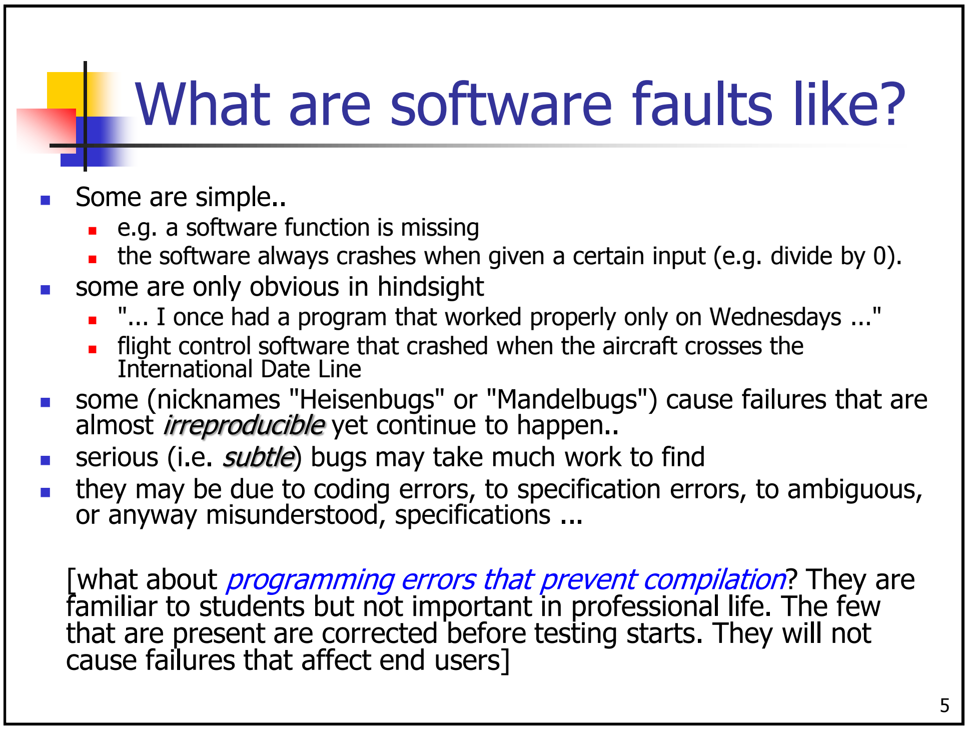
Testing is usually a large part (40-50%) of software development cost

Why testing?

The software development process is ***not perfect***
At each stage, developers *should* deliver items (requirements, designs, code) that they have checked to be correct; but *in reality*, these items contain some defects (*faults*, *"bugs"*), of which they remove some, but others become inputs to the next stage, for instance:


Defect insertion (unintentional)
6
23
48
1
requirements
2
design
5
coding
27
testing/ debugging
4
Residual defects post-testing
4
20
26
24
Defect removal (intentional)
Defect propagation

Once the software is in operation, the faults will at times cause failures (crashes, incorrect/mis-timed outputs, data corruption): some frequently, some rarely.

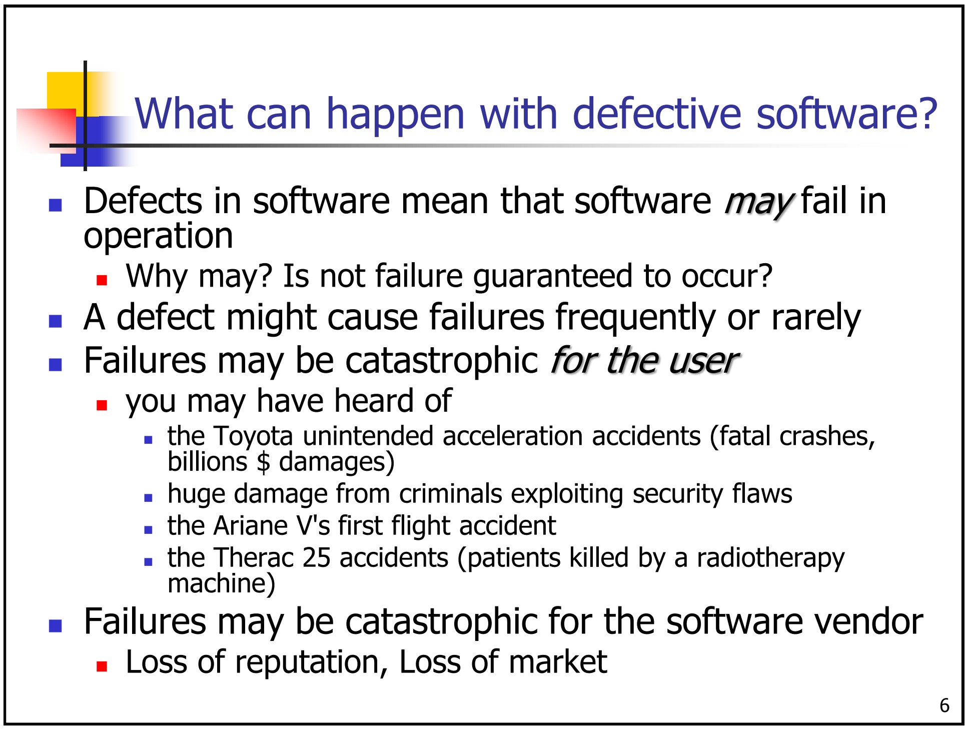


What are software faults like?

- Some are simple..
  - e.g. a software function is missing
  - the software always crashes when given a certain input (e.g. divide by 0).
- some are only obvious in hindsight
  - "... I once had a program that worked properly only on Wednesdays ..."
  - flight control software that crashed when the aircraft crosses the International Date Line
- some (nicknames "Heisenbugs" or "Mandelbugs") cause failures that are almost *irreproducible* yet continue to happen..
- serious (i.e. *subtle*) bugs may take much work to find
- they may be due to coding errors, to specification errors, to ambiguous, or anyway misunderstood, specifications ...

[what about *programming errors that prevent compilation*? They are familiar to students but not important in professional life. The few that are present are corrected before testing starts. They will not cause failures that affect end users]

5



What can happen with defective software?

- Defects in software mean that software *may* fail in operation
  - Why may? Is not failure guaranteed to occur?
- A defect might cause failures frequently or rarely
- Failures may be catastrophic *for the user*
  - you may have heard of
    - the Toyota unintended acceleration accidents (fatal crashes, billions $ damages)
    - huge damage from criminals exploiting security flaws
    - the Ariane V's first flight accident
    - the Therac 25 accidents (patients killed by a radiotherapy machine)
- Failures may be catastrophic for the software vendor
  - Loss of reputation, Loss of market

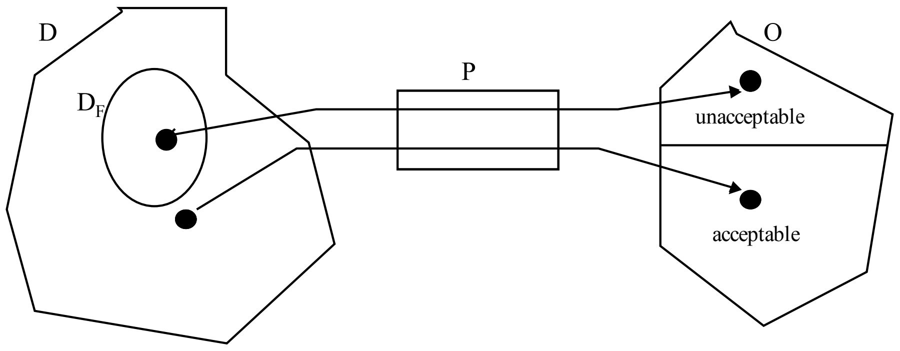
6

Role of testing

- It is essential
  - we can never be sure that the software we produced works well, unless we seriously test it
- It is not a substitute for producing software that is good (well-structured, based on solid designs, etc.) *before* testing
  - one must think before writing code, trying to avoid bugs, code which should work properly.
  - developers who develop without thinking, because "if there are bugs, testing will take care of them"
    - cause an ***expensive series of test-fix-test*** again cycles
    - and the software may ***never become good enough*** to use
- good thinking, design and coding make serious bugs ***rare***
  - and then testing helps ensure that those rare ones are removed

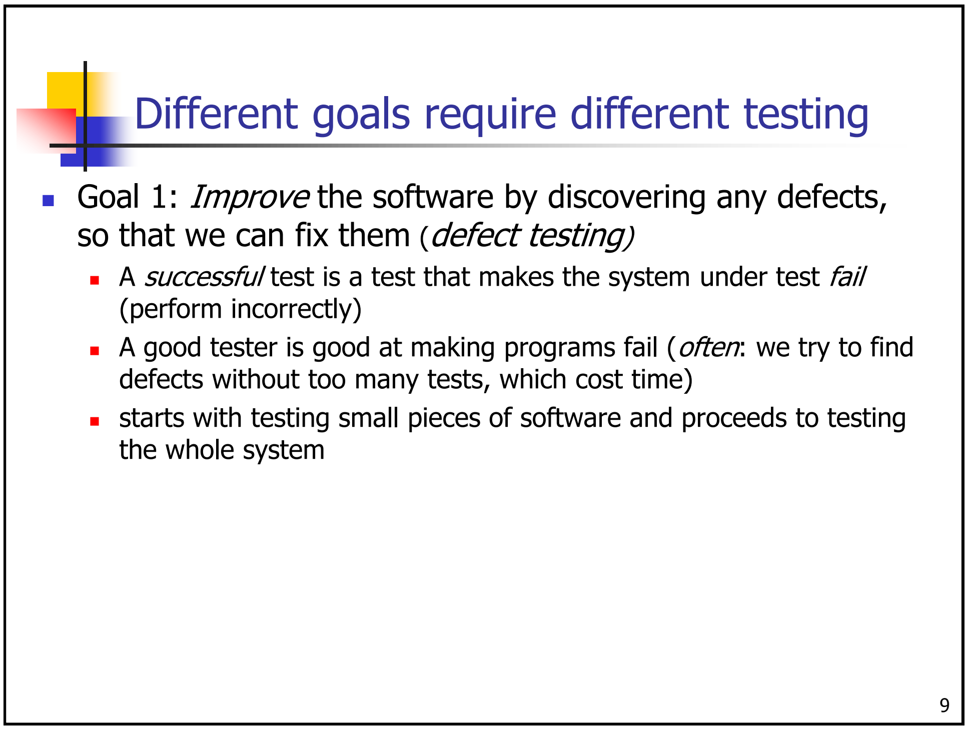
7

Software Failure Process



Conceptual model of the software failure process. Program execution is a mapping from the set D, of all possible demands (sequences of input values), into the set of output sequences, O.
For each demand, only some output sequences are correct ("acceptable").
The *failure set* D_F is the set of all demands that the program, P, does not execute correctly: they map into ***unacceptable output sequences***.

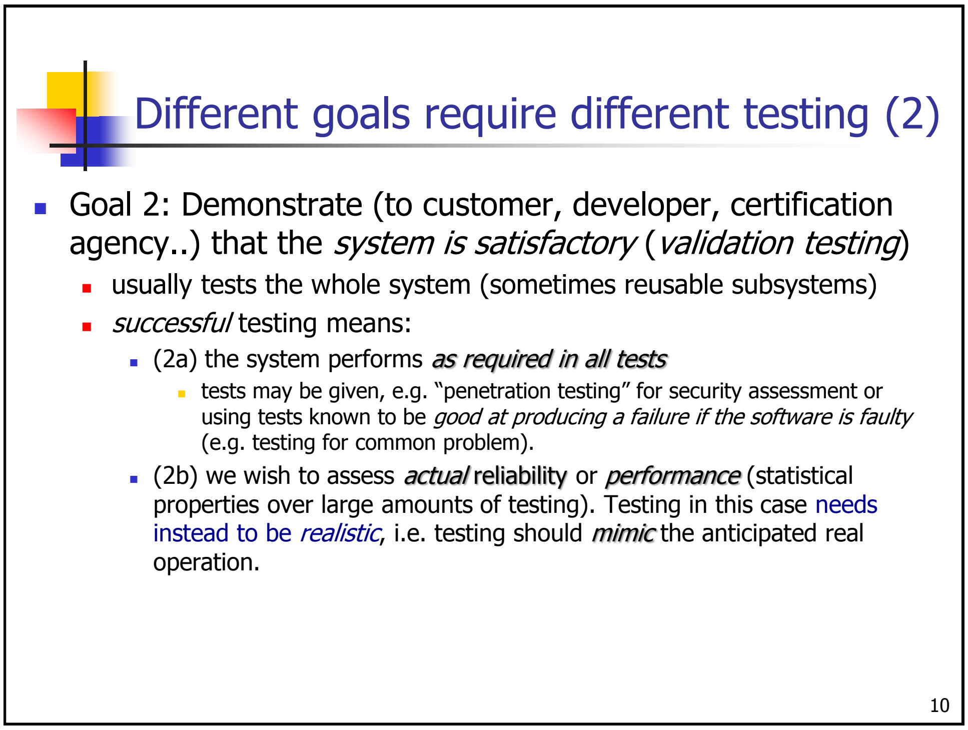
8



Different goals require different testing

- Goal 1: *Improve* the software by discovering any defects, so that we can fix them (*defect testing)*
  - A *successful* test is a test that makes the system under test *fail* (perform incorrectly)
  - A good tester is good at making programs fail (*often*: we try to find defects without too many tests, which cost time)
  - starts with testing small pieces of software and proceeds to testing the whole system

9



Different goals require different testing (2)

- Goal 2: Demonstrate (to customer, developer, certification agency..) that the *system is satisfactory* (*validation testing*)
  - usually tests the whole system (sometimes reusable subsystems)
  - *successful* testing means:
    - (2a) the system performs ***as required in all tests***
      - tests may be given, e.g. "penetration testing" for security assessment or using tests known to be *good at producing a failure if the software is faulty* (e.g. testing for common problem).
    - (2b) we wish to assess ***actual*** **reliability** or ***performance*** (statistical properties over large amounts of testing). Testing in this case needs instead to be *realistic*, i.e. testing should ***mimic*** the anticipated real operation.

10

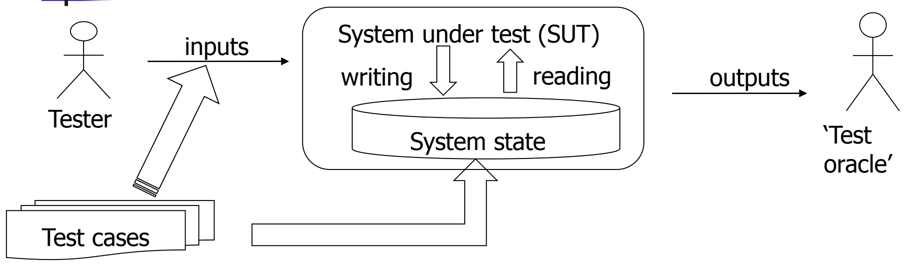
Stages of testing

- *Development* testing: *defect* testing (looking for bugs) during development, by programmers or specialised testers
- *Release* testing: to gain confidence before delivery to customer, usually by a separate testing team
  - some customers require "*independent*" testers (typically the case with "penetration testing")
- *User* testing: users (real or potential) test the system in their real tasks, in their own execution and work environment

11

Use case based testing

Use case based testing architecture



Test cases are specific scenarios that exercise ***ONE*** of the flows defined for a use case. The ***test oracle*** tells whether the system processed the test ***correctly or not***. An oracle may be a human (e.g. the tester) or a different system.

("oracle" is an ancient word for "prophet" or someone who could see hidden truths)

Each test case must define 3 things:
- Setup
- Data used in the test case by the tester
- Expected outcome

13

Use case based testing: the big picture

- ***Setup*** : the ***state of*** the system required ***before*** the test. To be defined in ***sufficient detail*** so as to produce that flow (main flow or one of the alternative flows) of the use case that you want to test
  - For instance, if we test the main flow of the UC "Login" using username 'in345' and password 'pass', then
    - in the Setup we must state that a user "in345/pass" has been recorded in the system DB
    - Only in this case we can test the main flow of Login using in345/pass.
  - to specify this, first read the preconditions of the use case
- ***Input data*** : the values that the tester must use in the test case. Recording input data is important so that one can repeat the tests, if necessary.
  - In the example above, the data that the tester uses in the test case for the main flow of Login must be recorded for testers, for instance:

    username: in345 , password: pass

  - These two specify for the tester the ***test case*** (manually generated)
- ……

14

Use case based testing: the big picture (2)

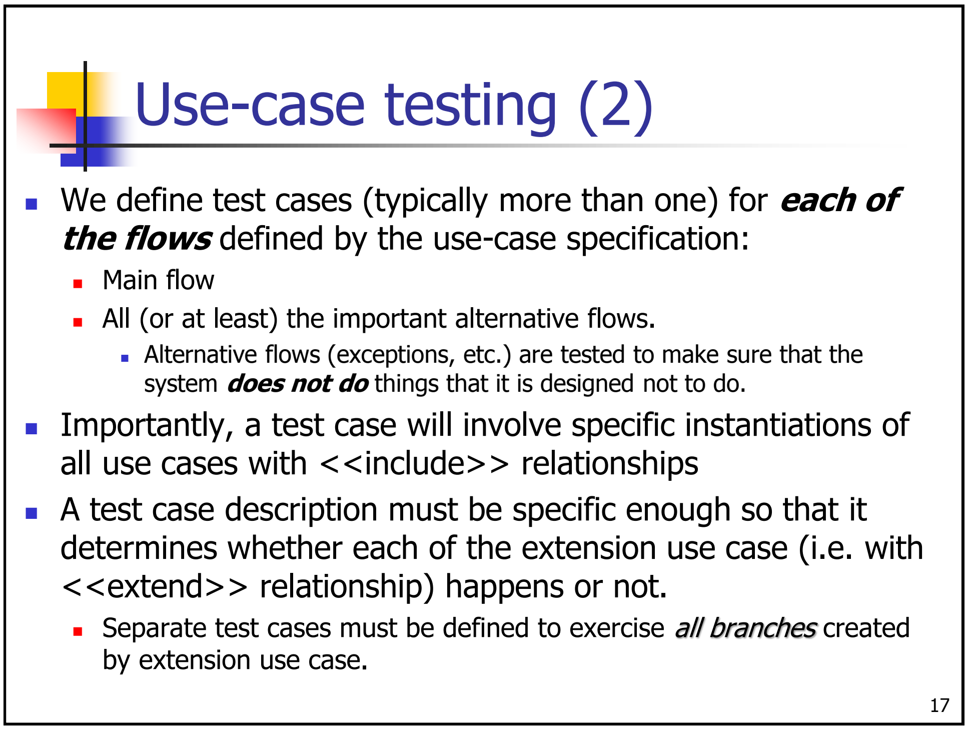
- …….
- The ***expected outcome***: detailed description of the outcome (behaviour of the system) that will be generated if the system processes the test case correctly.
- This (manually checked by the tester) gives us the ***test oracle***
- The outcome is a combination of:
  - ***observable*** results, something that the tester sees during the course of the test case, ***or/and***
  - ***hidden*** results, i.e. the successful processing by the system may lead to changes, which can only be established by undertaking additional ***dedicated checks*** after the test.
    - For instance, correct processing of a successful Login may result not only in showing the tester a specific new screen, but also in updating the list of users who are currently logged in
    - In this case the dedicated activity to check the test outcome would be to check that in345 has been added to the list of logged in users (e.g. by inspecting the respective data – a file or a table in a DB).

15

Use-case testing

- Use-case testing is often used in the following cases:
  - Acceptance testing by the customer
  - Demonstration that the functional requirements have been met (e.g. conducted by the testing team of the software vendor)
- Test cases are use case ***scenarios*** with ***concrete data***
  - i.e. how a use case (or several related use cases with <<include>>/<<extend>> relationships) will occur in real operation.

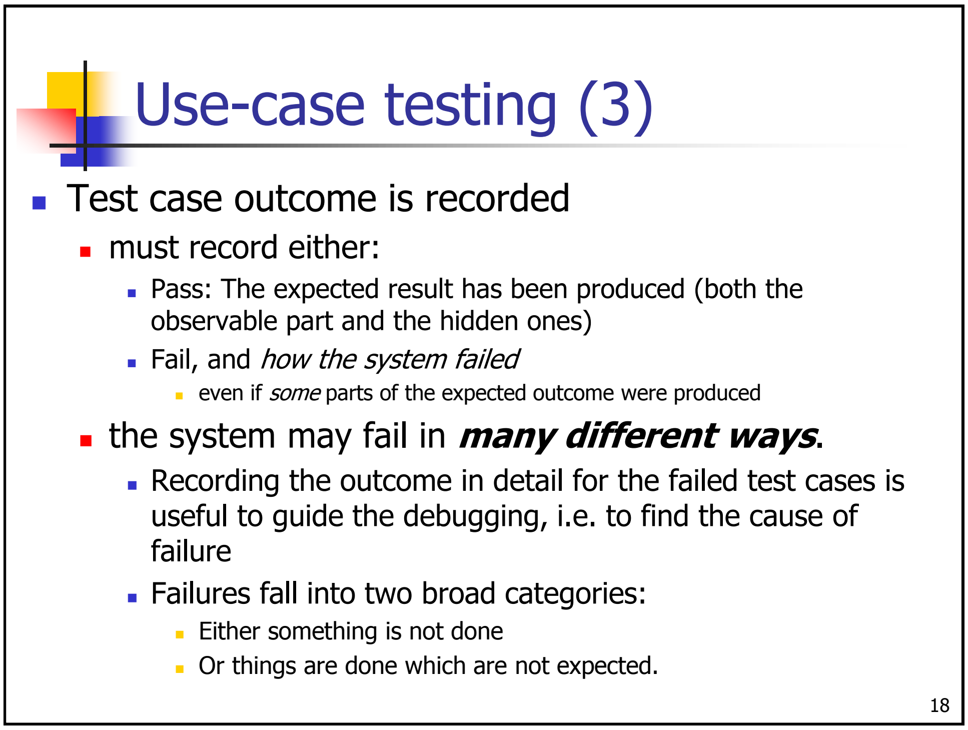
16



Use-case testing (2)

- We define test cases (typically more than one) for ***each of the flows*** defined by the use-case specification:
  - Main flow
  - All (or at least) the important alternative flows.
    - Alternative flows (exceptions, etc.) are tested to make sure that the system ***does not do*** things that it is designed not to do.
- Importantly, a test case will involve specific instantiations of all use cases with <<include>> relationships
- A test case description must be specific enough so that it determines whether each of the extension use case (i.e. with <<extend>> relationship) happens or not.
  - Separate test cases must be defined to exercise *all branches* created by extension use case.

17



Use-case testing (3)

- Test case outcome is recorded
  - must record either:
    - Pass: The expected result has been produced (both the observable part and the hidden ones)
    - Fail, and *how the system failed*
      - even if *some* parts of the expected outcome were produced
  - the system may fail in ***many different ways***.
    - Recording the outcome in detail for the failed test cases is useful to guide the debugging, i.e. to find the cause of failure
    - Failures fall into two broad categories:
      - Either something is not done
      - Or things are done which are not expected.

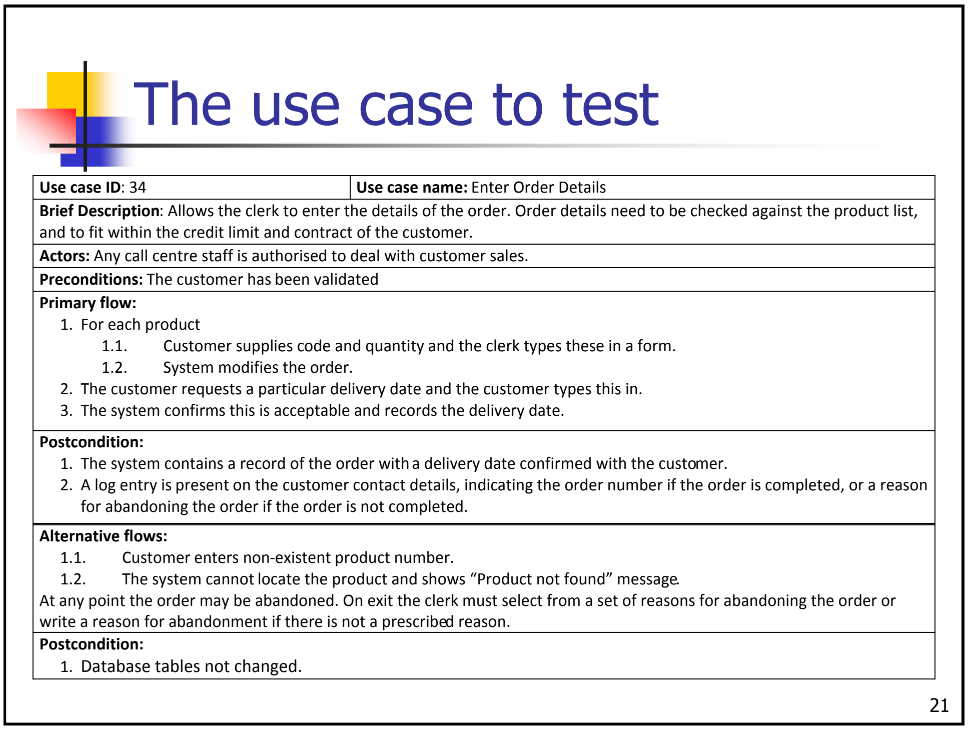
18

Use-case testing (4)

- How many test cases?
  - Test *every defined flow* of a use cases *at least once*.
    - Many test cases can be devised to test the same flow ***with different data***
  - Boundary cases
    - E.g. use large numbers to check for insufficient accuracy
    - Check how the software responds to "illegal" inputs
      - e.g. for a calculator: division by zero
      - read the alternate flows
    - Etc.

19

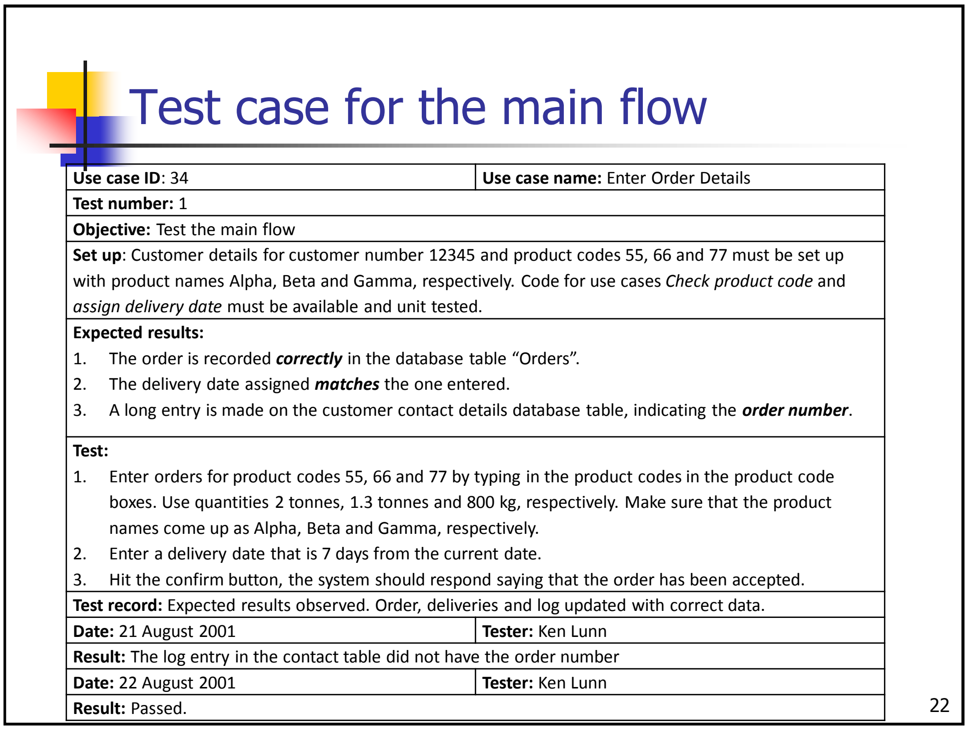
Use case based Testing: An Example



The use case to test

Use case ID: 34	**Use case name:** Enter Order Details
Brief Description: Allows the clerk to enter the details of the order. Order details need to be checked against the product list, and to fit within the credit limit and contract of the customer.	
Actors: Any call centre staff is authorised to deal with customer sales.	
Preconditions: The customer has been validated	
Primary flow: 1. For each product 1.1. Customer supplies code and quantity and the clerk types these in a form. 1.2. System modifies the order. 2. The customer requests a particular delivery date and the customer types this in. 3. The system confirms this is acceptable and records the delivery date.	
Postcondition: 1. The system contains a record of the order with a delivery date confirmed with the customer. 2. A log entry is present on the customer contact details, indicating the order number if the order is completed, or a reason for abandoning the order if the order is not completed.	
Alternative flows: 1.1. Customer enters non-existent product number. 1.2. The system cannot locate the product and shows “Product not found” message. At any point the order may be abandoned. On exit the clerk must select from a set of reasons for abandoning the order or write a reason for abandonment if there is not a prescribed reason.	
Postcondition: 1. Database tables not changed.	

21



Test case for the main flow

Use case ID: 34	**Use case name:** Enter Order Details
Test number: 1	
Objective: Test the main flow	
Set up: Customer details for customer number 12345 and product codes 55, 66 and 77 must be set up with product names Alpha, Beta and Gamma, respectively. Code for use cases *Check product code* and *assign delivery date* must be available and unit tested.	
Expected results: 1. The order is recorded ***correctly*** in the database table “Orders”. 2. The delivery date assigned ***matches*** the one entered. 3. A long entry is made on the customer contact details database table, indicating the ***order number***.	
Test: 1. Enter orders for product codes 55, 66 and 77 by typing in the product codes in the product code boxes. Use quantities 2 tonnes, 1.3 tonnes and 800 kg, respectively. Make sure that the product names come up as Alpha, Beta and Gamma, respectively. 2. Enter a delivery date that is 7 days from the current date. 3. Hit the confirm button, the system should respond saying that the order has been accepted.	
Test record: Expected results observed. Order, deliveries and log updated with correct data.	
Date: 21 August 2001	**Tester:** Ken Lunn
Result: The log entry in the contact table did not have the order number	
Date: 22 August 2001	**Tester:** Ken Lunn
Result: Passed.	

22

Test case for the Alternative flow

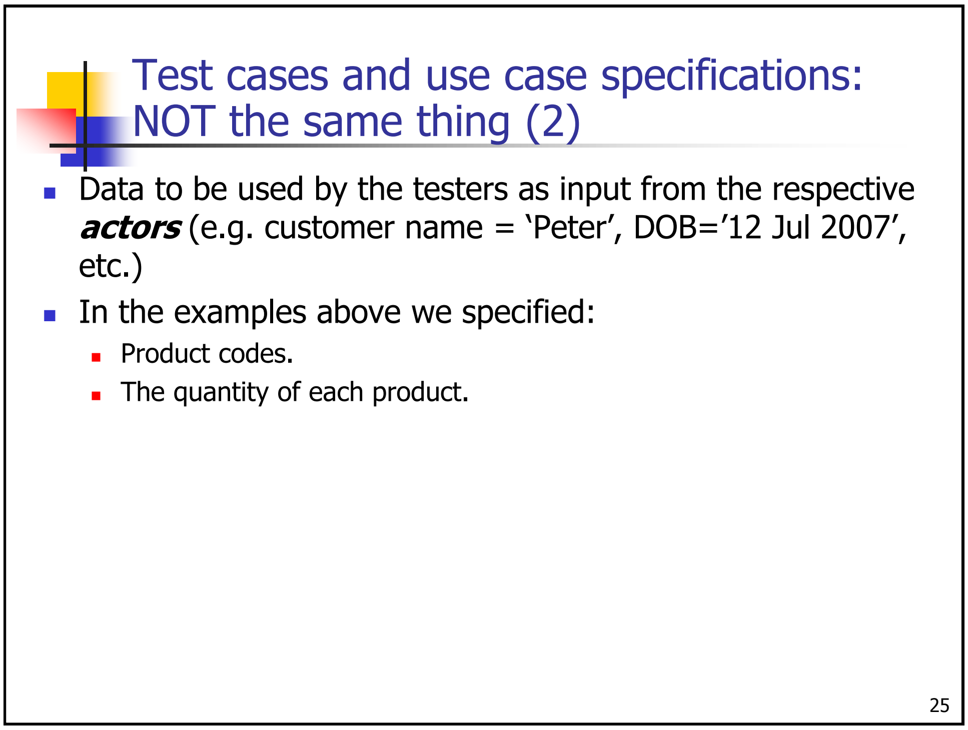
Use case ID: 34	**Use case name:** Enter Order Details
Test number: 2	
Objective: Test alternative flow 1 where the product code is unknown and has to be looked up in the products database.	
Set up: Customer details for customer number 12345 and product codes 55, 66 and 77 must be set up with product names Alpha, Beta and Gamma, respectively. Product code 88 must not exist. Code for use cases *Check product code* and *assign delivery date* must be available and unit tested..	
Expected results: 1. The order is NOT recorded in the orders database tables. 2. No delivery date is created. 3. No log entry is made on the customer contact details database table.	
Test: 1. Enter 88 as the product code. Activate the order modification function. 2. The system shows a warning message 'No such product exists'. 3. Hit button 'Cancel'.	
Test record: Observations consistent with the expected results. No DB tables updated.	
Date: 21 August 2001	**Tester:** Ken Lunn
Result: The product code was not retrieved and a blank was displayed.	
Date: 22 August 2001	**Tester:** Ken Lunn
Result: Passed.	

23

Test cases and use case specifications: NOT the same thing

- Test cases MUST provide the data used in the test!
  - This will help others *repeat the test* exactly
    - for regression testing, to verify the recorded test results.
- Setup data (i.e. the system state *before the test*) are essential to trigger the right flow and results to check.
  - The system state does NOT come from the actor only. Part of the state is something that the system remembers from previous use cases
    - software can be seen as large state-machine!!!
  - In the test case specify the *data values of the state* relevant to the particular test case
    - They MUST be such as to trigger the flow being tested.
    - A test case tests a *single flow*!!!
  - In the examples we specified the state in terms of DB contents (which can be set manually by the tester):
    - A valid customer (with ID 12345 is defined)
    - 3 products with names and ID are also specified.
    - for the 2nd test case we said explicitly: product with ID = 88 does not exist
    - What would have happened if product with ID 88 existed in the DB?

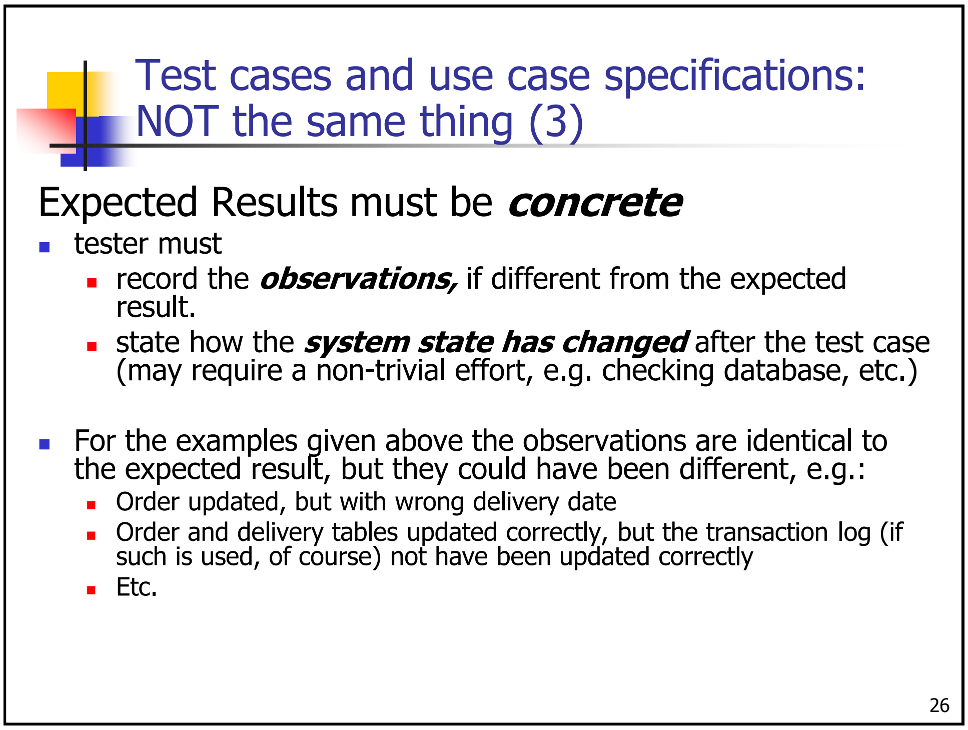
24



Test cases and use case specifications: NOT the same thing (2)

- Data to be used by the testers as input from the respective ***actors*** (e.g. customer name = 'Peter', DOB='12 Jul 2007', etc.)
- In the examples above we specified:
  - Product codes.
  - The quantity of each product.

25

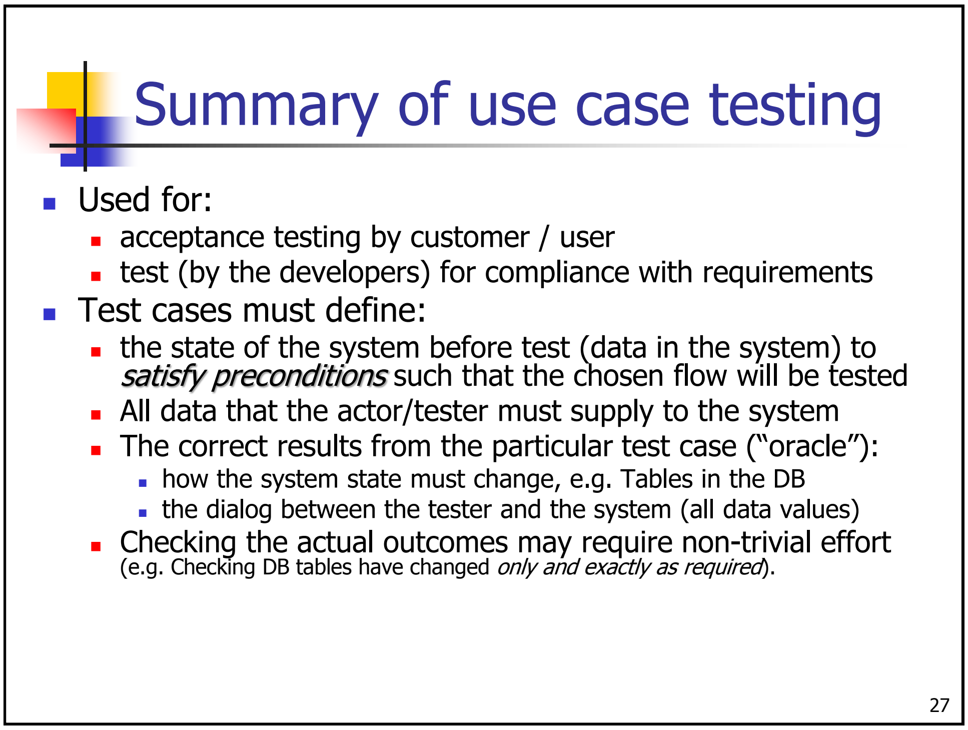


Test cases and use case specifications: NOT the same thing (3)

Expected Results must be ***concrete***

- tester must
  - record the ***observations,*** if different from the expected result.
  - state how the ***system state has changed*** after the test case (may require a non-trivial effort, e.g. checking database, etc.)
- For the examples given above the observations are identical to the expected result, but they could have been different, e.g.:
  - Order updated, but with wrong delivery date
  - Order and delivery tables updated correctly, but the transaction log (if such is used, of course) not have been updated correctly
  - Etc.

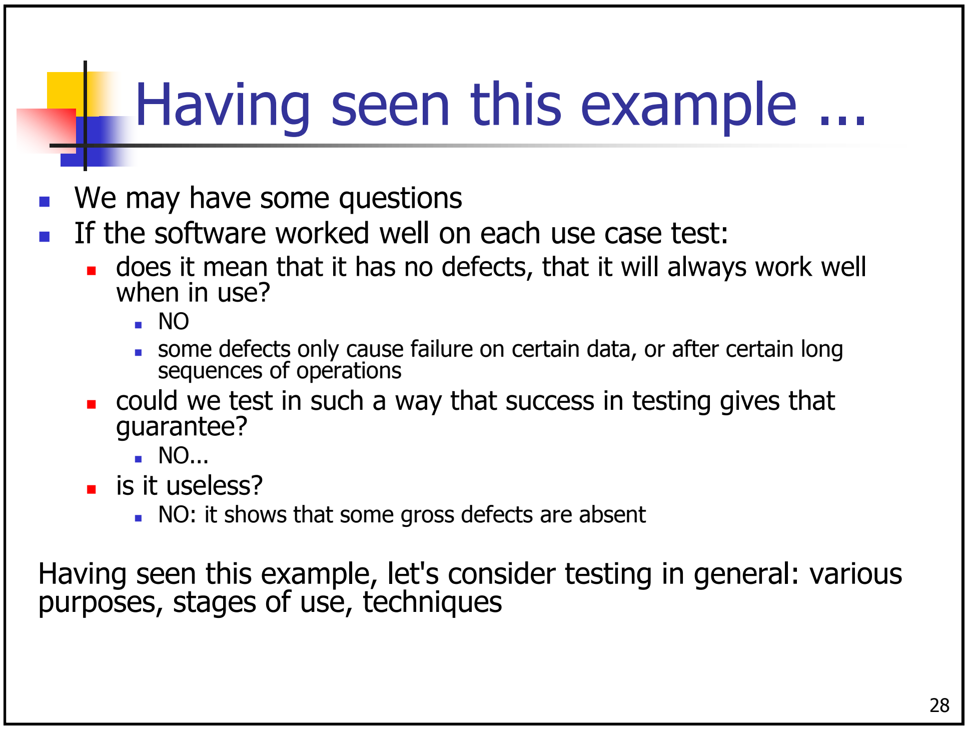
26



Summary of use case testing

- Used for:
  - acceptance testing by customer / user
  - test (by the developers) for compliance with requirements
- Test cases must define:
  - the state of the system before test (data in the system) to *satisfy preconditions* such that the chosen flow will be tested
  - All data that the actor/tester must supply to the system
  - The correct results from the particular test case ("oracle"):
    - how the system state must change, e.g. Tables in the DB
    - the dialog between the tester and the system (all data values)
  - Checking the actual outcomes may require non-trivial effort (e.g. Checking DB tables have changed *only and exactly as required*).

27



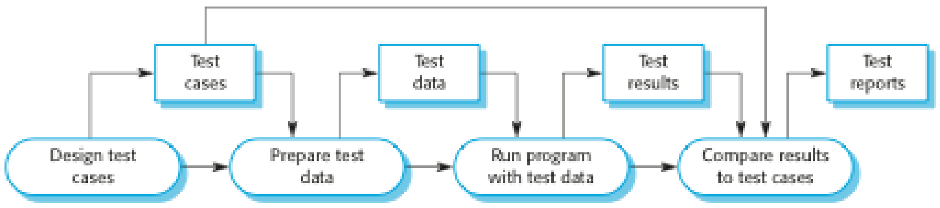
Having seen this example ...

- We may have some questions
- If the software worked well on each use case test:
  - does it mean that it has no defects, that it will always work well when in use?
    - NO
    - some defects only cause failure on certain data, or after certain long sequences of operations
  - could we test in such a way that success in testing gives that guarantee?
    - NO...
  - is it useless?
    - NO: it shows that some gross defects are absent

Having seen this example, let's consider testing in general: various purposes, stages of use, techniques

28

Software testing process

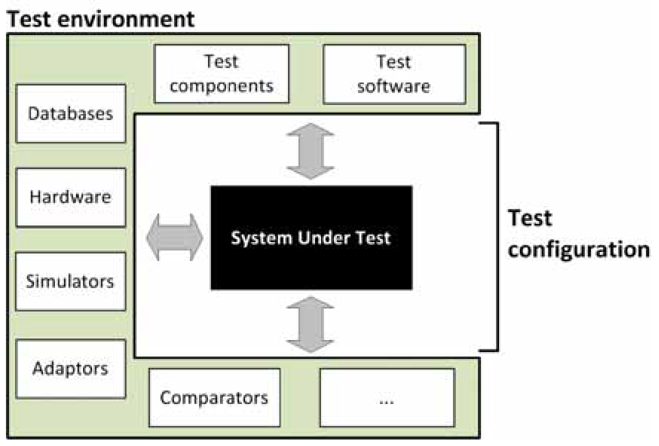


Ian Sommerville, Software Engineering, 9th Edition, chapter 8, p.210

29

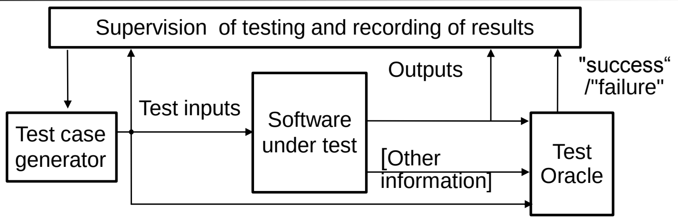
OMG "Unified Testing Profile"

- https://www.omg.org/spec/UTP/1.2/PDF



30

Test set-up: what is needed



Notes:

- *"oracle" in testing* means: *"the means by which we recognise failures from correct results"*: a crucial component.
- test generator, oracle, supervisory system may be human-only or automated.

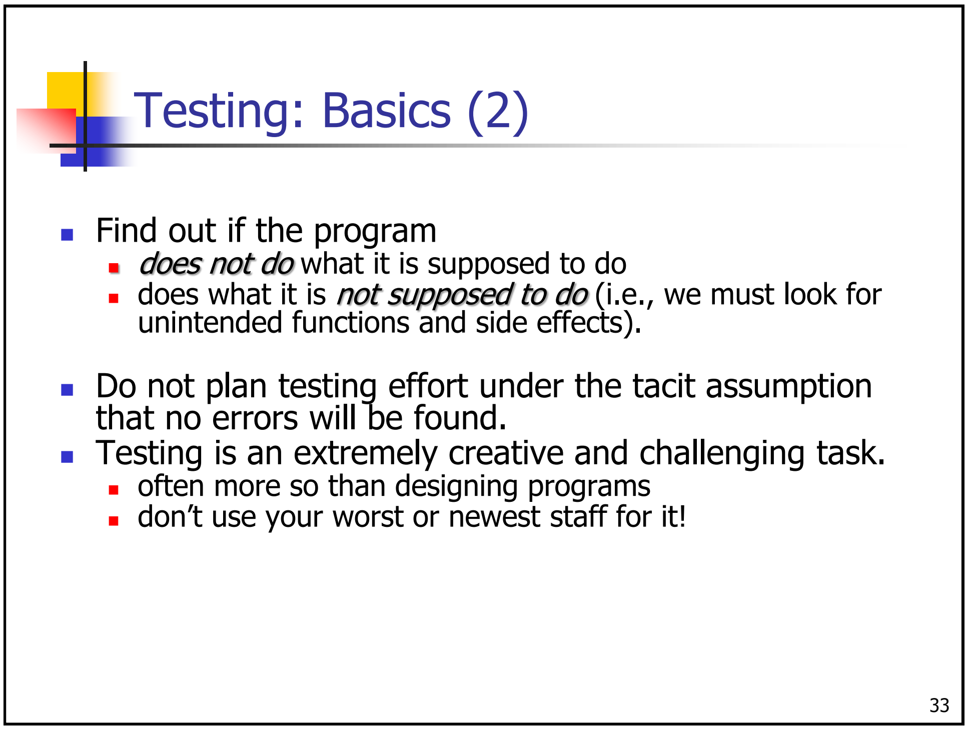
Automation is desirable and often feasible: it will massively reduce costs especially in regression testing, statistical testing

31

Testing: the basics, in summary

- A test case is specified by describing two parts:
  - input data (demands)
    - *including* the internal state of the software under test, because e.g.
      - the values of the attributes decide what an object's methods must do
      - the contents of database decide what a query must return
      - etc.
  - correct output (and state changes)
    - or more generally: how to check that the result is correct (a *test oracle*)
- The results of each test must be thoroughly checked
  - human testers may miss many failures: automated "oracles" help, but they are difficult to construct
- Test cases must be written for:
  - valid and expected input conditions
  - but also invalid and unexpected ones ('robustness testing').

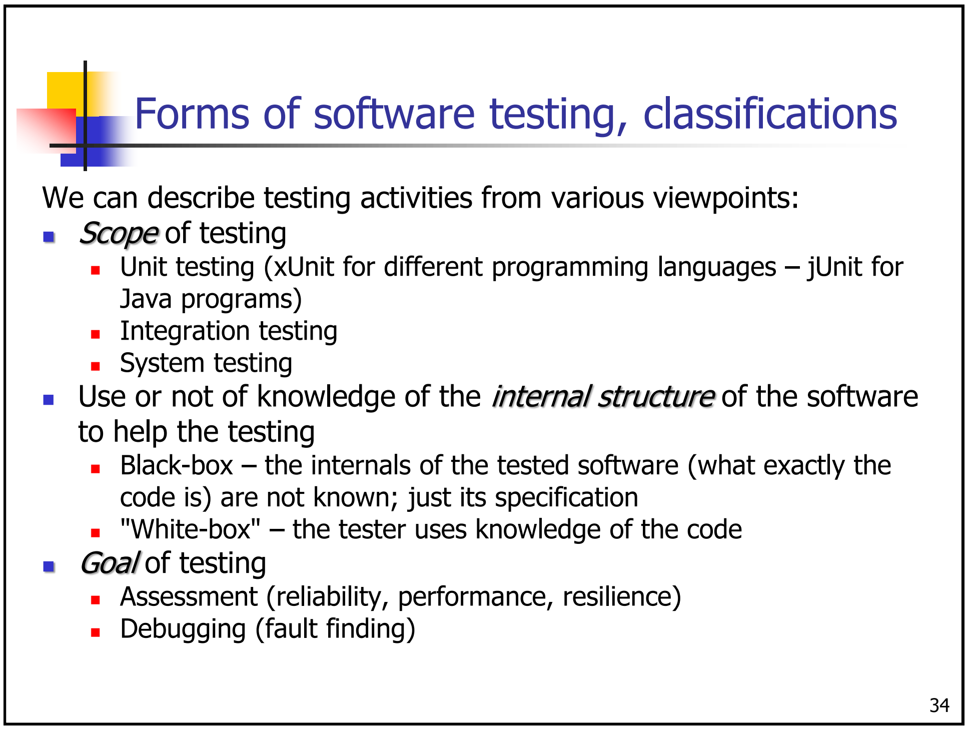
32



Testing: Basics (2)

- Find out if the program
  - *does not do* what it is supposed to do
  - does what it is *not supposed to do* (i.e., we must look for unintended functions and side effects).
- Do not plan testing effort under the tacit assumption that no errors will be found.
- Testing is an extremely creative and challenging task.
  - often more so than designing programs
  - don't use your worst or newest staff for it!

33



Forms of software testing, classifications

We can describe testing activities from various viewpoints:

- *Scope* of testing
  - Unit testing (xUnit for different programming languages – jUnit for Java programs)
  - 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)

34

Forms of software testing, classifications (2)

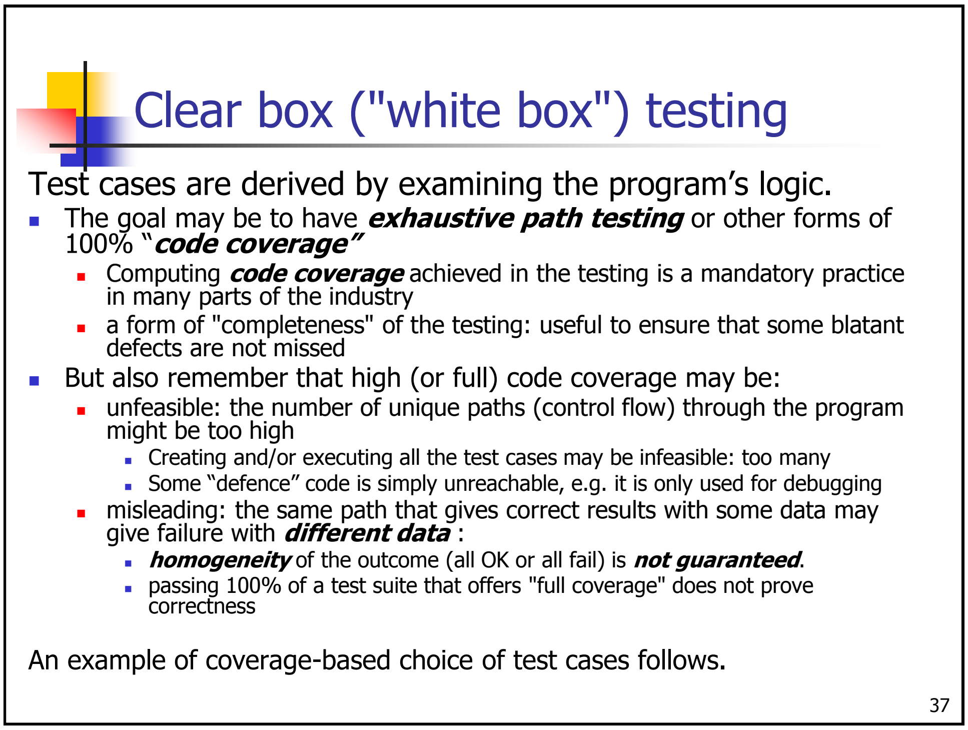
- By the phase of software lifecycle, the parties involved, the rules applied to decide that we have "tested enough":
  - regression testing (testing if bug fixes of the latest release have not introduced new faults, that did not exist in the previous releases),
  - acceptance testing (by client/stakeholder),
  - user testing,
  - beta testing (by potential users of software),
  - independent testing (by independent contractors),
  - coverage testing (will be covered in this lecture),
  - mutation testing – creating "mutants" by injecting faults (i.e. bugs) and using these to find out whether a "testing suite" (i.e. a set of test cases) will "kill" all mutants. If so, the testing suite is considered good.
- In security new types of testing emerged:
  - Penetration testing – testing if common vulnerabilities have been patched (i.e. cannot be exploited). Typically undertaken by an independent team.
  - Fuzzing – systematically looking for new exploitable vulnerabilities, e.g. "buffer overflow", etc.
    - Fuzzing was used for software debugging, too, before it was adapted to security.
  - Etc.

35

Black box testing

- Test cases derived solely from specification. Use – case based testing is a form of black-box testing.
  - Ideally should test for every possible state/input combination
  - This is infeasible. Instead we *sample* from the input/demand space either randomly or according to criteria, like:
    - test with some valid and some invalid inputs
    - use inputs that would reveal typical errors (e.g. extreme values)
    - try to be "*realistic*" (to be detailed later)

36



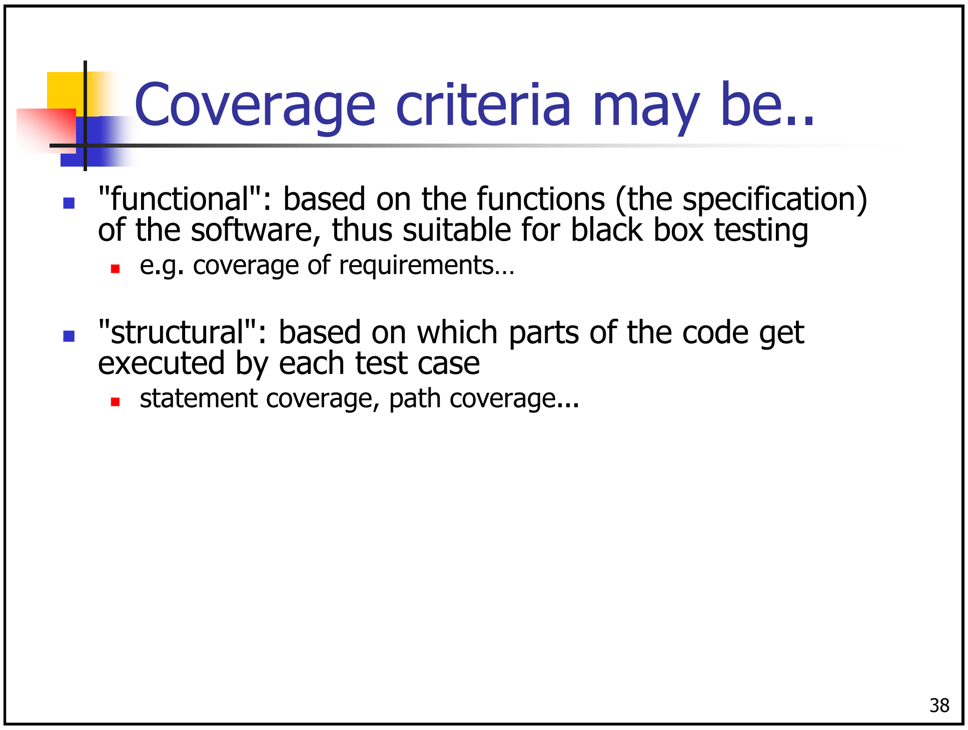
Clear box ("white box") testing

Test cases are derived by examining the program's logic.

- The goal may be to have ***exhaustive path testing*** or other forms of 100% "***code coverage***"
  - Computing ***code coverage*** achieved in the testing is a mandatory practice in many parts of the industry
  - a form of "completeness" of the testing: useful to ensure that some blatant defects are not missed
- But also remember that high (or full) code coverage may be:
  - unfeasible: the number of unique paths (control flow) through the program might be too high
    - Creating and/or executing all the test cases may be infeasible: too many
    - Some "defence" code is simply unreachable, e.g. it is only used for debugging
  - misleading: the same path that gives correct results with some data may give failure with ***different data*** :
    - ***homogeneity*** of the outcome (all OK or all fail) is ***not guaranteed***.
    - passing 100% of a test suite that offers "full coverage" does not prove correctness

An example of coverage-based choice of test cases follows.

37



Coverage criteria may be..

- "functional": based on the functions (the specification) of the software, thus suitable for black box testing
  - e.g. coverage of requirements…
- "structural": based on which parts of the code get executed by each test case
  - statement coverage, path coverage...

38

Functional testing and coverage criteria

- Coverage measures based on the functions of the software, thus suitable for black box testing, e.g.:
  - exercise all listed requirements
  - all use case flows (as in use – case based testing)
  - all possible sequences through an activity diagram that specifies a use case
  - all possible (non-repeating) sequences through the state machine that specifies a classifier (e.g. a class, a sub-system or event the entire system)
  - ...

39

Test coverage criteria

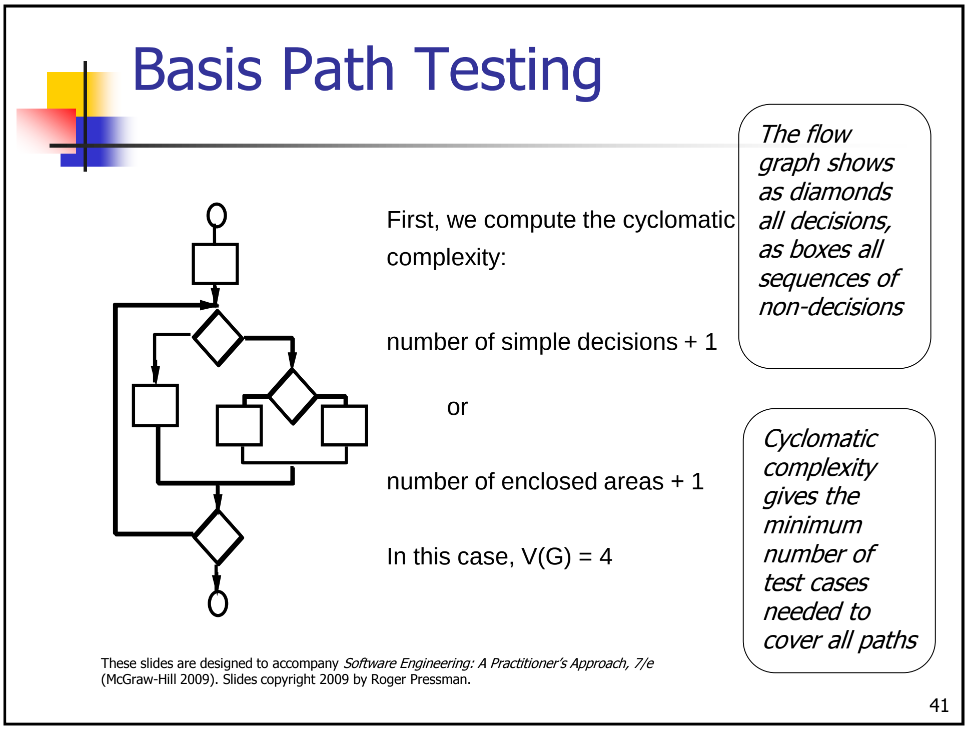
- Based on which parts of the code get executed by each test case: only feasible if we know the source code
  - e.g. infeasible for clients of proprietary software if the vendor does not grant access to source
  - we look at criteria, based on coverage of *control flow*
    - statement coverage, path coverage...
    - based on representing each program *as a graph*
      - all branch statements become decision "diamonds"
      - all sequences of non-branch statements become rectangles

For more details and examples, see e.g.

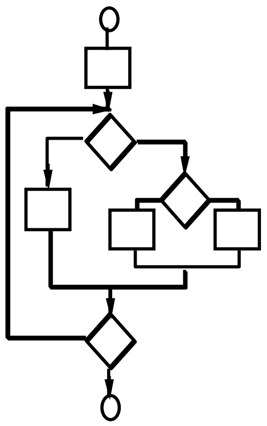
Roger Pressman, *Software Engineering: A Practitioner's Approach* (McGraw-Hill 2009)

Yogesh Singh, *Software testing*, Cambridge University Press, 2011 (e-book in City library)

40



Basis Path Testing



First, we compute the cyclomatic complexity:

number of simple decisions + 1

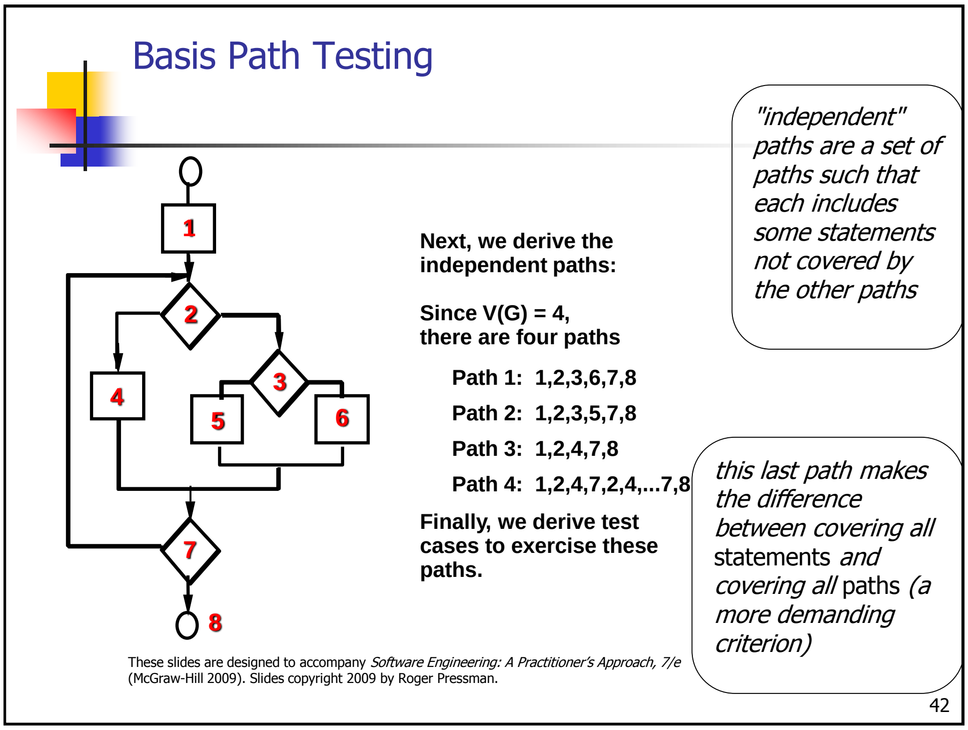
or

number of enclosed areas + 1

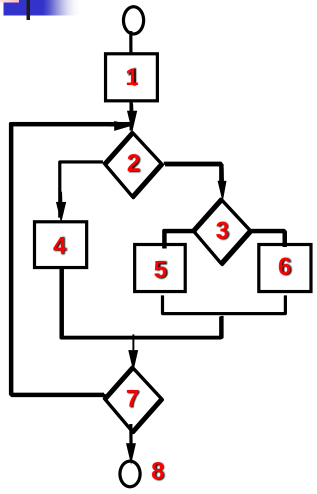
In this case, V(G) = 4

The flow graph shows as diamonds all decisions, as boxes all sequences of non-decisions

Cyclomatic complexity gives the minimum number of test cases needed to cover all paths



Basis Path Testing



Next, we derive the independent paths:

Since V(G) = 4, there are four paths

Path 1: 1,2,3,6,7,8

Path 2: 1,2,3,5,7,8

Path 3: 1,2,4,7,8

Path 4: 1,2,4,7,2,4,...7,8

Finally, we derive test cases to exercise these paths.

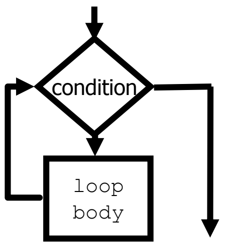
"independent" paths are a set of paths such that each includes some statements not covered by the other paths

this last path makes the difference between covering all statements *and covering all* paths *(a more demanding criterion)*

Test cases for a loop?

- e.g.
  `while (condition) {loop body}`
- say the loop has to be executed *n* times.
- Test cases for which:
  - <u>the while is false at the start</u> (no execution of the loop body)
  - <u>there must be *one* run through the loop</u>
  - two executions
  - more than 2, less than *n*
  - <u>exactly *n*</u>

<u>*underlining*</u> denotes "boundary" cases (between ranges that require different behaviours of the SUT): we choose them because *programming mistakes are often made about them*



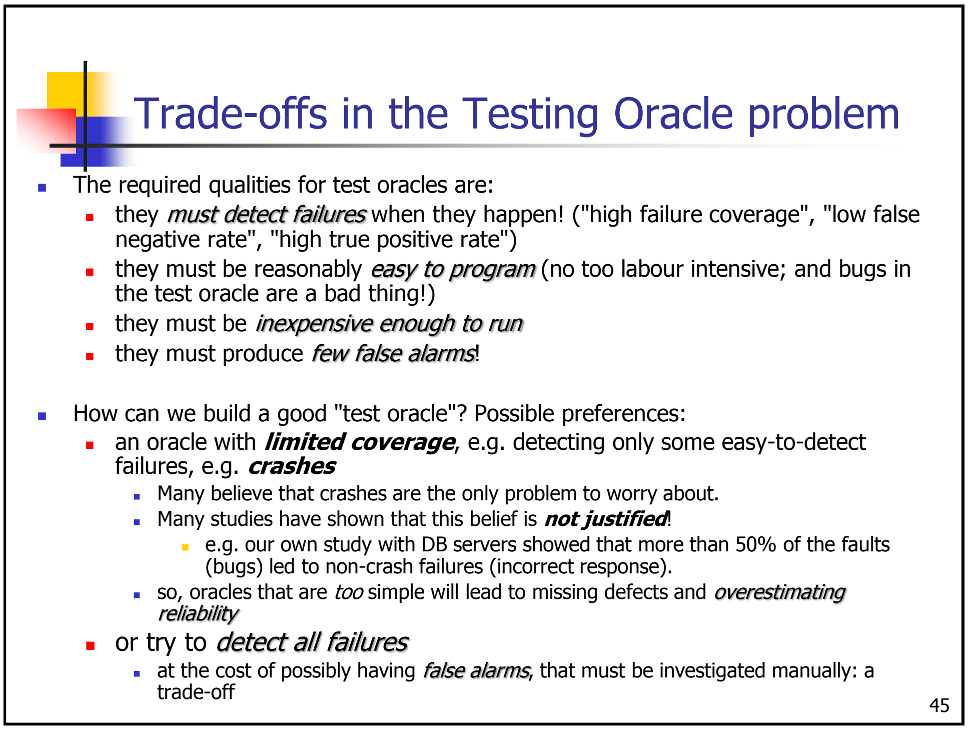
43

The Testing Oracle problem

- In many cases, whether the system is working correctly or not is ***not obvious***
  - sometimes the correct result cannot be ***pre-calculated***
  - realistic validation testing often involves very long sequences of tests
  - testing in real operation / with real users will create unpredictable sequences of tests
  - we then need to specify not "the correct outcome" of ***a particular test***, but a ***procedure*** for checking that the outcome is correct
- we can then automate this procedure and make affordable very ***extensive tests*** that would not be feasible with human "oracles"

- NB a well-designed test oracle can sometimes also be used as a *run-time check* to give *fault tolerance* to the software in normal use

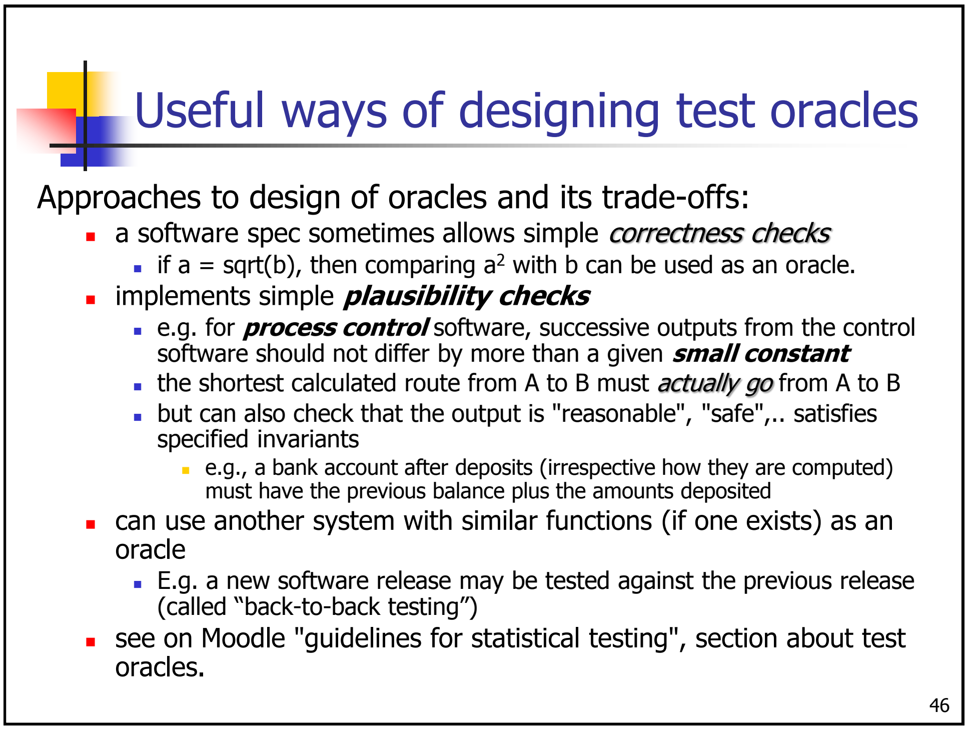
44



Trade-offs in the Testing Oracle problem

- The required qualities for test oracles are:
  - they *must detect failures* when they happen! ("high failure coverage", "low false negative rate", "high true positive rate")
  - they must be reasonably *easy to program* (no too labour intensive; and bugs in the test oracle are a bad thing!)
  - they must be *inexpensive enough to run*
  - they must produce *few false alarms*!
- How can we build a good "test oracle"? Possible preferences:
  - an oracle with ***limited coverage***, e.g. detecting only some easy-to-detect failures, e.g. ***crashes***
    - Many believe that crashes are the only problem to worry about.
    - Many studies have shown that this belief is ***not justified***!
      - e.g. our own study with DB servers showed that more than 50% of the faults (bugs) led to non-crash failures (incorrect response).
    - so, oracles that are *too* simple will lead to missing defects and *overestimating reliability*
  - or try to *detect all failures*
    - at the cost of possibly having *false alarms*, that must be investigated manually: a trade-off

45

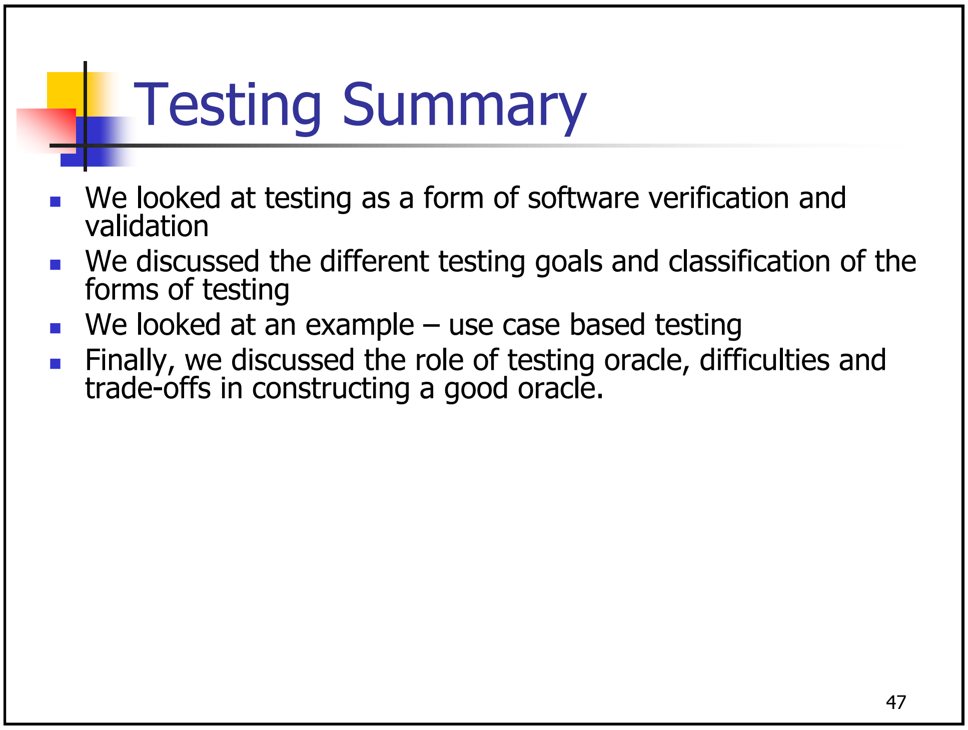


Useful ways of designing test oracles

Approaches to design of oracles and its trade-offs:

- a software spec sometimes allows simple *correctness checks*
  - if a = sqrt(b), then comparing a^2 with b can be used as an oracle.
- implements simple ***plausibility checks***
  - e.g. for ***process control*** software, successive outputs from the control software should not differ by more than a given ***small constant***
  - the shortest calculated route from A to B must *actually go* from A to B
  - but can also check that the output is "reasonable", "safe",.. satisfies specified invariants
    - e.g., a bank account after deposits (irrespective how they are computed) must have the previous balance plus the amounts deposited
- can use another system with similar functions (if one exists) as an oracle
  - E.g. a new software release may be tested against the previous release (called “back-to-back testing”)
- see on Moodle "guidelines for statistical testing", section about test oracles.

46



Testing Summary

- We looked at testing as a form of software verification and validation
- We discussed the different testing goals and classification of the forms of testing
- We looked at an example – use case based testing
- Finally, we discussed the role of testing oracle, difficulties and trade-offs in constructing a good oracle.

47