Software testing

Antonio Brogi

Department of Computer Science
University of Pisa

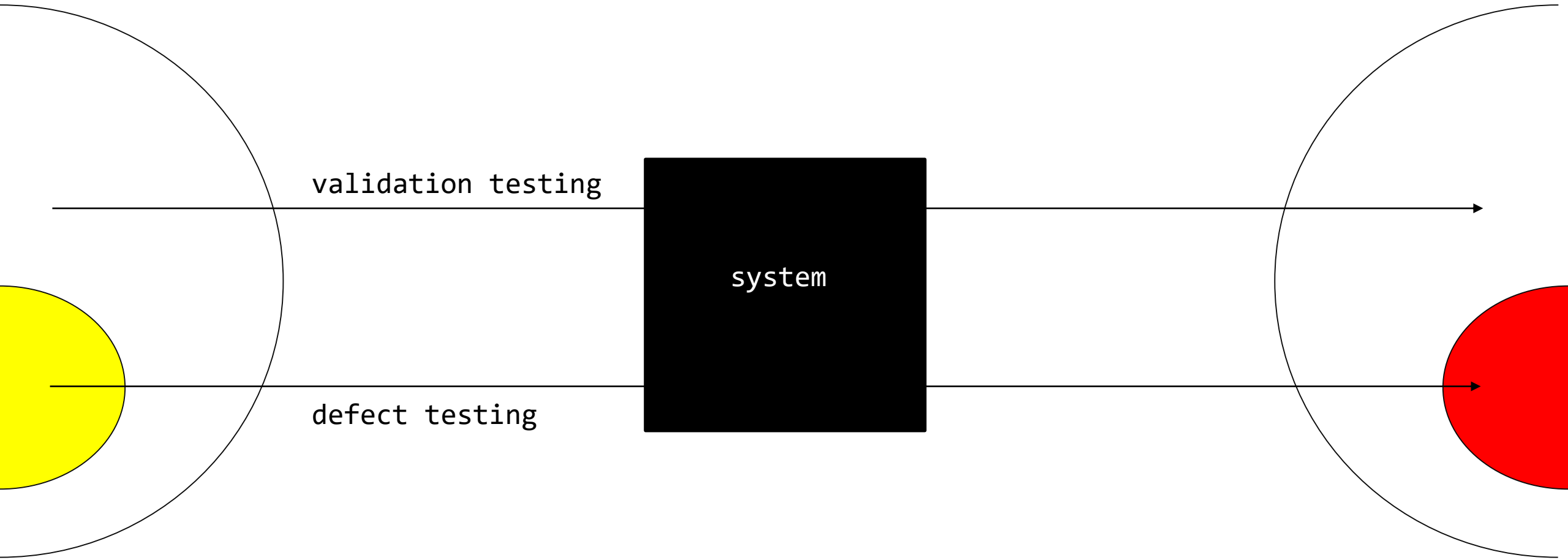
Testing?

Testing:

- execute program using artificial data
- check results of test run for errors/anomalies/non-functional info

Testing intended to:

- show that program does what it is intended to do (**validation testing**)
- discover program defects before it is put into use (**defect testing**)


validation testing
system
defect testing

[0,2:08]

At 36.7 seconds after H_0 (approx. 30 seconds after lift-off) the computer within the back-up inertial reference system, which was working on stand-by for guidance and attitude control, became inoperative. This was caused by an internal variable related to the horizontal velocity of the launcher exceeding a limit which existed in the software of this computer.

March 26, 2014

Nissan Recalls 990,000 Vehicles for Air Bag Malfunction

software problem in the occupant classification system might not detect an occupant in that seat

January 2016 – Google's Nest 'smart' thermostats left users, literally, in the cold because of a software update that went wrong

TSB BANK

April 2018 - Millions of TSB customers locked out of their accounts after IT upgrade led to an online banking outage

«Testing can only show the presence of errors, not their absence.»

«Testing can only show the presence of errors,
not their absence.»

[in exams, too ☺]

Verification & Validation (V&V)

Testing part of the broader software verification & validation process

Software verification: checking that software meets its stated functional and non-functional requirements

Software validation: checking that software meets customer's expectations

Verification & Validation (V&V)

Goal: Establish confidence that software is «fit for purpose»

- **Software purpose**

  Level of confidence depends on how critical the software to an organisation

- **User expectation**

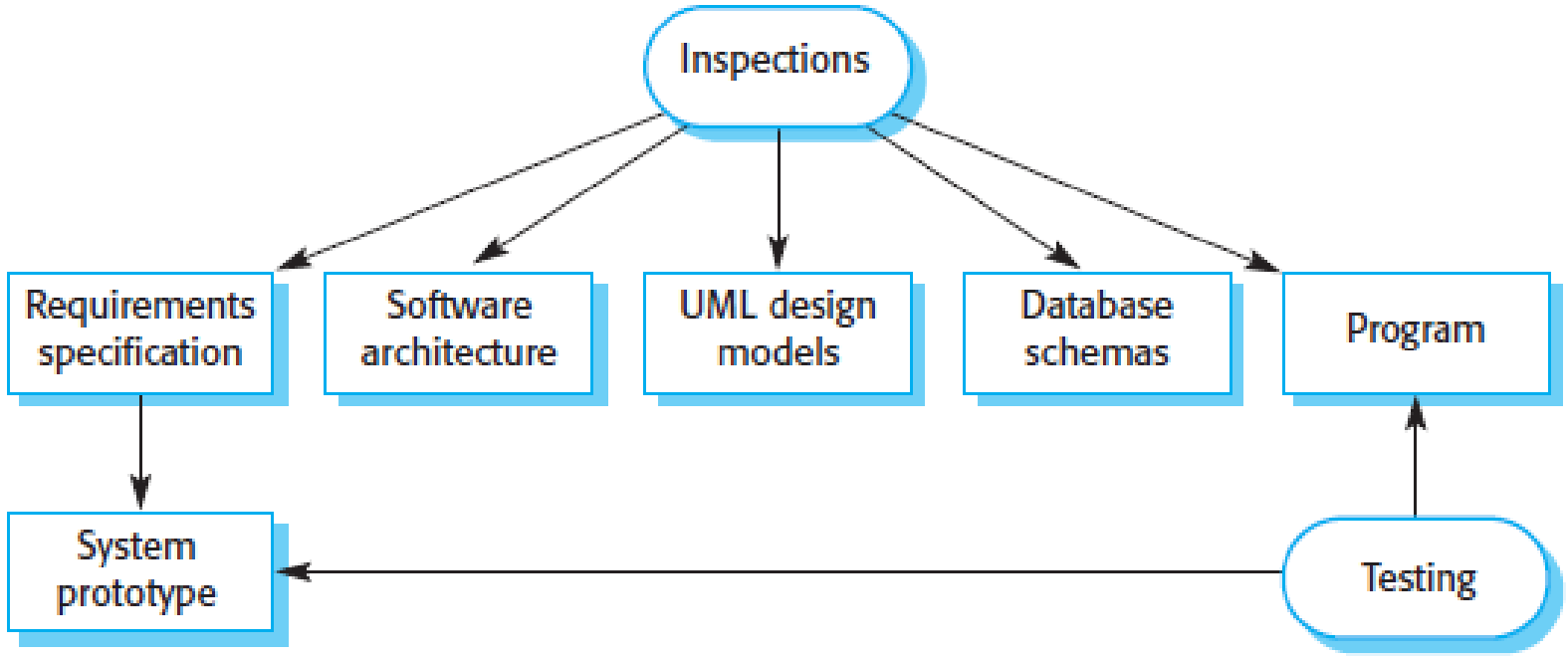
  Users may have low expectations for some (new) software, higher for more established products

- **Marketing environment**

  Getting a product to market early may be more important than finding all possible defects

Verification & Validation (V&V)

V&V also includes **inspections** that perform ***static*** analyses of requirements, design models, program source code

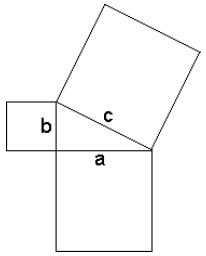


(testing concerned with exercising and observing ***dynamic*** behaviour)

Verification & Validation (V&V)

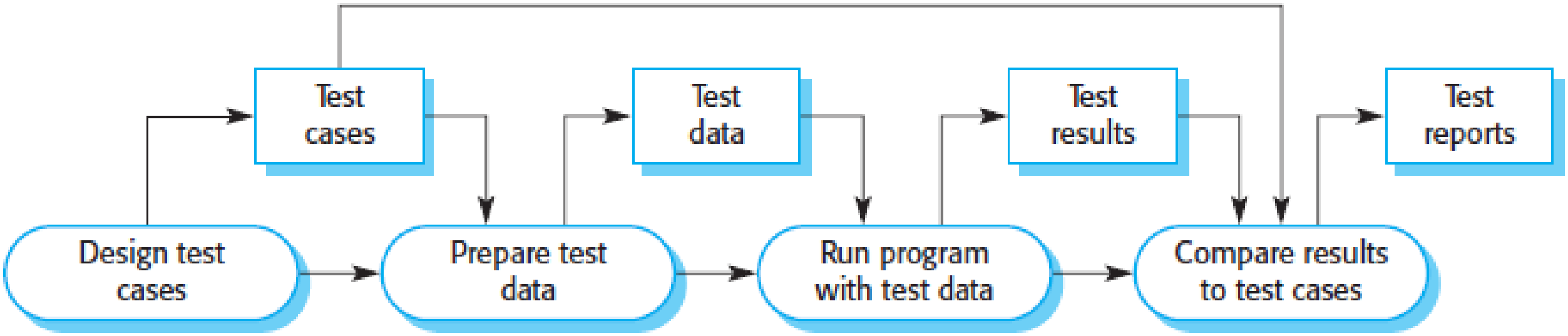
Advantages of inspections over testing:

- During testing, errors can mask other errors
- Incomplete systems can be inspected without additional costs
- Prove that $P \models Inv$



(However, inspections cannot replace software testing altogether)

Traditional testing process



Test cases specify test inputs and expected system outputs

Test data can be generated automatically, test cases cannot

Test execution can be automated

Traditional stages of testing

- **Development testing**

  system tested during development to discover bugs and defects

- **Release testing**

  separate testing team test complete version of the system before release to users

  aim: checking that system meets requirements of system stakeholders

- **User testing**

  (potential) users test the system in their own environment

- Testing?
- Development testing

Development testing

All testing activities carried out by team developing the system

- ***Unit testing***

  individual «program units» functionally tested

- ***Component testing***

  several units integrated to create composite components, testing interfaces

- ***System testing***

  components integrated, system tested as a whole, testing component interactions

Testing?

Development testing

- Unit testing

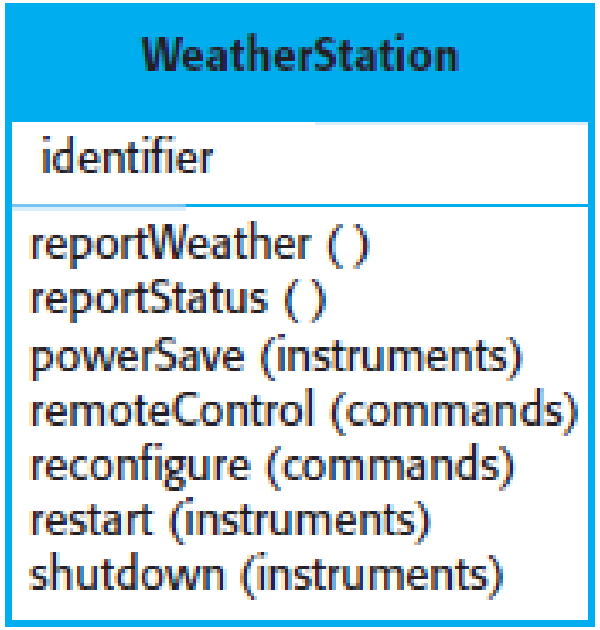
Unit testing

Testing program components, such as methods or object classes

Tests should

- call all methods with different input parameters
- provide coverage of all object features

Unit testing: Example



Test that object attribute is properly set up

Define test cases for all object methods

- Test methods in isolation
- Test inherited operations too
- Use a state model to test (all possible*) state transition sequences

Summary of Oct 17th class

- Summary of course so far
- Double-check of class health state (composition, homework)

- Software testing
  - Testing?
    - validation vs. defect testing
    - software verification and validation
    - inspections vs. testing
    - traditional testing process
  - Development testing
    - Unit testing

Unit testing

Whenever possible, **automate** unit testing

Use test automation frameworks (like Junit), which

- provide (extensible) generic classes
- can run all tests and report test results through a GUI

Parts of automated test

- *Setup* //to initialize the system as needed
- *Call* //calling method/object to be tested
- *Assertion* //comparing call result with expected results

Unit testing effectiveness

Effectiveness – unit tests should

- show that component, when used as expected, does what it is supposed to do
- also reveal defects in the component (e.g., with «abnormal inputs»)

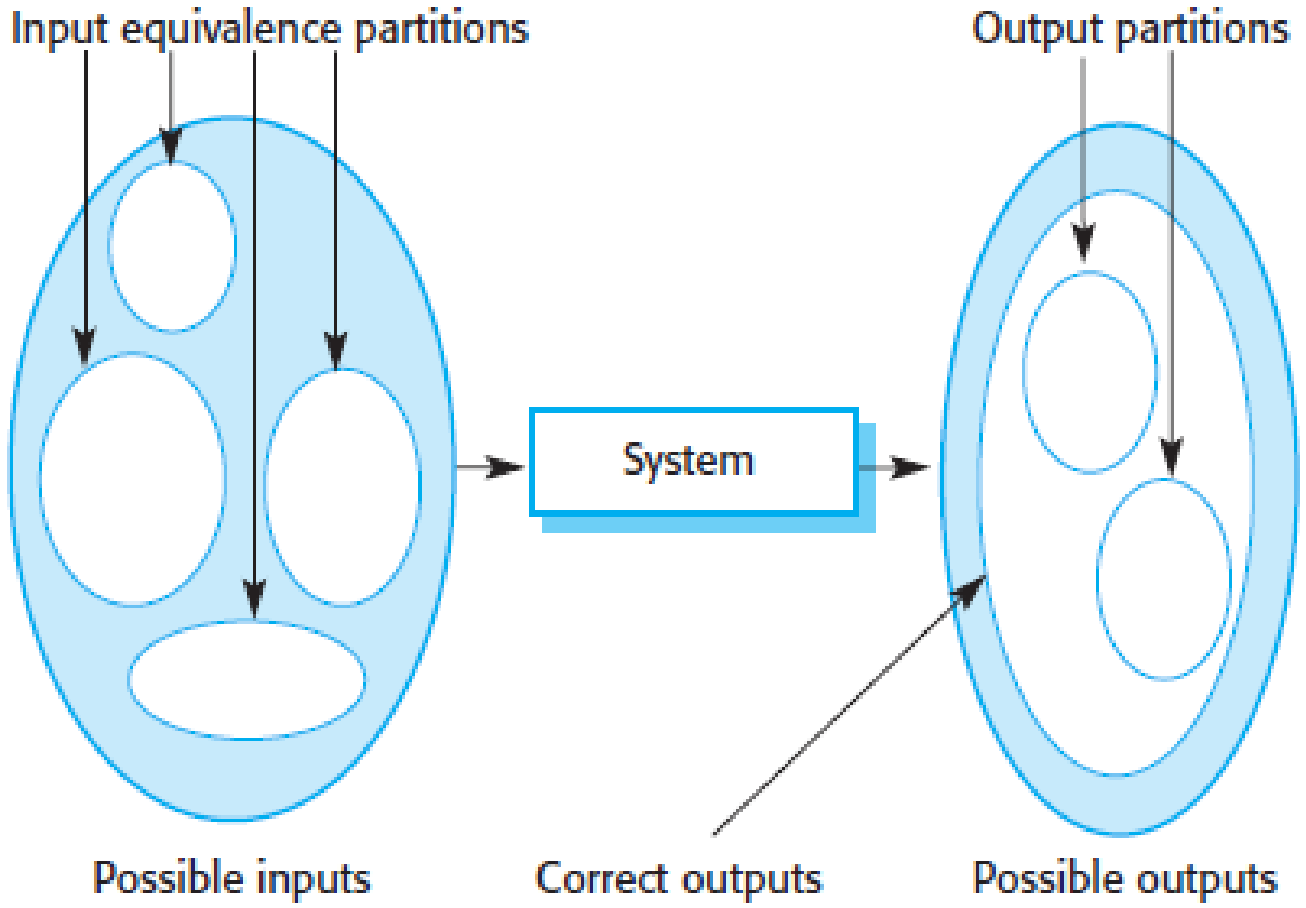
Two strategies

- ***Partition testing***
  - identify groups of inputs that should be processed the same
  - choose tests from within each of these groups
- ***Guideline-based testing***
  - choose test cases by using testing guidelines (that reflect previous experience of the kinds of errors that programmers often make when developing components)

Unit testing effectiveness

Partition testing

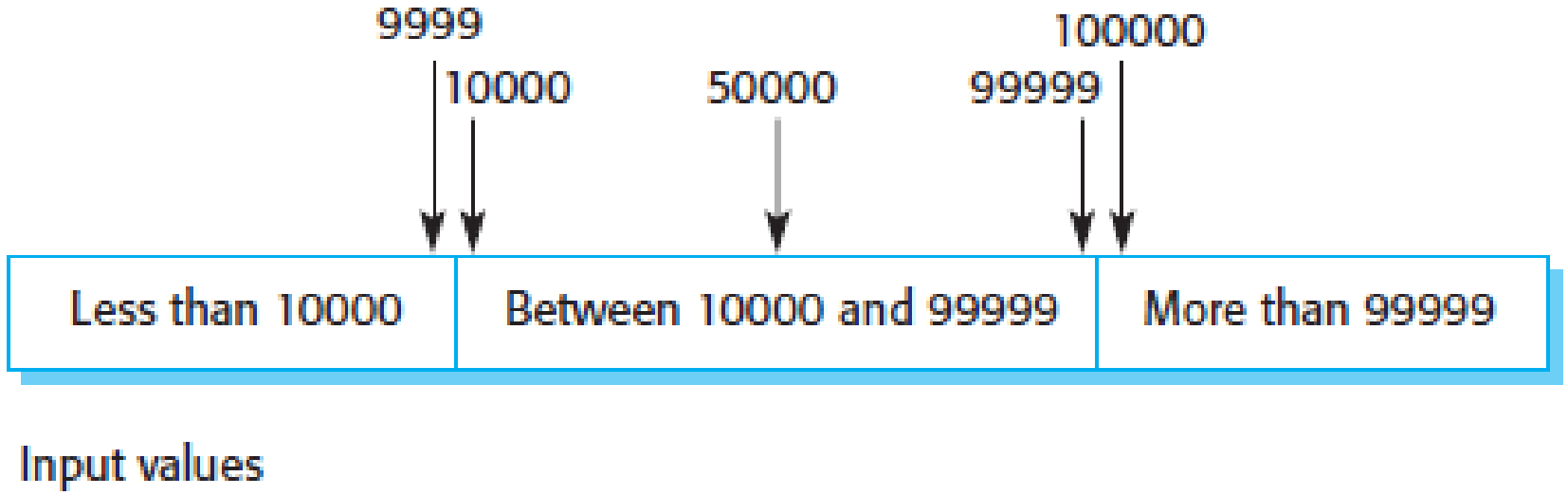
- Identify ***equivalence partitions*** – classes of input (and output) data that have «common characteristics» (e.g., positive/negative numbers, menu selections) and for which program normally behave in a comparable way



Unit testing effectiveness

Partition testing

- Choose test cases from each partition
- Rule of thumb: choose test cases on the boundaries and in the midpoint of each partition



Unit testing effectiveness

Testing guidelines

- For sequences
  - Test with sequences which have only a single value
  - Use sequences of different sizes in different tests
  - Derive tests so that the first, middle and last elements of the sequence are accessed
  - Test with sequences of zero length
- In general
  - Choose inputs that force the system to generate all error messages
  - Design inputs that cause input buffers to overflow
  - Repeat same input (series) many times
  - Force invalid outputs to be generated
  - Force computation results to be too large or too small

Unit testing criticisms (too)

Why Most Unit Testing is Waste

By James O Coplien

Unit tests are unlikely to test more than one trillionth of the functionality of any given method in a reasonable testing cycle

If you find your testers splitting up functions to support the testing process, you're destroying your system architecture and code comprehension along with it. Test at a coarser level of granularity.

Throw away tests that haven't failed in a year.

If X has business value and you can test X with either a system test or a unit test, use a system test – context is everything.

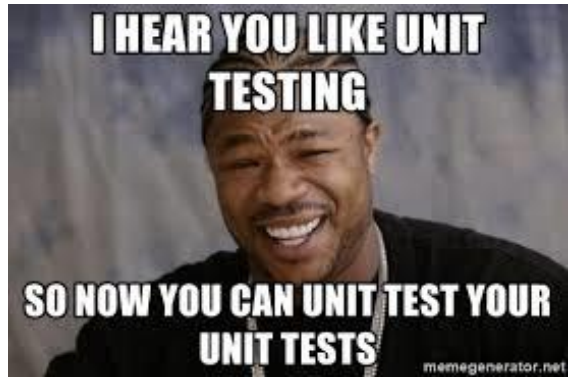
Vitaliy Pisarev
Senior System Architect
HPE

Just say no — Apr 20, 2017

The No. 1 unit testing best practice: Stop doing it

It always happens the same way: You write code and then run the unit tests, only to have them fail. Upon closer inspection, you realize that you added a collaborator to the production code but forgot to configure a mock object for it in the unit tests. The result: NullPointerExceptions everywhere.

We've all been there. At this point you slap your head, curse at your own stupidity, and mock out the new collaborator. Tests are green, and all is good.



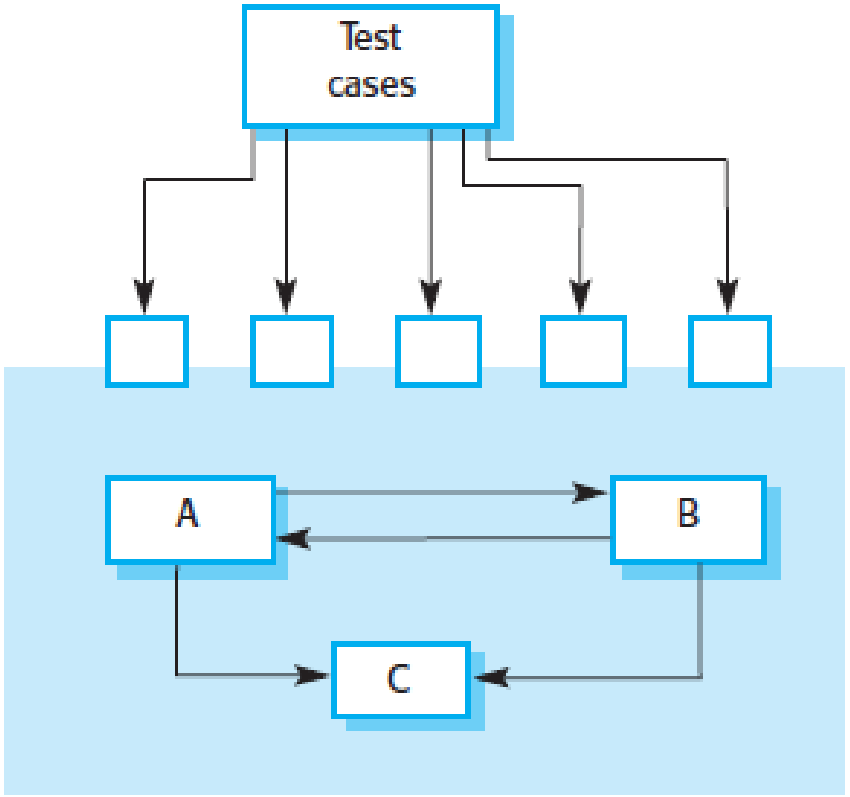
Testing?

Development testing

- Unit testing
- Component testing

Component testing

- Software components are often made up of several interacting objects
- You access the functionality of these objects through the interface of the composite component
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification



Component testing

- Types of interfaces:
  - *Parameter interfaces*

    Data passed from one method or procedure to another (e.g., in object methods)
  - *Procedural interfaces*

    Sub-system encapsulates procedures to be called by other sub-systems (e.g., objects and reusable components)
  - *Message passing interfaces*

    Sub-systems request services from other sub-systems (e.g., service-oriented systems)
  - *Shared memory interfaces*

    Block of memory shared between procedures/functions (e.g., in embedded systems)

Component testing

- Most common interface errors:
  - Interface misuse
    - Calling component makes an error in its use of called interface (e.g., parameters in wrong order/number)
  - Interface misunderstanding
    - Calling component embeds incorrect assumptions on behaviour of called component (e.g., binary search called with unordered array)
  - Timing errors
    - called and calling component operate at different speeds, out-of-date information accessed

Component testing

- Interface testing guidelines
  - Design tests so that parameters to a called procedure are at the extreme ends of their ranges
  - Always test pointer parameters with null pointers
  - Design tests which cause the component to fail
  - Use stress testing in message passing systems (many more messages than likely in practice)
  - In shared memory systems, vary the order in which components are activated

- Sometimes inspections are more effective than testing in looking for interface errors

Testing?

Development testing

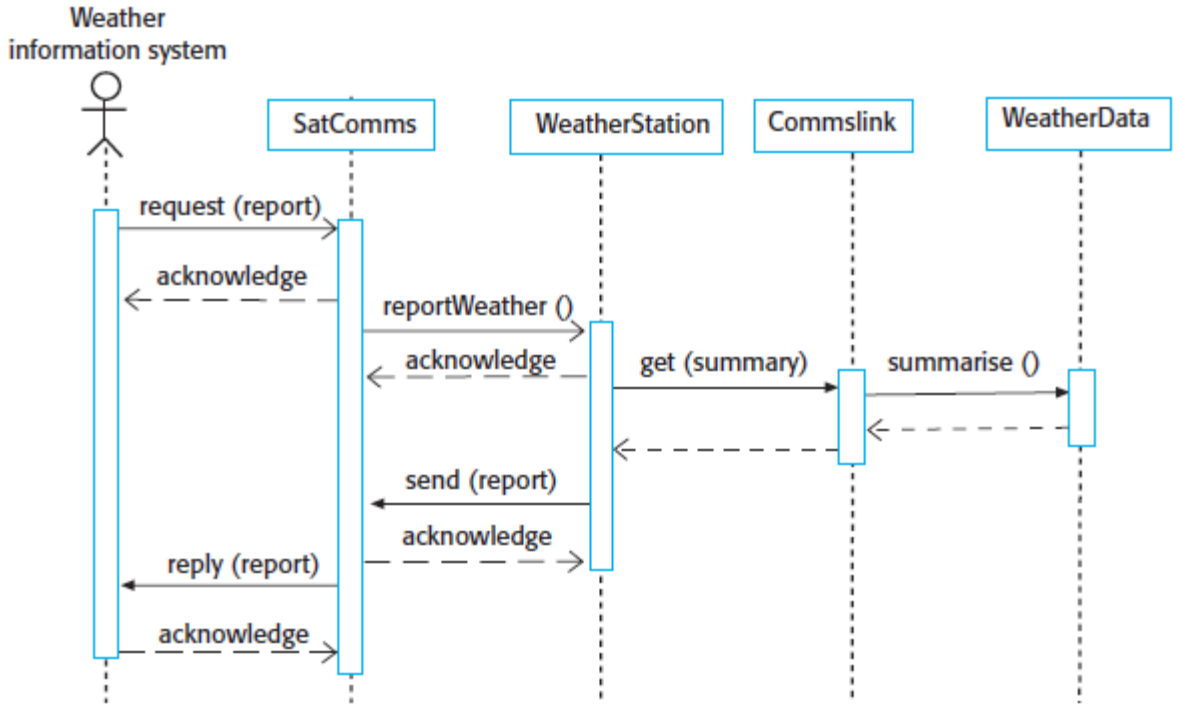
- Unit testing
- Component testing
- System testing

System testing

- During development, components are integrated to create a version of the system
  - Components developed by different developers/teams, off-the-shelf systems integrated
  - Testing the integrated system is a collective process, sometimes delegated to a separate testing team
- Focus: testing the interactions between components
- Checking that components are compatible, interact correctly and transfer the right data at the right time across their interfaces
- Testing the so-called “emergent behavior” of a system

System testing

- Use case-based testing
  - use-cases developed to identify system interactions can be used as a basis for system testing
  - each use case usually involves several system components so testing the use case forces interactions to occur
  - the sequence diagrams associated with the use case document the components and interactions that are being tested



System testing

- Testing policies
  - Exhaustive system testing is impossible
  - Testing policies which define required system test coverage are developed
  - Examples of testing policies
    - All system functions that are accessed through menus should be tested
    - Combinations of functions that are accessed through the same menu must be tested
    - Where user input is provided, all functions must be tested with correct & incorrect input

Testing?

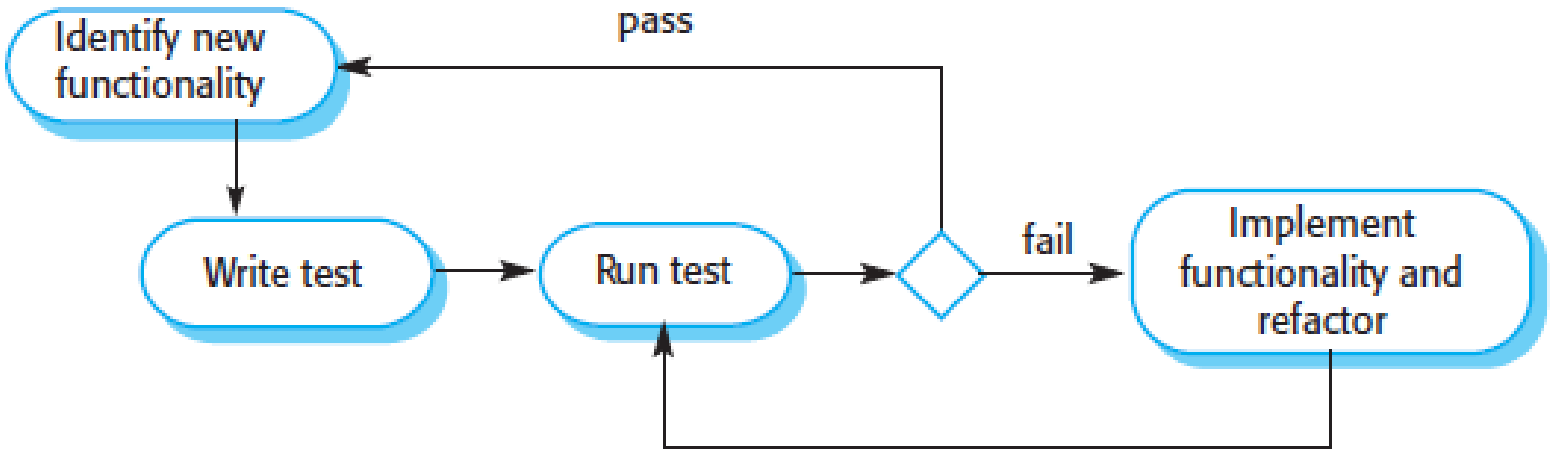
Development testing

- Unit testing
- Component testing
- System testing
- Test-Driven Development

Test-Driven Development (TDD)

- Approach to program development in which you inter-leave testing and code development
- Tests are written before code, ‘passing’ the tests is the critical driver of development
- You develop code incrementally, along with a test for each increment
- You don’t move on to next increment until the code that you have developed passes its test
- TDD was introduced as part of agile methods such as Extreme Programming

Test-Driven Development (TDD)



- Identify the increment of functionality that is required (normally 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
- 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

Test-Driven Development (TDD)

Benefits

- Code coverage
  - Every code segment that you write 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
  - You can always run regression tests to check that changes to the program have not introduced new bugs
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing

* testing the system to check that changes have not ‘broken’ previously working code

Testing?

Development testing

- Unit testing
- Component testing
- System testing
- Test-Driven Development

Release testing

Release testing

- Testing a particular release of a system that is intended for use outside of the development team
- Primary goal: to convince system supplier that it is good enough for use
  - Must show that system delivers its specified functionality, performance and dependability, and that it does not fail during normal use
- Usually a black-box testing process where tests are only derived from the system specification

Release testing & system testing

- Release testing is a form of system testing
  - A separate team (that has not been involved in the system development) should be responsible for release testing
  - System testing by the development team should focus on discovering bugs in the system (defect testing)
  - The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing)

Testing?

Development testing

- Unit testing
- Component testing
- System testing
- Test-Driven Development

Release testing

- Requirements-based testing

Requirements-based testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it
- Example (health care)

  Requirements

  - If a patient is known to be allergic to any particular medication, then prescription of that medication shall result in a warning message being issued to the system user
  - If a prescriber chooses to ignore an allergy warning, she shall provide a reason why this has been ignored

  Tests

  - Set up a patient record with no known allergies. Prescribe medication for allergies that are known to exist. Check that a warning message is not issued by the system.
  - Set up a patient record with a known allergy. Prescribe the medication to that the patient is allergic to, and check that the warning is issued by the system.
  - Set up a patient record in which allergies to two or more drugs are recorded. Prescribe both of these drugs separately and check that the correct warning for each drug is issued.
  - Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.
  - Prescribe a drug that issues a warning and overrule that warning. Check that the system requires the user to provide information explaining why the warning was overruled.

Testing?

Development testing

- Unit testing
- Component testing
- System testing
- Test-Driven Development

Release testing

- Requirements-based testing
- Scenario testing

Scenario testing

- Use typical scenarios of use to develop test cases
- A scenario is a (narrative) realistic story

George is a nurse who specializes in mental health care. One of his responsibilities is to visit patients at home to check that their treatment is effective and that they are not suffering from medication side effects.

On a day for home visits, George logs into the Mentcare system and uses it to print his schedule of home visits for that day, along with summary information about the patients to be visited. He requests that the records for these patients be downloaded to his laptop. He is prompted for his key phrase to encrypt the records on the laptop.

One of the patients whom he visits is Jim, who is being treated with medication for depression. Jim feels that the medication is helping him but believes that it has the side effect of keeping him awake at night. George looks up Jim's record and is prompted for his key phrase to decrypt the record. He checks the drug prescribed and queries its side effects. Sleeplessness is a known side effect, so he notes the problem in Jim's record and suggests that he visit the clinic to have his medication changed. Jim agrees, so George enters a prompt to call him when he gets back to the clinic to make an appointment with a physician. George ends the consultation, and the system re-encrypts Jim's record.

After finishing his consultations, George returns to the clinic and uploads the records of patients visited to the database. The system generates a call list for George of those patients whom he has to contact for follow-up information and make clinic appointments.

The scenario tests various features of the system:

- Authentication by logging on to the system
- Downloading and uploading of specified patient records to a laptop
- Home visit scheduling
- Encryption and decryption of patient records on a mobile device
- Record retrieval and modification
- Links with the drugs database that maintains side-effect information
- The system for call prompting

Testing?

Development testing

- Unit testing
- Component testing
- System testing
- Test-Driven Development

Release testing

- Requirements-based testing
- Scenario testing
- Performance testing

Performance testing

- To check that system can process its intended workload
- Operation profiles* employed to test whether performance requirements are being achieved

  * classes of system inputs and probability that those inputs will occur in normal use

- Stress testing for defect testing
  - Gradually increase load till passing system limits

Testing?

Development testing

- Unit testing
- Component testing
- System testing
- Test-Driven Development

Release testing

- Requirements-based testing
- Scenario testing
- Performance testing

User testing

User testing

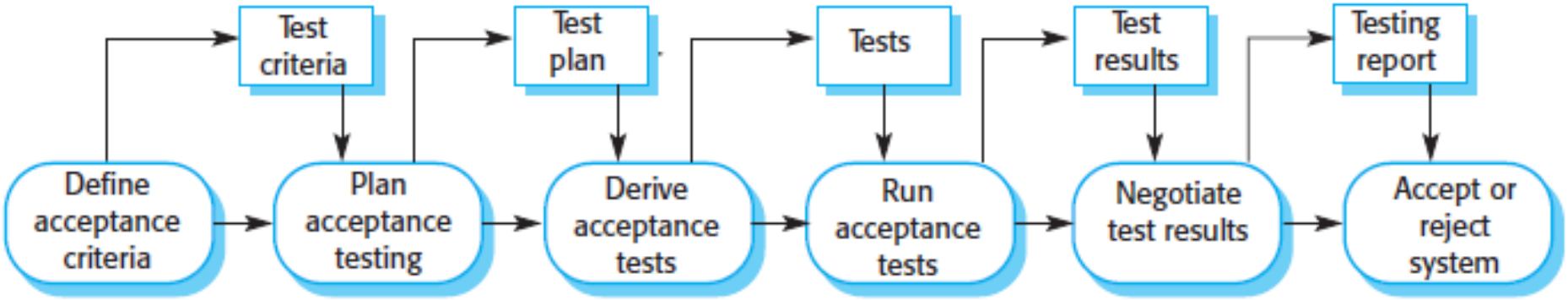
- Users provide input and advice on system testing
- Essential since user's real working environment cannot be fully replicated but it can impact on reliability/performance/usability of the system
- Types of user testing:
  - **Alpha testing**

    Users work with development team to test early releases of software
  - **Beta testing**

    Release made available to larger group of 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 system developers and deployed in the customer environment

Acceptance testing process



- Acceptance criteria should in principle be part of system contract (in practice requirements change during development process)
- Outcome of negotiations can be «conditional acceptance», e.g., when customer wants to start deployment even if there are some problems to be fixed

Agile methods and acceptance testing

- Agile methods do not include in principle acceptance testing as users are embedded in the development team, provide reqs (with users stories) and define tests
- Difficult to embed «typical users» with general knowledge of system will be used
- Many companies develop agile but have acceptance testing too

Summary

- Software testing
  - Testing?
    - validation vs. defect testing
    - software verification and validation
    - inspections vs. testing
    - traditional testing process
  - Development testing
    - Unit testing
      - partition vs. guideline-based testing
    - Component testing
      - types of interfaces, common interface errors
      - interface testing guidelines
    - System testing
      - use case-based testing
      - testing policies
    - Testing-Driven Development
      - definition, benefits
  - Release testing
    - Requirements-based testing
    - Scenario testing
    - Performance testing
  - User testing
    - Alpha|Beta|Acceptance testing

LOL

“Pay attention to zeros. If there is a zero, someone will divide by it.”

“Testers don’t break software, software is already broken.”