Testing

Mariano Ceccato

mariano.Ceccato@univr.it

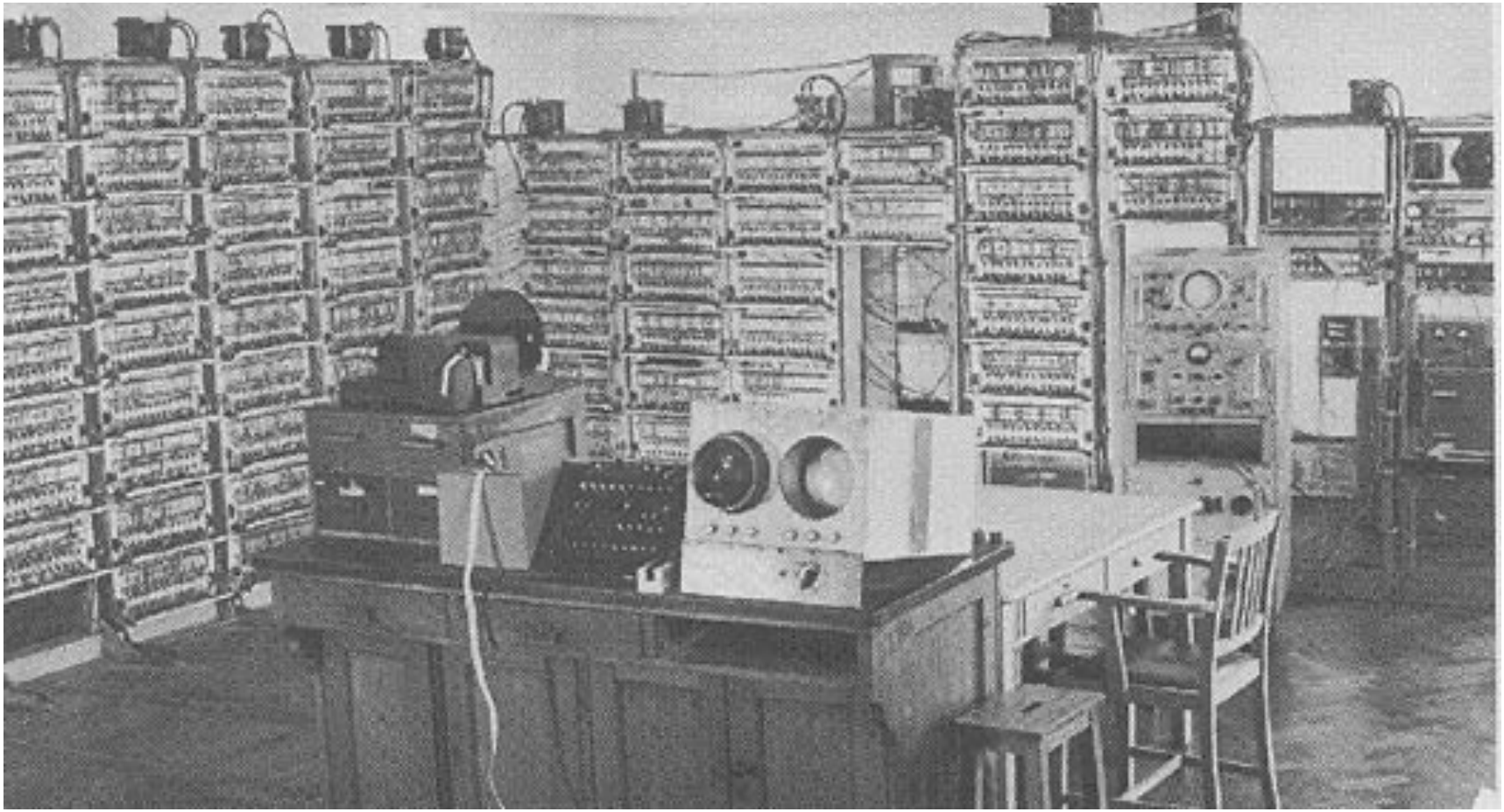
Software process activities

- **Specification**: defining what the system should do;
- **Design and implementation**: defining the organization of the system and implementing the system;
- **Validation**: checking that it does what the customer wants;
- **Evolution**: changing the system in response to changing customer needs.

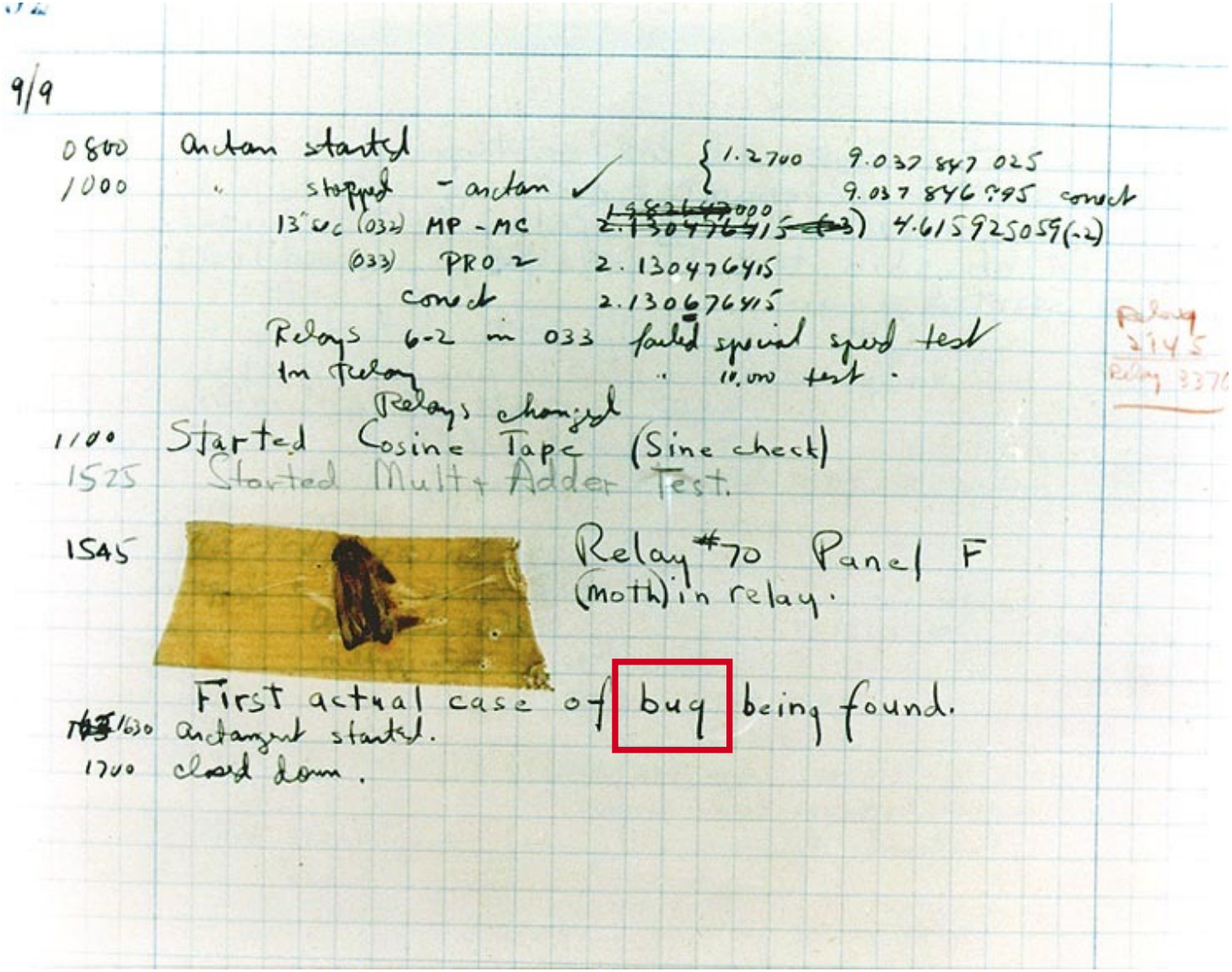
Program testing

- **Objective**: show that a program does what it is intended to do and to discover program defects before it is put into use
- To demonstrate to the developer and the customer that the software meets its requirements:
  - At least one test for every requirement
  - A test for each (main) system feature, plus combinations of these features
- To reveal when the software behavior is incorrect, undesirable or does not conform to its specification
  - E.g., system crashes, unwanted interactions with other systems, wrong results, data corruption → *bug.*

Mark II (1947)



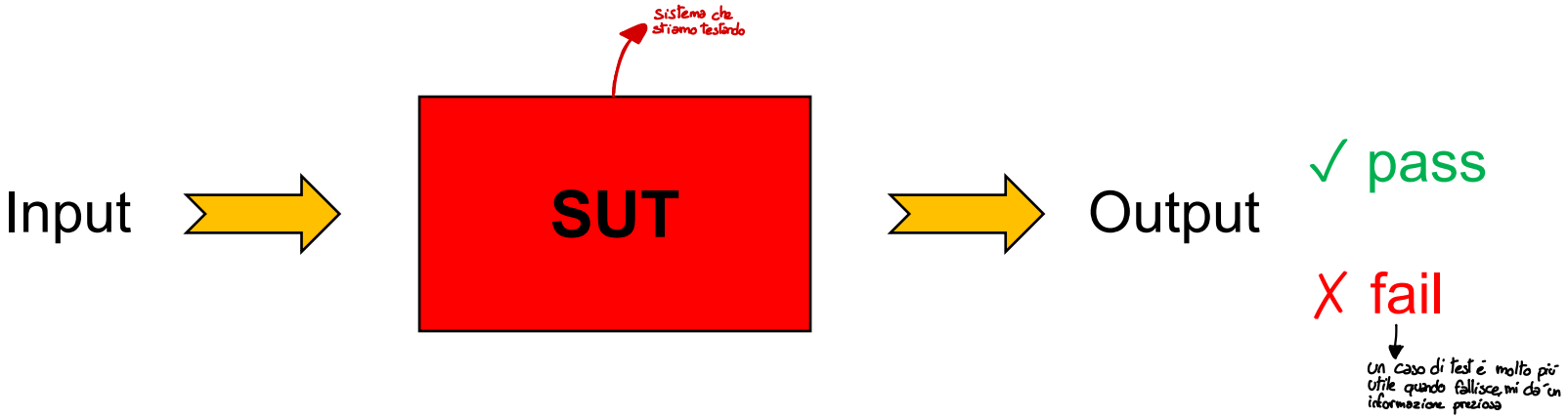
The first “bug”



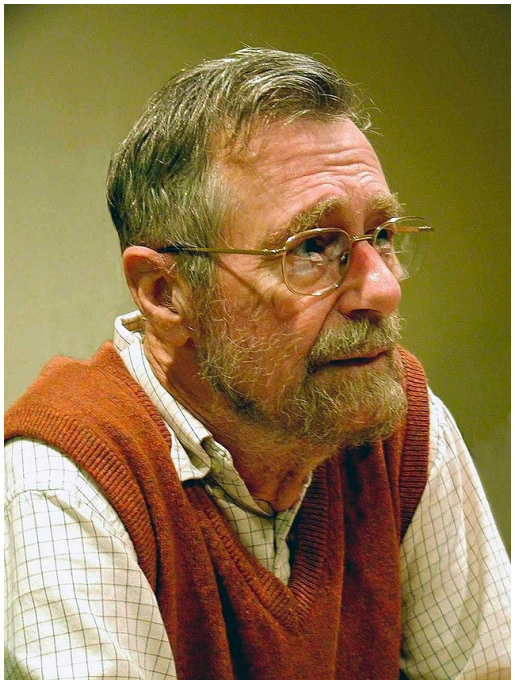
9/9

0800 Arctan started
1000 " stopped - arctan ✓ { 1.2700 9.037 847 025
9.037 846 795 corrct
13"cc (032) MP - MC ~~1.9821[illegible]000~~ ~~2.13047641[illegible]~~) 4.615925059(-2)
(033) PRO 2 2.130476415
corrct 2.130676415
Relays 6-2 in 033 failed special speed test
In relay " 10.00 test.
Relays changed
1100 Started Cosine Tape (Sine check)
1525 Started Multy Adder Test.
1545 Relay #70 Panel F
(moth) in relay.
First actual case of bug being found.
~~1700~~ 1630 Arctangent started.
1700 closed down.

Testing


Input
SUT
sistema che stiamo testando
Output
✓ pass
✗ fail
un caso di test è molto più utile quando fallisce, mi dà un informazione preziosa

Purpose of Testing



- Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance.
- It is always possible that a test you have overlooked could discover further problems with the system.

Tipicamente non si può testare esaustivamente un programma

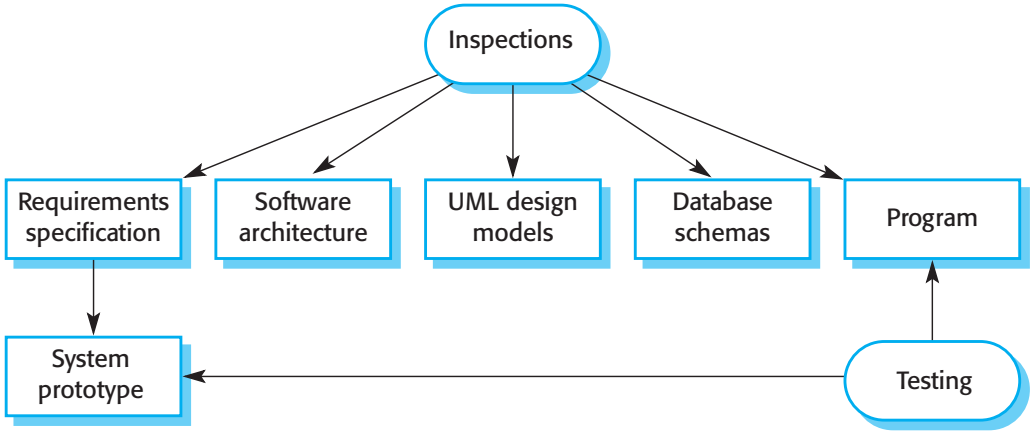
Edsger Dijkstra: "*Testing can only show the presence of errors, not their absence*"*

* Dijkstra, E. W. 1972. "The Humble Programmer." Comm. ACM 15 (10): 859–66. doi:10.1145/ 355604.361591

Confidence

- The software should do what the user really requires
- The final level of confidence depends on:
  - Software purpose
    - how critical the software is to an organization (security product Vs prof of concept)
  - User expectations
    - Expectations may be low on certain kinds of software (failures might be tolerated)
  - Marketing environment
    - Reaching the market early may be more important than finding defects in the program (e.g., to acquire market share before the competitors).

Inspection/review Vs testing



- Inspections are static (code is not run)
- Advantages
  - Not limited to system code
  - Errors are not masked
    - When testing, execution errors may mask subsequent errors
  - Can be performed on incomplete versions
  - Can enforce other quality attributed
    - Standard conformance for maintainability
    - Inefficiencies (e.g., in algorithm implementation)
    - Bad programming style

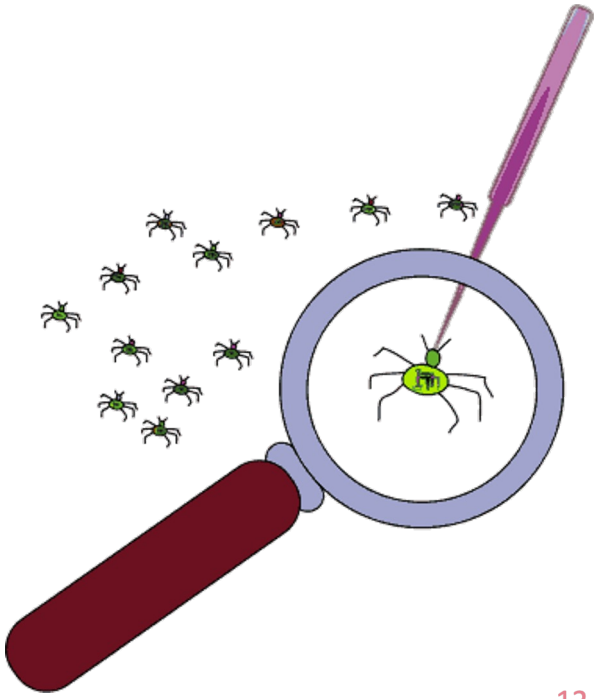
Stages of testing

1. **Development testing:** where the system is tested by developers during development to discover bugs and defects
2. **Release testing:** where a separate testing team test a complete version of the system before it is released to users
3. **User testing:** where users of a system (or internal marketing team members) test the system in their own environment

Stage 1:
Development testing

Development testing

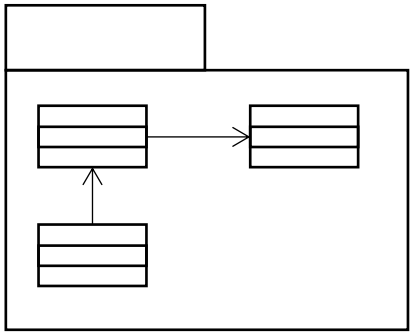
- Carried out by the team developing the system
  - In particular contexts (e.g., critical system) by a different dedicate team
- Objective: discover bugs in the software
  - Usually interleaved with debugging: the process of locating problems with the code and changing the program to fix these problems.
    - Step 1: fault localization
    - Step 2: fault removal



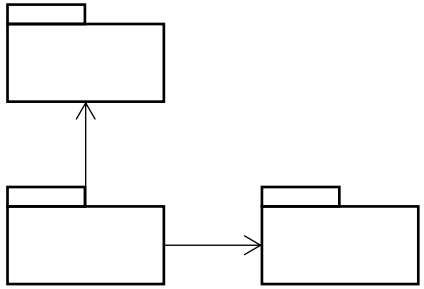
Development testing

- Unit testing:
  - Individual program units are tested
  - **Focus**: functionality of objects or methods

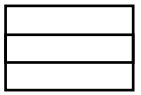
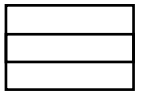
- Component testing
  - Several units are integrated to create composite components.
  - **Focus**: component interface



- System testing
  - Components are integrated and the system is tested as a whole
  - **Focus**: component interaction



Unit testing

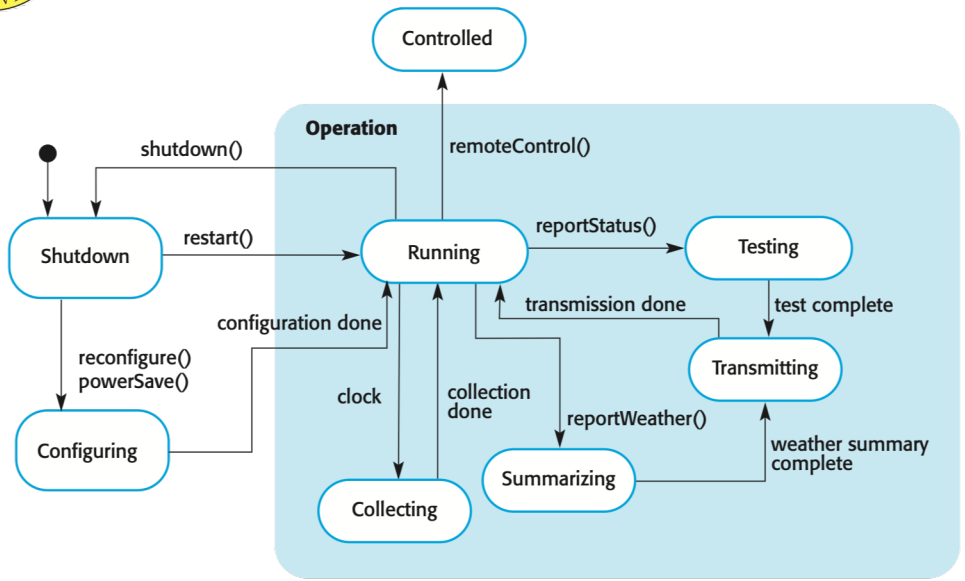



- A **unit** might be:
  - Individual functions or methods within an object
  - Object classes with several attributes and methods
  - Composite components with defined interfaces used to access their functionality

Testing a class

- Complete test coverage of a class involves
  - Testing all operations associated with an object
  - Setting and querying all object attributes
  - Exercising the object in all possible states
- Difficult to achieve complete coverage (e.g., inheritance)

Example

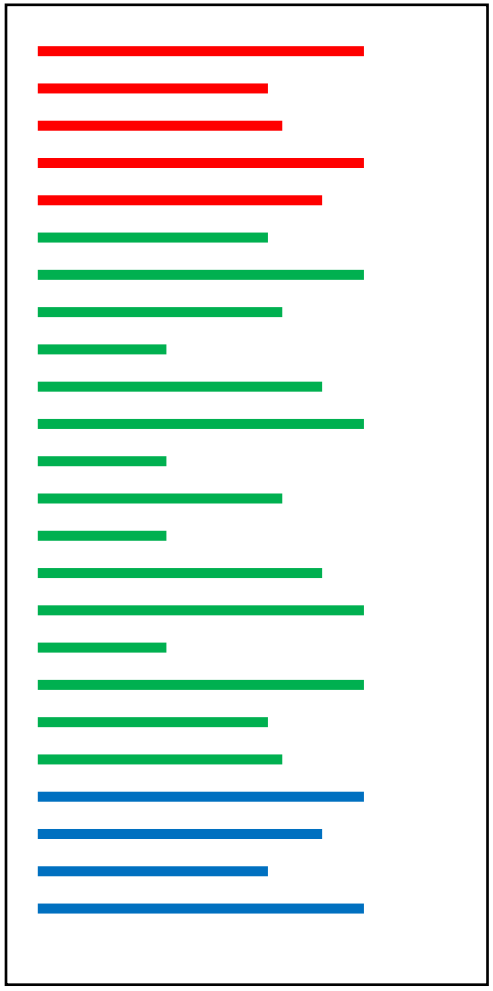


WeatherStation
identifier
reportWeather ( ) reportStatus ( ) powerSave (instruments) remoteControl (commands) reconfigure (commands) restart (instruments) shutdown (instruments)

Examples of state sequences that should be tested in the weather station:

- Shutdown → Running → Shutdown
- Configuring → Running → Testing → Transmitting → Running
- Running → Collecting → Running → Summarizing → Transmitting → Running

Automation of Unit testing



1. A setup part
   - The system is initialized and brought in a testable state
2. A call part,
   - Functionality to be tested are exercised
3. An assertion part
   - Actual result are compared with expected result
   - The test passes/fails

Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention!

Mock

- Dependencies towards units that has not been implemented
  - E.g., a database that is still not available
- Mock: a unit with the same interface as the external missing unit, that is used to simulate its functionality
  - E.g., a mock database with the same interface but with hardcoded constant data

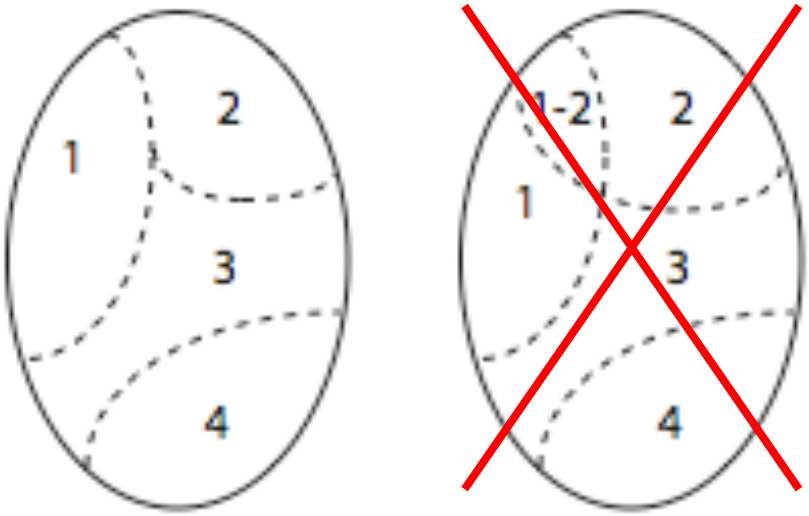
What scenarios to test?

Test scenarios that

1. reflect normal operation of a program. The test shows that the unit works as expected
2. resemble cases when common problems arise
   - Abnormal inputs
   - Attempt to produce wrong output or crashes

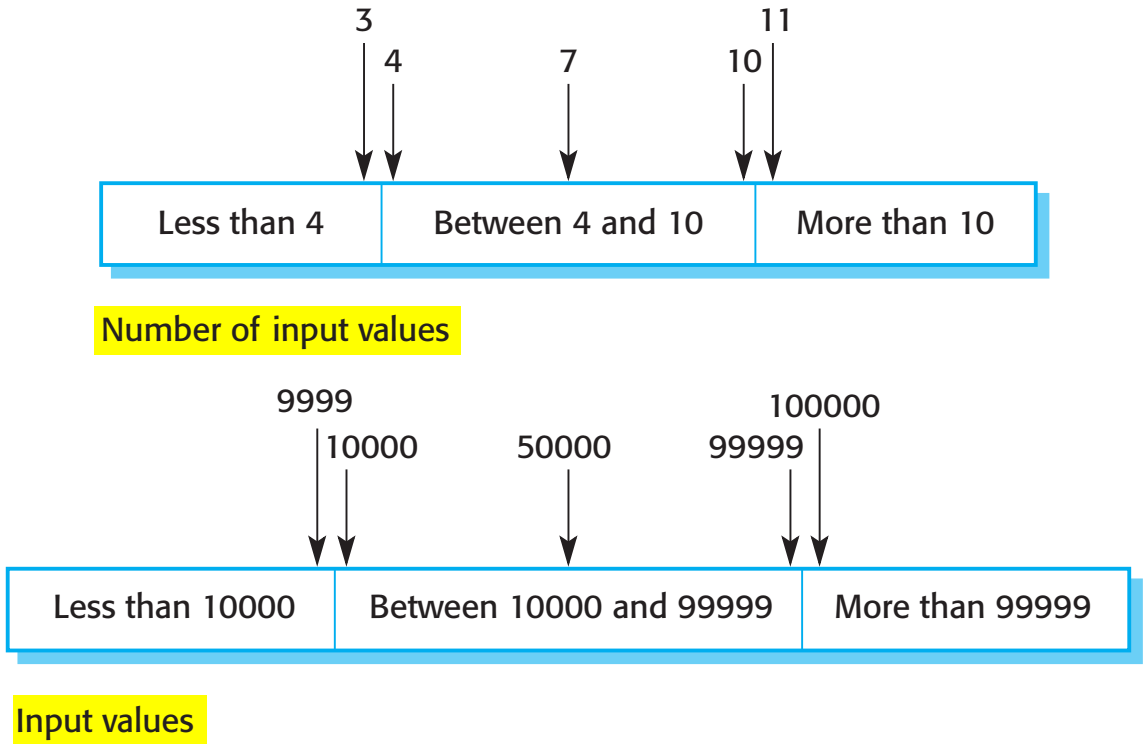
Equivalence partitioning

- **Input partitioning**: groups of inputs that have common characteristics and should be processed in the same way
  - Assumption: the behavior is the same within each group
  - A tests should be chosen from each group
  - The chosen input present the whole class



What test input data?

- Typical value: central point of the partition
- Atypical value: near to the limit of the partition (e.g., -1, 0, +1)
  - Cases maybe overlooked while developing the unit



Testing guidelines

- When coping with sequences (e.g., arrays, lists), chose
  - Multiple sequences with different number of elements
  - Write test cases that use the first, the middle and the last element of the sequence
  - Sequences with just 1 element or with no element (empty sequence)

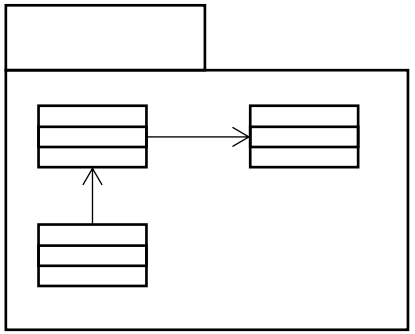
Testing guidelines

- Objective: reflect previous experience of the errors that programmers often make when developing units
  - Choose inputs that force the system to generate all error messages
  - Design inputs that cause input buffers to overflow
  - Repeat the same input or series of inputs numerous times
  - Force invalid outputs to be generated
  - Force computation results to be very large or very small

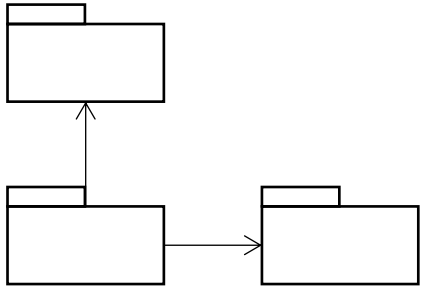
Development testing

- Unit testing:
  - Individual program units are tested
  - **Focus**: functionality of objects or methods

- Component testing
  - Several units are integrated to create composite components.
  - **Focus**: component interface

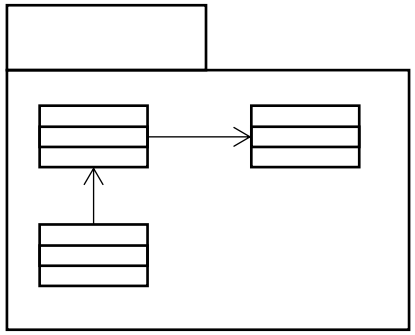


- System testing
  - Components are integrated and the system is tested as a whole
  - **Focus**: component interaction



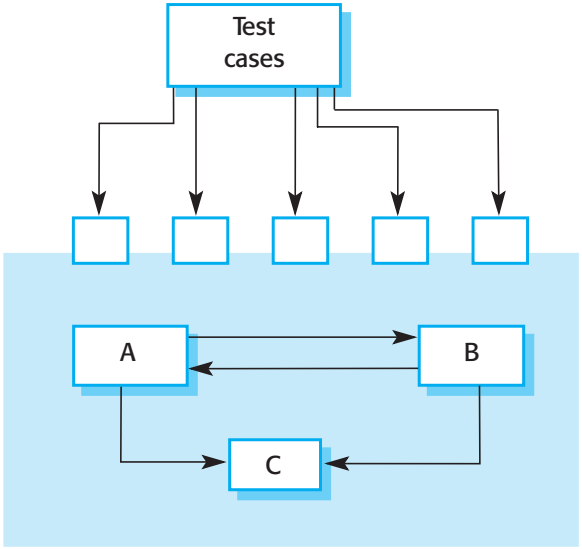
Component testing

- Test cases do not apply to individual units but to the interface of the composite component
- We assume that unit tests on the individual objects within the component have been completed
- Interface errors in the composite component may not be detectable by testing the individual objects, because these errors result from interactions between the objects in the component



Component interface testing

- **Objective**: detect faults due to interface errors or invalid assumptions about interfaces
- Interface types
  - **Parameter interfaces**: Data passed from one method or procedure to another
  - **Shared memory interfaces**: Block of memory is shared between procedures or functions (e.g., sensor integrated systems)
  - **Procedural interfaces:** Sub-system encapsulates a set of procedures to be called by other sub-systems
  - **Message passing interfaces:** Sub-systems request services from other sub-systems



Common interface errors

- **Interface misuse**: A calling component calls another component and makes an error in its use of its interface
  - E.g., parameters with wrong type or in the wrong order
- **Interface misunderstanding**: A calling component embeds assumptions about the behavior of the called component which are incorrect
  - E.g., passing a list to search into, that is *supposed* to be sorted, but it is not
- **Timing errors**: The called and the calling component operate at different speeds and out-of-date information is accessed
  - E.g., reading before the message is ready
  - E.g., real-time systems with shared memory interface or message passing interface

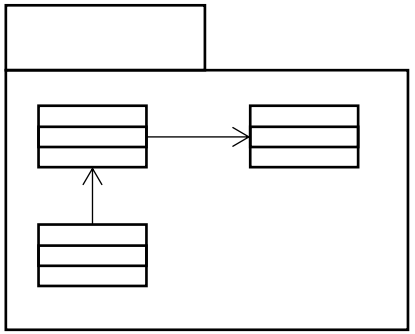
Interface testing guidelines

- Call procedure with parameter values at the extreme ends of their ranges.
- In case parameters are pointers, test with null pointers
- Design tests which cause the component to fail
  - Mismatch in failure assumptions are a common specification misunderstanding
- Use stress testing in message passing systems (to reveal timing problem)
- In shared memory systems, vary the order in which components are activated (to reveal implicit assumptions between producer and consumer)

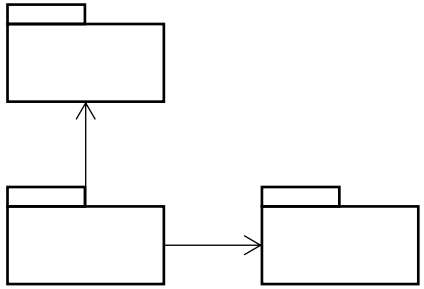
Development testing

- Unit testing:
  - Individual program units are tested
  - **Focus**: functionality of objects or methods

- Component testing
  - Several units are integrated to create composite components.
  - **Focus**: component interface

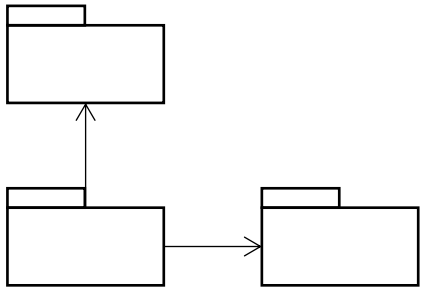


- System testing
  - Components are integrated and the system is tested as a whole
  - **Focus**: component interaction



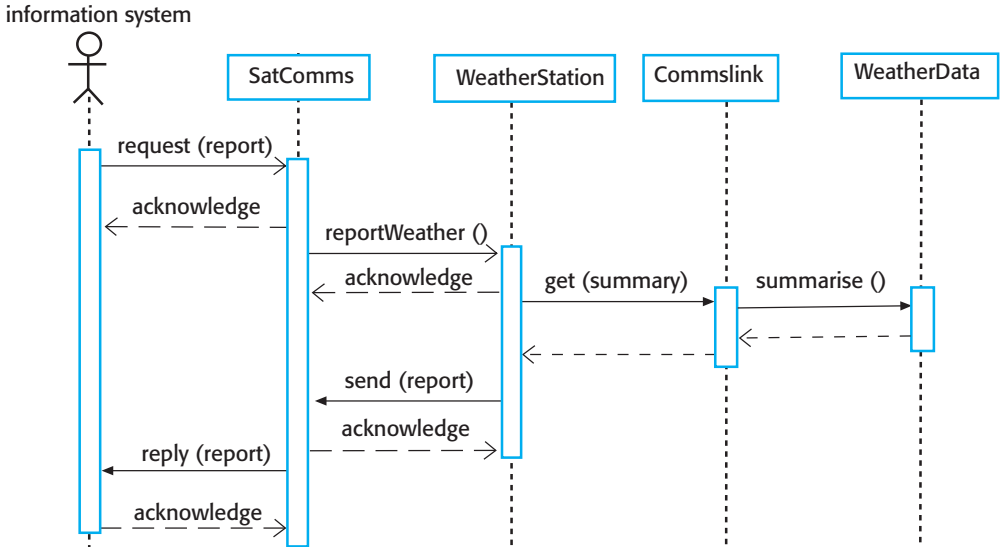
System test

- System testing during development involves integrating components to create a version of the system to be tested
- The focus is testing the interactions between components
  - Some system functionalities only become testable when you put components together
  - To detect wrong hypotheses/assumptions made by developers on other components
- Checks that
  - components are compatible
  - components interact correctly
  - components transfer the right data at the right time across their interfaces.



System test based on use cases

- The 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 these interactions to occur
- The sequence diagrams associated with the use case documents the components and interactions that are being tested
  - Tests should also take exceptions into account (not completely reported in sequence diagrams) and ensure that they are correctly handled



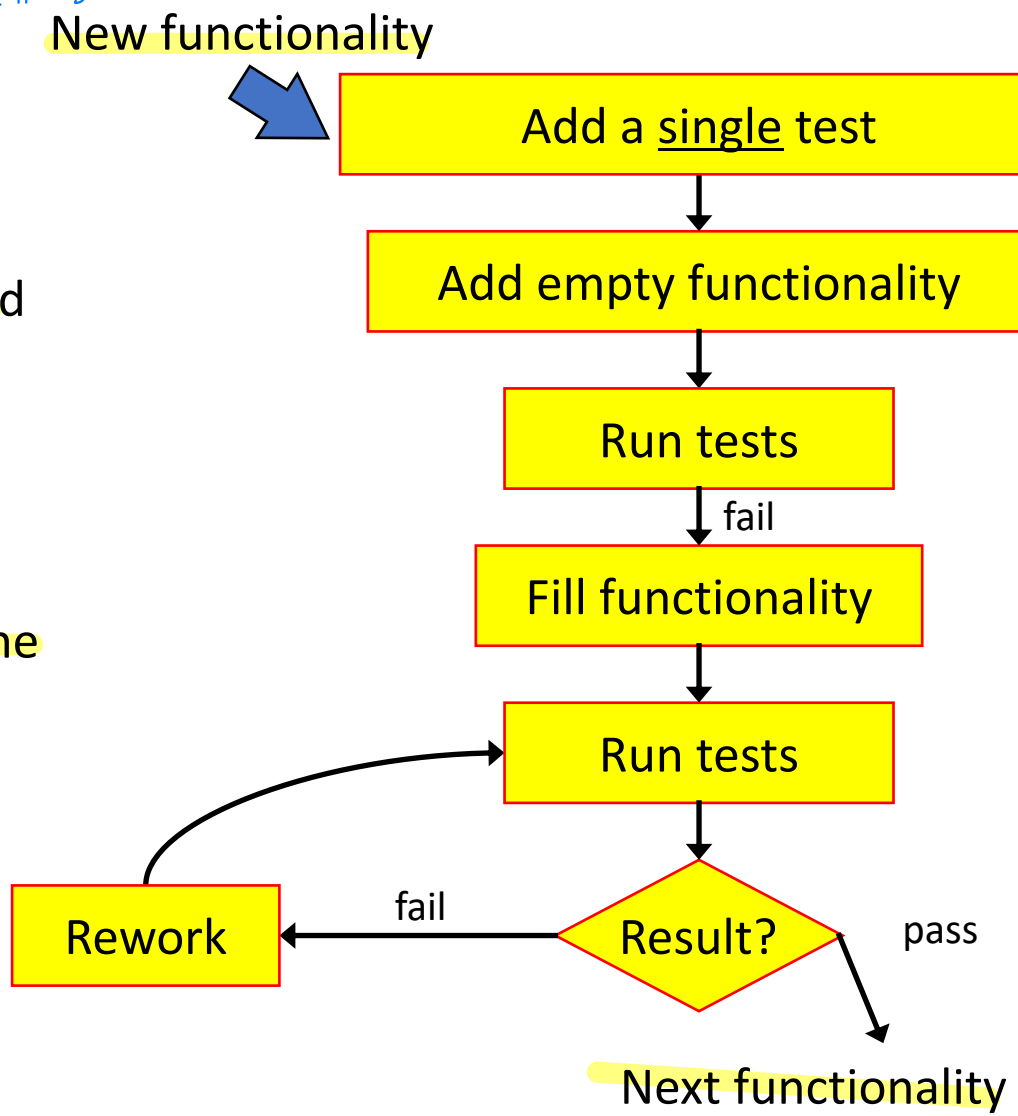
Testing policies

- Testing all the possible executions in a system is impossible
- Policies to consider testing adequate.
- Examples of policies:
  - All the instructions in the program should be executed by at least one test
  - 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 both correct and incorrect input.

Test driven development

- Introduced with agile software development (XP: eXtreme Programming)
  1. Identify the incremental feature to be implemented
  2. Write one (or more) test cases for it
  3. Run the test(s): it fails!
  4. Implement the feature
  5. When all the tests pass, the feature is released

- Objective: clarify what the new feature is about before start implementing it



Advantages of TDD

- **Code coverage**: there is at least a test case for each segment of system code
  ↳ alta copertura delle funzionalità
- **Regression test**: it is always possible to (automatically) run all the test cases, so existing features are tested each time a new feature is added
- **Simplified debugging**: a failing test is related to a single feature, it should be obvious where the fault is localized
- **System documentation**: tests can be read to understand the system features

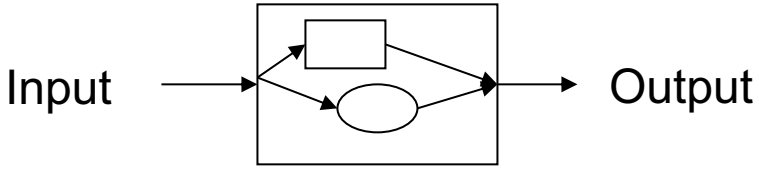
Stages of testing

1. **Development testing:** where the system is tested by developers during development to discover bugs and defects
2. **Release testing:** where a separate testing team test a complete version of the system before it is released to users
3. **User testing:** where users of a system (or internal marketing team members) test the system in their own environment

Stage 2: Release testing

Release testing

- Usually responsibility of a team different than the development team
- Check if requirements are met, so the system can be delivered to users
- Usually a black-box testing process: tests are only derived from the system specification



Release Vs 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).

Requirement based release testing

- Examining each requirement and developing a test or tests for it

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, they shall provide a reason why this has been ignored

Tests:

1. 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.

2. 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.

3. 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.

4. Prescribe two drugs that the patient is allergic to. Check that two warnings are correctly issued.

5. 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.

Scenario testing

- Scenario: story that describes one way in which the system might be used
  - Realistic
  - Includes multiple requirements
  - In case scenarios and user stories are available from the *requirement engineering* process, they can be directly used as testing scenarios

Performance testing

- **Objective**: ensure that the system can process its intended load
- Tests should reflect the operational profile of the system
  - tests that resemble the actual mix of work that will be handled by the system
    - E.g., 90% transaction type A, 5% of type B, the rest of type C, D and E
- Series of tests where the load is steadily increased until the system performance becomes unacceptable
- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behavior.

Stages of testing

1. **Development testing:** where the system is tested by developers during development to discover bugs and defects
2. **Release testing:** where a separate testing team test a complete version of the system before it is released to users
3. **User testing:** where users of a system (or internal marketing team members) test the system in their own environment

Stage 3:
User testing

Type of user testing

- Influences from the user's working environment can impact the reliability, performance, usability, and robustness of a system
- Development testing and release testing are not enough
  - Final users or customers give their input and suggestions on system testing

ambiente di simulazione molto simile a quello finale

Alpha testing

Few users of the software work with the development team to test the software at the developer's site

Beta testing

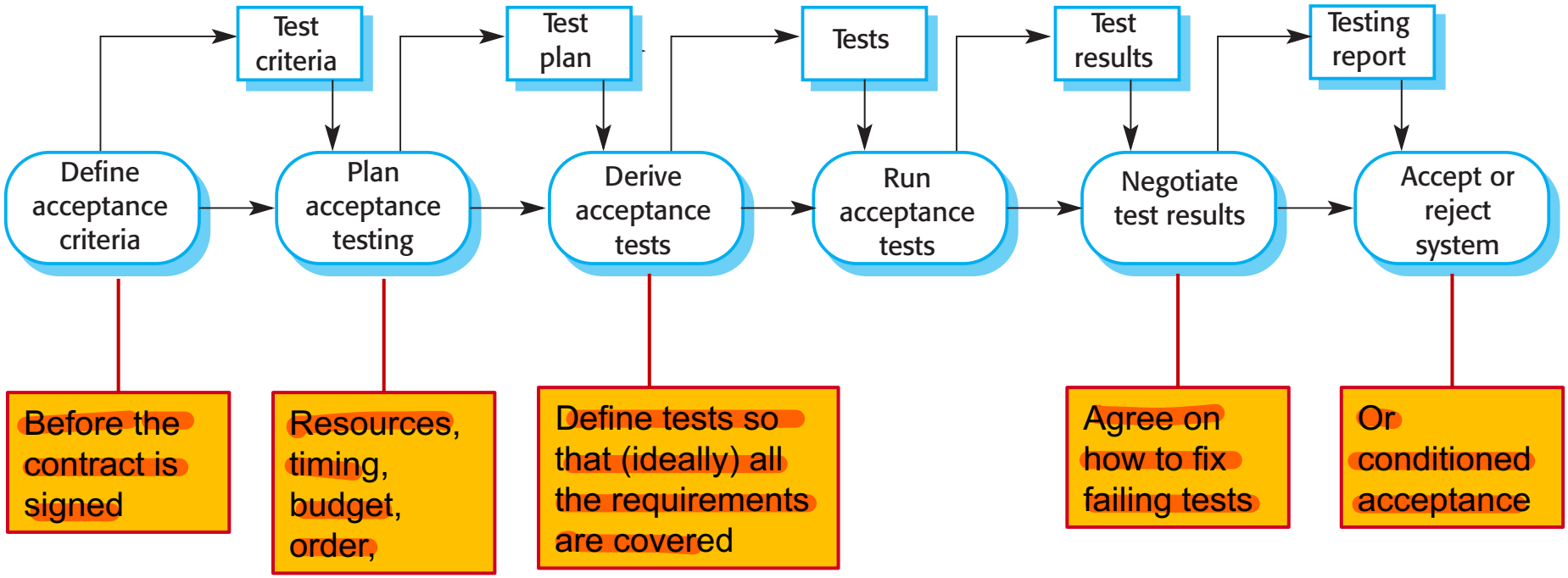
A large group of users experiment with the system and to raise problems that they discover to system developers

Acceptance testing

Customers test a system to decide whether or not it is ready to be accepted and deployed in the customer environment.

The acceptance testing process

- Acceptance imply that final payment should be made for the software



Agile methods in acceptance testing

- In agile methods, the user/customer is part of the development team and is responsible for making decisions on the acceptability of the system
- Tests are defined by the user/customer and are integrated with other tests in that they are run automatically when changes are made
- There is no separate acceptance testing process
- Main problem here is whether or not the embedded user is *typical* and can represent the interests of all system stakeholders

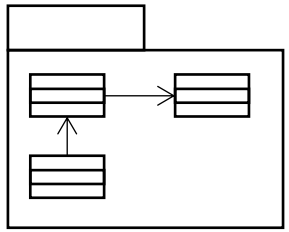
Summary

Stages of testing

1. **Development testing:** where the system is tested by developers during development to discover bugs and defects
2. **Release testing:** where a separate testing team test a complete version of the system before it is released to users
3. **User testing:** where users of a system (or internal marketing team members) test the system in their own environment

1. **Development testing:**
   - Unit testing

   - Component testing



   - System testing