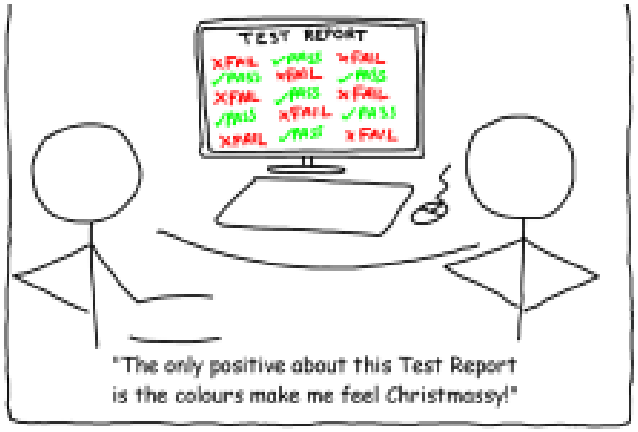
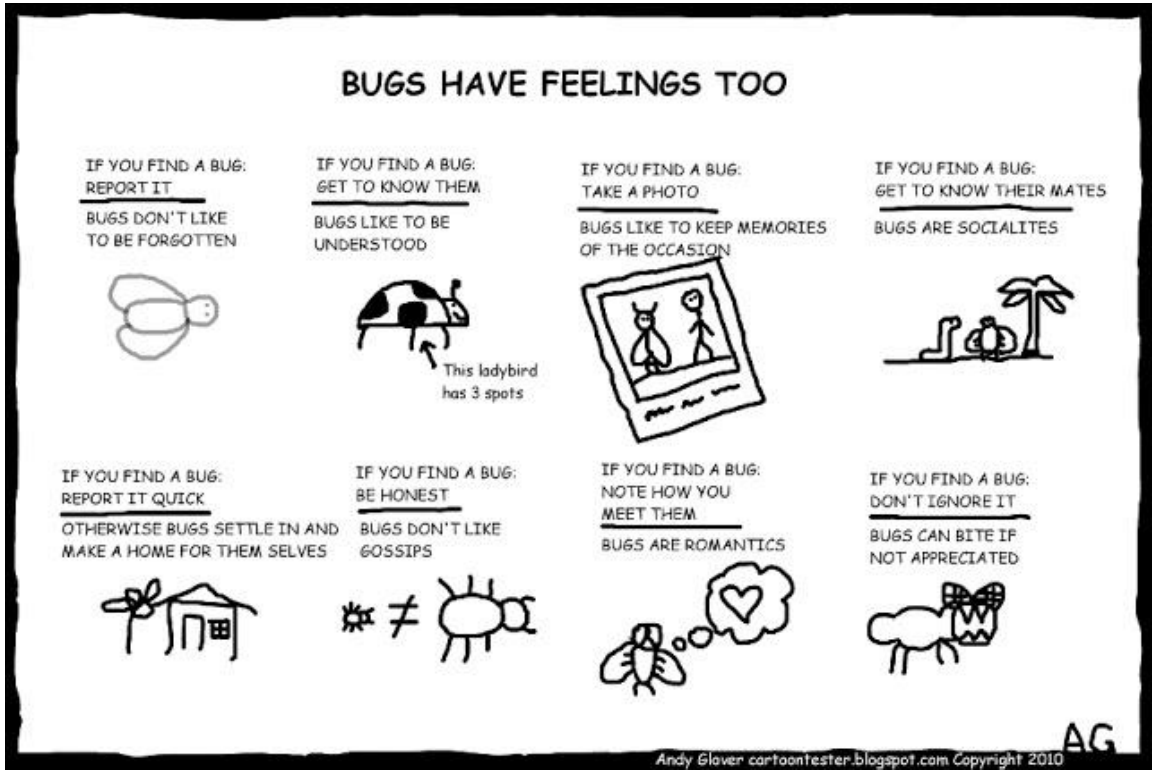
Software Testing

Why do we test?





A bit of facts (1)

- February 25, 1991, Dhahran, Saoudi Arabia, Operation „Desert Storm”
  - A Patriot system failed to track and intercept an incoming Iraqi Scud missile.
  - 28 killed, roughly 100 wounded
  - … The fix was on the way

A bit of facts (2)

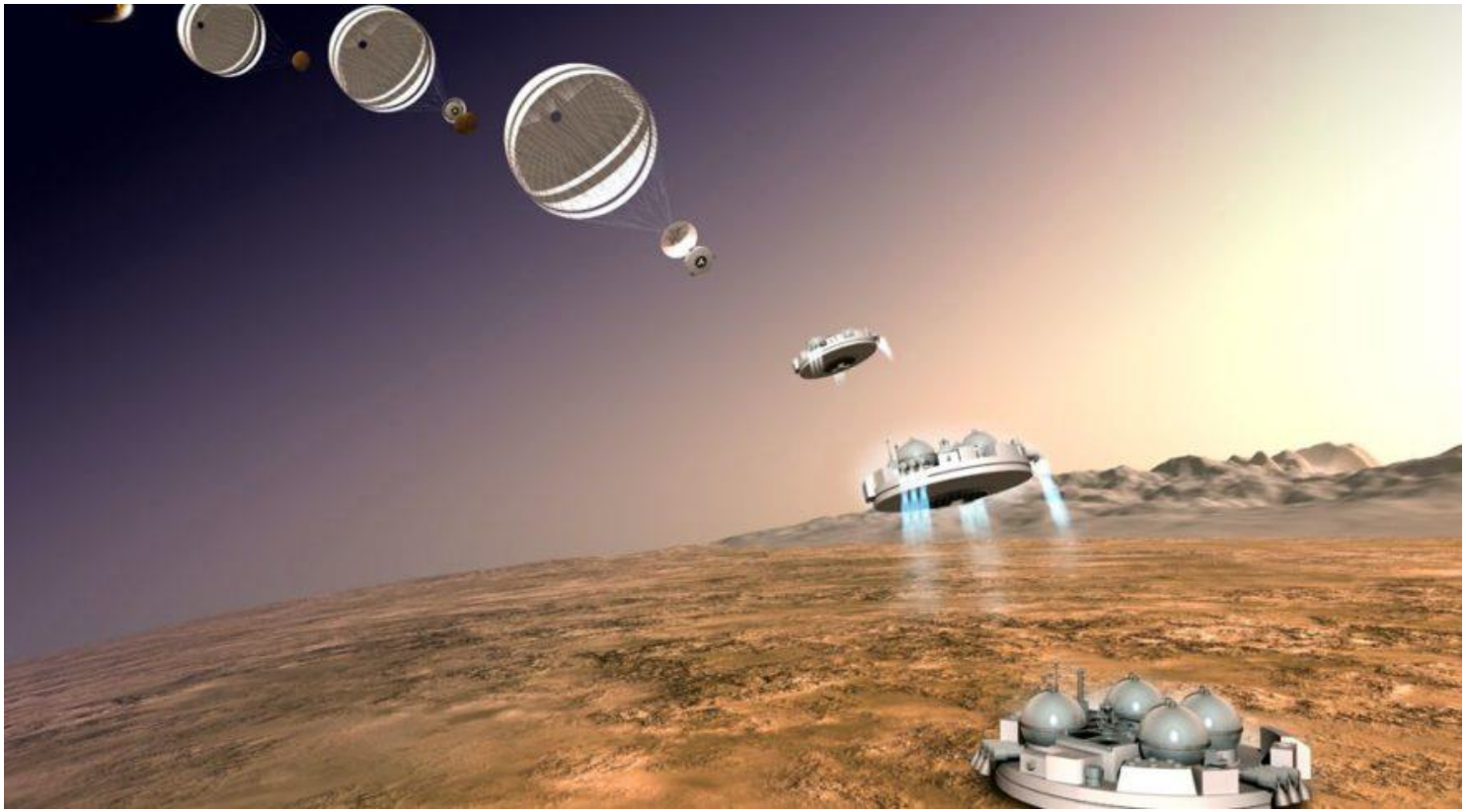
- June 4 1996, Kourou, Guiana Space Centre
  - Ariane 5 virgin voyage…
  - http://www.youtube.com/watch?feature=player_detailpage&v=gp_D8r-2hwk#t=33
  - http://www.ima.umn.edu/~arnold/disasters/ariane5rep.html

A bit of facts (3)

- NASA Mars Climate Orbiter (23.09.1999)

A bit of facts (4)

- Schiaparelli lander (19.10.2016)



A bit of facts (5)

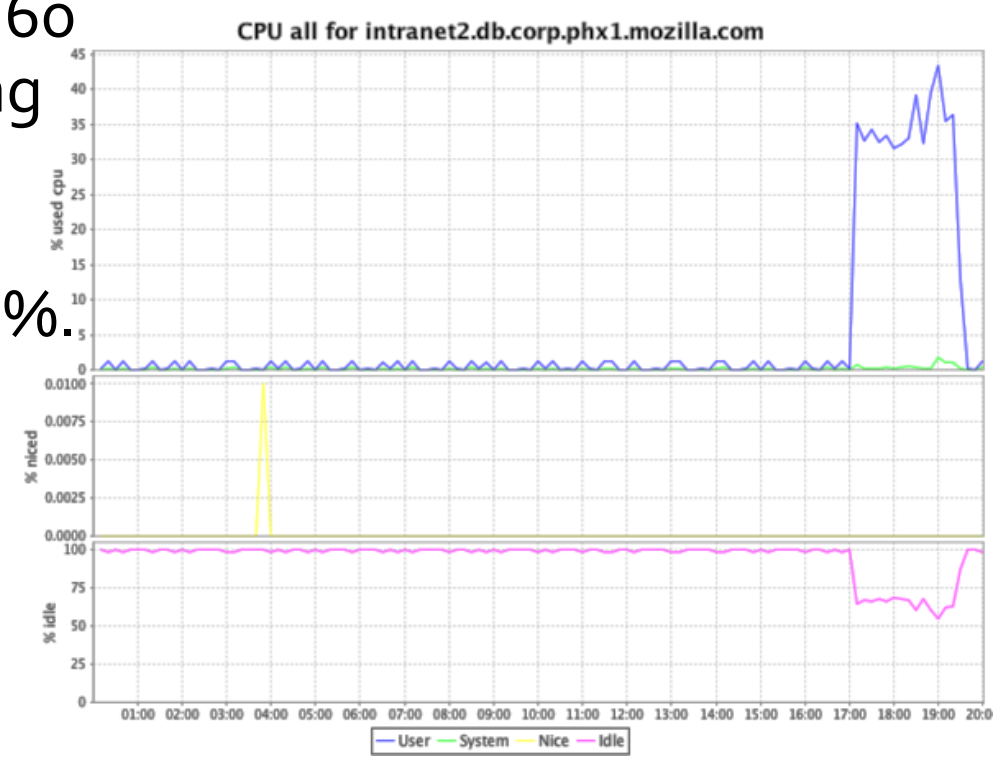
- August 2003, south-eastern part of the USA
  - Blackout
    - memory corruption error caused by concurrency problems
    - (Www.icfi.com/Markets/Energy/doc_files/blackout-economic-costs.pdf)

A bit of facts (6)

- June 1985 – January 1987,
  - Therac-25 – a linear accelerator,
    - several cancer patients receiving deadly radiation overdoses
  - Race condition: events do not happen in the order the programmer intended -> bugs
  - https://en.wikipedia.org/wiki/Therac-25
- 2001, Instituto Oncologico Nacional, Panama City
  - Treatment-planning software from Multidata Systems incorrectly calculated radiation dosages
  - 28'th patients received excessive amounts of radiation, with fatal consequences for several.
  - https://en.wikipedia.org/wiki/Instituto_Oncologico_Nacional
  - …

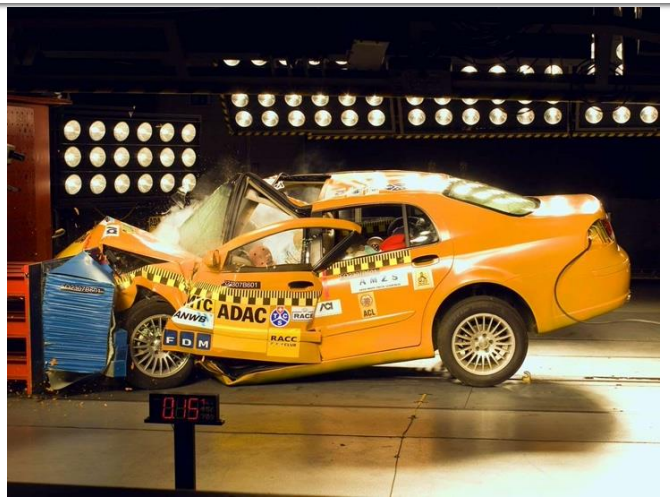
A bit of facts (7)

- 30.06.2012 godz. 23:59:60
- Some computers running under the control of the Linux system begin to use the processor in 100%.
  - LinkedIn, Reddit, …



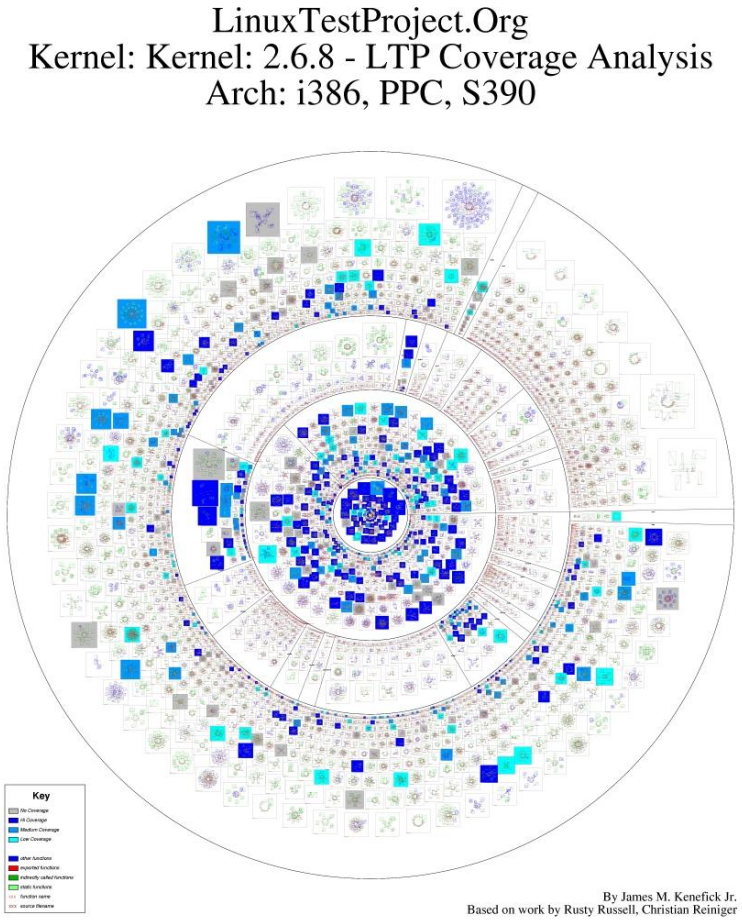
- 31.07.2012 godz. 23.59.59
  - NTP servers announce the addition of a leap second

Test does not always mean the same …



How to carry testing?

jennycham.co.uk



A bit of facts, cont. Glass R.L., 2002 „Facts and Fallacies of Software Engineering", Addison-Wesley (1)

- Software that a typical programmer believes to be thoroughly tested has often had only about 55 to 60 percent of its logic paths executed.
  - **Using automated support**, such as coverage analyzers, can raise that roughly to 85 to 90 percent.
  - It is nearly impossible to test software at the level of 100 percent of its logic paths.

A bit of facts, cont. Glass R.L., 2002 „Facts and Fallacies of Software Engineering", Addison-Wesley (2)

- Even if 100% code coverage with the test could be possible, there are not enough testing criteria.
  - About 35% of software defects result from the lack of specific paths
    - skip errors
  - About 40% of software defects are the result of an exceptional combination of paths
    - combinatorial errors

A bit of facts, cont. Glass R.L., 2002 „Facts and Fallacies of Software Engineering", Addison-Wesley (3)

- Rigorous software inspections can remove up to 90 percent of errors before the first test case starts.
  - Studies show that inspection costs are lower than the costs of testing needed to remove the same errors
- But despite these advantages, inspections can not and should not replace testing.

A bit of facts, cont. Glass R.L., 2002 „Facts and Fallacies of Software Engineering", Addison-Wesley (4)

- Retrospections (so-called post-implementation reviews) are considered an important method from the point of view of determining user satisfaction and process improvement.
  - But most organizations do not do it …

Topics covered

- Software verification and validation
- Testing during development / production
- Release testing, User testing
- Formal verification of programs

Verification vs validation

- **Verification**

  "Are we building the product right".

  - The software should conform to its specification.

- **Validation**

  "Are we building the right product".

  - The software should do what the user really requires.

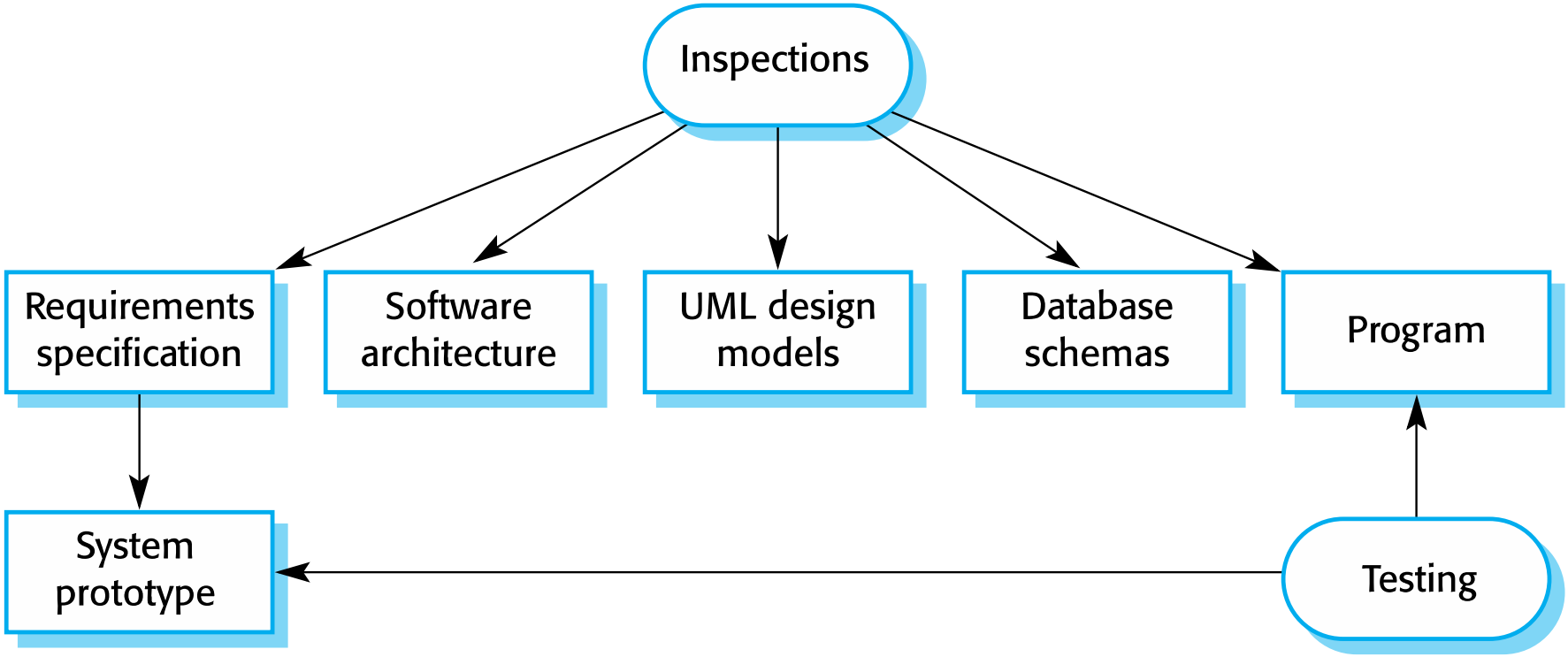
V & V confidence

- Aim of V & V is to establish confidence that the system is 'fit for purpose'.
- Depends on system's purpose, user expectations and marketing environment
  1. Software purpose
     - The level of confidence depends on how critical the software is to an organisation.
  2. User expectations
     - Users may have low expectations of certain kinds of software.
  3. Marketing environment
     - Getting a product to market early may be more important than finding defects in the program.

Inspections and testing

- **Software inspections -** Concerned with analysis of the static system representation to discover problems (static verification)
  - May be supplement by tool-based document and code analysis.
- **Software testing -** Concerned with exercising and observing product behaviour (dynamic verification)
  - The system is executed with test data and its operational behaviour is observed.

Inspections and testing



Software inspections

- These involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections not require execution of a system so may be used before implementation.
- They may be applied to any representation of the system (requirements, design, configuration data, test data, etc.).
- They have been shown to be an effective technique for discovering program errors.

Inspections and testing

- Inspections and testing are complementary and not opposing verification techniques.
- Both should be used during the V & V process.
- Inspections can check conformance with a specification but not conformance with the customer's real requirements.
- Inspections cannot check non-functional characteristics such as performance, usability, etc.

Testing

- **Running** the program in the context of artificial data and reasoning based on the results of its actions

What does it mean that the test was successful?

Program testing goals

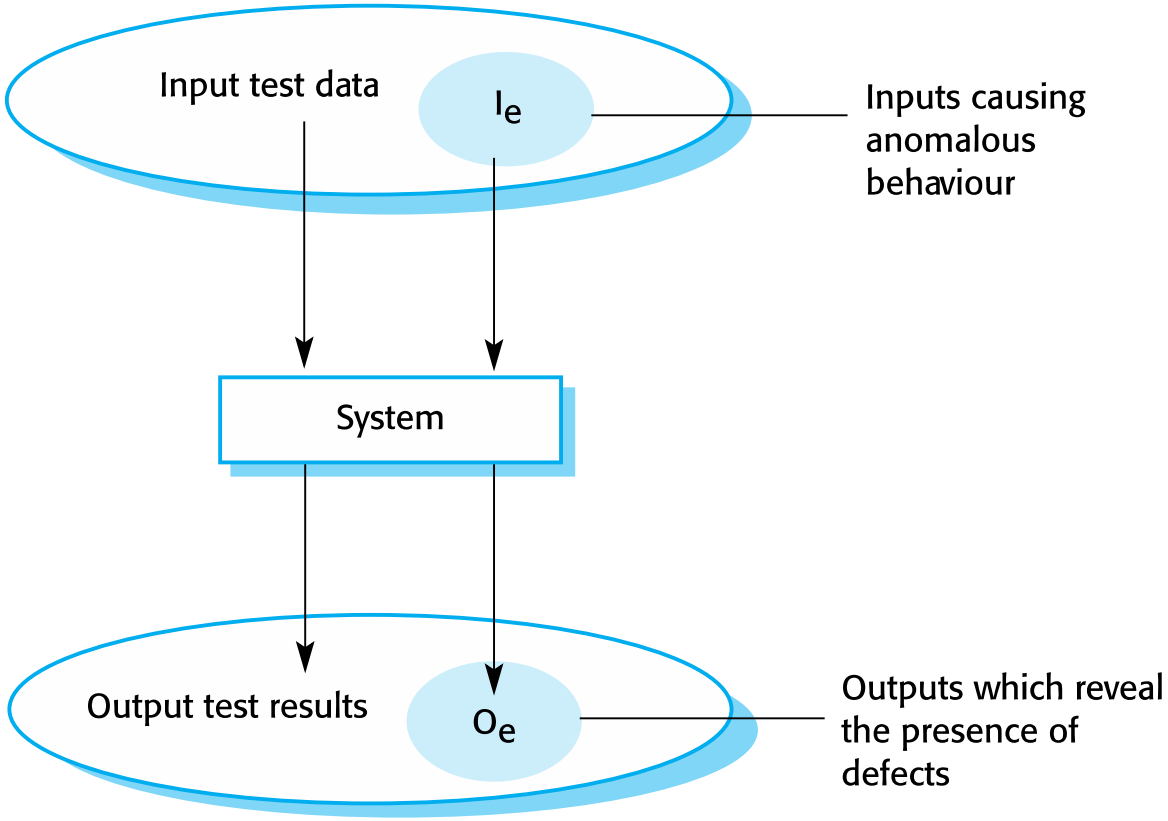
1. To demonstrate to the developer and the customer that the software meets its requirements.
   - For custom software, this means that there should **be at least** one test for every requirement in the requirements document. For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.

2. To discover situations in which the behavior of the software is incorrect, undesirable or does not conform to its specification.
   - Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations and data corruption.

Validation and defect testing

- Validation testing
  - To demonstrate to the developer and the system customer that the software meets its requirements
  - A successful test shows that the system operates as intended.
- Defect testing
  - To discover faults or defects in the software where its behaviour is incorrect or not in conformance with its specification
  - A successful test is a test that makes the system perform incorrectly and so exposes a defect in the system.

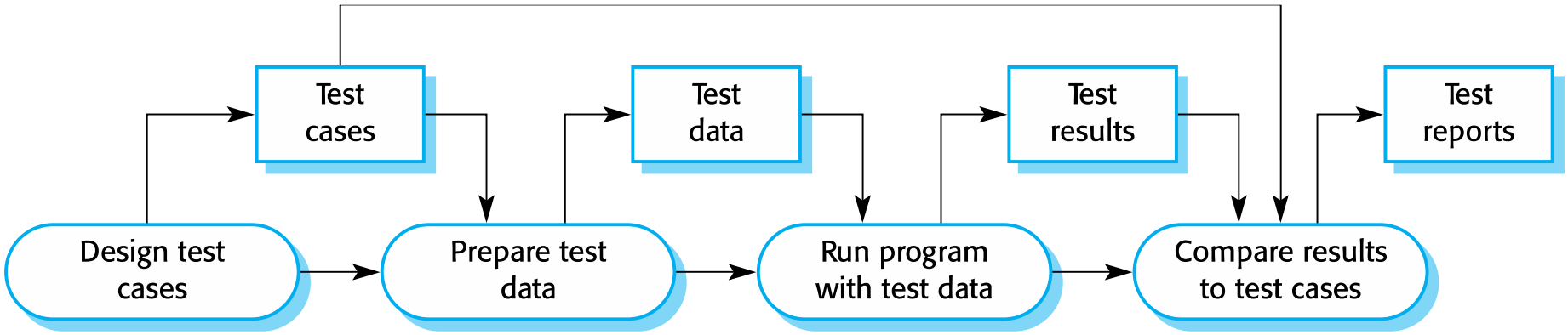
An input-output model of program testing



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

Edsger Dijkstra

A model of the software testing process



Stages of testing

1. **Development testing**, where the system is tested 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 or potential users of a system test the system in their own environment.

Stage 1: Development testing

- Development testing includes all testing activities that are carried out by the team developing the system.
  1. **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
  2. **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
  3. **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Unit testing

- Unit testing is the process of testing individual components in **isolation**.
  - External dependencies must be provided in a controlled manner
    - Simulation code, mock objects
- It is a defect testing process.
- Units may 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 (e.g., several classes in the package).
    - The unit test concerns the implementation of the component.

Unit testing (2)

- Automate execution
- Test coverage
- External dependencies:
  - may:
    - provide non-deterministic results; act slowly; have difficult conditions to reproduce; do not exist yet …
  - … they must have „*doubles*"

Unit tests and external dependencies

- The *double* is for the viewer (tested unit) "transparent"
- Types of *double* dependencies
  - **Dummy objects**
    - „filler" object
  - **Fake objects**
    - „non-productive" implementation
  - **Stubs**
    - They provide fixed responses to selected calls
  - **Mocks**
    - Objects built by specifying the manner in which they should be called (behavior verification)

Stub vs Mock

```
public interface MailService {
  public void send (Message msg);
}
public class MailServiceStub implements MailService {
  private List<Message> messages = new ArrayList<Message>();
  public void send (Message msg) {
    messages.add(msg);
  }
  public int numberSent() {
    return messages.size();
  }
}
```

Stub

```
class OrderStateTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    MailServiceStub mailer = new MailServiceStub();
    order.setMailer(mailer);
    order.fill(warehouse);
    assertEquals(1, mailer.numberSent());
  }
```

Test

Mock & test

```
class OrderInteractionTester...
  public void testOrderSendsMailIfUnfilled() {
    Order order = new Order(TALISKER, 51);
    Mock warehouse = mock(Warehouse.class);
    Mock mailer = mock(MailService.class);
    order.setMailer((MailService) mailer.proxy());

    mailer.expects(once()).method("send");
    warehouse.expects(once()).method("hasInventory")
      .withAnyArguments()
      .will(returnValue(false));

    order.fill((Warehouse) warehouse.proxy());
  }
}
```

http://martinfowler.com/articles/mocksArentStubs.html

Automated testing

- Whenever possible, unit testing should be automated so that tests are run and checked without manual intervention.
- In automated unit testing, you make use of a test automation framework (such as JUnit) to write and run your program tests.
- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success of otherwise of the tests.

Automated test components

1. A **setup** part, where you initialize the system with the test case, namely the inputs and expected outputs.
2. A **call** part, where you call the object or method to be tested.
3. An **assertion** part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful if false, then it has failed.
4. A **tear down** part where you clean up (e.g., removing a leftover condition that could affect other tests)

Unit test effectiveness

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.
2. If there are defects in the component, these should be revealed by test cases.

- This leads to 2 types of unit test case:
  1. The first of these should reflect normal operation of a program and should show that the component works as expected.
  2. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

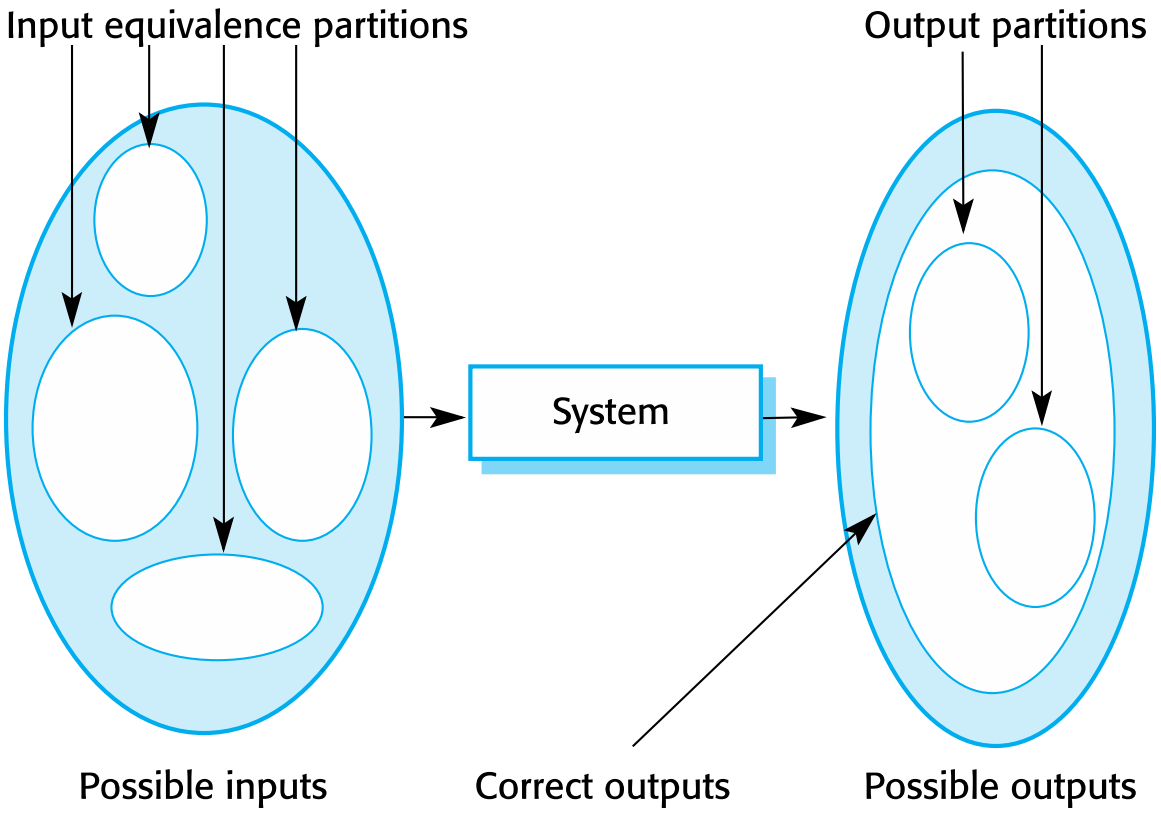
Testing strategies

1. Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way.
   - You should choose tests from within each of these groups.
2. Guideline-based testing, where you use testing guidelines to choose test cases.
   - These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

Partition testing

- Input data and output results often fall into different classes where all members of a class are related.
  - positive numbers, negative numbers, and menu selections.
- Each of these classes is an equivalence partition or domain where the program behaves in an equivalent way for each class member.
- Test cases should be chosen from each partition.

Equivalence partitioning



Sample - brainstorm

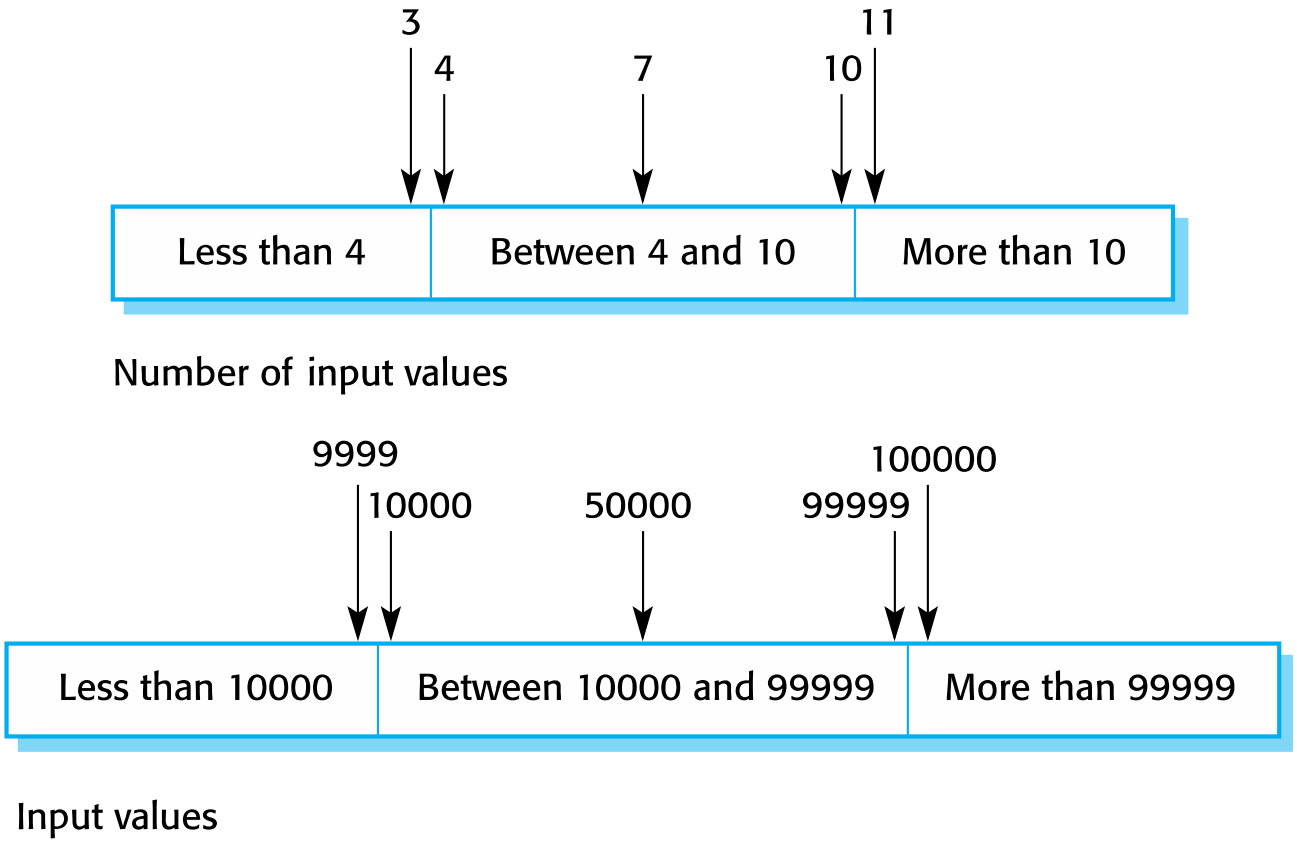
- A field can accept integer values between 20 and 50.
- What tests should you try?

A Test Ideas List for Integer-Input Tests

- Common answers to the exercise would include:

Test	Why it's interesting	Expected result
20	Smallest valid value	Accepts it
19	Smallest -1	Reject, error msg
0	0 is always interesting	Reject, error msg
Blank	Empty field, what's it do?	Reject? Ignore?
49	Valid value	Accepts it
50	Largest valid value	Accepts it
51	Largest +1	Reject, error msg
-1	Negative number	Reject, error msg
4294967296	2^32, overflow integer?	Reject, error msg

Equivalence partitions



General testing guidelines

- 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 too large or too small.

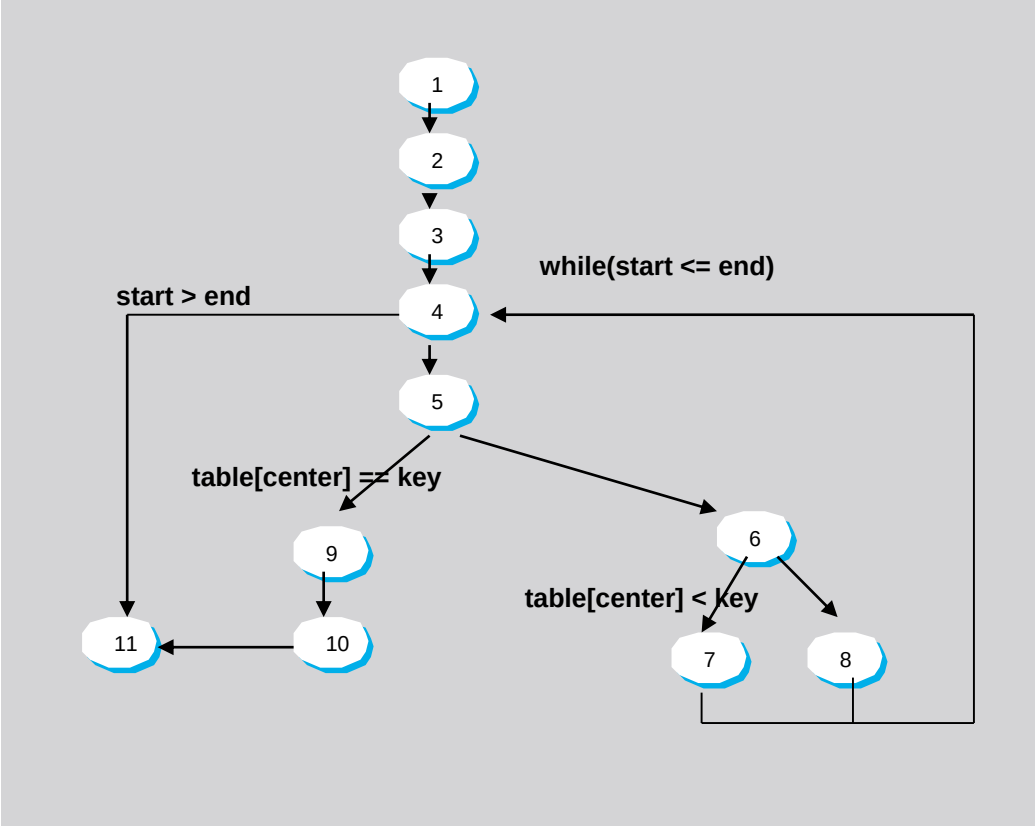
Structural tests

- So called **white-box testing**
- Developed on the basis of knowledge about the structure and implementation of software
- The goal is to ensure that each program statement is executed at least once (not all possible program paths).
- It is possible to develop for relatively small components

Path testing

- A variation of structural testing, the purpose of which is to examine each independent program execution path
- The starting point is the development of a stream graph (flow graph) of the program
  - Nodes represent decisions (decision instructions)
  - The edges represent the flow of control
- The method is mainly used as part of unit testing

Binary search – stream graph



Independent paths

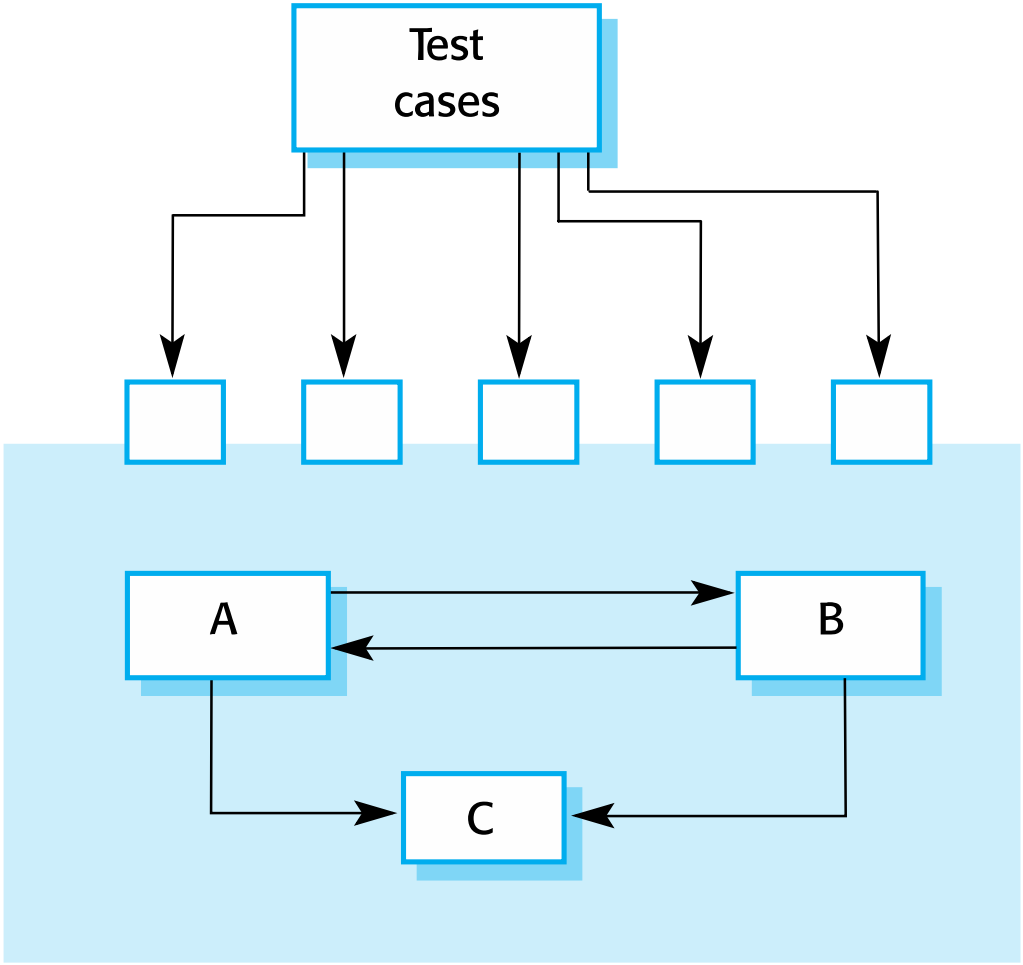
- 1,2,3,4,5,9,10,11
- 1,2,3,4,11
- 1,2,3,4,5,6,7,4…
- 1,2,3,4,5,6,8,4…

- Test cases should run all paths

Component testing

- Software components are often composite components that are made up of several interacting objects.
  - For example, in the weather station system, the reconfiguration component includes objects that deal with each aspect of the reconfiguration.
- You access the functionality of these objects through the defined component interface.
- Testing composite components should therefore focus on showing that the component interface behaves according to its specification.
  - You can assume that unit tests on the individual objects within the component have been completed.

Component testing - Interface testing



Interface testing

- Objectives are to detect faults due to interface errors or invalid assumptions about interfaces.
- Interface types
  1. Parameter interfaces Data passed from one method or procedure to another.
  2. Shared memory interfaces Block of memory is shared between procedures or functions.
  3. Procedural interfaces Sub-system encapsulates a set of procedures to be called by other sub-systems.
  4. Message passing interfaces Sub-systems request services from other sub-systems

Interface errors

1. 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.
2. Interface misunderstanding
   - A calling component embeds assumptions about the behaviour of the called component which are incorrect.
3. Timing errors
   - The called and the calling component operate at different speeds and out-of-date information is accessed.

Interface testing guidelines

1. Design tests so that parameters to a called procedure are at the extreme ends of their ranges.
2. Always test pointer parameters with null pointers.
3. Design tests which cause the component to fail.
4. Use stress testing in message passing systems.
5. In shared memory systems, vary the order in which components are activated.

System testing vs component testing

- System testing during development involves integrating components to create a version of the system and then testing the integrated system.
  1. During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.
  2. Components developed by different team members or sub-teams may be 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.

System testing considerations

- The focus in system testing is testing the interactions between components.
- System testing checks 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.

Use-case testing for system testing

- 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.

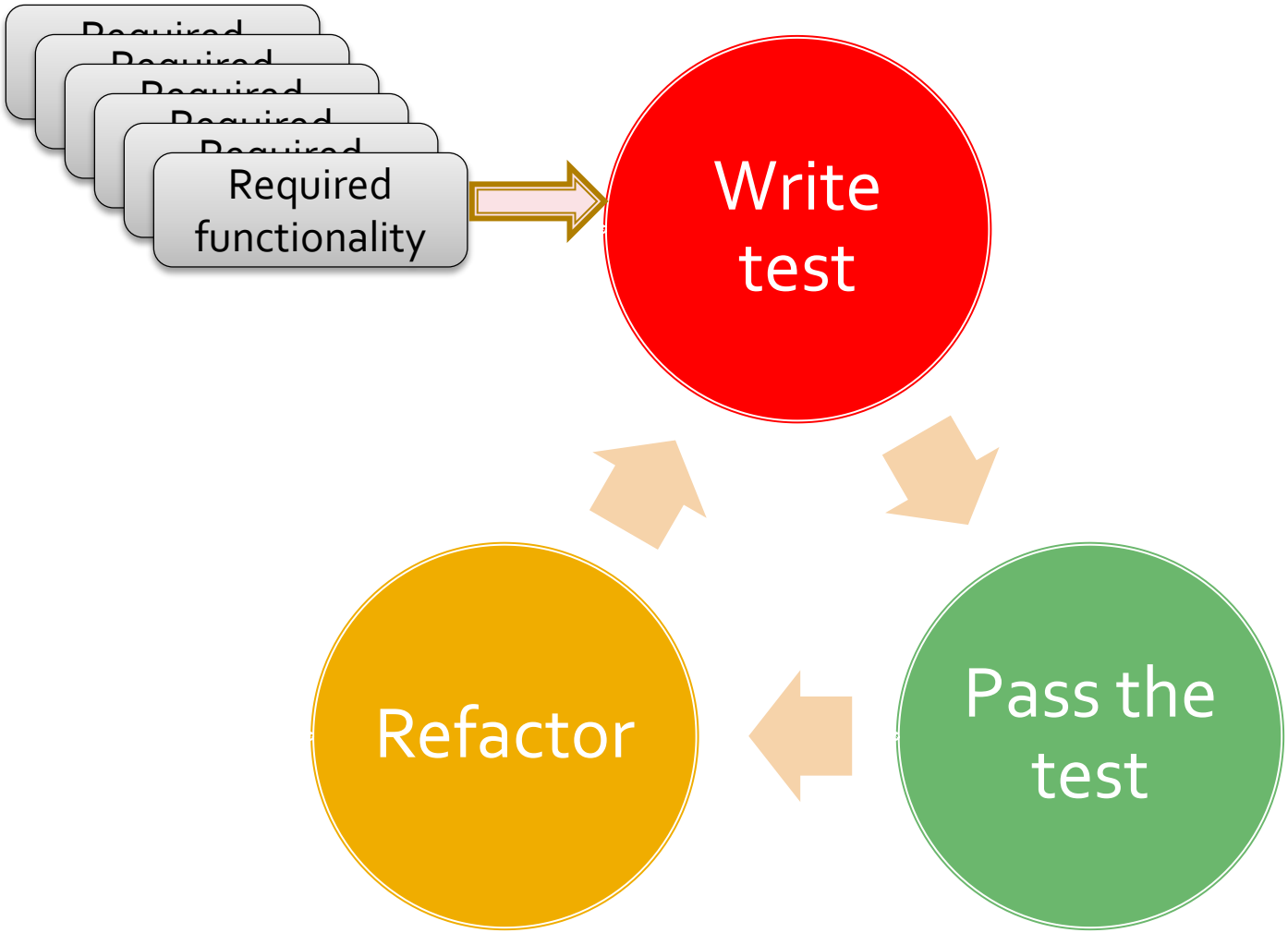
Testing policies

- Exhaustive system testing is impossible so testing policies which define the required system test coverage may be developed.
- Examples of testing policies:
  1. All system functions that are accessed through menus should be tested.
  2. Combinations of functions (e.g. text formatting) that are accessed through the same menu must be tested.
  3. Where user input is provided, all functions must be tested with both correct and incorrect input.

Test-driven development

- Test-driven development (TDD) is an approach to program development in which you inter-leave testing and code development.
- 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 code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Test-driven development process



Benefits of test-driven development

1. Code coverage
   - Every code segment that you write has at least one associated test so all code written has at least one test.
2. Regression testing
   - A regression test suite is developed incrementally as a program is developed.
3. Simplified debugging
   - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
4. System documentation (tests naming is important!)
   - The tests themselves are a form of documentation that describe what the code should be doing. System documentation
5. Focused on API

Regression testing

- Regression testing is testing the system to check that changes have not 'broken' previously working code.
- 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.
- Tests must run 'successfully' before the change is committed.

Stage 2: Release testing

1. Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.
2. The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use.
3. Release testing is usually a black-box testing process where tests (another name for this is 'functional testing')
   - are only derived from the system specification,
   - Testers are not aware of the implementation.

Release testing and system testing

- Release testing is a form of system testing.
- Important differences:
  - 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).

Requirements based testing

- Requirements-based testing involves examining each requirement and developing a test or tests for it.
- MHC-PMS 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.

Requirements 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.

Performance testing

- Part of release testing may involve testing the emergent properties of a system, such as performance and reliability.
- Tests should reflect the profile of use of the system.
- Performance tests usually involve planning a series of tests where the load is steadily increased until the system performance becomes unacceptable.

Stress testing

- Stress testing is a form of performance testing where the system is deliberately overloaded to test its failure behaviour.
  1. It tests the failure behavior of the system.
  2. It stresses the system and may cause defects to come to light that would not normally be discovered.

- Particularly relevant to distributed systems

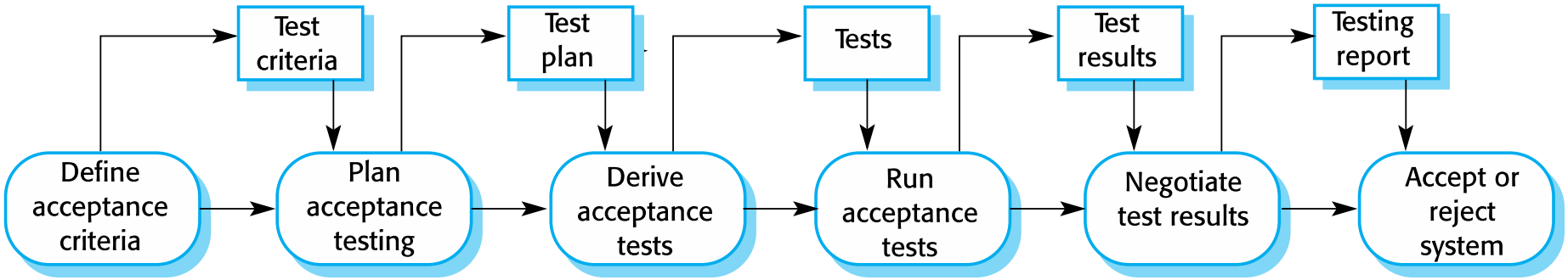
Stage 3: User testing

- User or customer testing is a stage in the testing process 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.
  - The reason for this is that influences from the user's working environment have a major effect on the reliability, performance, usability and robustness of a system. These cannot be replicated in a testing environment.

Types 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 system developers and deployed in the customer environment. Primarily for custom systems.

The acceptance testing process



Agile methods and acceptance testing

- In agile methods, the user/customer is part of the development team
  - 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'
  - can he/she represent the interests of all system stakeholders.

System testing in a production environment

- A/B testing (Online Controlled Experimentation)
  - Observation of users' behawior

- Canary deployment
  - Redirecting a subset of users to the new functionality

Formal software verification

- The formal process of proving the program's compliance with the requirements
  - Formulating requirements in a formal language (logic, propositional calculus)
  - Converting the program to a model (simplified)
  - Model confrontation with requirements
- Sample verification techniques
  - Model checking, Hoare logic, rule inference

Key points

- Testing can only show the presence of errors in a program. It cannot demonstrate that there are no remaining faults.

  Edsger Dijkstra·

- Development testing is the responsibility of the software development team. A separate team should be responsible for testing a system before it is released to customers.

- Development testing includes unit testing, in which you test individual objects and methods component testing in which you test related groups of objects and system testing, in which you test partial or complete systems.

Key points

- When testing software, you should try to 'break' the software by using experience and guidelines to choose types of test case that have been effective in discovering defects in other systems.
- Wherever possible, you should write automated tests. The tests are embedded in a program that can be run every time a change is made to a system.
- Test-driven development is an approach to development where tests are written before the code to be tested.
- Scenario testing involves inventing a typical usage scenario and using this to derive test cases.
- Acceptance testing is a user testing process where the aim is to decide if the software is good enough to be deployed and used in its operational environment.