CITS5502 – Testing and software quality

Unit coordinator: Arran Stewart

Outline

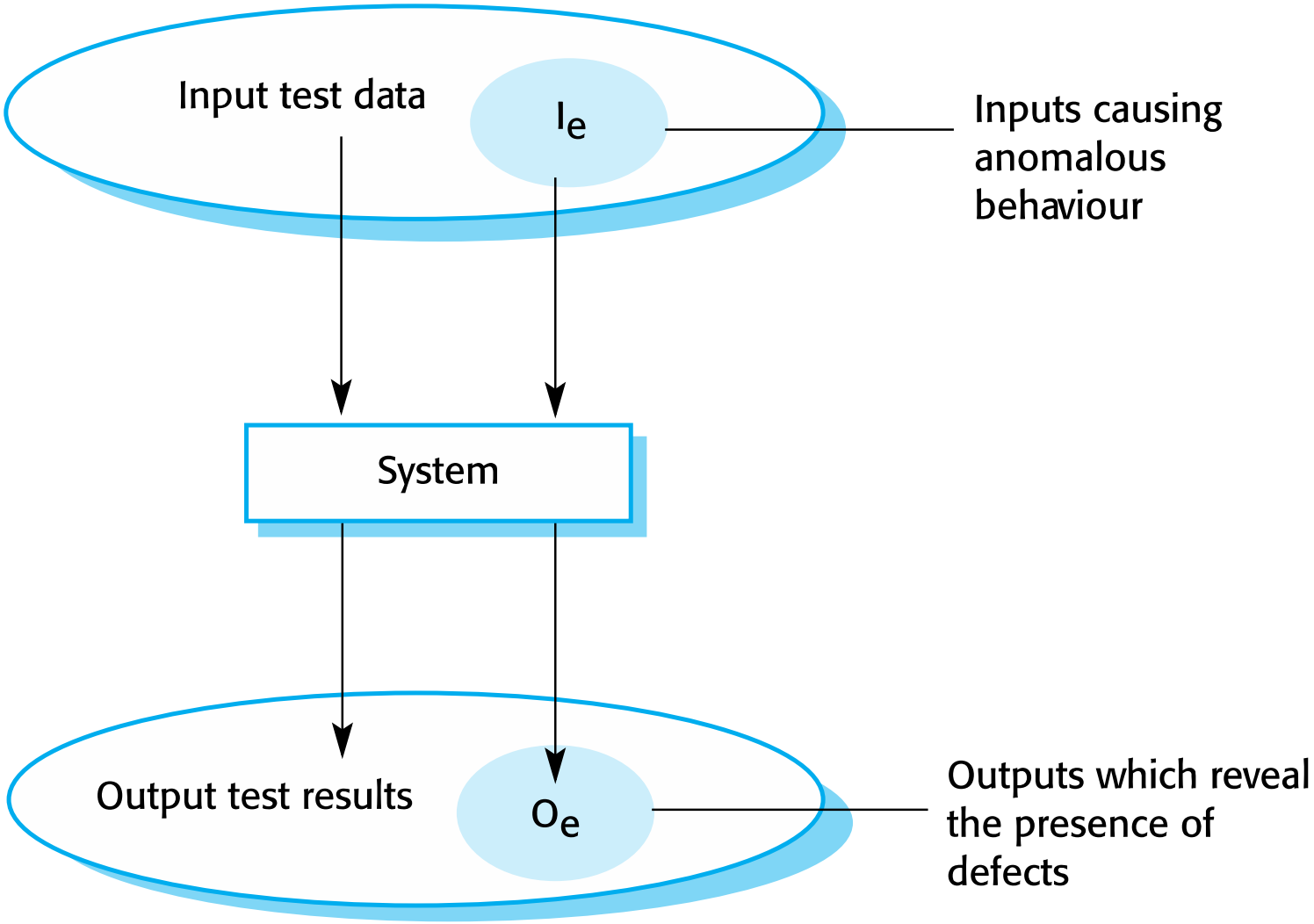
We look at processes and metrics relating to software testing and software quality.

We won't examine testing *techniques* in detail – it's assumed you have some familiarity with unit testing from other languages.

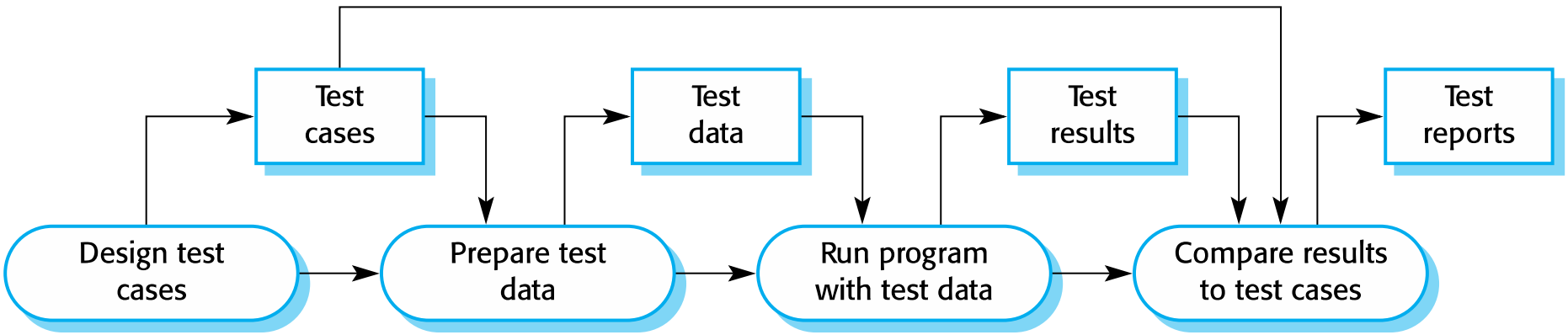
Software testing

- Testing is intended to discover software system defects before it is put into use.
- When testing software, a program (or some part of it) is executed using artifical data.
- The results of the run can be checked to see whether the software exhibits any deviations from its expected behaviour – either a *fault* in its functional attributes, or a deviation from its desired non-functional attributes.
- Testing can only reveal the *presence* of errors, not their *absence*
- Testing is part of the more general process of *verification* and *validation* process, and is typically combined with static validation techniques

A model of testing



Testing process



Typical stages in testing

- Development testing, where the system is tested during development to discover bugs and defects.
- Release testing, where a separate testing team test a complete version of the system before it is released to users.
- User testing, where users or potential users of a system test the system in their own environment.

Development testing

Development testing includes all testing activities that are carried out by the team developing the system.

- Unit testing, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.
- Component testing, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.
- 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.

Verification and validation

As used in software engineering, a definition adapted from project management is typically used for these terms:

- *Validation.* The assurance that a product, service, or system meets the needs of the customer and other identified stakeholders. It often involves acceptance and suitability with external customers.
- *Verification.* The evaluation of whether or not a product, service, or system complies with a regulation, requirement, specification, or imposed condition. It is often an internal process.

(Source: PMBOK Guide, 6th ed)

Verification vs validation

Verification:

- "Are we building the product right?"
- The software should conform to its specification.

Validation

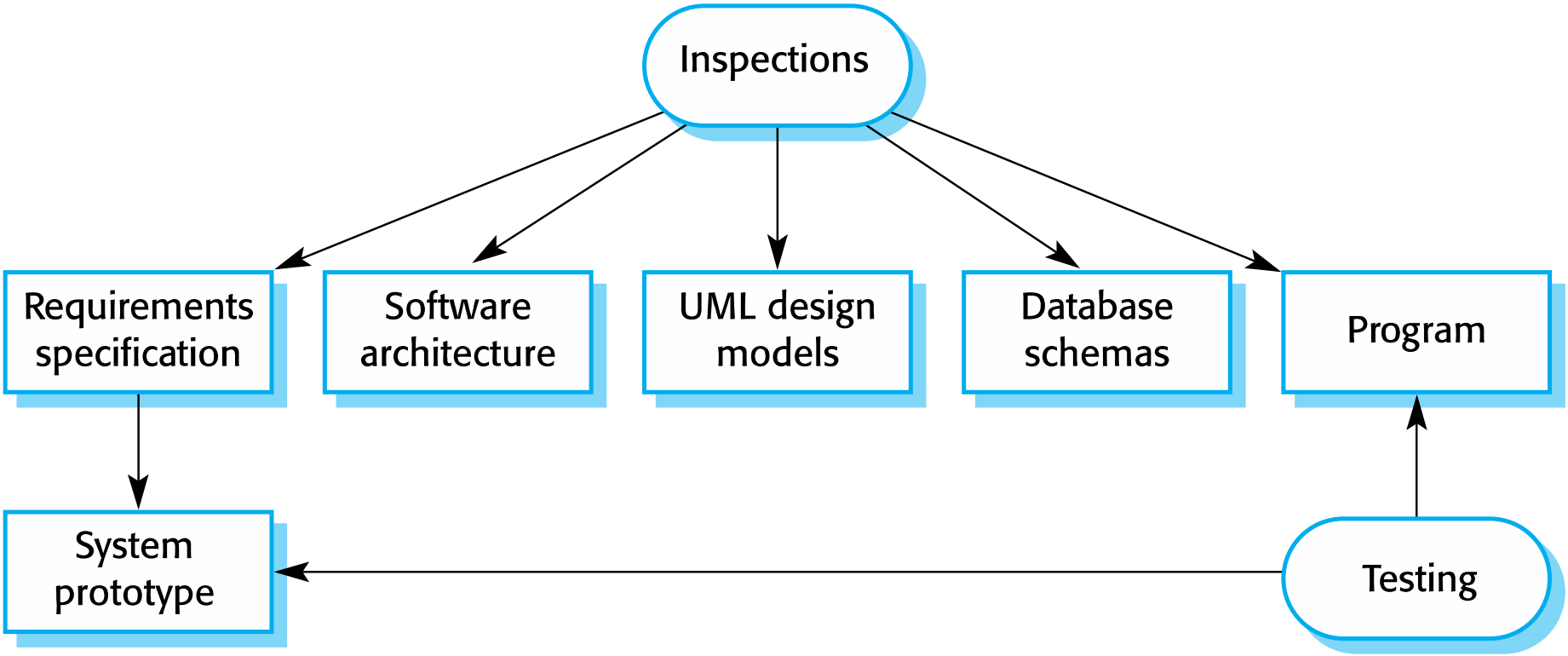
- "Are we building the right product?"
- The software should do what the user really requires.

Inspections and testing

- Software testing investigates the *dynamic* behaviour of the system – what does it do when run?
- Software inspections are concerned with analysis of *static* artifacts of the system (e.g. source code, binaries, documentation, models, etc.)

  Inspections may be manual, automatic, or some combination of the two.

Inspecting artifacts



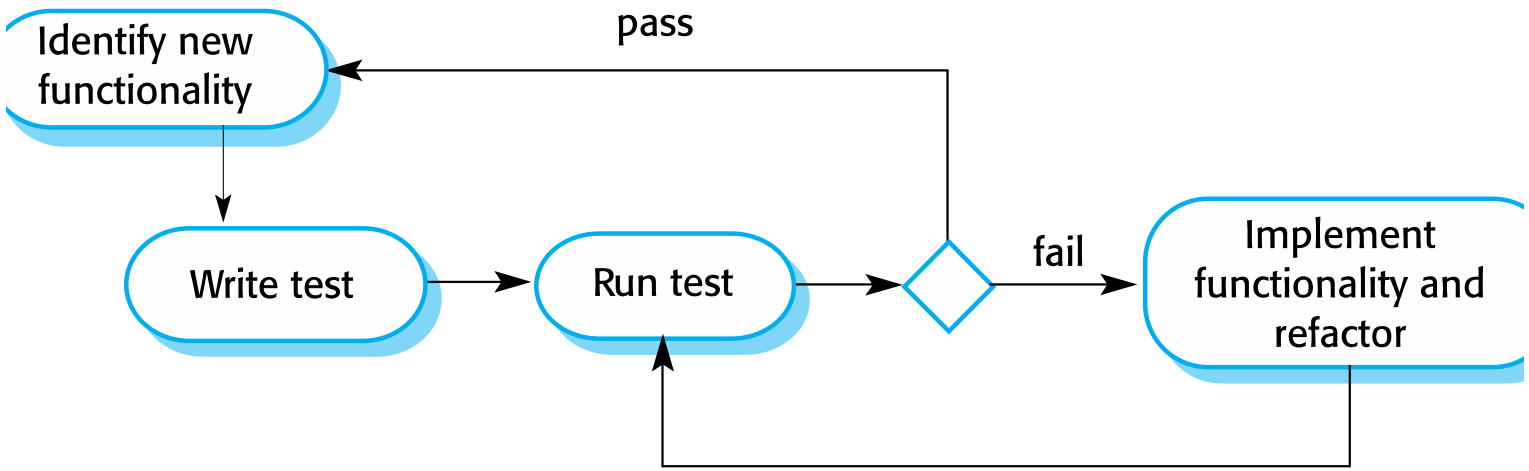
Inspections

- Manual software inspections involve people examining the source representation with the aim of discovering anomalies and defects.
- Inspections do 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.

Test-driven development

- In a strict waterfall model, testing is a stage performed *after* implementation.
- In Test-driven development (TDD), testing and code development are interleaved.
- 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.

TDD process


Identify new functionality
pass
Write test
Run test
fail
Implement functionality and refactor

TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. 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.

Benefits of test-driven development

- 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.
- Simplified debugging
  - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
  - The tests themselves are a form of documentation that describe what the code should be doing.

How much testing is enough?

- For even small programs, *exhaustive testing* (i.e., testing every possible input) is simply not possible.
- Thus, testing can never demonstrate the *absence* of defects.
- That being so, how do we know when we've done enough testing?

How much testing is enough?

- Ideally, we would like there to be no faults in our system.
- In other words, we want it to have high *reliability* (conformance with specification).
- A common metric for reliability is *mean time between failures* (MTBF) (sometimes you may see *mean time to defect*, MTTD).
  - This is the average time the system will run without experiencing a failure.

How much testing is enough?

- The same problem leads to difficulties with measuring the effectiveness of testing processes.
- We would like our *test efficiency* (how effective our tests are – the ratio of defect found to defects present) to be 100%.
- But that would requires us to know how many defects exist but haven't been found – which by definition, we don't know.

Estimating number of defects present

However, there a number of ways the defects in the system and its reliability can be estimated.

We can use data from:

- Industry bodies
- Previous project
- Extrapolating from test history of the current project, and curve fitting
- Reliability models (Poisson processes)

Estimating defects – assumptions

Using data from any of these sources requires us to make assumptions.

For instance, using data from a previous project requires us to assume that the present project is sufficiently similar for it to be a good model.

Using historical data from the current project might require us to assume that (for instance) all testing engineers have similar skill and effectiveness, and/or that this doesn't alter over time.

Other methods – error seeding

The idea:

- Deliberately insert a certain number of bugs into a system
- Ask our software engineers to test the new system
- Measure what proporition are found –
  - e.g. if 20 bugs are inserted and 15 found, our test efficience is 75%

Error seeding assumptions

This too relies on assumptions – e.g. that we are able to deliberately insert bugs of a sort that accurately reflect the bugs that turn up.

Other method – mutation analysis

Mutation analysis (sometimes regarded as an automated form of error seeding) uses a similar idea.

- We modify a program's source code in small ways: e.g. we might change a "<=" sign to "<" in a conditional statement, or alter the value of a constant (string or number) found in the source code
- Each modified version is called a *mutant*.

The assumption here is that these are the sorts of errors a programmer might make in practice.

If we can think of other typical errors, we can create mutation operations for them.

Mutation analysis

We then run our tests on the mutants – we expect that our tests should reject the mutant versions (called "killing the mutants").

If they don't, then something is either very wrong with our tests (they are not detecting mutated code) or with our software (it contains code which either is not being tested, or which seems to have no effect on program behaviour – dead code).

Reliability models

We can also use *models* to estimate the reliability of our software.

A simple model is the *Poisson distribution*

Poisson distribution

- This measures systems where in any given time interval, we know the average time between some event occurring, but when exactly the events do occur is random.
- Example: It might be that the *aveerage rate* at which we receive mail is constant, even though we don't know in advance when any particular message will arrive.

Poisson distribution – assumptions

These distributions make a number of assumptions . . .

- Defects are independent
  - In fact, there are distinct patterns of types of defects
  - A problem with analysis or design can lead to a cluster of many defects
- The size of the system is constant
  - Whereas in fact, systems typically hrow
- Each defect is fixed in the same time interval in which it was discovered
  - In fact, “low priority” defects may be left until “next release”
- No new defects are introduced in the fixing process
  - In large systems with poor regression testing frameworks, the chance of a correct fix may be low.