SYSC-3020 – Introduction to Software Engineering

Software Validation, Verification and Testing

Software Bugs …

- Bug related to the year

  A 104 years old woman received an invitation to a kindergarten (1992).

- Interface misuse

  Underground train in London left a station without the driver (1990).

- Over budget project

  Failure in an automated luggage system in an airport (1995).

- NASA mission to Mars:

  Incorrect conversion from imperial to metric leads to loss of Mars satellite (1999)

- Ariane 5 Flight 501

  The space rocket was destroyed (1996).

- Therac-25

  Radiation therapy and X-ray machine killed several patients (1985-1987).

Software Bugs - Cost

- "Impact of Inadequate Software Testing on US Economy"
  - Who?
    - National Institute of Standards and Technology (NIST), a US federal agency.
  - What?
    - Studies in the manufacturing and transportation equipment sectors, to assess the cost to the U.S. economy of inadequate software testing infrastructure.
  - Results (annual cost):
    - Estimation: **$5.85 billion**
    - Projection to the entire U.S. economy: **$59.5 billion**
  - `http://www.nist.gov/director/prog-ofc/report02-3.pdf`

- Open source software development projects have been shown to lack "attention to basic, accepted, and mature testing techniques"
  - L. Zhao, S. Elbaum, Quality assurance under the open source development model, Journal of Systems and Software 66 (1) (2003) 65–75.

Cost of Testing

- You are going to spend about half of your development budget on testing, whether you want to or not.
- In real world usage, testing is the principle post design activity
- Restricting early testing usually increases costs

Definitions (Verification vs. Validation)

- ***Software Verification****:*
  - The goal is to find as many latent defects as possible before delivery
  - Checking whether the system adheres to properties termed as *verification properties*
  - **Constructing the system well**

- ***Software Validation****:*
  - The goal is to gain confidence in the software, shows it meets its specifications
  - Relationship with other software engineering activities (e.g., Requirements elicitation, Analysis)
  - **Constructing the right system**

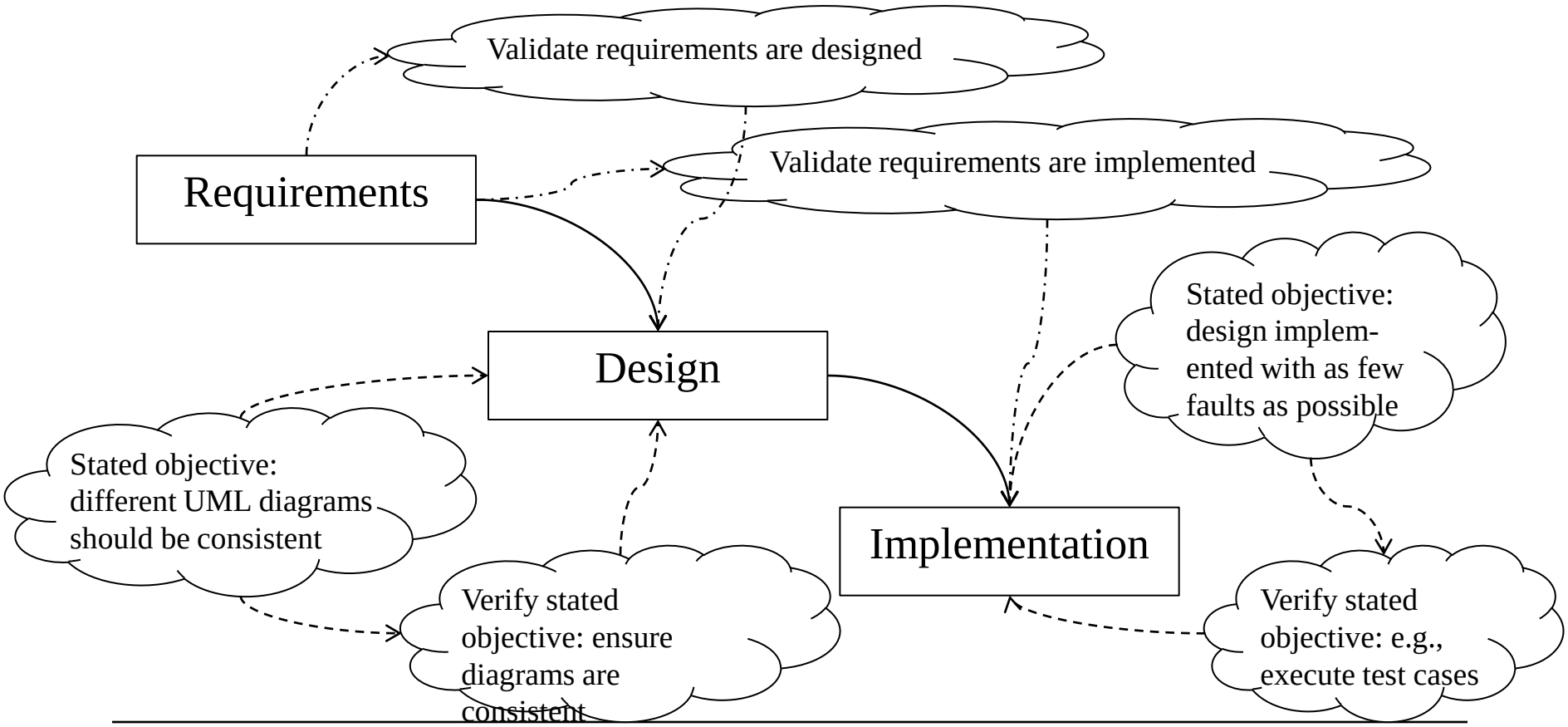
Definitions (Verification vs. Validation)

- ***Software Verification****: IEEE definition (Std 610.12.1990)*
  - *The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.*

- ***Software Validation****: IEEE definition (Std 610.12.1990)*
  - *The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.*

Definitions (Verification vs. Validation)

Consider the waterfall software development model and its three first phases.



Software testing is **one** V&V technique!

Definitions (V&V Techniques)

- Static Techniques: i.e., without any execution of the system
  - Inspections: Techniques aimed at *systematically* verifying non-executable software artifacts with the intent of finding as many defects as possible, as early as possible
  - Mathematical Proof: Proof of the program against its formal specification
  - Model Checking: Verifying properties of the system using models (e.g., finite state machines, petri nets)
- Dynamic Techniques: i.e., through the execution of the system
  - Symbolic Execution: Inputs supplied to the system are symbolic
  - Verification Testing (or simply, Testing): Inputs supplied to the system are valued
    - The most used V&V technique

What should we Do during Software Testing?

- "Program testing can be used to show the presence of bugs, but never to show their absence" [Dijkstra, 1972]

- Even *exhaustive testing*, i.e., testing a software system using all the possible inputs, is most of the time impossible
  - Examples:
    - A program that computes the factorial function (n!=n.(n-1).(n-2)…1)
      - Exhaustive testing = running the program with 0, 1, …, 100, …!
    - A compiler (e.g., javac)
      - Exhaustive testing = compiling every possible (Java) program

What should we Do during Software Testing?

Therefore

- No absolute certainty can be gained from testing
- Testing should be integrated with other verification activities, e.g., inspections
- Main goal: demonstrate the software can be depended upon, i.e., *sufficient dependability*

Therefore, when testing, we need to know when to stop creating and executing tests!

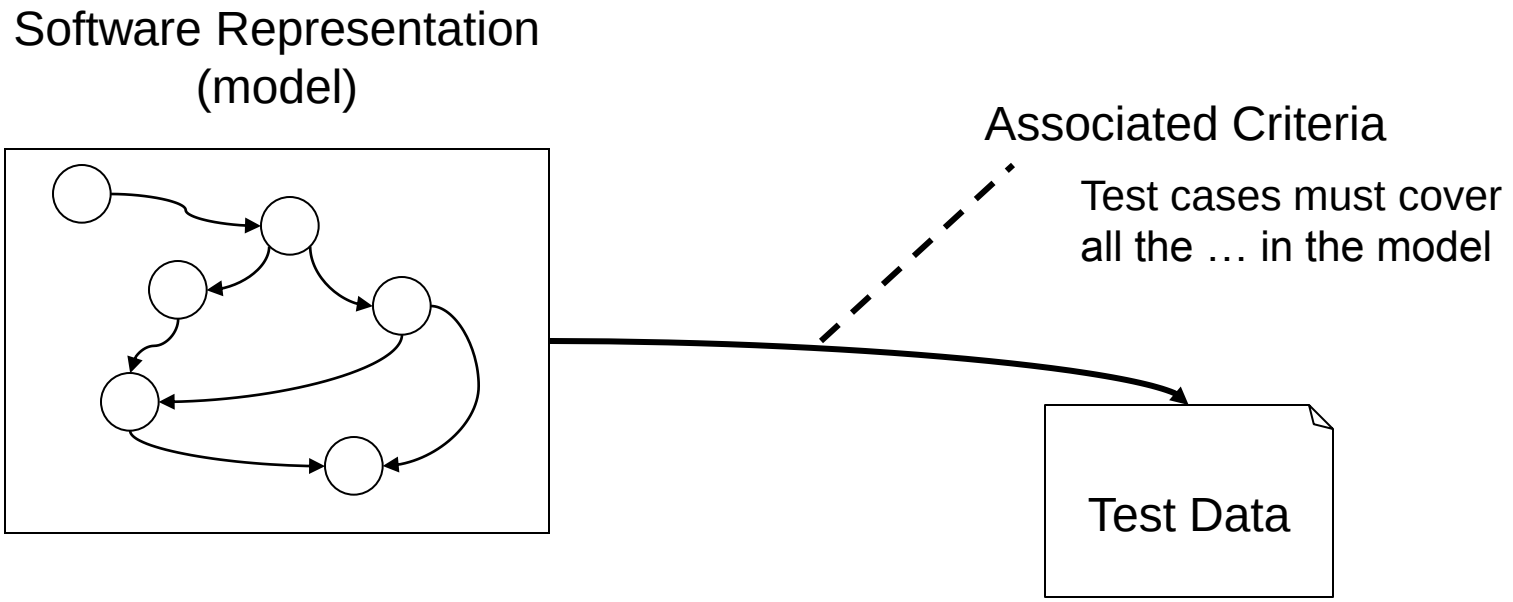
- How can we do that with rationale?

- How do **you** do that?

What should we Do during Software Testing?

- Technique used to reduce the number of inputs (i.e., test cases):
  - Testing criteria group input elements into (equivalence) classes
  - One input in selected in each class (notion of test data coverage)
  - Criteria are used to decide which test inputs to use
  - Criteria are used to decide when to stop testing

Test Data Coverage



Representation of

- the specification ⇒ Black-Box Testing
- the implementation ⇒ White-Box Testing

Black-box vs. White-box Testing

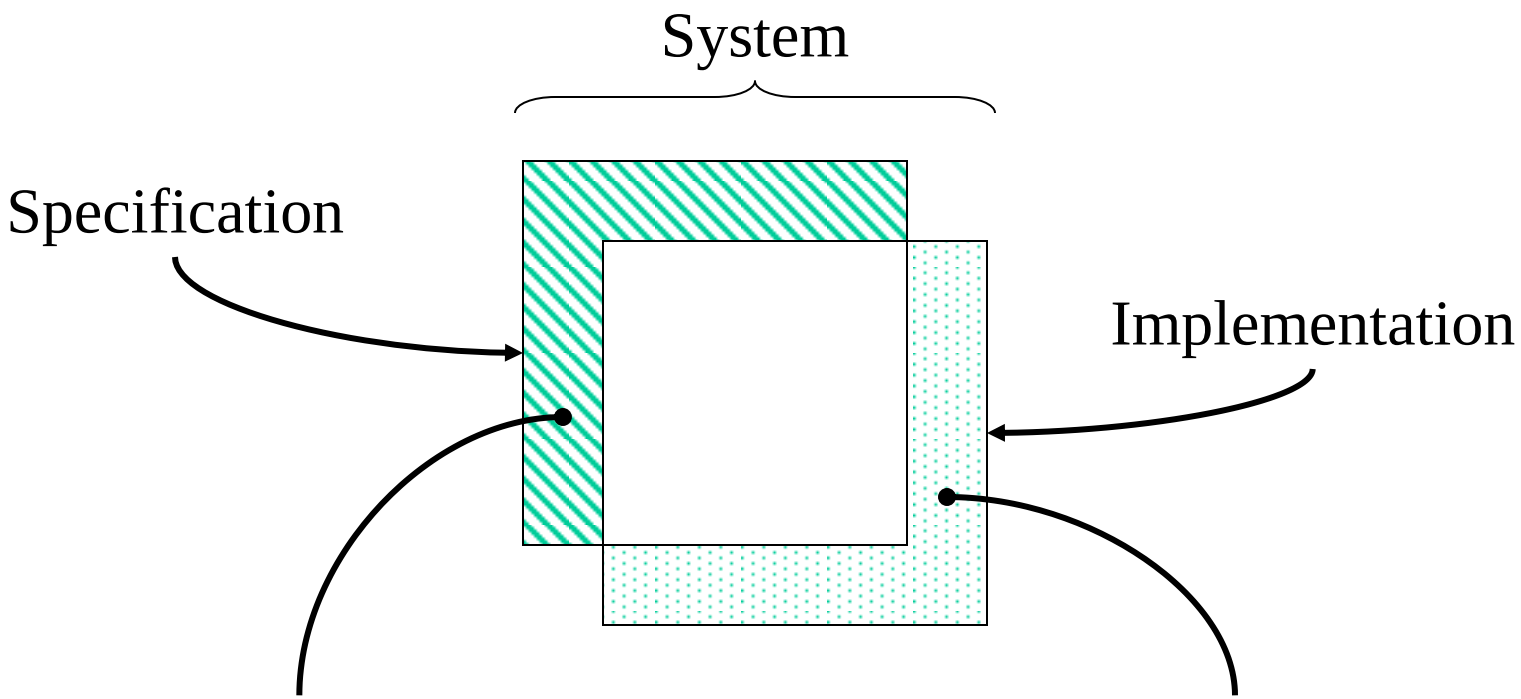
- ☺ Check conformance with the specification
- ☺ It scales up (different techniques at different granularity levels)

- ☹ It depends on the specification and the degree of detail
- ☹ Do not know how much of the system is being tested
- ☹ What if the system performs some unexpected, undesirable task?

- ☺ Based on control and data flow coverage criteria
- ☺ It allows you to be confident about how much of the system is being tested

- ☹ It does not scale up (mostly applicable at unit and integration testing levels)
- ☹ It cannot reveal missing functionalities (part of the specification that is not implemented)

Black-box vs. White-box Testing



Missing functionality:
Cannot be revealed by white-box techniques

Unexpected functionality:
Cannot be revealed by black-box techniques

Adequacy Criterion

- Given a criterion C for a model M
  - The *coverage ratio* of a test set T is the proportion of the elements in M defined by C covered by the test set.
  - A test set T is said to be adequate for C, or simply C-adequate, when the coverage ratio achieves 100% for criterion C.
- Example 1:
  - M is the control flow graph of a function
  - C is all the statements
- Example 2:
  - M is a set of scenarios of execution (e.g., from use cases)
  - C is all the scenarios

Using a Test Adequacy Criterion

	Example
Test model + Test criterion This is the test objective/requirement	I want to cover statements
Test criterion $\Rightarrow$ Test case specifications	Test case 1 will cover statements 1, 2, 3, 10, 11 … Test case 2 will cover statements 1, 2, 3, 10, 15, … …
Test case specification $\Rightarrow$ identifying test data/input	To execute test case 1, I need to execute with input value 10 To execute test case 2, I need to execute with input value 20 …

Marick's Recommandation

Brian Marick recommends the following approach:

1. Generate functional tests from requirements and design to try every function under well-defined conditions (recall the discussion on "when to stop testing).
2. Check the structural coverage after the functional tests are all verified to be successful.
   - How much of the code have we hit? Is this enough?
   - There are numerous tools to tell you how much of the code your tests execute (simply Google "java code coverage")
3. Where the structural coverage is imperfect, generate functional tests (not structural) that induce the additional coverage.

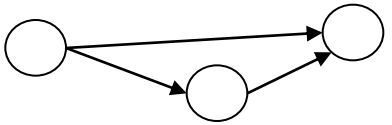
Result?

- Test suite that is adequate w.r.t. a black box criterion (by construction at step 1)
- Test suite that is adequate w.r.t. a white boc criterion (by construction at step 3)

Test Criteria Based on Structure of Test Model [Offutt]

- Graphs

Method body
Methods and calls
Components interactions
State and transitions
Sequence diagrams
…



- Logical Expressions

(not X or not Y) and A and B

Can appear in:
- State machine
- Source code
- Software specification
- …

- Input Domain Characterization

A: {0,1,>1}

B: {600,700,800}

C: {swe,cs,isa,infs}

Describes the input domain of the software under test (method, component, system)

- Syntactic Structures

```
if (x>y)
      z = x - y;
else
      z = 2 * x
```

Based on a grammar, or other syntactic definition
- e.g., mutation testing

Basic Testing Definitions

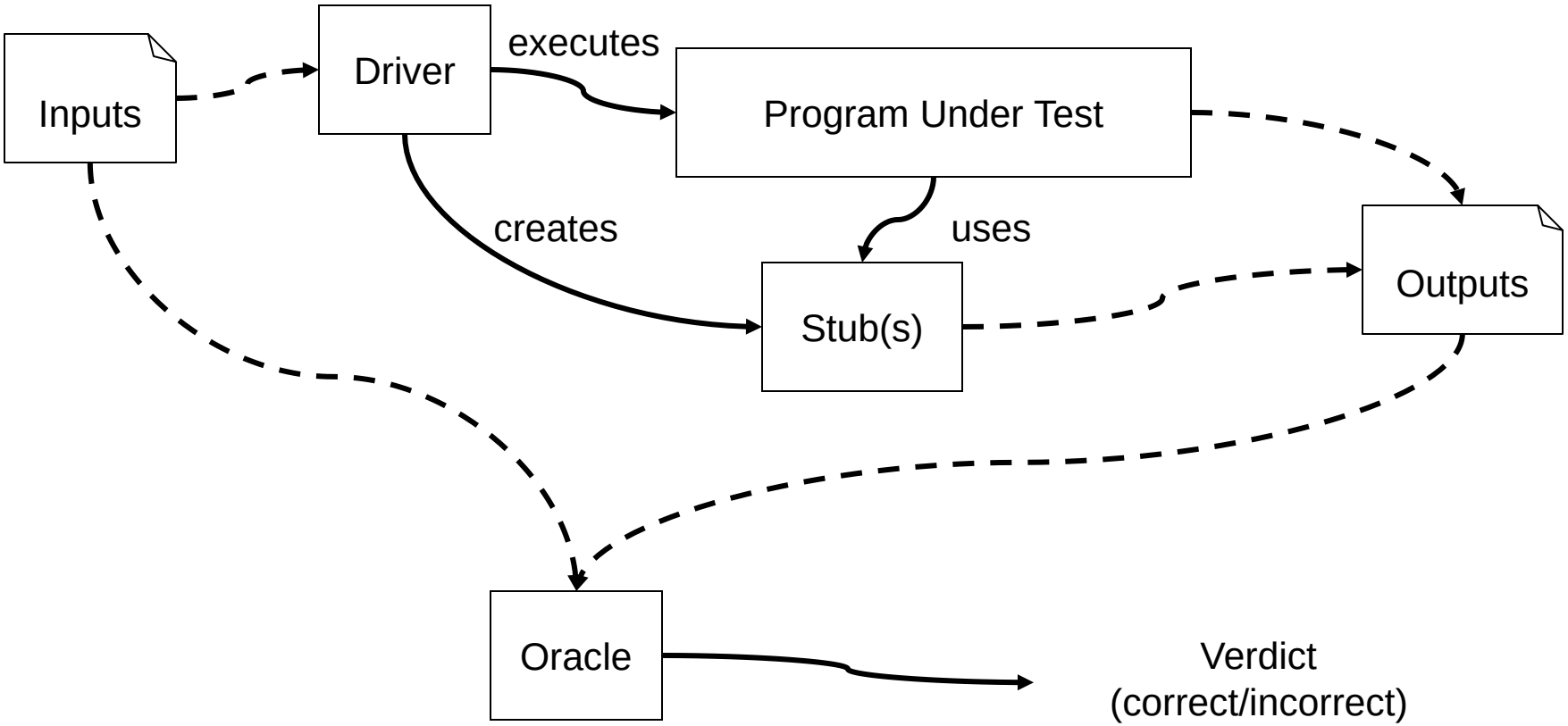
- Fault → Error → Failure

  Three conditions necessary for a failure to be observed

  – Reachability: the location(s) in the program that contain the fault must be reached
  – Infection: the state of the program must be incorrect
  – Propagation: the infected state must propagate to cause some output of the program to be incorrect.

- Test case → Test set
- Test scaffolding: Test stub and Test driver
  – Test stubs and drivers enable components to be isolated from the rest of the system for testing
- Oracle

Summary of Definitions (1)



Observability vs. Controllability

- Software Observability :

  How easy it is to observe the behavior of a program in terms of its outputs, effects on the environment and other hardware and software components

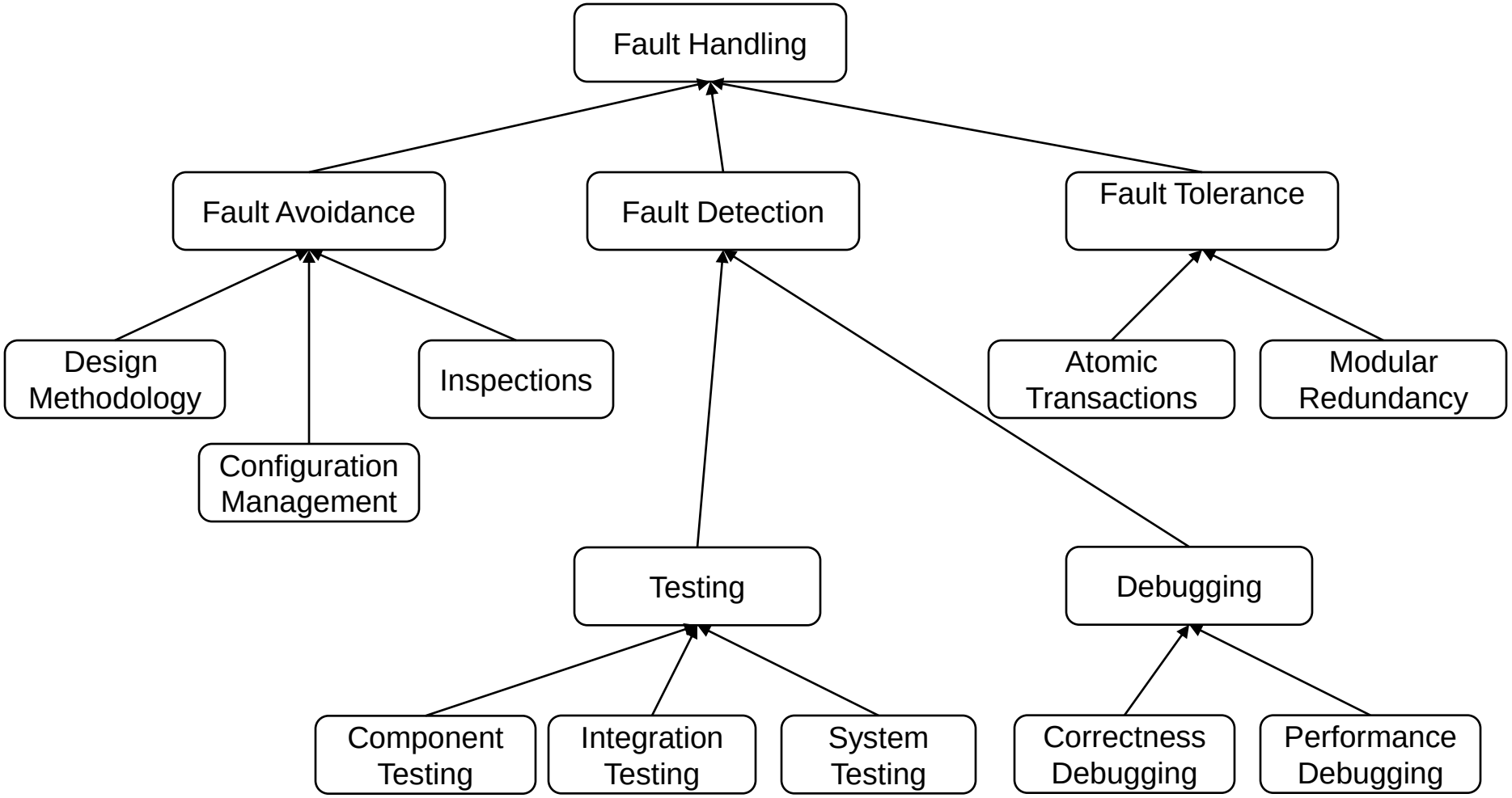
  - Software that affects hardware devices, databases, or remote files have low observability

- Software Controllability :

  How easy it is to provide a program with the needed inputs, in terms of values, operations, and behaviors

  - Easy to control software with inputs from keyboards
  - Inputs from hardware sensors or distributed software is harder
  - Data abstraction reduces controllability and observability

Dealing with Software Faults



Interested in Further Details?

- Title: Introduction to Software Testing
- Authors: Paul Ammann, Jeff Offutt
- Publisher: Cambridge University Press
- ISBN: 978-0-521-88038-1