# Software Testing

Fault Errors and Failures. RIPR Model

# Fault, Error and Failure

- Software Fault: A static defect in the software.
- Software Error: An incorrect internal state that is the manifestation of some fault.
- Software Failure: External, incorrect behavior with respect to the requirements or another description of the expected behavior.

# RIPR model

- A test exposes a fault only if **all four** conditions hold:
  - **Reachability** – the test **executes the faulty (defect) code**.
  - **Infection** – executing that fault **corrupts program state** (a wrong value appears somewhere) (error).
  - **Propagation** – the corrupted state **flows to an observable output/behaviour**.
  - **Revealability** – the test's **oracle checks** that output and can **distinguish** right from wrong.

# RIPR model

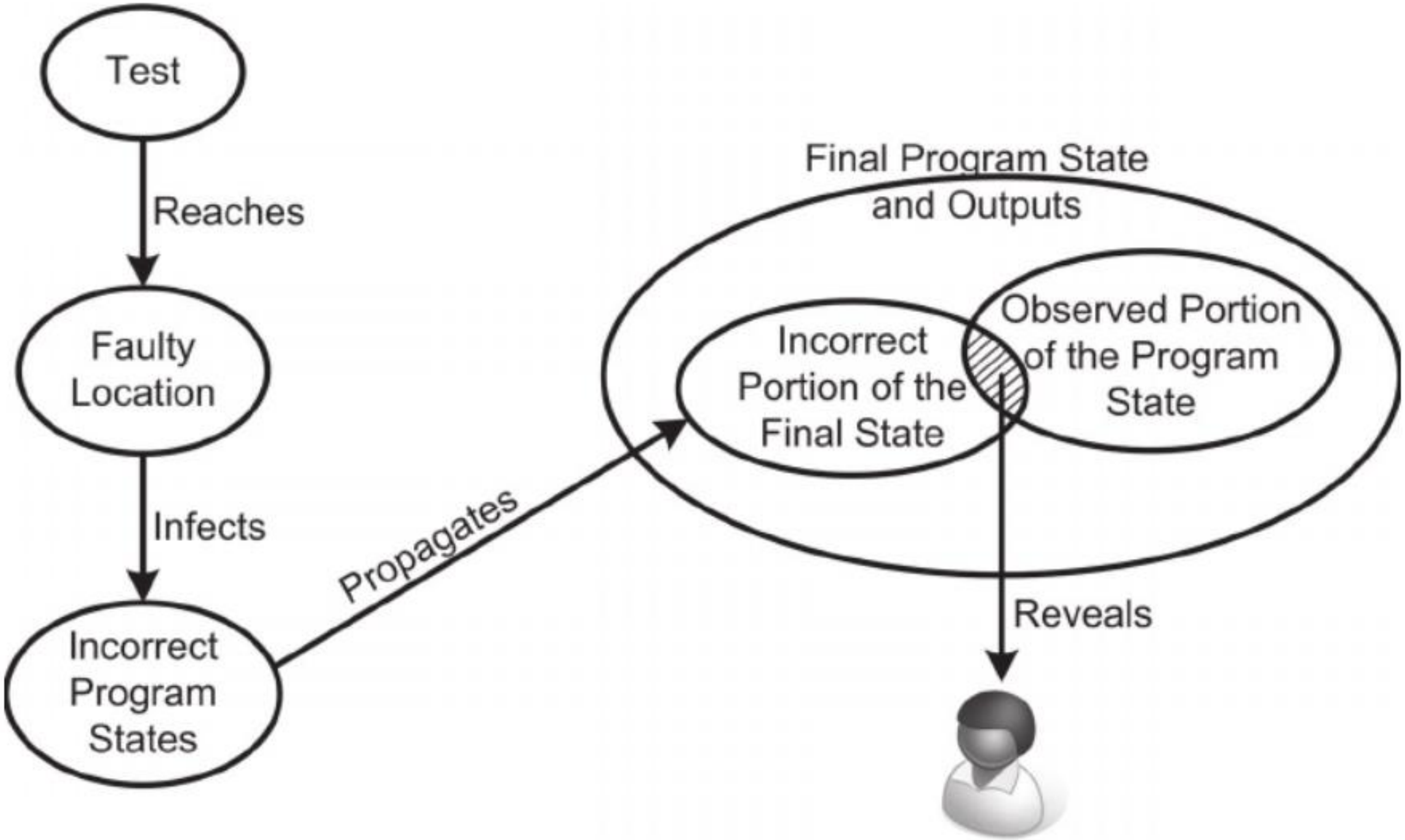



**Figure 2.1.** Reachability, Infection, Propagation, Revealability (RIPR) model.

# RIPR Simple Example

```
int max2(int a, int b) {
  if (a > b) return a;
  else return a;   // BUG: should return b
}
```

- Reachability: Use inputs that go to the else branch, e.g., a=2, b=5.
- Infection: Because of the bug, result becomes 2 (wrong) → state infected.
- Propagation: That wrong value is returned (visible at the interface).
- Revealability: Your test asserts max2(2,5) == 5. If you don't assert (or assert the wrong thing), the failure isn't revealed.

# Defect detection techniques

- methods used in software development to find and identify errors, or "defects," in software before it is released.
- The primary goal is to improve software quality, reduce the cost of fixing issues, and ensure the product meets its requirements.
- These techniques are broadly categorized into **static** and **dynamic** methods.

### Static testing

- Testing without executing the program
  - Software inspection and some forms of analysis
  - Effective at finding certain kinds of problems such as problems that can lead to faults when the program is modified

### Dynamic testing

- Testing by executing the program with real inputs
- **Unit testing, integration testing, system testing, acceptance testing, …**



# Testing Categories

# Defect detection techniques

- Static Techniques
  - **Inspections**: A formal process where a team of developers, designers, and testers systematically reviews the source code. This is a very thorough method for finding defects that might not be apparent during execution.
  - **Walkthroughs**: A less formal review where the developer guides a team through the code, explaining its logic and design. The team provides feedback and identifies potential issues.
  - **Static Analysis**: Automated tools scan the source code for common errors, potential bugs, security vulnerabilities, and violations of coding standards. This is a highly efficient method for early defect detection.

# Defect detection techniques

- Dynamic Techniques
  - **Unit Testing**: The smallest components of the software (e.g., individual functions or methods) are tested in isolation to ensure they work correctly. This is usually done by the developers themselves.
  - **Integration Testing**: This type of testing checks how different software modules interact with each other, uncovering defects that arise when components are combined.
  - **System Testing**: The entire, integrated system is tested to verify that it meets the specified requirements. This includes functional, performance, and security testing.
  - **Acceptance Testing**: This is the final stage of testing where the software is validated against the user's requirements. It's often performed by end-users to ensure the product is ready for release.

# Verification and Validation

- Verification
  - Evaluation of software system that help in determining whether the product of a given development phase satisfy the requirements established before the start of that phase
    - Building the product correctly

- Validation
  - Evaluation of software system that help in determining whether the product meets its intended use
    - Building the correct product

# Validation and Verification (IEEE)

**Validation**

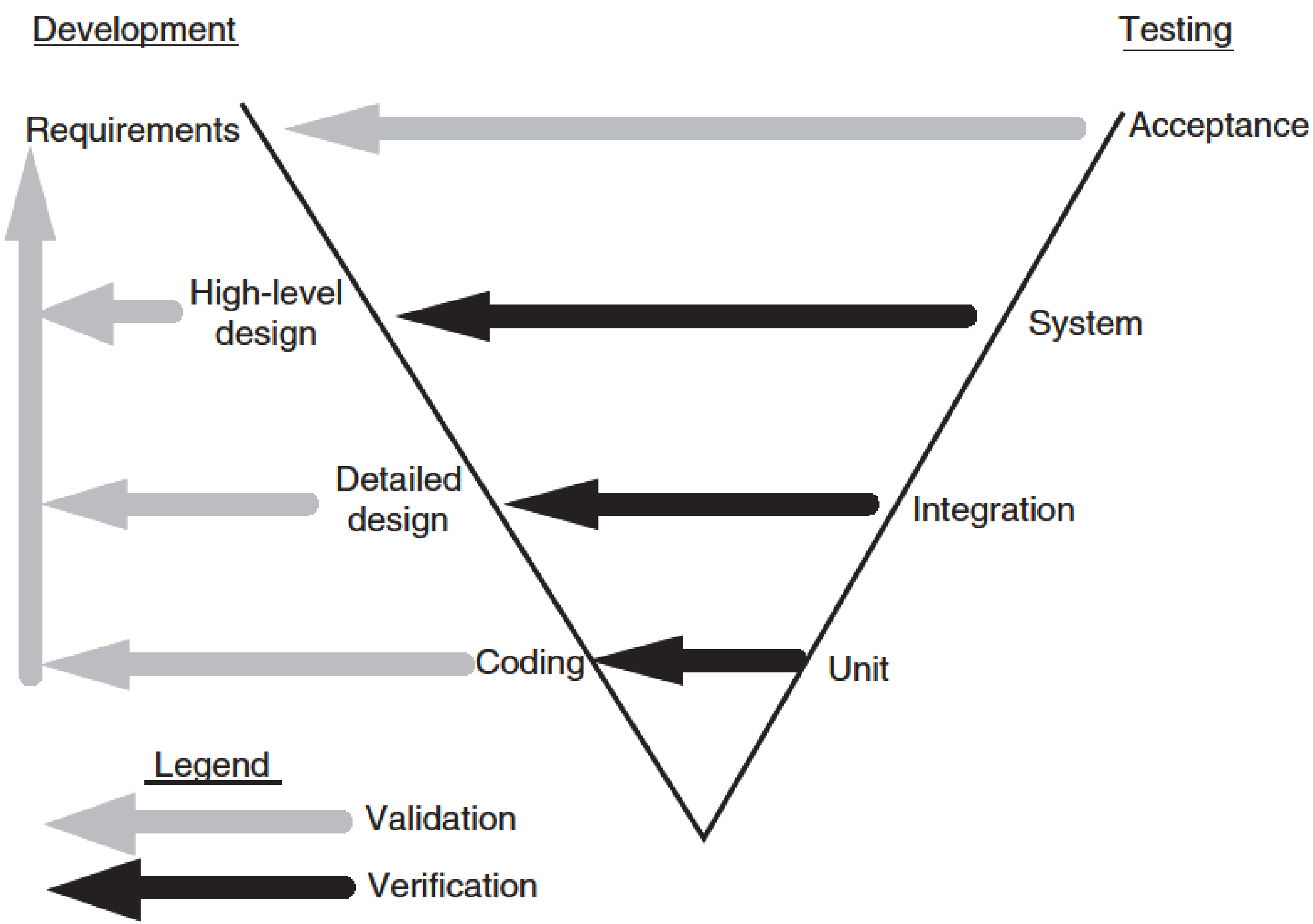
- Evaluate software at the end of software development
- Ensure compliance with the intended usage
- Done by experts in the intended usage of the software, not developers

**Verification**

- Evaluate software at a given phase of the development process

  Fulfill the requirements established during the previous phase
- Requires technical background on the software
- Done by developers at the various stages of development

# Development and Testing in V-model



# Goals based on
# **Test Process Maturity**

Beizer's scale for test process maturity

- Level 0: There is no difference between testing and debugging
- Level 1: The purpose of testing is to show correctness
- Level 2: The purpose of testing is to show that the software does not work
- Level 3: The purpose of testing is not to prove anything specific, but to reduce the risk of using the software
- Level 4: Testing is a mental discipline that helps all IT professionals develop higher quality software

Level 0: Testing is the same as debugging

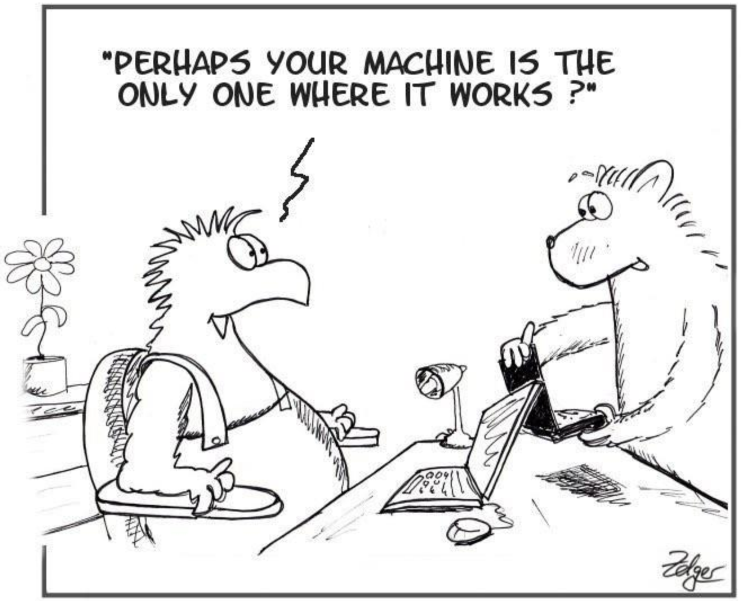

- Adopted by most CS students ☹
  - Get programs to compile
  - Debug with few arbitrarily chosen inputs or those provided by instructors
  - This approach is often ad hoc and unplanned, and testing is performed only by the developer.
  - It's a reactive, unsystematic process.



# Level 0 – Testing is Debugging

# Levels 1 – Software Works

## Level 1: To show correctness (developer-biased view)



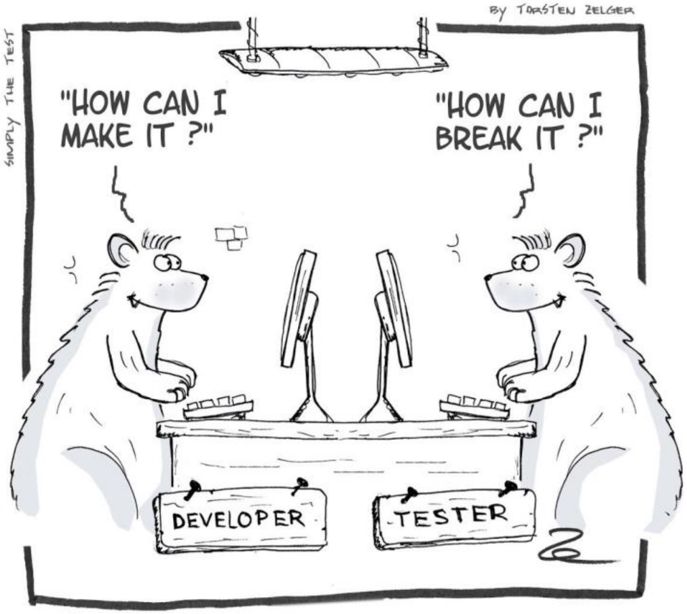

It works on my machine

- Organizations at this level focus on "happy path" testing, where test cases are designed to show that the software meets its specified requirements.
- The mindset is to prove correctness, not to find defects.
- No strict goal, no stopping rule or formal test technique
- No quantitatively way to evaluate; test managers are powerless
- This can create a false sense of security and often misses edge cases and potential failures.

# Levels 2 – Software Doesn't Work

Level 2: To show failure (tester-biased view)

- A negative view puts testers and developers into an adversarial relationship – bad team morale
- What do we know if "no failures"?


They were not so much different, but they had different goals

# "Mature" Testing

Correctness cannot generally be achieved or demonstrated through testing.

Testing can only show the presence of failure, not the absence.

Developers and testers should be on the same boat.

How can we move to a team approach?

# Levels 3 – Risk Reduction

Level 3: To reduce the risk of using the software

- There are risks when using software
  - Some may be small with unimportant consequences
  - Some may be big with important consequences, or even catastrophic
- Testers and developers cooperate to reduce risk

# Levels 4 – Quality Improvement

Level 4: To increase quality of the software

- Testing should be an integral part of the development process
- Testers become technical leaders → measuring and improving software quality

  Help developers improve the ability to produce quality software

  Train developers

- Testers and developers cooperate to improve the quality

- Example: A team practices **Test-Driven Development (TDD)**. Before writing a single line of feature code, the developer writes a test case. This forces them to think about edge cases and clear interfaces early. Because the code is "designed for testability," the final product is naturally higher quality and has fewer bugs from the start.

# How “Mature” is Your Testing?

Are you at level 0, 1, 2, 3, or 4?

We hope to train you to become “change agents” (level 4)

## *Why testing is so hard*

1. Exhaustive testing is impossible
2. We need to know when to stop testing
3. There is no silver bullet in software testing
4. Faults happen in some places more than others
5. Bug-free software does not exist
6. Testing is context-dependent
7. Verification is not validation



# Principles of Software Testing

# Tactical Goals: Each Test

*"If you don't know why you're conducting each test, it won't be very helpful" – Jeff Offutt*

- What is objective and requirement of each test?
- What fact does each test try to verify?
- What are the threshold reliability requirements?
- What are the planned coverage levels?
- How many tests are needed?