

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Back on Proofs

Most of you don't see the point of proofs. I know

- ▶ I even agree (to a given point: we need 2 legs to code – theory and practice)

*Beware of bugs in the above code; I have only proved it correct,  
not tried it.*  
– D.E. Knuth.

- ▶ **Cost/gain ratio**: if you cannot afford to loose, prove it correct  
I hope that Nuclear Power Plants are [partially] proved
- ▶ Can give you a competitive advantage:  
With these, you may accept more complicated contracts
- ▶ You're studying in Nancy, there is a local history of algorithm proofs
- ▶ There will be 1/4 of points on proofs at the exam ...

What's expected at the exam

- ▶ Well, that's similar to when you write code
- ▶ I don't bother a missing } in the code, as long as the idea is here
- ▶ I don't bother a partially wrong proof, as long as the method is here

And now, let's see that tests that you guys prefer are even worst

# Introduction

## Bugs

- ▶ Bugs are inevitable in complex software system
- ▶ A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

[http://www.theregister.co.uk/2009/08/14/critical\\_linux\\_bug/](http://www.theregister.co.uk/2009/08/14/critical_linux_bug/)

# Introduction

## Bugs

- ▶ Bugs are inevitable in complex software system
- ▶ A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

[http://www.theregister.co.uk/2009/08/14/critical\\_linux\\_bug/](http://www.theregister.co.uk/2009/08/14/critical_linux_bug/)

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

<http://www.esecurityplanet.com/features/article.phpr/3860131/>

[Microsoft-Warns-About-17-year-old-Windows-Bug.htm](#)

# Introduction

## Bugs

- ▶ Bugs are inevitable in complex software system
- ▶ A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

[http://www.theregister.co.uk/2009/08/14/critical\\_linux\\_bug/](http://www.theregister.co.uk/2009/08/14/critical_linux_bug/)

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

[http://www.esecurityplanet.com/features/article.phpr/3860131/](http://www.esecurityplanet.com/features/article.phpr/3860131/Microsoft-Warns-About-17-year-old-Windows-Bug.htm)

Microsoft-Warns-About-17-year-old-Windows-Bug.htm

July 2008 25 years old bug found in BSD (seekdir() wrongly implemented)

[http://www.vnode.ch/fixing\\_seekdir](http://www.vnode.ch/fixing_seekdir)

# Introduction

## Bugs

- ▶ Bugs are inevitable in complex software system
- ▶ A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

[http://www.theregister.co.uk/2009/08/14/critical\\_linux\\_bug/](http://www.theregister.co.uk/2009/08/14/critical_linux_bug/)

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

<http://www.esecurityplanet.com/features/article.phpr/3860131/>

Microsoft-Warns-About-17-year-old-Windows-Bug.htm

July 2008 25 years old bug found in BSD (seekdir() wrongly implemented)

[http://www.vnode.ch/fixing\\_seekdir](http://www.vnode.ch/fixing_seekdir)

July 2008 33 years old bug found in Unix (buffer overflow in YACC)

[http://www.computerworld.com/s/article/9108978/Developer\\_fixes\\_33\\_year\\_old\\_Unix\\_bug](http://www.computerworld.com/s/article/9108978/Developer_fixes_33_year_old_Unix_bug)

# Introduction

## Bugs

- ▶ Bugs are inevitable in complex software system
- ▶ A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

[http://www.theregister.co.uk/2009/08/14/critical\\_linux\\_bug/](http://www.theregister.co.uk/2009/08/14/critical_linux_bug/)

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

<http://www.esecurityplanet.com/features/article.phpr/3860131/>

Microsoft-Warns-About-17-year-old-Windows-Bug.htm

July 2008 25 years old bug found in BSD (seekdir() wrongly implemented)

[http://www.vnode.ch/fixing\\_seekdir](http://www.vnode.ch/fixing_seekdir)

July 2008 33 years old bug found in Unix (buffer overflow in YACC)

[http://www.computerworld.com/s/article/9108978/Developer\\_fixes\\_33\\_year\\_old\\_Unix\\_bug](http://www.computerworld.com/s/article/9108978/Developer_fixes_33_year_old_Unix_bug)

## Chasing bugs

- ▶ Once identified, use print statements of IDE's debugger to hunt them down
- ▶ But how to discover all bugs in the system, even those with low visibility?

# Introduction

## Bugs

- ▶ Bugs are inevitable in complex software system
- ▶ A bug can be very visible or can hide in your code until a much later date

## Bugs can hide very well

Aug 2009 8 years old bug found in Linux (handling not implemented kernel fctions)

[http://www.theregister.co.uk/2009/08/14/critical\\_linux\\_bug/](http://www.theregister.co.uk/2009/08/14/critical_linux_bug/)

Jan 2010 Microsoft fixes a 17 years old bug (in code allowing NT to run 16bits apps)

<http://www.esecurityplanet.com/features/article.phpr/3860131/>

Microsoft-Warns-About-17-year-old-Windows-Bug.htm

July 2008 25 years old bug found in BSD (seekdir() wrongly implemented)

[http://www.vnode.ch/fixing\\_seekdir](http://www.vnode.ch/fixing_seekdir)

July 2008 33 years old bug found in Unix (buffer overflow in YACC)

[http://www.computerworld.com/s/article/9108978/Developer\\_fixes\\_33\\_year\\_old\\_Unix\\_bug](http://www.computerworld.com/s/article/9108978/Developer_fixes_33_year_old_Unix_bug)

## Chasing bugs

- ▶ Once identified, use print statements of IDE's debugger to hunt them down
  - ▶ But how to discover all bugs in the system, even those with low visibility?
- ⇒ Testing and Quality Assurance practices



# Why to test?

*Testing can only prove the presence of bugs, not their absence.*  
– E. W. Dijkstra

## Perfect Excuse

- ▶ Don't invest in testing: system will contain defects anyway

## Counter Arguments

- ▶ The more you test, the less likely such defects will *cause harm*
- ▶ The more you test, the more *confidence* you will have in the system

# Who should Test?

Fact: Programmers are not necessarily the best testers

- ▶ Programming is a constructive activity: try to make things work
- ▶ Testing is a destructive activity: try to make things fail

## In practice

- ▶ **Best case:** Testing is part of quality assurance
  - ▶ done by developers when finishing a component (unit tests)
  - ▶ done by a specialized test team when finishing a subsystem (integration tests)
- ▶ **Common case:** done by rookies
  - ▶ testing seen as a beginner's job, assigned to least experienced team members
  - ▶ testing often done after completion (if at all)
  - ▶ but very difficult task; impossible to completely test a system
- ▶ **Worst case** (unfortunately very common too): no one does it
  - ▶ Not productive  $\Rightarrow$  not done [yet], postponed “by a while”
  - ▶ But without testing, productivity decreases, so less time, so less tests

*Debugging is twice as hard as writing the code in the first place.  
Therefore, if you write the code as cleverly as possible, you are, by  
definition, not smart enough to debug it.* – Kernighan

# What is “Correct” ?

different meanings depending on the context

## Correctness

- ▶ A system is correct if it behaves according to its specification
- ⇒ An absolute property (i.e., a system cannot be "almost correct")
- ⇒ ... undecidable in theory and practice

## Reliability

- ▶ The user may rely on the system behaving properly
- ▶ Probability that the system will operate as expected over a specified interval
- ⇒ Relative property (system mean time between failure (MTTF): 3 weeks)

## Robustness

- ▶ System behaves reasonably even in circumstances that were not specified
- ⇒ Vague property (specifying abnormal circumstances  $\leadsto$  part of the requirements)

# Terminology

## Avoid the term "Bug"

- ▶ Implies that mistakes somehow creep into the software from the outside
- ▶ Imprecise because mixes various "mistakes"

## Error: incorrect software behavior

- ▶ A deviation between the specification and the running system
- ▶ A manifestation of a defect during system execution
- ▶ Inability to perform required function within specified limits
- ▶ *Example*: message box text said "Welcome null."
- ▶ **Transient error**: only with certain inputs; **Permanent error**: for any input

## Fault: cause of error

- ▶ Design or coding mistake that may cause abnormal behavior
- ▶ *Example*: account name field is not set properly.
- ▶ A fault is not an error, but it can lead to them

## Failure: particular instance of a general error, caused by a fault

# Quality Control Techniques

Large systems bound to have faults. How to deal with that?

**Fault Avoidance:** Prevent errors by finding faults before the release

- ▶ **Development methodologies:**  
Use requirements and design to minimize introduction of faults  
Get clear requirements; Minimize coupling
- ▶ **Configuration management:** don't allow changes to subsystem interfaces
- ▶ **[Formal] Verification:** find faults in system execution  
Maturity issue; Assumes requirements, pre/postconditions are correct & adequate
- ▶ **Review:** manual inspection of system by team members  
shown effective at finding errors

**Fault Detection:** Find existing faults without recovering from the errors

- ▶ **Manual tests:** Use debugger to move through steps to reach erroneous state
- ▶ **Automatic Testing:** tries to expose errors in planned way

**Fault Tolerance:** When system can recover from failure by itself

- ▶ Recovery from failure (*example:* DB rollbacks, FS logs)
- ▶ Sub-system redundancy (*example:* disk RAID-1)

# Quality Control Techniques

Large systems bound to have faults. How to deal with that?

**Fault Avoidance:** Prevent errors by finding faults before the release

- ▶ **Development methodologies:**  
Use requirements and design to minimize introduction of faults  
Get clear requirements; Minimize coupling
- ▶ **Configuration management:** don't allow changes to subsystem interfaces
- ▶ **[Formal] Verification:** find faults in system execution  
Maturity issue; Assumes requirements, pre/postconditions are correct & adequate
- ▶ **Review:** manual inspection of system by team members  
shown effective at finding errors

**Fault Detection:** Find existing faults without recovering from the errors

- ▶ **Manual tests:** Use debugger to move through steps to reach erroneous state
- ▶ **Automatic Testing:** tries to expose errors in planned way ← **We are here**

**Fault Tolerance:** When system can recover from failure by itself

- ▶ Recovery from failure (*example:* DB rollbacks, FS logs)
- ▶ Sub-system redundancy (*example:* disk RAID-1)

# Testing Concepts

## Recapping generic terms

- ▶ **Error**: Incorrect software behavior
- ▶ **Fault**: Cause of the error (programming, design, etc)
- ▶ **Failure**: Particular instance of a general error, caused by a fault

## Component

- ▶ A part of the system that can be isolated for testing
- ⇒ an object, a group of objects, one or more subsystems

## Test Case

- ▶ {inputs; expected results} set exercising component to cause failures
- ▶ Boolean method: whether component's answer matches expected results
- ▶ "expected results" includes exceptions, error codes ...

## Test Stub

- ▶ Partial implementation of components on which the tested compnt depends
- ▶ dummy code providing necessary input values and behavior to run test cases

## Test Driver

- ▶ Partial implementation of a component that depends on the tested part
- ▶ a "main()" function that executes a number of test cases

# Testing Concepts

## Recapping generic terms

- ▶ **Error**: Incorrect software behavior
- ▶ **Fault**: Cause of the error (programming, design, etc)
- ▶ **Failure**: Particular instance of a general error, caused by a fault

## Component

- ▶ A part of the system that can be isolated for testing (through *stub* and *driver*)
- ⇒ an object, a group of objects, one or more subsystems

## Test Case

- ▶ {inputs; expected results} set exercising component to cause failures
- ▶ Boolean method: whether component's answer matches expected results
- ▶ "expected results" includes exceptions, error codes ...

## Test Stub

- ▶ Partial implementation of components on which the tested compnt depends
- ▶ dummy code providing necessary input values and behavior to run test cases

## Test Driver

- ▶ Partial implementation of a component that depends on the tested part
- ▶ a "main()" function that executes a number of test cases



# Tests Campaign Planing

## Goal

- ▶ Should *verify* the requirements (are we building the product right?)
- ▶ NOT *validate* the requirements (are we building the right product?)

## Definitions

- ▶ **Testing**: activity of executing a program with the intent of finding a defect  
⇒ A successful test is one that finds defects!
- ▶ **Testing Techniques**: Techniques to find yet undiscovered mistakes  
⇒ **Criterion**: Coverage of the system
- ▶ **Testing Strategies**: Plans telling *when* to perform *what* testing technique  
⇒ **Criterion**: Confidence that you can safely proceed with the next activity

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# White Box Testing

Focuses on internal states of objects

Use internal knowledge of the component to craft input data

- ▶ *Example:* internal data structure = array of size 256  
⇒ test for size = 255 and 257 (near boundary)
- ▶ Internal structure include design specs (like diagram sequence)
- ▶ Derive test cases to maximize structure coverage, yet minimize # of test cases

Coverage criteria: Path testing

- ▶ every statement at least once
- ▶ all portions of control flow (= branches) at least once
- ▶ all possible values of compound conditions at least once (condition coverage)
  
- ▶ all portions of data flow at least once
- ▶ all loops, iterated at least 0, once, and N times (loop testing)

# White Box Testing

Focuses on internal states of objects

Use internal knowledge of the component to craft input data

- ▶ *Example:* internal data structure = array of size 256  
⇒ test for size = 255 and 257 (near boundary)
- ▶ Internal structure include design specs (like diagram sequence)
- ▶ Derive test cases to maximize structure coverage, yet minimize # of test cases

Coverage criteria: Path testing

- ▶ every statement at least once
- ▶ all portions of control flow (= branches) at least once
- ▶ all possible values of compound conditions at least once (condition coverage)  
Multiple condition coverage  $\leadsto$  all true/false combinations for all simple conditions  
Domain testing  $\leadsto$   $\{a < b; a == b; a > b\}$
- ▶ all portions of data flow at least once
- ▶ all loops, iterated at least 0, once, and N times (loop testing)

# White Box Testing

Focuses on internal states of objects

Use internal knowledge of the component to craft input data

- ▶ *Example:* internal data structure = array of size 256  
⇒ test for size = 255 and 257 (near boundary)
- ▶ Internal structure include design specs (like diagram sequence)
- ▶ Derive test cases to maximize structure coverage, yet minimize # of test cases

Coverage criteria: Path testing

- ▶ every statement at least once
- ▶ all portions of control flow (= branches) at least once
- ▶ all possible values of compound conditions at least once (condition coverage)  
Multiple condition coverage  $\leadsto$  all true/false combinations for all simple conditions  
Domain testing  $\leadsto$   $\{a < b; a == b; a > b\}$
- ▶ all portions of data flow at least once
- ▶ all loops, iterated at least 0, once, and N times (loop testing)

Main issue: white box testing negates object encapsulation

# Black Box Testing

Component  $\equiv$  "black box"

## Test cases derived from external specification

- ▶ Behavior only determined by studying inputs and outputs
- ▶ Derive tests to maximize coverage of spec elements yet minimizing # of tests

## Coverage criteria

- ▶ All exceptions
- ▶ All data ranges (incl. invalid input) generating different classes of output
- ▶ All boundary values

## Equivalence Partitioning

- ▶ For each input value, divide value domain in classes of equivalences:
  - ▶ Expects value within  $[0, 12] \rightsquigarrow$  negative value, within range, above range
  - ▶ Expects fixed value  $\rightsquigarrow$  below that value, expected, above
  - ▶ Expects value boolean  $\rightsquigarrow$  {true, false}
- ▶ Pick a value in each equivalence class (randomly or at boundary)
- ▶ Predict output, derive test case

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Testing Strategies

## Unit testing

- ~> Looks for errors in objects or subsystems

## Integration testing

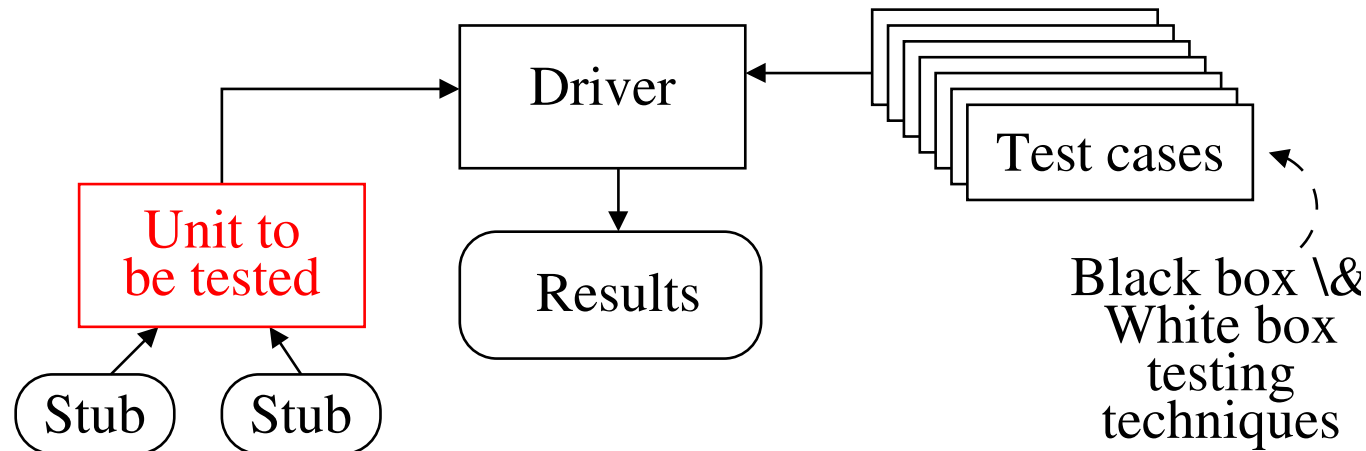
- ~> Find errors with connecting subsystems together
- ▶ **System structure testing:** integration testing all parts of system together

## System testing

- ~> Test entire system behavior as a whole, wrt use cases and requirements
- ▶ **functional testing:** test whether system meets requirements
- ▶ **performance testing:** nonfunctional requirements, design goals
- ▶ **acceptance testing:** done by client



# Unit Testing



## Why?

- ▶ Locate small errors (= within a unit) fast

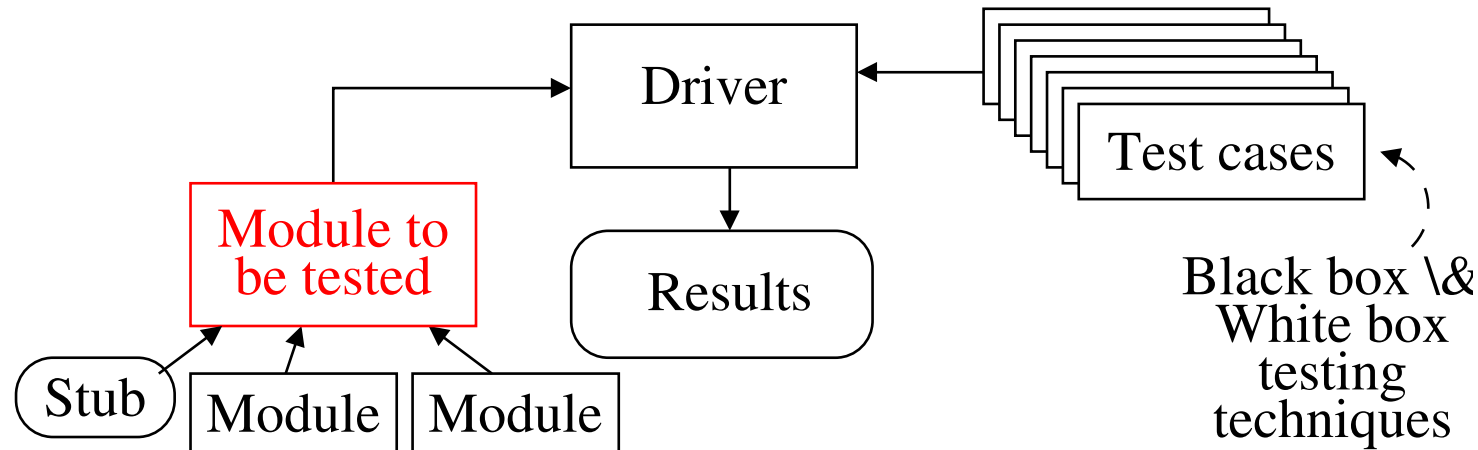
## Who?

- ▶ Person developing the unit writes the tests

## When?

- ▶ At the latest when a unit is delivered to the rest of the team
  - ▶ No test  $\Rightarrow$  no unit
- ▶ Write the test first, i.e. before writing the unit
  - $\Rightarrow$  help to design the interface right

# Integration Testing



## Why?

- ▶ Sum is more than parts, interface may contain faults too

## Who?

- ▶ Person developing the module writes the tests

## When?

- ▶ **Top-down:** main module before constituting modules
- ▶ **Bottom-up:** constituting modules before main module
- ▶ **In practice:** a bit of both

**Remark:** Distinction between unit testing and integration testing not that sharp

# Regression Testing

Ensure that things that used to work still work after changes

## Regression test

- ▶ Re-execution of tests to ensure that changes have no unintended side effects
- ▶ Tests must avoid regression (= degradation of results)
- ▶ Regression tests must be repeated *often*  
(after every change, every night, with each new unit, with each fix,...)
- ▶ Regression tests may be conducted manually
  - ▶ Execution of crucial scenarios with verification of results
  - ▶ Manual test process is slow and cumbersome  
⇒ preferably completely automated

## Advantages

- ▶ Helps during iterative and incremental development + during maintenance

## Disadvantage

- ▶ Up front investment in maintainability is difficult to sell to the customer
- ▶ Takes a lot of work: more test code than production code

# Acceptance Testing

## Acceptance Tests

- ▶ conducted by the end-user (representatives)
- ▶ check whether requirements are correctly implemented  
borderline between verification ("Are we building the system right?")  
and validation ("Are we building the right system?")

## Alpha- & Beta Tests

- ▶ Acceptance tests for "off-the-shelves" software (many unidentified users)
- ▶ Alpha Testing
  - ▶ end-users are invited at the developer's site
  - ▶ testing is done in a controlled environment
- ▶ Beta Testing
  - ▶ software is released to selected customers
  - ▶ testing is done in "real world" setting, without developers present

# Other Testing Strategies

## Recovery Testing

- ▶ Forces system to fail and checks whether it recovers properly
- ~> For fault tolerant systems

## Stress Testing (Overload Testing): Tests extreme conditions

- ▶ e.g., supply input data twice as fast and check whether system fails

## Performance Testing: Tests run-time performance of system

- ▶ e.g., time consumption, memory consumption
- ▶ first do it, then do it right, then do it fast

## Back-to-Back Testing

- ▶ Compare test results from two different versions of the system
- ~> requires N-version programming or prototypes
- git version control system does so to isolate regressions (`bisect` command)

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Tool support

## Test Harness

- ▶ Framework merging all test code in environment
- ▶ Main example for Python is called **unittest**
- ▶ Others exist for almost all languages CppUnit, JUnit, ...

## Test Verifiers

- ▶ Measure test coverage for a set of test cases
- ▶ coverage for Python, JCov for Java, gcov for gcc, ...

## Test Data Generators

- ▶ Assist in selecting test data
- ▶ testdata is a Python testdata generator

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion



# Introduction

## What is pytest?

- ▶ It is a unit testing framework for Python.
- ▶ It provides tools for easy implementation of unit test plans
- ▶ It eases execution of tests
- ▶ It provides reports of test executions

## What pytest is NOT?

- ▶ It cannot design your test plan
- ▶ It does only what you tell it to
- ▶ It does not fix bugs for you

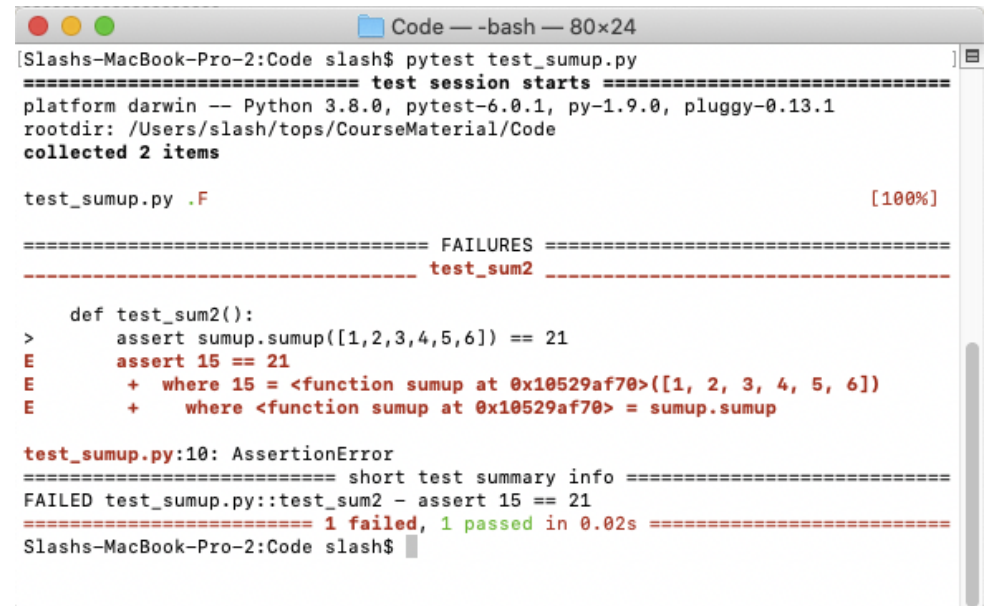
## pytest

- ▶ `pytest.org`
- ▶ Documentation `docs.pytest.org`

# Structure of pytest tests

## Running a test suite consists of

- ▶ Setting up test environment
- ▶ For each test
  - ▶ Setting test up
  - ▶ Invoking test function
  - ▶ Tearing test down
- ▶ Tearing down everything
- ▶ Report result



```
Slashes-MacBook-Pro-2:Code slash$ pytest test_sumup.py
===== test session starts =====
platform darwin -- Python 3.8.0, pytest-6.0.1, py-1.9.0, pluggy-0.13.1
rootdir: /Users/slash/tops/CourseMaterial/Code
collected 2 items

test_sumup.py .F [100%]

===== FAILURES =====
test_sum2

    def test_sum2():
>     assert sumup.sumup([1,2,3,4,5,6]) == 21
E       assert 15 == 21
E       + where 15 = <function sumup at 0x10529af70>([1, 2, 3, 4, 5, 6])
E       + where <function sumup at 0x10529af70> = sumup.sumup

test_sumup.py:10: AssertionError
===== short test summary info =====
FAILED test_sumup.py::test_sum2 - assert 15 == 21
===== 1 failed, 1 passed in 0.02s =====
Slashes-MacBook-Pro-2:Code slash$
```

# Doing the tests

## Test functions

- ▶ It is where the tests are performed
- ▶ Need one function per test case (which may call helper functions)
- ▶ Name must start with `test*` in `pytest`
- ▶ We recommend the following naming convention of `test_`

## Verifying results

- ▶ All tests are verified with assertions.
- ▶ `pytest` has an `assert` construct for that

# Assertion Example

```
import pytest
import sumup

def test_sum():
    assert sumup.sumup([1, 2, 3, 4, 5]) == 15
    assert sumup.sumup([0, 1, 2]) >= 2

def test_sum2():
    assert sumup.sumup([0, 1, 2]) == 3

def test_sum3():
    assert sumup.sumup([1, 3, 4]) != 12

def testit():
    assert sumup.sumup([1, 3, 4]) == 8
```

# Setting up test environment

## Purpose

- ▶ Get things ready for testing.
- ▶ Create common instances, variables and data to use in tests.

## pytest fixtures

- ▶ Set-up the environment before calling a test or a set of tests
- ▶ Annotation in Python
  - ▶ `@pytest.fixture`
  - ▶ defines functions
  - ▶ functions are given as parameters to the test functions
  - ▶ function names hold fixture variables/values when exiting :
    - ▶ `yield`
    - ▶ `return` statement

# A fixture example

```
import pytest
import sumup

@pytest.fixture
def connect():
    import requests
    response = requests.get('https://festor.me/testdata.txt
    _?inline=false', allow_redirects=True)
    print(response.text)
    result = [int(i) for i in response.text.split(",")] # A nice
    print(result)
    yield result
    print("shutting_down")

def test_basique(connect):
    assert sumup.sumup(connect[:-1]) == connect[-1]
```

# Cleaning test environment: Tearing down methods

---

## Purpose

- ▶ Clean up after testing
- ▶ I.e., closing any files or connexions, etc
- ▶ Not used as often as setup methods

## Tearing down

- ▶ Code following the `yield` statement in the fixture function
- ▶ Explicitly writing a `fin()` function embedded in the fixture function

## Fixture scoping

- ▶ function: executed once per test
- ▶ class: executed once per test class
- ▶ module: executed once per module
- ▶ session : executed once per session

# Going further with pytestt: TDD

## Test-Driven Development

- ▶ That's a methodology to write code
- ▶ Aims at ease/productivity + code quality

## Principle: Write the Test Cases First (before the code)

- ▶ Ensures that the codes actually get written
- ▶ Improves the interface: you're user of your own code before coding it

## That's easy and pleasant to do

- ▶ It's one of the “agile development methodologies”, very light-weighted  
More than just TDD in agile methods (but too long to say it all here)
- ▶ Eclipse correction suggestion and ability to generate stubs very helpful
- ▶ Try it for your next project



# Right BICEPS

Thinking of all mandatory test cases is difficult

- ▶ I.e., challenging to discover all the ways a code might fail
- ▶ **Good news:** Experience quickly gives a feel for what is likely to fail

Beginners need a bit of help (until they get experienced)

- ▶ Guidelines on what can fail
- ▶ Reminders of areas that are important to test
- ▶ These guidelines are not very complex, but quite useful/powerful
- ▶ See [Software Systems and Architecture](#) [Scott Miller] for details

# Right-BICEP

## Guidelines in a Nutshell

- ▶ **R**ight: Are the results right?
- ▶ **B**: Are all the **b**oundary conditions **CORRECT**?
- ▶ **I**: Can you check **i**nverse relationships?
- ▶ **C**: Can you **c**ross-check results using other means?
- ▶ **E**: Can you force **e**rror conditions to happen?
- ▶ **P**: Are **p**erformance characteristics within bounds?

## Right?

- ▶ We need to compute what the correct result should be to test
- ▶ Quite often these can be inferred from the specification
- ▶ If the "right" results cannot be determined... you shouldn't be writing code!
- ▶ If spec not completed [by client], assume what's correct, and fix afterward

## B: Boundary Tests (1/3)

Discovering boundary conditions is crucial!

- ▶ This is where most of the bugs generally live
- ▶ These are also the "edges" of our code

Remember our little experience

- ▶ We had to refine it several time our specifications
  - ▶ Triangle with negative length
  - ▶ Sort an empty array
- ▶ The algorithm in exercise 3 of proof lab were false
  - ▶ Failed to find smallest value if at the end of the array

## B: Boundary Tests (2/3)

### Example of boundary conditions

- ▶ Totally bogus, inconsistent input values: filename of " #()\*% )Q\*#%&@"
- ▶ Badly formatted data: e-mail address without TLD (zastre@foo)
- ▶ Empty or missing values: 0, 0.0, "", null
- ▶ Values above some reasonable expectation: age of 10,000; #children == 30
- ▶ Duplicates in lists meant to be free of duplicates
- ▶ Ordered lists that are not ordered
  - Also: Presorted lists passed to sort algorithms? reverse-sorted?
- ▶ Things which arrive out of order? or out of expected order?

## B: Boundary Tests (3/3)

Another guideline for boundaries: CORRECT

- ▶ **Conformance:** Does the value conform to an expected value?
- ▶ **Ordering:** Is the set of values ordered or unordered as appropriate?
- ▶ **Range:** Is the value within reasonable minimum and maximum values?
- ▶ **Reference:** Does the code reference anything external that isn't under control?
- ▶ **Existence:** Does the value exist? (e.g., is non-null, non-zero, present in a set)
- ▶ **Cardinality:** Are there exactly enough values?
- ▶ **Time:** Is everything happening in order? At the right time? In time?

# I: Check Inverse Relationships

Some methods can be checked almost trivially

- ▶ Data inserted in table should appear in a search immediately afterwards
- ▶ Lossless compression algorithm  $\rightsquigarrow$  data uncompressed to the original value
- ▶ Check square-root calculation by squaring result (ensure it is "close enough")

## Inverse Gotchas

- ▶ You usually write the function/method and its inverse
- ▶ What if both are buggy? errors gets be masked
- ▶ Ideally, the inverse function is written by somebody else
  - ▶ Square root example: use built-in multiplication
  - ▶ Database insert: vendor-provided search routine to test insert

## C: Cross-check Using Other Means

### Idea 1: For methods without an inverse

- ▶ Use a different algorithm to compute the result, and compare to yours
- ▶ For example, use a  $O(n^2)$  sorting algorithm to check your  $O(n \times \log(n))$  one

### Idea 2: Ensure that different pieces of the class's data “add up”

- ▶ Example: library system with book, copies
- ▶ “books out” + “books in library” should equal “total copies”

# E: Force Error Conditions

Production code defensive to system failures

- ▶ Disks: fill up
- ▶ Networks: go down
- ▶ E-mail: gets lost
- ▶ Programs: crash

This also should to be tested

- ▶ Easy: invalid parameters
- ▶ Harder: environmental errors

Environmental errors/constraints

- ▶ Out of memory; Out of disk space
- ▶ Network availability and errors
- ▶ System load
- ▶ Limited colour palette; Very high or very low video resolution
- ▶ ...



# P: Performance Characteristics

What is the time performance as:

- ▶ Input size grows?
- ▶ Problem sets become more complex

Idea: “regression test” on performance characteristics

- ▶ Ensure that version  $N+1$  is not awfully slower than version  $N$

Very hard to ensure

- ▶ Bad performance can come from external factors
- ▶ Performance not portable from machine to machine
- ▶ (even harder to ensure *automatically*)

# When to stop writing tests?

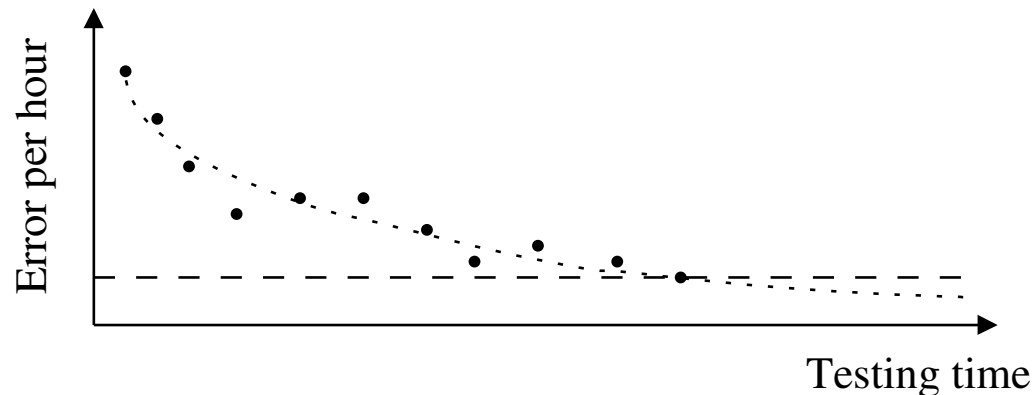
*Testing can only prove the presence of defects, not their absence.*  
– E. W. Dijkstra

## Cynical answer (sad but true)

- ▶ You're never done: each run of the system is a new test
  - ⇒ Each bug-fix should be accompanied by a new regression test
- ▶ You're done when you are out of time/money
  - ▶ Include test in project plan and **do not give in to pressure**
  - ▶ ... in the long run, tests **save** time

## Statistical testing

- ▶ Test until you've reduced failure rate under risk threshold



# Why to test? (continued)

Because it helps ensuring that the system matches its specification

But not only (more good reason to test)

- ▶ Traceability
  - ▶ Tests helps tracing back from components to the requirements that caused their presence
- ▶ Maintainability
  - ▶ Regression tests verify that post-delivery changes do not break anything
- ▶ Understandability
  - ▶ Newcomers to the system can read the test code to understand what it does
  - ▶ Writing tests first encourage to make the interface really useable

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Introduction

## Design by Contract

- ▶ Programming methodology trying to prevent code to diverge from specs
- ▶ Mistakes are possible
  - ▶ while transforming requirements into a system
  - ▶ while system is changed during maintenance

# Introduction

## Design by Contract

- ▶ Programming methodology trying to prevent code to diverge from specs
- ▶ Mistakes are possible
  - ▶ while transforming requirements into a system
  - ▶ while system is changed during maintenance

## What's the difference with Testing?

- ▶ Testing tries to diagnose (and cure) errors after the facts
- ▶ Design by Contract tries to prevent certain types of errors

# Introduction

## Design by Contract

- ▶ Programming methodology trying to prevent code to diverge from specs
- ▶ Mistakes are possible
  - ▶ while transforming requirements into a system
  - ▶ while system is changed during maintenance

## What's the difference with Testing?

- ▶ Testing tries to diagnose (and cure) errors after the facts
- ▶ Design by Contract tries to prevent certain types of errors

## Design by Contract is particularly useful in an Object-Oriented context

- ▶ preventing errors in interfaces between classes  
incl. subclass and superclass via subcontracting
- ▶ preventing errors while reusing classes  
incl. evolving systems, thus incremental and iterative development

Example of the Ariane 5 crash

# Introduction

## Design by Contract

- ▶ Programming methodology trying to prevent code to diverge from specs
- ▶ Mistakes are possible
  - ▶ while transforming requirements into a system
  - ▶ while system is changed during maintenance

## What's the difference with Testing?

- ▶ Testing tries to diagnose (and cure) errors after the facts
- ▶ Design by Contract tries to prevent certain types of errors

## Design by Contract is particularly useful in an Object-Oriented context

- ▶ preventing errors in interfaces between classes  
incl. subclass and superclass via subcontracting
  - ▶ preventing errors while reusing classes  
incl. evolving systems, thus incremental and iterative development
- Example of the Ariane 5 crash

**Use Design by Contract in combination with Testing!**



# What is Design By Contract?

*View the relationship between two classes as a formal agreement, expressing each party's rights and obligations. – Bertrand Meyer*

## Example: Airline Reservation

	Obligations	Rights
Customer	<ul style="list-style-type: none"><li>▶ Be at Paris airport at least 3 hour before scheduled departure time</li><li>▶ Bring acceptable baggage</li><li>▶ Pay ticket price</li></ul>	<ul style="list-style-type: none"><li>▶ Reach Los Angeles</li></ul>
Airline	<ul style="list-style-type: none"><li>▶ Bring customer to Los Angeles</li></ul>	<ul style="list-style-type: none"><li>▶ No need to carry passenger who is late</li><li>▶ has unacceptable baggage</li><li>▶ or has not paid ticket</li></ul>

- ▶ Each party expects benefits (rights) and accepts obligations
- ▶ Usually, one party's benefits are the other party's obligations
- ▶ Contract is declarative: it is described so that both parties can understand *what* service will be guaranteed without saying *how*

# Connecting back to Hoare logic

## Pre- and Post-conditions + Invariants

- ▶ Obligations are expressed via pre- and post-conditions

*If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied.*

pre-condition:  $x \geq 9$     post-condition:  $x \geq 13$

component:  $x := x + 5$

- ▶ and invariants

*For all calls you make to me, I will make sure the invariant remains satisfied.*

## Isn't this pure documentation?

- (a) Who will register these contracts for later reference (the notary)?
- (b) Who will verify that the parties satisfy their contracts (the lawyers)?

# Connecting back to Hoare logic

## Pre- and Post-conditions + Invariants

- ▶ Obligations are expressed via pre- and post-conditions

*If you promise to call me with the precondition satisfied, then I, in return promise to deliver a final state in which the postcondition is satisfied.*

pre-condition:  $x \geq 9$     post-condition:  $x \geq 13$

component:  $x := x + 5$

- ▶ and invariants

*For all calls you make to me, I will make sure the invariant remains satisfied.*

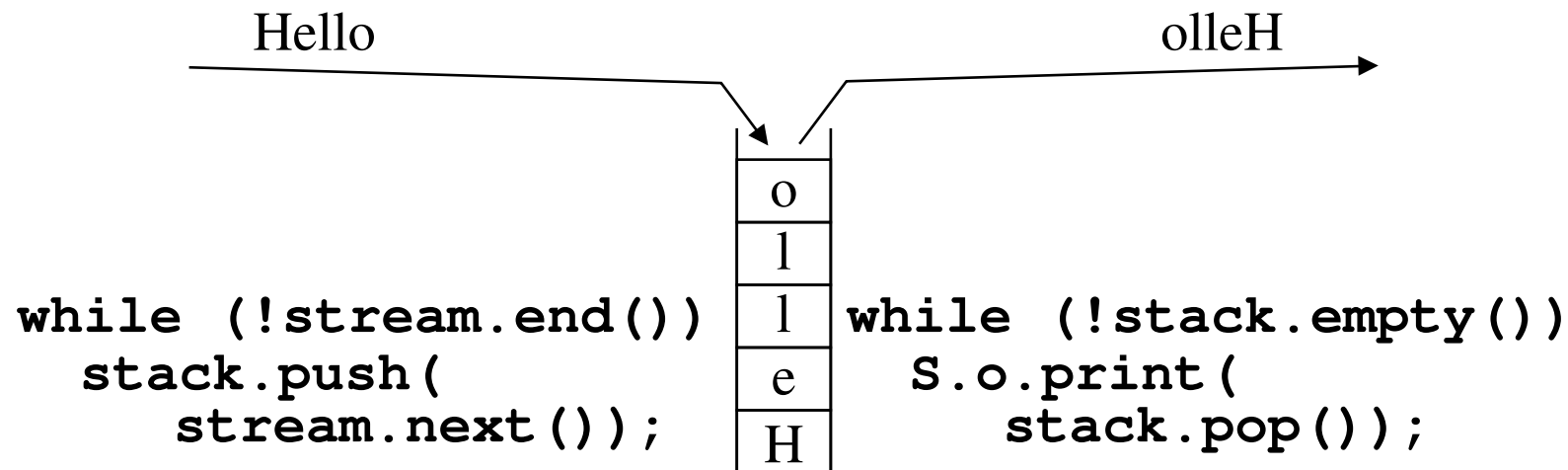
## Isn't this pure documentation?

- (a) Who will register these contracts for later reference (the notary)?  
The source code
- (b) Who will verify that the parties satisfy their contracts (the lawyers)?  
The running system

# Example: Stack

## Specification

- ▶ Given
  - ▶ A stream of characters, length unknown
- ▶ Requested
  - ▶ Produce a stream containing the same characters but in reverse order
  - ▶ Specify the necessary intermediate abstract data structure



# Example: Stack Specification

class stack

invariant: (isEmpty (this)) or (! isEmpty (this))

/\* Implementors promise that invariant holds after all methods return  
(incl. constructors)\*/

public char pop ()

require: !isEmpty(this)

ensure: true

/\* Clients' promise (precondition) \*/

/\* Implementors' promise (postcondition)  
Here: nothing \*/

public void push(char)

require: true

ensure: (!isEmpty(this))  
and (top(this)==char)

/\* Implementors' promise:  
Matches specification \*/

public void top (char) : char

require: ...

ensure: ...

/\* left as an exercise \*/

public void isEmpty() : boolean

require: ...

ensure: ...

# Defensive Programming

## Redundant checks

- ▶ Redundant checks are the naive way for including contracts in the source code

```
public char pop () {  
    if (isEmpty (this)) {  
        ... //Error-handling  
    } else {  
        ...}  
}
```

This is redundant code: it is the responsibility of the client to ensure the pre-condition!

## Redundant Checks Considered Harmful

- ▶ Extra complexity  
due to extra (possibly duplicated) code ... which must be verified as well
- ▶ Performance penalty  
Redundant checks cost extra execution time
- ▶ Wrong context
  - ▶ How severe is the fault? How to rectify the situation?
  - ▶ A service provider cannot assess the situation, only the consumer can.
  - ▶ Again: What happens if the precondition is not satisfied?

# Assertions

Any boolean expression we expect to be true at some point

## Advantages

- ▶ Help in writing correct software (formalizing invariants, and pre/post-conditions)
- ▶ Aid in maintenance of documentation (specifying contracts **in the source code**)  
⇒ tools to extract interfaces and contracts from source code
- ▶ Serve as test coverage criterion (Generate test cases that falsify assertions)
- ▶ Should be configured at compile-time (to avoid performance penalties in prod)

## What happens if the precondition is not satisfied?

- ▶ When an assertion does not hold, throw an exception

# Assertions in Programming Languages

## Eiffel

- ▶ Eiffel is designed as such ... but only used for correction (not documentation)

## C++

- ▶ assert.h does not throw an exception, but close program
- ▶ Possible to mimick. Documentation extraction rather difficult

## Smalltalk

- ▶ Easy to mimic, but compilation option requires some language idioms
- ▶ Documentation extraction is possible (style JavaDoc)

## Java

- ▶ Assert is standard since Java 1.4 ... very limited
- ▶ JML provide a mechanism ... but not ported to Java 5 (damn genericity)
- ▶ Modern Jass seems very promising, but needs more polishing



# Design by Contract vs. Testing

They serve the same purpose

- ▶ Design by contract *prevents* errors; Testing *detect* errors
- ~> One of them should be sufficient!

They are complementary

None of the two guarantee correctness ... but the sum is more than the parts

- ▶ Testing detects wide range of coding mistakes  
... design by contract prevents specific mistakes due to incorrect assumption
- ▶ Design by contract ease black box testing by formalizing spec
- ▶ Condition testing verify whether all assertions are satisfied  
(whether parties satisfy their obligations)

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Fuzzing big picture

1. Intercept the data an application gets from its environment
2. **Fuzz it**, i.e. provide data vaguely resembling to expected one  
(sort of model-checking, exploring only execution paths close to the usual one)

## Why? Motivation

- ▶ It's a security assessment method:  
if you can get the application to segfault, it must be a buffer overflow to exploit
- ▶ Easy to write some tests w/o system knowledge  
⇒ launch a fuzzer when arriving in company, it may find something interesting
- ▶ One of the best price/quality ratio

## Target applications

- ▶ Classically, network protocols; Recently used on media files

# How to actually fuzz the data?

## Brute force: random

- ▶ Pick a byte in the stream, and change its value
- 😊 Really easy to do
- 😞 Get easily caught by checksums
- 😞 “Interesting changes” are rather improbable

## Refined manner: templating

- ▶ Understand the logic of the stream
- 😊 Pass checksums and easy validity protection levels
- 😞 Rather hard to do
- 😞 Fuzzing process becomes stream-dependent  
no longer first tool to use: you need to understand stream first

## Some research leads

- ▶ **Stateful fuzzing:** Build a protocol automaton, test invalid transitions
- ▶ **Automatic templating:** Extend file format metagrammar, use almost valid data

# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# Formal Methods

**Goal:** Develop safe software using automated methods

- ▶ Strong mathematical background

# Formal Methods

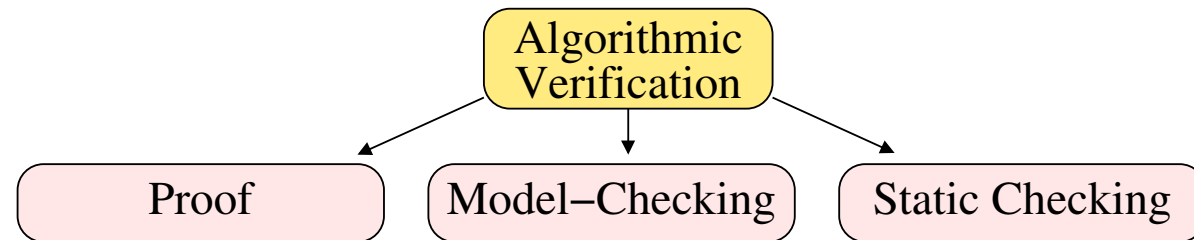
**Goal:** Develop safe software using automated methods

- ▶ Strong mathematical background
- ▶ Safe  $\equiv$  respect some given properties

## Kind of properties shown

- ▶ **Safety:** the car does not start without the key
- ▶ **Liveness:** if I push the break paddle, the car will eventually stop

# Existing Formal Methods



## Proof of programs

- ▶ In theory, applicable to any class of program
- ▶ In practice, quite tedious to use  
often limited to help a specialist doing the actual work (system state explosion)

## Model-checking

- ▶ **Goal:** Shows that a system:
  - (safety) never evolves to a faulty state from a given initial state
  - (liveness) always evolve to the wanted state (stopping) from a given state (breaking)
- ☹ Less generic than proof: lack of faulty states **for all** initial state?
- 😊 Usable by non-specialists (at least, by *less-specialists*)

## Static Checking

- ▶ Automatic analyse of the source code (data flow analysis; theorem proving)
- ▶ Completely automated, you should use these tools, even if partial coverage



# Example of problem to detect: Race Condition

$x$  is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result :  $x = 7$

- a. Read the value of shared variable  $x$  and store it locally
- b. Modify the local value (add 5 or 2)
- c. Propagate the local variable into the shared one

# Example of problem to detect: Race Condition

$x$  is a shared variable; *Alice* adds 2, *Bob* adds 5; Correct result :  $x = 7$

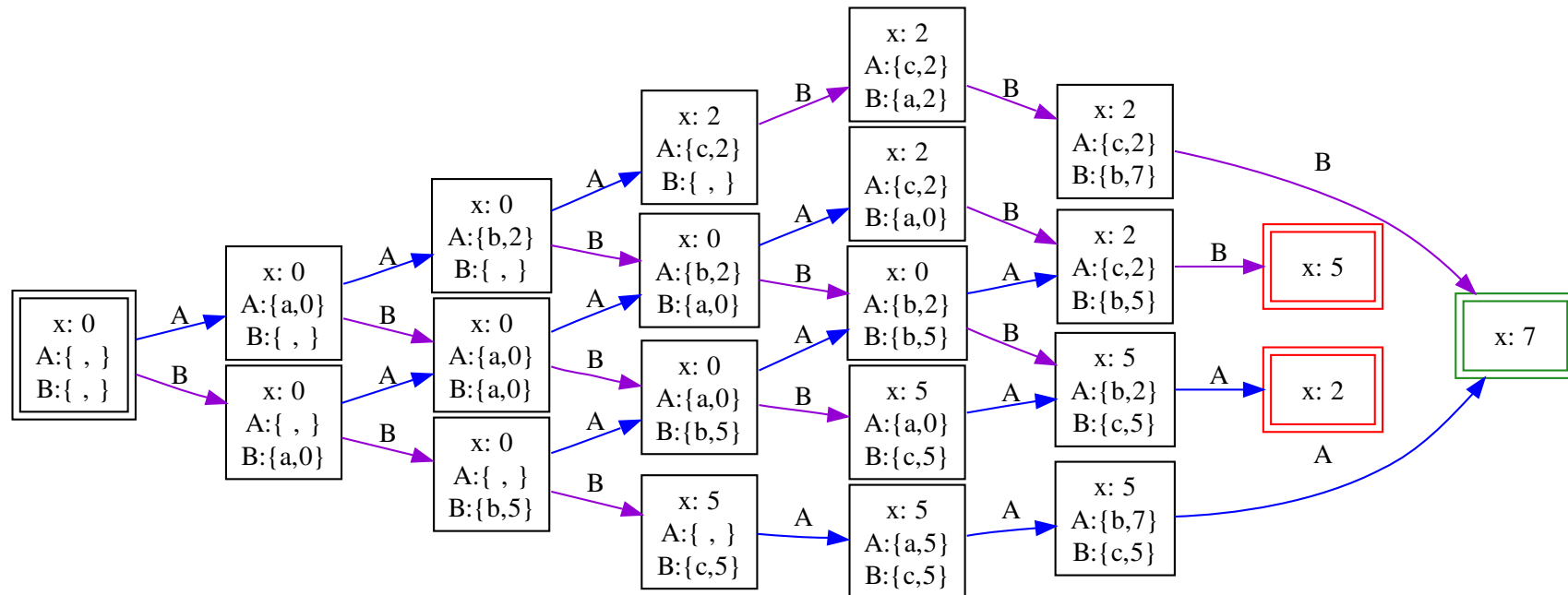
- a. Read the value of shared variable  $x$  and store it locally
  - b. Modify the local value (add 5 or 2)
  - c. Propagate the local variable into the shared one
- 
- ▶ Execution of *Alice* then *Bob* or opposite: result = 7
  - ▶ Interleaved execution: result = 2 or 5 (depending on last propagator)

# Example of problem to detect: Race Condition

$x$  is a shared variable; *Alice* adds 2, *Bob* adds 5; **Correct result** :  $x = 7$

- Read the value of shared variable  $x$  and store it locally
  - Modify the local value (add 5 or 2)
  - Propagate the local variable into the shared one
- ▶ Execution of *Alice* **then** *Bob* or opposite: **result** = 7
  - ▶ Interleaved execution: **result** = 2 or 5 (depending on last propagator)

**Model-checking:** traverse graph of executions checking for properties

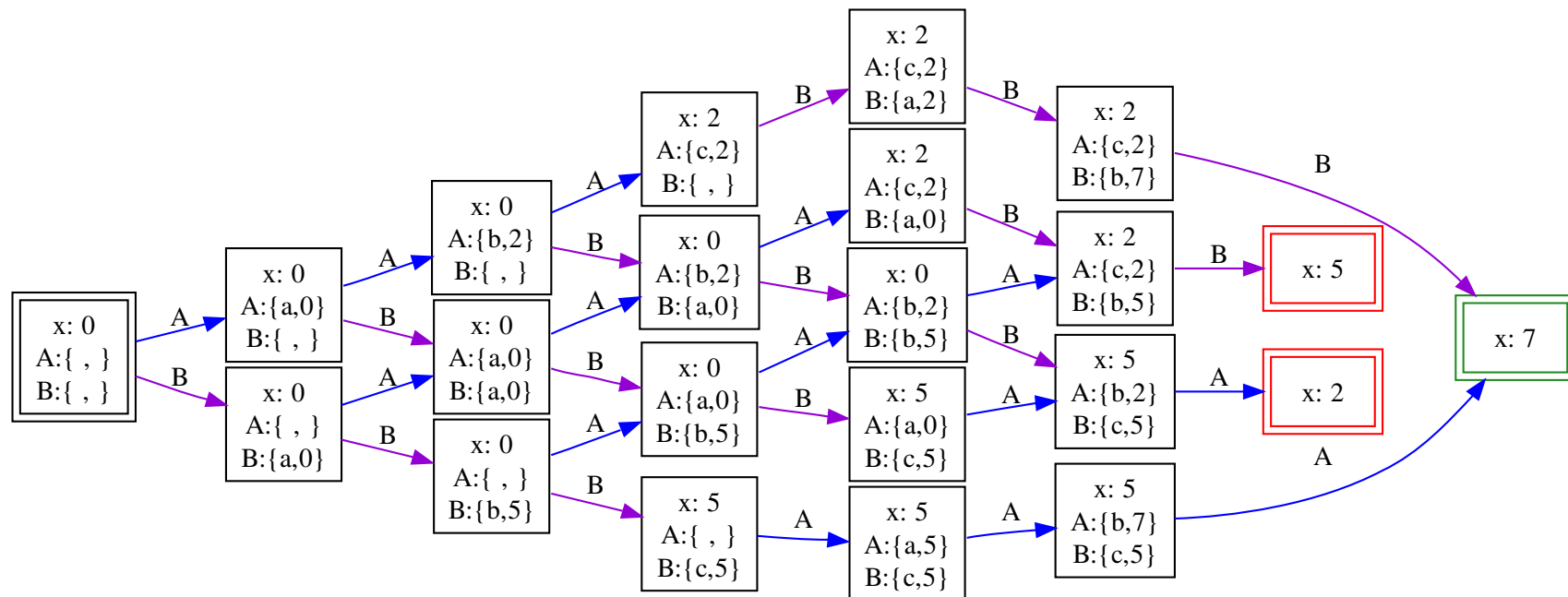


# Example of problem to detect: Race Condition

$x$  is a shared variable; *Alice* adds 2, *Bob* adds 5; **Correct result** :  $x = 7$

- Read the value of shared variable  $x$  and store it locally
  - Modify the local value (add 5 or 2)
  - Propagate the local variable into the shared one
- ▶ Execution of *Alice* **then** *Bob* or opposite: **result** = 7
  - ▶ Interleaved execution: **result** = 2 or 5 (depending on last propagator)

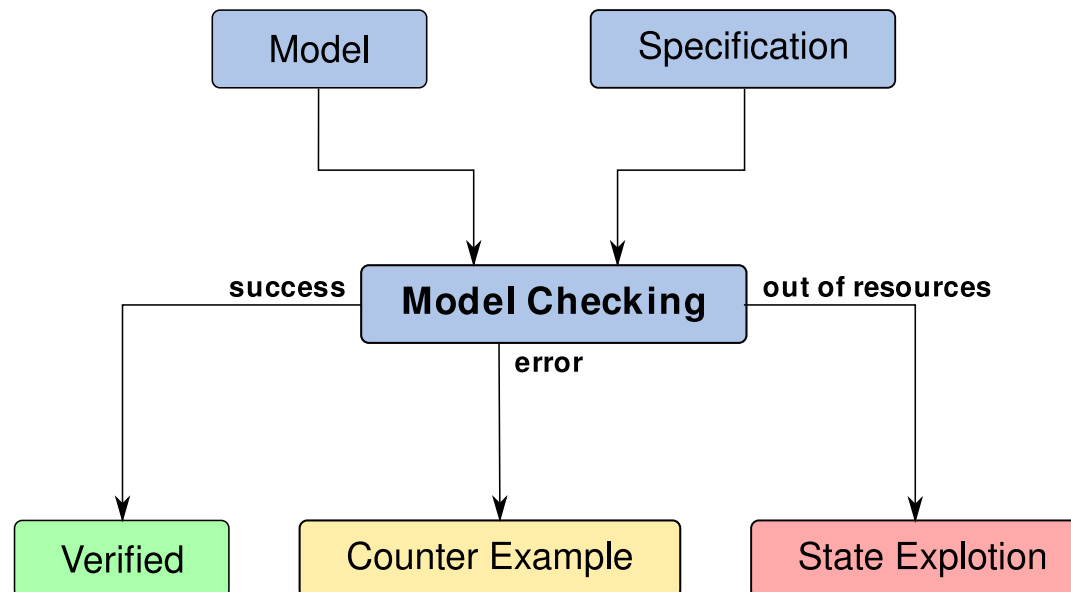
**Model-checking:** traverse graph of executions checking for properties



- ▶ Safety: assertions on each node
- ▶ Liveness by studying graph (cycle?)

# Model-Checking Big Picture

1. User writes **Model** (formal writing of algorithm) and **Specification** (set of properties)
2. Each decision point in model (if, input data)  $\leadsto$  a branch in model state space
3. Check safety properties on each encountered node (state)
4. Store encountered nodes (to avoid looping) and transitions (to check liveness)
5. Process until:
  - ▶ State space completely traversed ( $\Rightarrow$  model verified against this specification)
  - ▶ One of the property does not hold (the path until here is a counter-example)
  - ▶ We run out of resource ("state space explosion")



# Fourth Chapter

## Testing

- Introduction
- Testing Techniques
  - White Box Testing
  - Black Box Testing
- Testing Strategies
  - Unit Testing
  - Integration Testing
  - Regression Testing
  - Acceptance Testing
- Testing in Practice
  - pytest
  - Right BICEP + CORRECT
- Other Techniques for Practical Software Correctness
  - Design By Contract
  - Fuzzing
  - Formal Methods
- Conclusion

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

Example code

```
1 import java.io.*;
2 public class Foo{
3     private byte[] b;
4     private int length;
5     Foo(){ length = 40;
6         b = new byte[length]; }
7     public void bar(){
8         int y;
9         try {
10             FileInputStream x =
11                 new FileInputStream("z");
12             x.read(b,0,length);
13             x.close();}
14         catch(Exception e){
15             System.out.println("Oopsie");}
16         for(int i=1; i<=length; i++){
17             if (Integer.toString(50) ==
18                 Byte.toString(b[i]))
19                 System.out.print(b[i] + " ");
20         }
21     }
22 }
```

Defects of this code

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

## Example code

```
1 import java.io.*;
2 public class Foo{
3     private byte[] b;
4     private int length;
5     Foo(){ length = 40;
6         b = new byte[length]; }
7     public void bar(){
8         int y;
9         try {
10             FileInputStream x =
11                 new FileInputStream("z");
12             x.read(b,0,length);
13             x.close();}
14         catch(Exception e){
15             System.out.println("Oopsie");}
16         for(int i=1; i<=length; i++){
17             if (Integer.toString(50) ==
18                 Byte.toString(b[i]))
19                 System.out.print(b[i] + " ");
20         }
21     }
22 }
```

## Defects of this code

l8 W: Unused variable

l12 W: Return of read() ignored

l14 W: Stream may not be closed

l17 E: == to compare strings

l18 E: may access out of bound

l18 FP: possible null-dereference to b



# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

Example code

```
1 import java.io.*;
2 public class Foo{
3     private byte[] b;
4     private int length;
5     Foo(){ length = 40;
6         b = new byte[length]; }
7     public void bar(){
8         int y;
9         try {
10             FileInputStream x =
11                 new FileInputStream("z");
12             x.read(b,0,length);
13             x.close();}
14     catch(Exception e){
15         System.out.println("Oopsie");}
16     for(int i=1; i<=length; i++){
17         if (Integer.toString(50) ==
18             Byte.toString(b[i]))
19             System.out.print(b[i] + " ");
20     }
21 }
22 }
```

## Defects of this code

l8 W: Unused variable

l12 W: Return of read() ignored

l14 W: Stream may not be closed

l17 E: == to compare strings

l18 E: may access out of bound

l18 FP: possible null-dereference to b

## Some tools for Java

Name	Input	Interface	Technology
Bandera	Source	CL, GUI	Model checking
ESC/Java	Source+spec	CL, GUI	Theorem proving
FindBugs	Bytecode	CL,GUI,Ant,IDE	Syntax, data-flow
JLint	Bytecode	CL	Syntax, data-flow
PMD	Source	CL,GUI,Ant,IDE	Syntax

# A Comparison of Bug Finding Tools for Java

Rutar, Almazan, Foster, U. Maryland – ISSRE04

## Example code

```
1 import java.io.*;
2 public class Foo{
3     private byte[] b;
4     private int length;
5     Foo(){ length = 40;
6         b = new byte[length]; }
7     public void bar(){
8         int y;
9         try {
10             FileInputStream x =
11                 new FileInputStream("z");
12             x.read(b,0,length);
13             x.close();}
14     catch(Exception e){
15         System.out.println("Oopsie");}
16     for(int i=1; i<=length; i++){
17         if (Integer.toString(50) ==
18             Byte.toString(b[i]))
19             System.out.print(b[i] + " ");
20     }
21 }
22 }
```

## Defects of this code

- l8 W: Unused variable
- l12 W: Return of read() ignored
- l14 W: Stream may not be closed
- l17 E: == to compare strings
- l18 E: may access out of bound
- l18 FP: possible null-dereference to b

## Some tools for Java

Name	Input	Interface	Technology
Bandera	Source	CL, GUI	Model checking
ESC/Java	Source+spec	CL, GUI	Theorem proving
FindBugs	Bytecode	CL,GUI,Ant,IDE	Syntax, data-flow
JLint	Bytecode	CL	Syntax, data-flow
PMD	Source	CL,GUI,Ant,IDE	Syntax

## No tool is perfect/sufficient

- ▶ All detect something
- ▶ Some give false positive (b init'd in ctor)
- ▶ All have false negative

# Conclusion

Failure is not an option. It comes bundled with software.

- ▶ **Testing** searches for defects  
But not that easy and endless?
- ▶ **Proof** tries to ensure that there is no defect  
But quite heavy-weighted
- ▶ **Automatic tools** (static checking, theorem provers) may help  
But none is enough/sufficient (false negatives); all have false positives
- ▶ **Design by Contract** constitutes a global (methodological) answer  
Too bad that nobody use it / that Java offers no tool support for it (yet?)

## Optimistic Last Note

- ▶ That's a hot research topic, things move fast
- ▶ Tools improve quickly, you really should learn to use them
- ▶ Methodologies exist (DBC, TDD), you should try to follow them

# Bibliography for this chapter (and previous one)

## Lectures

- ▶ [Testing, Debugging, and Verification](#) (W. Ahrendt, R. Hähnle – U. Göteborgs)  
[www.cse.chalmers.se/edu/year/2009/course/TDA566](http://www.cse.chalmers.se/edu/year/2009/course/TDA566)
- ▶ [Software Engineering](#) (Serge Demeyer – U. Antwerpen)  
<http://www.lore.ua.ac.be/Teaching/SE3BAC/>
- ▶ [Software Systems and Architecture](#) (Scott Miller – U. of Victoria)  
<http://samiller.ece.uvic.ca/courses/SENG271/2009/05/>
- ▶ [JUnit](#) (Dirk Hasselbalch – U. Copenhagen)  
<http://isis.ku.dk/kurser/blob.aspx?feltid=217458>

## Other

- ▶ [Test Infected: Programmers Love Writing Tests](#) (Tutorial on JUnit)  
[http://www.cril.univ-artois.fr/~leberre/MI32001/TESTING/junit3.7/doc/testinfected/ntesting\\_fr.htm](http://www.cril.univ-artois.fr/~leberre/MI32001/TESTING/junit3.7/doc/testinfected/ntesting_fr.htm)