



Intro to Unit Testing

James Brucker

Many Kinds of Software Testing

- ❑ Test **requirements** - consistent? unambiguous?
- ❑ Test **application design** - does it satisfy requirements?
Consistent with *Vision*? Anything not in requirements?
- ❑ **Unit Testing - test individual methods and functions**
- ❑ Integration Testing
- ❑ End-to-End or Functional Testing
- ❑ Acceptance Testing
- ❑ Usability Testing

Why Test?

1. *Saves time!*

- *Testing is faster than fixing "bugs".*

2. *Testing finds more errors than debugging.*

3. *Prevent re-introduction of old errors (regression errors).*

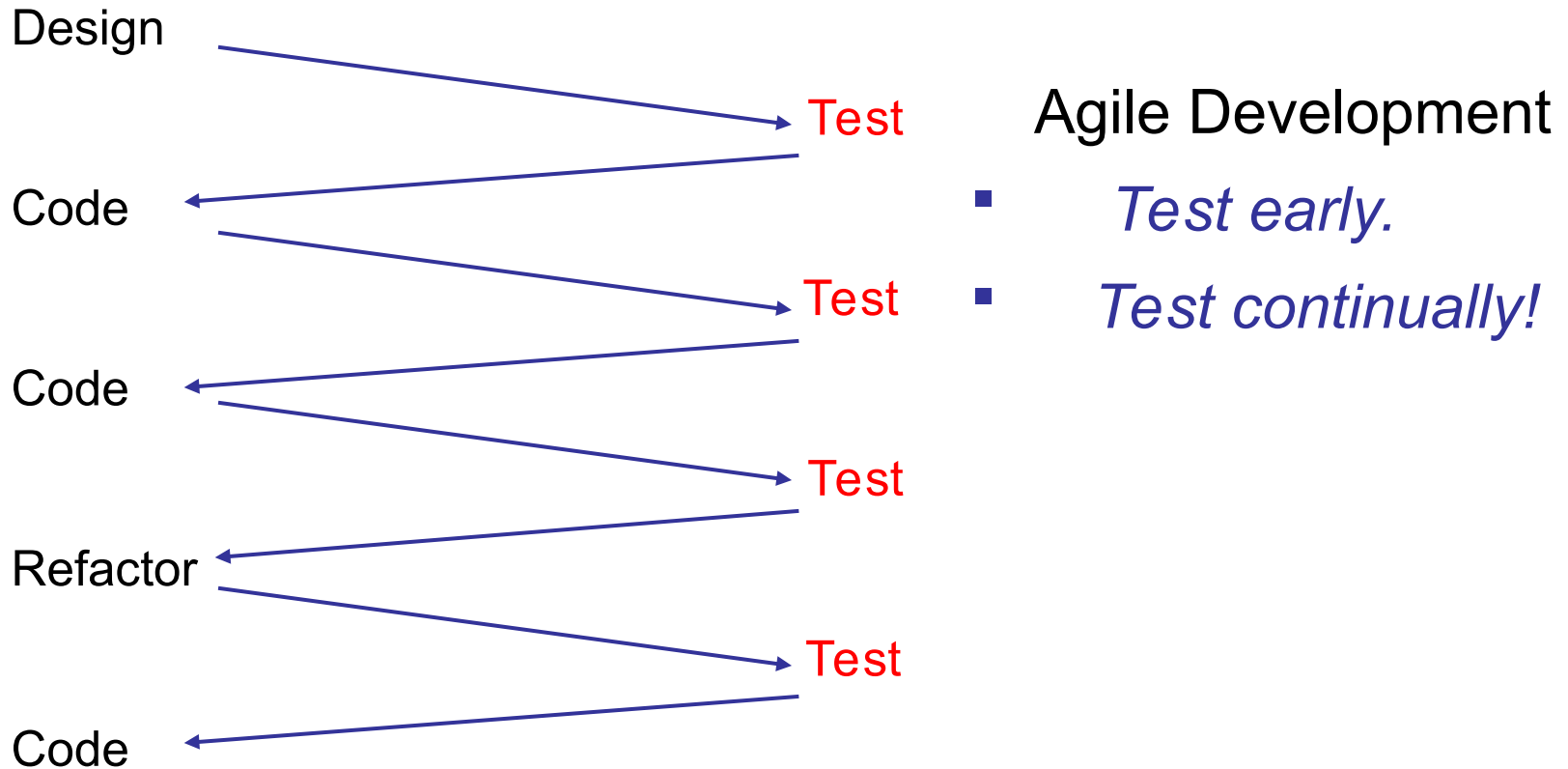
Programmers often **recreate** an error that was already fixed when they modify code.

4. *Validate software: does it match the specification?*

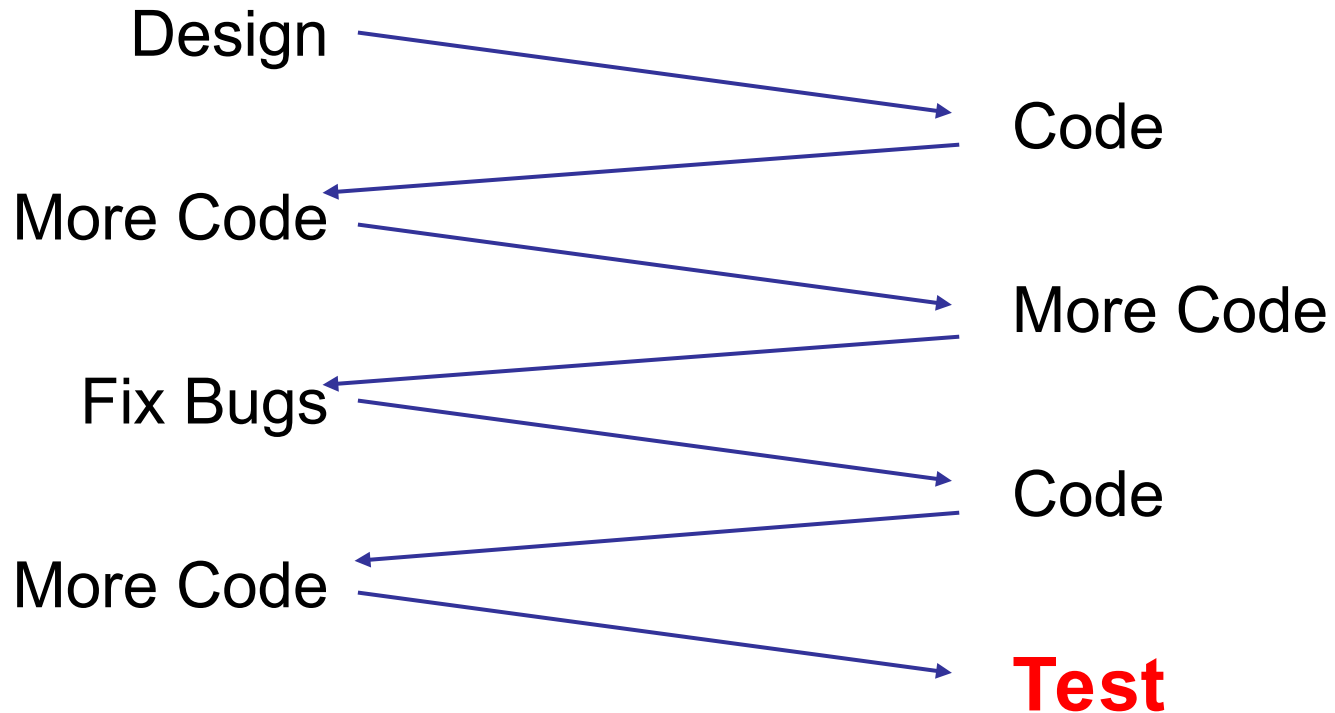
Psychological Advantages

- ❑ *Keeps you focused on current task.*
- ❑ *Increase satisfaction.*
- ❑ *Confidence to make changes.*

Test Frequently During Development



Testing Done Wrong



When to Test?

- ***While you are coding.***
- ***Whenever you fix or modify existing code.***
- ***Before & after refactoring.***
- ***When the environment changes*** (upgrade a package, "pull" new code, change Python version, change OS).

Test-Driven Development (TDD)

Write the **tests first** ...

*write tests for what the code should do.
the tests will fail, but that's OK.*

Then write code that passes the tests.

The Cost of Fixing "faults"

Discover & fix a defect **early** is **much cheaper** (100X) than to fix it **after** code is integrated.

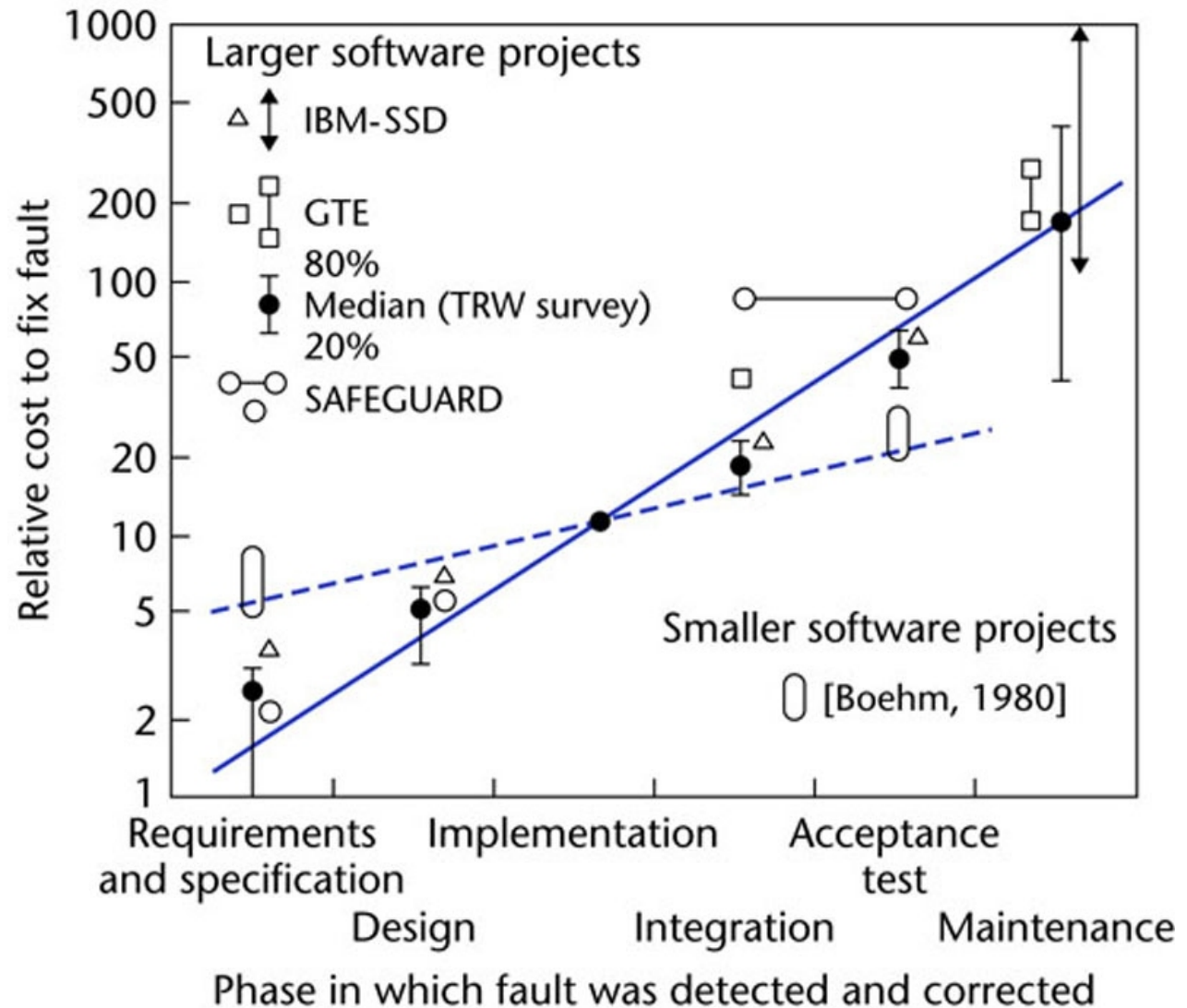
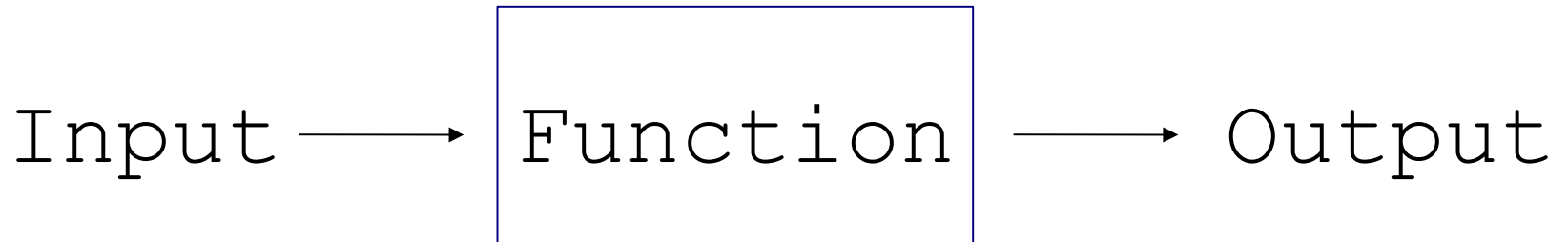


Figure 1.5

What to Test?

In unit testing, we test functions or methods.

Test that inputs produce the expected results.

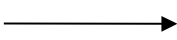


Test State, Too

Many operations change the **state** of an object or component.

You should test the expected state, too.

start () → Stopwatch



is **running** ?



is **elapsed** increasing?

How to Test?

We **can not** test all possible inputs & outputs.

- Divide input into **categories** or **sets**.
- Discover "rules" that apply to different sets of input.
- Test a few samples from each set, category, or class.
 - Test **boundary** values.
 - Test "**typical**" values.
 - Test "**extreme**" values.
 - Test **impossible** values.
 - Try to make the code **fail**.

Example: gcd(a,b)

gcd(a:int, b:int) = greatest common divisor

gcd(24, 30) -> 6

gcd(3, 7) -> 1 (no common factors)

Rule: gcd is always positive

gcd(80, -15) -> 5

gcd(-7, -3) -> 1

Rule: gcd involving zero is positive

gcd(8, 0) -> 8

gcd(0, -8) -> 8

Edge Case: something that may go wrong

gcd(0, 0) -> 1

Defining Test Cases

Test Case	Example Arguments
Two positive ints with common factor	(30, 35), (48, 20), (36, 999)
Two int with no common factor	(1, 50), (50, 3), (370, 999), (1,1)
One or both args are negative	(-30,45), (72,-27), (-1,-2)
One or both args are zero	(99, 0), (0, 7), (0, -7), (0, 0)
Extreme case to test algorithm efficiently terminates	(123*123457890123, 123*789012345890)

Don't Rely on Manual Tests

Automate

Automate

Automate

Why?

Python Unit Test Libraries

Doctest - tests in code provide documentation

Unittest - the standard, based on JUnit

Pytest - simple yet powerful package for concise tests. Can execute doctests & unittests, too.

Tools to Enhance Testing:

Mock objects - "fake" objects for external components

Hamcrest - declarative rules of "intent" to help write readable, powerful matching rules for tests.

Python unittest

```
import unittest
```

```
class TestGcd(unittest.TestCase):
```

```
    def test_gcd_positive_values(self):  
        """Should return positive gcd."""  
        self.assertEqual(5, gcd(30, 35))  
        self.assertEqual(4, gcd(48, 20))
```

```
    def test_gcd_no_common_factors(self):  
        """gcd of relatively primes values is 1."""  
        self.assertEqual(1, gcd(30, 49))  
        self.assertEqual(1, gcd(27, 29))  
        self.assertEqual(1, gcd(44, 1))
```

Doctest

```
def gcd(a: int, b: int):  
    """Return the greatest common divisor two ints.  
  
    Examples:  
>>> gcd(24, 30)  
6  
>>> gcd(24, -36)  
12  
>>> gcd(24, 49)  
1  
>>> gcd(0, 15)  
15  
"""
```

Provides documentation.

Each test is a different category of input.

Pytest

```
import pytest

def test_gcd_positive_values():
    """Should return positive gcd."""
    assert 5 == gcd(30, 35)
    assert 4 == gcd(48, 20)

def test_gcd_no_common_factors():
    """gcd of relatively primes values is 1."""
    assert 1 == gcd(30, 49)
    assert 1 == gcd(27, 29)
    assert 1 == gcd(44, 1)
```

Run: `pytest -v`
`pytest -v test_file_name.py`

Parameterize: reuse test code

```
import pytest

@pytest.mark.parametrize( # "parametrize" is not typo
    "a, b, expected",
    [
        (30, 35, 5),
        (48, 20, 4),
        (27, 29, 1),
    ])
def test_gcd_positive_values(a, b, expected):
    assert expected == gcd(a, b)
```

Run a test with multiple sets of values.

unittest has parameterized tests, too.

FIRST - guide for good tests

Fast

Independent - can run any subset of tests in any order

Repeatable - always get same result

Self-checking - test knows if it passed or failed

Timely - written at same time as the code to test

References

`unittest` in Python Library

- Search for "`unittest`" on main page of Library
- You should have Library **installed on your computer**

Hitchhiker's Guide to Python Testing

Short introduction to several testing tools

<https://docs.python-guide.org/writing/tests/>

Test-Driven Development in Python, 2E

"The TDD book" for Python. Uses Django.