Tests

Tests in Software Design

- Unit tests are essential for **iterative design**.
- They provide a **safety net** during refactoring.
- Good design makes testing **easier**.
- Tests help **verify** that our design decisions work correctly.

What type of tests do we use?

- *Unit test* checks if the module methods are working as expected.
- *Regression test* checks if refactored code is working as before.
- *Integration test* checks if two or more modules are working as expected.
- *Acceptance test* checks if all the requirements are implemented and working as expected.

Real Example: Employee Salary Feature

The Feature: Give employees a salary raise

```python
class Employee:
    def __init__(self, first_name, last_name, salary, shift):
        self.first_name = first_name
        self.last_name = last_name
        self.salary = salary
        self.shift = shift

    def raise_salary(self, factor):
        """Multiply current salary by the given factor"""
        self.salary = self.salary * factor

    def get_full_name(self):
        return f"{self.first_name} {self.last_name}"
```

Testing the Salary Feature

- When we make a feature, we must make the tests that check the feature:
    - **Think like a bank auditor:** Test every calculation!

```python
import unittest
from employee import Employee

class TestEmployee(unittest.TestCase):
    def test_get_full_name(self):
        # Test basic name formatting
        employee = Employee("John", "Doe", 50000, "9-5")
        self.assertEqual(employee.get_full_name(), "John Doe")

    def test_raise_salary(self):
        # Test salary calculation
        employee = Employee("Jane", "Smith", 2000, "9-5")
        employee.raise_salary(1.1)  # 10% raise
        self.assertEqual(employee.salary, 2200)

    def test_zero_raise(self):
        # Edge case: no raise
        employee = Employee("Bob", "Jones", 3000, "9-5")
        employee.raise_salary(1.0)  # 0% raise
        self.assertEqual(employee.salary, 3000)
```

Unit Test

- **Unit tests** are testing each module's functionality.
- Unit tests are **part of the development process**.
- Software engineers must make unit tests when they make modules.

Running Your Tests

In Command Line:

```
python -m unittest test_employee.py -v
```

In VS Code:

- Click the green arrow next to test methods
- Right-click → "Run Tests"

Expected Output:

```
test_get_full_name ... ok
test_raise_salary ... ok
test_zero_raise ... ok

Ran 3 tests in 0.002s
OK
```

Regression Testing: Preventing Backslides

Analogy: Home security system after renovation

- Test all sensors still work after changes
- Run automatically when you modify the house

In software: Run ALL tests after code changes

```
#!/bin/bash
# run_all_tests.sh

echo "Running all tests after code changes..."
python -m unittest test_employee.py
python -m unittest test_reports.py
python -m unittest test_database.py
echo "All tests completed!"
```

Integration Testing: Testing Teamwork

Analogy: Testing a restaurant kitchen

- Can the chef and sous chef work together?
- Does the ordering system talk to the kitchen?

```python
class TestEmployeeReportIntegration(unittest.TestCase):

    def test_salary_report_generation(self):
        # Test Employee + Report modules together
        emp = Employee("Alice", "Wonder", 5000, "9-5")
        emp.raise_salary(1.2)

        report = SalaryReport()
        result = report.generate_for_employee(emp)

        self.assertIn("Alice Wonder", result)
        self.assertIn("6000", result)  # 5000 * 1.2
```

Acceptance Testing: The Final Exam

Analogy: Test driving a completed car

- Does it start?
- Can you drive safely?
- Do all features work as advertised?

Requirements as the Guideline

- **What to Test:** Requirements define exactly what the system should do
- **How to Verify:** Each requirement becomes a test case
- **Pass/Fail Criteria:** Clear, measurable expectations

Methods:

- **Automated scripts** - Run user scenarios automatically
- **Manual testing** - Human testers use the software
- **User acceptance** - Real users try it out

Modern Testing with Pytest

Pytest makes testing easier and more powerful:

```
pip install pytest
```

Simple test example:

```python
def calculate_bonus(salary, performance_score):
    return salary * (performance_score / 100)

def test_bonus_calculation():
    # Test normal case
    assert calculate_bonus(50000, 10) == 5000

def test_zero_bonus():
    # Test edge case
    assert calculate_bonus(50000, 0) == 0

def test_high_performance():
    # Test high performer
    assert calculate_bonus(50000, 25) == 12500
```

Running Pytest

```
$ pytest
========================= test session starts ==========================
collected 3 items

test_bonus.py ...                                                [100%]

========================== 3 passed in 0.01s ===========================
```

If a test fails:

```
$ pytest
========================= FAILURES =========================
_________________ test_bonus_calculation __________________

    def test_bonus_calculation():
>       assert calculate_bonus(50000, 10) == 6000  # Wrong!
E       assert 5000 == 6000

test_bonus.py:8: AssertionError
```

Testing Best Practices

Write Clear Test Names

```
# Good
def test_salary_raise_with_valid_factor()

# Bad
def test1()
```

Test Edge Cases

- Zero values, negative numbers
- Empty strings, null values
- Maximum/minimum boundaries

Keep Tests Fast

- Unit tests should run in milliseconds
- Use mock objects for external dependencies

Summary: Testing is Your Safety Net

Remember the analogies:

- **Unit tests** = Testing each light switch
- **Integration tests** = Testing room connections
- **Acceptance tests** = Final home inspection
- **Regression tests** = Security check after renovation

Key takeaway: Good tests = Confident coding = Happy users!