Week 10 Lecture 1

⌄ Class	BSCCS2003
🕐 Created	@November 7, 2021 5:18 PM
📎 Materials	
# Module #	53
⌄ Type	Lecture
# Week #	10

Application Testing

Why?

Does something work as intended

- Requirements → specifications
- Respond correctly to inputs
- Respond with reasonable time
- Installation and environment
- Usability and Correctness

Static vs Dynamic

Static testing:

- Code review, correctness proof

Dynamic testing:

- Functional tests
- Apply suitable inputs

White-box testing

- Detailed knowledge of implementation
- Can examine internal variables, counters
- Tests can be created based on knowledge of internal structure

- Pro:
  - More detailed information available, better tests
- Con:
  - Can lead to focusing on less important parts because code is known
  - Does not encourage clean abstraction
  - Too much information?

Black-box testing

- Only interfaces are available, not the actual code
- Tests based on how it would look from outside
- Pro:
  - Closer to real usage scenario
  - Encourages (enforces) clean abstraction of interface
- Con:
  - May miss corner cases that would have been obvious if internal structure was known
  - Debugging is harder - even if it failed, why did it fail?

Grey-box testing

- Hybrid approach between white-box and black-box
- Enforce interface as far as possible
- Internal structure mainly used for debugging, examining variables, etc.

Regressions

- Maintain series of tests starting from basic development of code
  - Each test is for some specific feature or set of features
- **Regression:** loss of functionality introduced by some changes in the code
- Future modifications to code should not break existing code
- Sometimes necessary
  - Update tests
  - Update API versions etc.

Coverage

- How much of the code is covered
  - Every line is executed at least once - 100% code coverage
  - Does not guarantee "correctness" in all conditions
  - There may be more complex paths or other conditions that can cause failure
- Branch coverage, condition coverage, function coverage

Example

```
int foo (int x, int y)
{
  int z = 0;
  if ((x > 0) && (y > 0))
  {
    z = x;
  }
  return z;
}
```

Function coverage

- Test invokes `foo()` at least once

Statement coverage

- Example: `foo(1,1)`
  - All statements in the code will be executed

Branch coverage

- At least two tests needed:

- `foo(1,1)`
  - Branch taken

- `foo(1,0)`
  - Branch not taken

Condition coverage

- At least two tests needed

- `foo(0,1)`
  - First condition fails, second succeeds

- `foo(1,0)`
  - First condition succeeds, second fails

- Note: does not guarantee branch coverage

Summary

- Requirements specified by the user
- Creating suitable tests can itself be challenging
- How much knowledge of the code internals should the testers have?
- Separation of concerns:
  - ideally testers should be able to generate test cases based only on spec and without knowing code
- Code coverage useful metric
  - Does not guarantee all scenarios actually tested

Week 10 Lecture 2

⊙ Class	BSCCS2003	
🕐 Created	@November 7, 2021 5:52 PM	
📎 Materials		
# Module #	54	
⊙ Type	Lecture	
# Week #	10	

Levels of Testing

Initial requirements gathering

- Who are the stakeholders?
  - Students: log in and see the courses
  - Admins: manage students
  - Teachers: update/manage course materials
- Functionality
  - Each group has different needs
- Non-functional requirements
  - Page colour, font, logo

Example: Student page

Functional

- Latest updates
- Register exam hall preferences
- Download hall ticket
- Update course registration
- View completed courses
- ...

Non-functional

- Header/Footer colours

- Copyright notice and extra information

- Logo

- Fonts

- ...

Requirements gathering

- Extensive discussions with end-users required

- Avoid language ambiguity

- Capture use cases and examples

- Start thinking about test cases and how the requirements will be validated

Units of implementation

- Break functional requirements down to small, implementable units

- Example:

    - view course list

    - edit course status

    - edit exam preferences

    - download completion certificate

- Each one may become a single controller

    - May also combine multiple into a single controller

Unit Testing

- Test each individual unit of implementation

- May be single controllers

    - May even be a part of a controller

- Clearly define inputs and expected outputs

- Testable in isolation?

    - Can each unit be tested without the entire system?

    - Create artificial data set to check whether a single update works

Example: Unit tests

Student registers for a course

- Create dummy DB

    - One student

    - One course

- Test

    - Controller to add course for student

    - Form to be displayed

    - Invalid student ID, course ID - error codes?

    - Add student more than once?

Integration

- Application consists of multiple modules:

    - Student management

- Course management
- Payment interfaces
- Admin interface

Integration testing

- Example of integration:
  - Student + Payment gateway
  - Student + Course + Admin
  - All of the above ...
- Potential problems
  - Individual units work - combined system does not
  - Dependencies violations in server - redesign?
- Continuous integration
  - Combined with version control systems: CI
  - Each commit to main branch triggers a re-evaluation of integration tests
  - Multiple times a day possible

System-level testing

- One step beyond integration
- Includes server, environment
- Mainly black-box: should validate final usage

Example: onlinedegree

- Deploy to final environment: Google app-engine
- Tests domain used
- Confirm all aspects of behaviour
- Non-functional tests
  - Performance under load
  - Number of instances, scaling
  - Cost

System testing Automation

- Has to simulate actual user interaction
- Browser automation frameworks
  - Selenium (Example)
- Includes database, persistent connections, etc
- Typically a complete secondary system

User Acceptance Testing

- Deploy final system
- Tested by restricted set of users - pilot
- "Beta" testing
  - Beta-software: pre-production

Week 10 Lecture 3

	Class	BSCCS2003
⊘	Created	@November 7, 2021 6:28 PM
⊘	Materials	
#	Module #	55
⊘	Type	Lecture
#	Week #	10

Test generation

API-based testing

- Application Programming Interface: abstraction for system design
- Standard representation for APIs
  - OpenAPI, Swagger etc.
- Can they also generate test cases

Use cases

- Import API definition from standard like OpenAPI
- Generate tests for specific endpoints, scenarios
- Record API traffic
- Inject possible problem cases based on known techniques
- Data validation tests

Abstract Tests

- Semi-formal verbal description:
  - Make a request to `'/'` endpoint
  - Ensure that result contains text
    "Hello world"

```
def test_hello(client):
    """Verify home page."""
    rv = client.get('/')
    assert b'Hello World' in rv.data
```

Model-based testing

Example: Authenticate user before showing information

- Scenarios:

    - User already logged in - page shown

    - User not yet logged in - redirect to the login page

    - Forgot password - after resetting, come back to the desired page

- Model:

    - Possible states (logged in, password reset, ...)

    - Possible transitions

    - Generate tests for the possible transitions

Models and Abstract Tests

- Abstract tests apply to generic models

- Create model for system-under-test

- Derive "executable" tests by combining abstract test information with model

(G)UI Testing

- User interface: visual output

- Usually GUI - even for web-based systems

    - But specific details of graphical display may be different in web-based systems

- Tests:

    - Are specific elements present on page

    - Are navigation links present

    - What happens on random click on some part of the page

Browser automation

- Some tests cannot be directly run programmatically

    - Browser is required, just requests not sufficient

- Example:

    - IRCTC or SBI website - captcha protected

    - Some user input also required - cannot be completely automated

- Request generation:

    - Python requests library

    - Capybara (ruby), ...

- Direct browser automation:

    - Selenium framework - actually instantiate a browser

Security testing

- Generate invalid inputs to test app behaviour

- Try to crash server - overload, injection, etc

- Black-box or White-box approaches

- Fuzzing or Fuzz-testing

- Generate large number of random/semi-random inputs

Week 10 Lecture 4

⊙ Class	BSCCS2003	
🕐 Created	@November 7, 2021 6:58 PM	
📎 Materials		
# Module #	54	
⊙ Type	Lecture	
# Week #	10	

pytest

What?

- Framework to make testing easier in Python
- Opinionated:
    - Provides several defaults to make it easier to write tests
- Helpful features:
    - Can automatically set up environment, tear down after test etc.
    - Test fixtures, monkeypatching etc.

NOTE: python standard library includes `unittest` - `pytest` is an alternative with some more features

Example

```
# content of test_sample.py
def func(x):
  return x + 1

def test answer():
  assert func(3) == 5
```

```
$ pytest
=============== test session starts ========================
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y,
pluggy-1.x.y
cachedir: $PYTHON PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test sample.py F
[100%]

=============== FAILURES ============================
_____ test_answer _____

    def test answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test sample.py:6: AssertionError
=============== short test summary info ====================
FAILED test sample.py::test answer - assert 4 == 5
=============== 1 failed in 0.12s ==========================
```

Test for exceptions

```python
# content of test_sysexit.py
import pytest

def f():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

Temporary directory etc.

```python
# content of test_tmpdir.py
def test_needsfile(tmpdir):
    print(tmpdir)
    assert 0
```

```
$ pytest -q test_tmpdir.py
F                                                              [100%]
=================================== FAILURES ===================================
_____ test_needsfiles _____

tmpdir = local('PYTEST_TMPDIR/test_needsfiles0')

    def test_needsfiles(tmpdir):
        print(tmpdir)
>       assert 0
E       assert 0

test_tmpdir.py:3: AssertionError
--------------------------- Captured stdout call ---------------------------
PYTEST_TMPDIR/test_needsfiles0
========================= short test summary info =========================
FAILED test_tmpdir.py::test_needsfiles - assert 0
1 failed in 0.12s
```

Test fixtures

- Set up some data before test

- Remove after test

- Examples:

  - initialize dummy database

  - Create dummy users, files

Example: Test fixture

```
import pytest

@pytest.fixture
def setup_list():
  return ["apple", "banana"]

def test_apple(setup_list):
  assert "apple" in setup_list

def test_banana(setup_list):
  assert "banana" in setup_list

def test_mango(setup_list):
  assert "mango" in setup_list
```

Result: text fixture

```
test_fruit.py ..F                              [100%]

========== FAILURES ============================
_____ test_mango _____

setup_list = ['apple', 'banana']

    def test_mango(setup_list):
>       assert "mango" in setup_list
E       AssertionError: assert 'mango' in ['apple', 'banana']

test_fruit.py:14: AssertionError
========== short test summary info ==========
FAILED test_fruit.py::test_mango - AssertionError: assert 'mango' in ['apple', 'banana']
========== 1 failed, 2 passed in 0.01s ======
```

Conventions

- Test discovery starts from current dir or **testpaths** variable

  - Recurse into subdirectories unless specified not to

- Search for files name `test_*.py` or `*_test.py`

- From those files:

  - `test` prefixed test functions or methods outside of class

  - `test` prefixed test functions or methods inside `Test` prefixed test classes (without an `__init__` method)

- Also supports standard python `unittest`

Testing Flask Applications

- Create a `client` fixture - known to Flask

- Set up dummy database, temp dir, etc. in fixture

- Use `requests` library to generate queries

Fixture setup

```
import os
import tempfile
import pytest
from flaskr import create_app
from flaskr.db import init_db
```

```
@pytest.fixture
def client():
    db_fd, db_path = tempfile.mkstemp()
    app = create_app({'TESTING': True, 'DATABASE': db_path})

    with app.test_client() as client:
        with app.app_context():
            init_db()
        yield client

    os.close(db_fd)
    os.unlink(db_path)
```

Test example

```
def test_empty_db(client):
    """Start with a blank database."""
rv = client.get('/')
assert b'No entries here so far' in rv.data
```

Testing login and other features

```
def login(client, username, password):
    return client.post('/login', data = dict(
        username = username,
        password = password
    ), follow_redirects = True)

def logout(client):
    return client.get('/logout', follow_redirects = True)
```

```
def test_login_logout(client):
    """Make sure login and logout works."""
    username = flaskr.app.config["USERNAME"]
    password = flaskr.app.config["PASSWORD"]

    rv = login(client, username, password)
    assert b'You were logged in' in rv.data

    rv = logout(client)
    assert b'You were logged out' in rv.data

    rv = login(client, f"{username}x", password)
    assert b'Invalid username' in rv.data

    rv = login(client, username, f'{password}x')
    assert b'Invalid password' in rv.data
```

Evaluation

```
import pytest
import os.path

class TestWeek1PublicCases:
    # Test case to check if the contact.html file exists
    def test_public_case1(self, student_assignment_folder):
        file_path = student_assignment_folder + "contact.html"
        assert os.path.isfile(file_path) == True

    # Test case to check if the resume.html file exists
    def test_public_case5(self, student_assignment_folder):
        file_path = student_assignment_folder + "resume.html"
        assert os.path.isfile(file_path) == True
```

Summary

- Automated testing is essential to get confidence in design

- Regression testing:

  - ensure previously passed tests do not start failing

- Test generation process:

- mix of manual and automated

Continuous testing essential for overall system stability