Assignment 3
Application of Automated Test Generation in Practice

Casper Kristiansson

January 6, 2024

1 Comparison of Manually and Automatically Generated Tests

The code for both the manually created tests and the automatically generated test can be seen in section 3. When comparing the two different created test cases groups I want to mostly focuses on the coverage, functionality and maintainability/readability. If we start of with coverage we can use a tool like pytest-cov [1] to get the coverage. After manually creating the tests and than running the coverage I was able to on the first try be able to get a score of 100%. As mentioned in assignment 2 it was seen that the automatically generated test also got a score of 100% on the coverage.

A issue I talked about in assignment 2 is that the automatically generated tests sometimes didn't properly test certain functions. For example it never tried to manage actual balances in the bank account beside test case 6 which sent in a negative number. Instead the tests generated sent in Booleans instead of integers. While this works in python because Booleans will be converted to either 0 or 1.

Another issue is that the automatically generated tests didn't cross check information. For example in a test where it tries to withdraw money it only checked if the return message was "Withdrawal successful" and didn't check if the new balance is reflected. This means that if there is a bug with the amount withdrawn (instead of minus it could add balance) it would have sneaked past the tests.

Lastly a issue with the automatically generated tests is that the maintainability/readability is really hard. In certain of the automatically generated tests such as test 13 contains 7 different assert statements. If one of these would have failed it would be hard to track down why exactly it failed. The automatically generated tests also have bad variable names which makes it hard to track down what exactly is being tested.

2 Task 2

When comparing both manually created tests and automatically generated tests there are both strength and weaknesses in them both. The paper "A Comparative Study of Manual and Automated Testing for Industrial Control Software" [2] found that automatically generated tests achieved (avg 19min) a 90% in time reduction to test code than manually tested code (avg 165min)

The paper stated that both the manually tests and automatically generated tests had similar coverage but the automatically generated tests lacked certain types of fault detection. Manually tested code was much more effective in detecting logical, negation, and timer faults. The paper "A Controlled Experiment in Testing of Safety-Critical Embedded Software" [3] also highlights that manually tested code is much more effective in detecting faults in the code.

The studies suggested that while automatically generated tests can be effective in certain situations but due to better fault detection in manual testing it could be more attracting to use manual testing. This aligns a bit with mu conclusion in section 1 where it is highlighted that the automatically generated tests would have missed certain issues and errors in the code.

Overall most of the papers highlight that the testing procedure of code is faster using automated generated tests but they found that manually generated tests has a much better fault detection. What can be drawn from this is that code that often are under development or needs high fault tolerance should have either combined manually created tests with automated generated tests or just manually created tests.

While in this course I created a extremely simple class that was tested with both manually test and automatically generated tests the "real world" will consist of much more advanced examples. This means that in certain situations automatically generated tests might find faults that a human might have missed.

3 Listing of Tests

3.1 Manually Created Tests

```python
@pytest.fixture
def banking_system():
    return BankingSystem()

def test_create_account_success(banking_system):
    res = banking_system.create_account("001", "Casper", 100)
    assert res == "Account created successfully"

def test_create_account_negative_balance(banking_system):
    res = banking_system.create_account("002", "Casper", -100)
    assert res == "Initial balance must be non-negative"

def test_create_duplicate_account(banking_system):
    banking_system.create_account("003", "Casper", 50)
    res = banking_system.create_account("003", "Casper", 50)
    assert res == "Account already exists"

def test_deposit_success(banking_system):
    banking_system.create_account("004", "Casper", 100)
    res = banking_system.deposit("004", 50)
    assert res == "Deposit successful"

def test_deposit_non_existing_account(banking_system):
    res = banking_system.deposit("999", 50)
    assert res == "Account not found"

def test_deposit_negative_amount(banking_system):
    banking_system.create_account("005", "Casper", 100)
    res = banking_system.deposit("005", -50)
    assert res == "Deposit amount must be positive"

def test_withdraw_success(banking_system):
    banking_system.create_account("006", "Casper", 100)
    res = banking_system.withdraw("006", 50)
    assert res == "Withdrawal successful"

def test_withdraw_non_existing_account(banking_system):
    res = banking_system.withdraw("999", 50)
    assert res == "Account not found"

def test_withdraw_negative_amount(banking_system):
    banking_system.create_account("007", "Casper", 100)
    res = banking_system.withdraw("007", -50)
    assert res == "Withdrawal amount must be positive"

def test_withdraw_insufficient_funds(banking_system):
    banking_system.create_account("008", "Casper", 50)
    res = banking_system.withdraw("008", 100)
    assert res == "Insufficient funds"


def test_get_balance_existing_account(banking_system):
    banking_system.create_account("009", "Casper", 100)
    res = banking_system.get_balance("009")
    assert res == 100

def test_get_balance_non_existing_account(banking_system):
    res = banking_system.get_balance("999")
    assert res == "Account not found"

def test_get_balance_after_deposit(banking_system):
    banking_system.create_account("010", "Casper", 100)
    banking_system.deposit("010", 50)
    res = banking_system.get_balance("010")
    assert res == 150

def test_get_balance_after_withdrawal(banking_system):
    banking_system.create_account("011", "Casper", 100)
    banking_system.withdraw("011", 50)
    res = banking_system.get_balance("011")
    assert res == 50
```

3.2 Automated Generated Tests

```
@pytest.mark.xfail(strict=True)
def test_case_0():
    banking_system_0 = module_0.BankingSystem()
    banking_system_0.create_account(
        banking_system_0, banking_system_0, banking_system_0
    )

def test_case_1():
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.deposit(banking_system_0,
    banking_system_0)
    assert var_0 == "Account not found"

def test_case_2():
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.withdraw(banking_system_0,
    banking_system_0)
    assert var_0 == "Account not found"

def test_case_3():
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.get_balance(banking_system_0)
    assert var_0 == "Account not found"

def test_case_5():
    bool_0 = False
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(bool_0, bool_0, bool_0)
    assert var_0 == "Account created successfully"
    assert banking_system_0.accounts == {False: {"name": False, "
    balance": False}}

@pytest.mark.xfail(strict=True)
def test_case_6():
    none_type_0 = None
```

```python
    int_0 = -425
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(int_0, int_0, int_0)
    assert var_0 == "Initial balance must be non-negative"
    var_0.withdraw(none_type_0, none_type_0)

@pytest.mark.xfail(strict=True)
def test_case_7():
    bool_0 = False
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(bool_0,
    banking_system_0, bool_0)
    assert var_0 == "Account created successfully"
    assert len(banking_system_0.accounts) == 1
    var_1 = banking_system_0.deposit(bool_0, bool_0)
    assert var_1 == "Deposit amount must be positive"
    banking_system_0.deposit(bool_0, var_1)

def test_case_8():
    bool_0 = True
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(banking_system_0,
    bool_0, bool_0)
    assert var_0 == "Account created successfully"
    assert len(banking_system_0.accounts) == 1
    var_1 = banking_system_0.withdraw(banking_system_0, bool_0)
    assert var_1 == "Withdrawal successful"
    var_2 = banking_system_0.deposit(var_0, banking_system_0)
    assert var_2 == "Account not found"
    var_3 = banking_system_0.create_account(banking_system_0, var_0
    , bool_0)
    assert var_3 == "Account already exists"

@pytest.mark.xfail(strict=True)
def test_case_9():
    bool_0 = True
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(banking_system_0,
    bool_0, bool_0)
    assert var_0 == "Account created successfully"
    assert len(banking_system_0.accounts) == 1
    var_1 = banking_system_0.withdraw(banking_system_0, bool_0)
    assert var_1 == "Withdrawal successful"
    module_1.object(*var_0)

def test_case_10():
    bool_0 = False
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(bool_0, bool_0, bool_0)
    assert var_0 == "Account created successfully"
    assert banking_system_0.accounts == {False: {"name": False, "
    balance": False}}
    var_1 = banking_system_0.get_balance(bool_0)
    assert var_1 is False

@pytest.mark.xfail(strict=True)
def test_case_11():
    bool_0 = True
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.get_balance(bool_0)
    assert var_0 == "Account not found"
    var_1 = banking_system_0.create_account(bool_0,
    banking_system_0, bool_0)
    assert var_1 == "Account created successfully"
    assert len(banking_system_0.accounts) == 1
    var_2 = banking_system_0.deposit(bool_0, bool_0)
    assert var_2 == "Deposit successful"
    var_3 = module_1.object()
    var_2.create_account(var_2, var_2, var_2)

@pytest.mark.xfail(strict=True)
def test_case_12():
    bool_0 = False
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(banking_system_0,
    bool_0, bool_0)
    assert var_0 == "Account created successfully"
    assert len(banking_system_0.accounts) == 1
    var_1 = banking_system_0.withdraw(banking_system_0, bool_0)
    assert var_1 == "Withdrawal amount must be positive"
    banking_system_0.deposit(banking_system_0, banking_system_0)

@pytest.mark.xfail(strict=True)
def test_case_13():
    bool_0 = True
    banking_system_0 = module_0.BankingSystem()
    var_0 = banking_system_0.create_account(banking_system_0,
    bool_0, bool_0)
    assert var_0 == "Account created successfully"
    assert len(banking_system_0.accounts) == 1
    var_1 = banking_system_0.withdraw(banking_system_0, bool_0)
    assert var_1 == "Withdrawal successful"
    var_2 = banking_system_0.get_balance(var_0)
    assert var_2 == "Account not found"
    var_3 = banking_system_0.withdraw(banking_system_0, bool_0)
    assert var_3 == "Insufficient funds"
    var_4 = banking_system_0.deposit(var_2, banking_system_0)
    assert var_4 == "Account not found"
    var_5 = banking_system_0.deposit(bool_0, var_4)
    assert var_5 == "Account not found"
    var_4.withdraw(var_0, var_2)
```

References

[1] “Welcome to pytest-cov’s documentation! — pytest-cov 4.1.0 documentation,” https://pytest-cov.readthedocs.io/en/latest/, (Accessed on 01/06/2024).

[2] E. Enoiu, D. Sundmark, A. Čaušević, and P. Pettersson, “A comparative study of manual and automated testing for industrial control software,” in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2017, pp. 412–417.

[3] E. P. Enoiu, A. Cauevic, D. Sundmark, and P. Pettersson, “A controlled experiment in testing of safety-critical embedded software,” in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 1–11.