Assignment 2
Practical Exercise on Automated Test Design Techniques

Casper Kristiansson

January 6, 2024

1 Taxonomy

I choose to continue with the automated test generation tool Pynguin. I think the specific details in the taxonomy are still sufficient and don't need any modifications.

1. **Software Artifact**
   (a) **Implementation**: Source, generates test based on python source code.
   (b) **Characteristics**: Application/System because it generates tests for Python applications/systems
   (c) **Programming Language**: Python
2. **Implementation**
   (a) **Code**: Method level because the testing tools generate unit tests for different methods
   (b) **Monitoring**: Dynamic, because it executes the code and then monitors its behavior
3. **Test Generation**
   (a) **Objective**: Code coverage, "...aims to automatically generate unit tests that maximize code coverage." [1].
   (b) **Technology**: Search-Based Testing, Chapter 3 [1].
4. **Test Execution**
   (a) **Online/Offline**: Offline
5. **Test Oracle**
   (a) **Categories**: Manual, "So far, Pynguin focuses on test-input generation and excludes the generation of oracles." [1].

2 Example Program

I chose to create a simple class in Python which represents a simple bank system. The class is able to create multiple accounts, get balance, withdraw money, and lastly deposit money. In each function, I handle simple error cases such as the account doesn't exist or insufficient balance during withdrawal.

```python
class BankingSystem:
    def __init__(self):
        self.accounts = {}

    def create_account(self, account_number, name, initial_balance)
    :
        if account_number in self.accounts:
            return "Account already exists"
        if initial_balance < 0:
            return "Initial balance must be non-negative"
        self.accounts[account_number] = {
            "name": name,
            "balance": initial_balance
        }
        return "Account created successfully"

    def deposit(self, account_number, amount):
        if account_number not in self.accounts:
            return "Account not found"
        if amount <= 0:
            return "Deposit amount must be positive"
        self.accounts[account_number]["balance"] += amount
        return "Deposit successful"

    def withdraw(self, account_number, amount):
        if account_number not in self.accounts:
            return "Account not found"
        if amount <= 0:
            return "Withdrawal amount must be positive"
        if amount > self.accounts[account_number]["balance"]:
            return "Insufficient funds"
        self.accounts[account_number]["balance"] -= amount
        return "Withdrawal successful"

    def get_balance(self, account_number):
        if account_number not in self.accounts:
            return "Account not found"
        return self.accounts[account_number]["balance"]
```

3 Automated Generated Test Cases

When using the tool Pynguin tool to automatically generate tests we get some interesting test cases generated. The tool generated a total of 13 test cases. Each test case tests the expected behaviour and after running a coverage test on the test cases it got a score of 100%. While it is not mentioned in the assignment I

want to quickly note the problem with the generated test cases. For the test case 9 (test_case_9) the test tries to withdraw money. A good testing behaviour for this would be to withdraw and than check that the new balance has the correct amount. But rather than withdrawing a proper amount (ex 100) it withdraws using True. While this works in python (adding True to a int results in True being 1) it doesn't really test the true behaviour of the class. Here are a few note worthy test cases generated:

```
@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_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
    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_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)
```

References

[1] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 2020, pp. 9–24.