Iftichi Albert Ioan Calin

Grupa 342

Raport: Impactul Dezvoltării Bazate pe Teste (TDD) asupra Calității Software-ului: Un Studiu Empiric

Introducere

Acest raport prezintă pe scurt conceptele și metodele abordate în articolul "The Impact of Test-Driven Development on Software Quality: An Empirical Study", publicat în jurnalul VFAST Transactions on Computer Sciences. Studiul explorează modul în care Test-Driven Development (TDD) influențează calitatea software-ului și oferă perspective interesante despre beneficiile și provocările acestei metodologii.

Concepte Cheie

1. **Ce este Dezvoltarea Bazată pe Teste (TDD)**:
    - TDD este o metodă de dezvoltare software unde testele sunt scrise înaintea codului efectiv. Procesul include cicluri scurte de dezvoltare, fiecare începând cu scrierea unui test care descrie o nouă funcționalitate sau îmbunătățire.
    - După scrierea testului, se scrie minimul de cod necesar pentru a trece testul. Odată ce testul este trecut, codul este îmbunătățit fără a-i modifica comportamentul.
2. **Calitatea Software-ului**:
    - Articolul examinează calitatea software-ului folosind diverse măsuri precum numărul de erori (defecte), ușurința de întreținere a codului și timpul de dezvoltare.
    - Calitatea software-ului se referă la reducerea numărului de erori și la creșterea ușurinței cu care codul poate fi înțeles, modificat și extins.

Metodele Studiului

3. **Colectarea Datelor**:
    - Studiul a adunat date din proiecte software reale în care s-a folosit TDD. Aceste date au fost comparate cu proiecte similare care nu au folosit TDD pentru a vedea diferențele în calitatea software-ului.
    - Dezvoltatorii au fost intervievați și li s-au aplicat chestionare pentru a obține informații despre experiențele lor cu TDD.

Iftichi Albert Ioan Calin

Grupa 342

4. **Metrici de Evaluare**:
    - **Numărul de Defecte**: Numărul de erori în cod. Mai puține erori înseamnă o calitate mai bună.
    - **Ușurința de Întreținere**: Cât de ușor poate fi modificat codul. Asta include claritatea și structura codului.
    - **Timpul de Dezvoltare**: Timpul necesar pentru a dezvolta funcționalități folosind TDD comparativ cu metodele tradiționale.
5. **Analiza Comparativă**:
    - Proiectele dezvoltate cu TDD au fost comparate cu proiecte similare dezvoltate prin metode tradiționale pentru a evalua diferențele în calitatea software-ului.
    - Tehnici statistice au fost folosite pentru a analiza datele și a determina semnificația diferențelor observate.

Concluzii ale Studiului

6. **Beneficiile TDD**:
    - **Mai puține Erori**: Proiectele care au folosit TDD au avut un număr mai mic de erori comparativ cu cele dezvoltate prin metode tradiționale.
    - **Întreținere mai Ușoară**: Codul dezvoltat cu TDD a fost mai ușor de înțeles și de modificat, ceea ce înseamnă o mentenabilitate mai mare.
    - **Calitate mai Bună**: Dezvoltatorii au raportat că TDD a dus la un cod de calitate superioară, datorită ciclurilor de feedback rapide și testării continue.
7. **Provocările TDD**:
    - **Timp Inițial mai Mare**: TDD necesită mai mult timp la început, deoarece scrierea testelor și îmbunătățirea codului sunt etape suplimentare.
    - **Curba de Învățare**: Adoptarea TDD necesită o schimbare în modul de gândire al dezvoltatorilor și poate necesita timp pentru a deveni eficienți.
8. **Recomandări**:
    - **Adoptare treptată**: Este recomandată adoptarea treptată a TDD pentru a permite dezvoltatorilor să se obișnuiască cu noua metodologie.
    - **Training și Suport**: Asigurarea de training și resurse pentru dezvoltatori pentru a le facilita tranziția la TDD.
    - **Monitorizare Continuă**: Monitorizarea continuă a calității pentru a evalua impactul TDD și pentru a face ajustări necesare în procese.

Iftichi Albert Ioan Calin

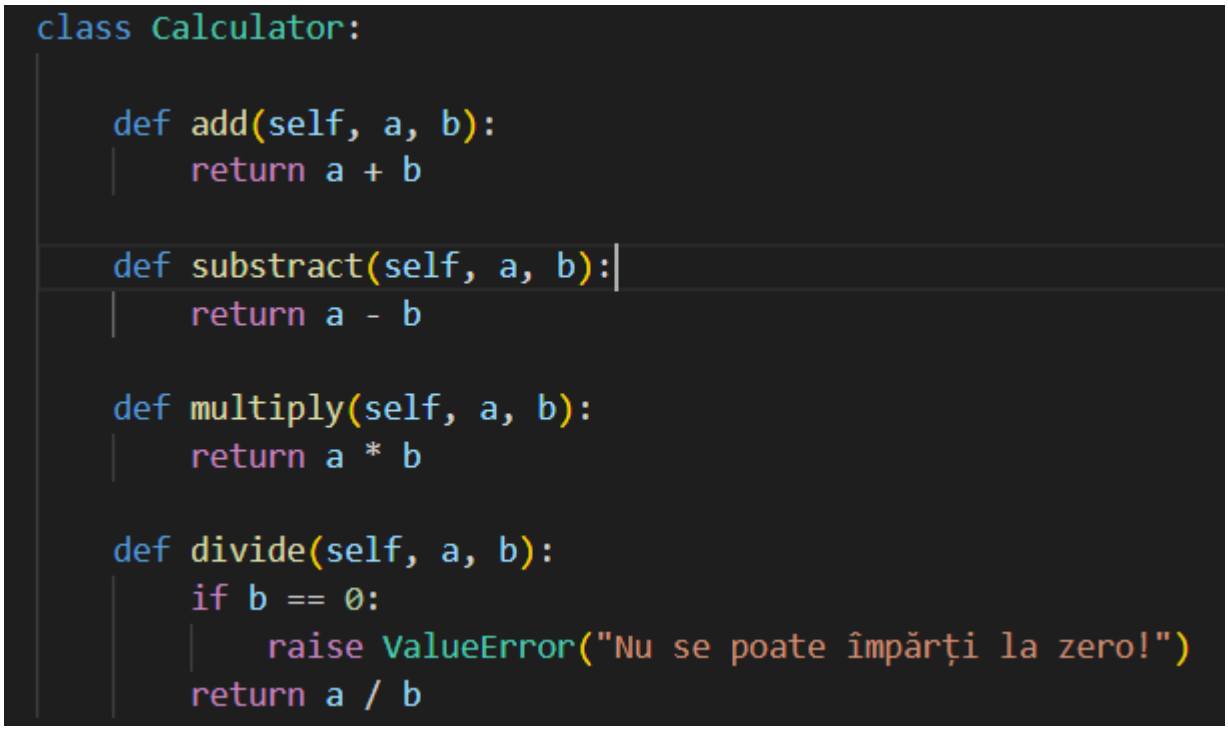
Grupa 342

Concluzie

Articolul "The Impact of Test-Driven Development on Software Quality: An Empirical Study" oferă o perspectivă detaliată asupra beneficiilor și provocărilor TDD. Studiul demonstrează că, deși TDD poate necesita o investiție inițială mai mare în timp și resurse, beneficiile pe termen lung, cum ar fi calitatea superioară a codului și întreținerea mai ușoară, fac din aceasta o metodologie valoroasă pentru dezvoltarea software-ului. Aceste descoperiri sunt esențiale pentru a înțelege impactul TDD și pentru a încuraja adoptarea sa în practici de dezvoltare software.

Demo aplicatie

Am definit o clasă numită "Calculator" care conține patru metode pentru efectuarea operațiilor matematice de adunare, scădere, înmulțire și împărțire. Fiecare metodă primește două argumente, a și b, reprezentând numerele pe care se efectuează operația.



```python
class Calculator:

    def add(self, a, b):
        return a + b

    def substract(self, a, b):
        return a - b

    def multiply(self, a, b):
        return a * b

    def divide(self, a, b):
        if b == 0:
            raise ValueError("Nu se poate împărți la zero!")
        return a / b
```

Iftichi Albert Ioan Calin

Grupa 342

1. Metoda `add(self, a, b)` adună cele două numere primite ca argumente și returnează rezultatul adunării.

2. Metoda `substract(self, a, b)` scade al doilea număr (b) din primul număr (a) și returnează rezultatul scăderii.

3. Metoda `multiply(self, a, b)` înmulțește cele două numere primite ca argumente și returnează rezultatul înmulțirii.

4. Metoda `divide(self, a, b)` împarte primul număr (a) la al doilea număr (b), cu condiția ca b să nu fie zero (pentru a evita împărțirea la zero) și returnează rezultatul împărțirii.

Apoi, într-un alt fișier, am importat clasa `Calculator` din fișierul "calculator.py" și am instanțiat un obiect de tip `Calculator` numit `calc`. Ulterior, am apelat metodele definite în cadrul clasei `Calculator` pentru a efectua operații matematice și a afișa rezultatele acestora.

```
from calculator import Calculator

calc = Calculator()

print(calc.add(1, 2))         # Output: 3
print(calc.substract(5, 3))    # Output: 2
print(calc.multiply(4, 3))    # Output: 12
print(calc.divide(10, 2))     # Output: 5
```

În exemplul de utilizare, sunt efectuate următoarele operații și se afișează rezultatele:

- Adunarea a două numere (1 și 2), rezultând 3.

- Scăderea unui număr (3) din altul (5), rezultând 2.

- Înmulțirea a două numere (4 și 3), rezultând 12.

- Împărțirea unui număr (10) la altul (2), rezultând 5.

Iftichi Albert Ioan Calin

Grupa 342

În cazul împărțirii, se afișează un mesaj de eroare în cazul împărțirii la zero.

```
PS C:\Users\kln> & C:/Users/kln/anaconda3/python.exe c:/Users/kln/Desktop/ProiectTSS/main.py
3
2
12
5.0
```

În continuare am realizat două fișiere pentru teste unitare pentru clasa `Calculator` definită în primul cod. Testele unitare sunt utilizate pentru a verifica corectitudinea funcționării metodelor dintr-o clasă.

```python
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        self.assertEqual(self.calc.add(4, 6), 10)
        self.assertEqual(self.calc.add(-1, 2),1)
        self.assertEqual(self.calc.add(-3, 5), 2)

    def test_substract(self):
        self.assertEqual(self.calc.substract(12, 5), 7)
        self.assertEqual(self.calc.substract(-1, 6), -7)
        self.assertEqual(self.calc.substract(-1, -4), 3)

    def test_multiply(self):
        self.assertEqual(self.calc.multiply(9, 8), 72)
        self.assertEqual(self.calc.multiply(-1, 1), -1)
        self.assertEqual(self.calc.multiply(-1, -1), 1)

    def test_divide(self):
        self.assertEqual(self.calc.divide(24, 6), 4)
        self.assertEqual(self.calc.divide(-1, 1), -1)
        self.assertEqual(self.calc.divide(-1, -1), 1)
        with self.assertRaises(ValueError):
            self.calc.divide(10, 0)

if __name__ == '__main__':
    unittest.main()
```

Iftichi Albert Ioan Calin

Grupa 342

```python
class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        self.assertEqual(self.calc.add(1, 2), 3)
        self.assertEqual(self.calc.add(0, 0), 0)
        self.assertEqual(self.calc.add(-1, -1), -2)
        self.assertEqual(self.calc.add(-1, 1), 0)
        self.assertEqual(self.calc.add(1000, 2000), 3000)
        self.assertEqual(self.calc.add(1.5, 2.5), 4.0)

    def test_substract(self):
        self.assertEqual(self.calc.substract(10, 5), 5)
        self.assertEqual(self.calc.substract(0, 0), 0)
        self.assertEqual(self.calc.substract(-1, -1), 0)
        self.assertEqual(self.calc.substract(-1, 1), -2)
        self.assertEqual(self.calc.substract(2000, 1000), 1000)
        self.assertEqual(self.calc.substract(2.5, 1.5), 1.0)

    def test_multiply(self):
        self.assertEqual(self.calc.multiply(3, 7), 21)
        self.assertEqual(self.calc.multiply(0, 0), 0)
        self.assertEqual(self.calc.multiply(-1, -1), 1)
        self.assertEqual(self.calc.multiply(-1, 1), -1)
        self.assertEqual(self.calc.multiply(100, 200), 20000)
        self.assertEqual(self.calc.multiply(1.5, 2.0), 3.0)

    def test_divide(self):
        self.assertEqual(self.calc.divide(10, 2), 5)
        self.assertEqual(self.calc.divide(-1, -1), 1)
        self.assertEqual(self.calc.divide(-1, 1), -1)
        self.assertEqual(self.calc.divide(2000, 1000), 2)
        self.assertEqual(self.calc.divide(4.5, 1.5), 3.0)
        with self.assertRaises(ValueError):
            self.calc.divide(10, 0)

if __name__ == '__main__':
    unittest.main()
```

Fiecare test este o metodă definită în clasa `TestCalculator`, care este o subclasă a `unittest.TestCase`. Metoda `setUp` este folosită pentru a inițializa un obiect `Calculator` înainte de fiecare test, asigurând astfel un mediu pentru testare.

În interiorul fiecărei metode de test, sunt apelate metodele corespunzătoare ale obiectului `Calculator`, iar rezultatele sunt verificate folosind *assertEqual*. De exemplu,

Iftichi Albert Ioan Calin

Grupa 342

`self.assertEqual(actual, expected)` compară valoarea returnată de metodă cu rezultatul așteptat.

Pentru fiecare operație matematică (adunare, scădere, înmulțire și împărțire), sunt testate mai multe scenarii pentru a acoperi diverse cazuri. De exemplu, în testul pentru împărțire, se verifică atât rezultatele corecte când împărțitorul nu este zero, cât și cazul în care împărțitorul este zero, în care se așteaptă să fie ridicată o excepție de tip `ValueError`.

În aceste exemple, sunt acoperite diverse scenarii, inclusiv operații cu numere întregi și cu virgulă, precum și diverse combinații de numere pozitive și negative.

[1] Dua Agha, Rashida Sohail, Ramsha Qaboolio, Sania Bhatti, *Test Driven Development and Its Impact on Program Design and Software Quality: A Systematic Literature Review,* https://www.vfast.org/journals/index.php/VTCS/article/view/1494/1228, 09.06.2024.

Iftichi Albert Ioan Calin

Grupa 342

Mai jos avem un exemplu în care algoritmul reușește să treacă toate testele:

Observăm că dacă adăugăm un test pentru care valoarea returnată de metoda din clasa Calculator este diferită de cea reală, testul va genera o eroare. Pentru aceasta, am adăugat la linia 22 un test conform căruia, prin înmulțirea lui 5 cu 6 așteptăm rezultatul 1, astfel generând un fail.

Iftichi Albert Ioan Calin

Grupa 342

```python
import unittest
from calculator import Calculator

class TestCalculator(unittest.TestCase):
    def setUp(self):
        self.calc = Calculator()

    def test_add(self):
        self.assertEqual(self.calc.add(4, 6), 10)
        self.assertEqual(self.calc.add(-1, 2),1)
        self.assertEqual(self.calc.add(-3, 5), 2)

    def test_substract(self):
        self.assertEqual(self.calc.substract(12, 5), 7)
        self.assertEqual(self.calc.substract(-1, 6), -7)
        self.assertEqual(self.calc.substract(-1, -4), 3)

    def test_multiply(self):
        self.assertEqual(self.calc.multiply(9, 8), 72)
        self.assertEqual(self.calc.multiply(-1, 1), -1)
        self.assertEqual(self.calc.multiply(-1, -1), 1)
        self.assertEqual(self.calc.multiply(5, 6), 1)

    def test_divide(self):
        self.assertEqual(self.calc.divide(24, 6), 4)
        self.assertEqual(self.calc.divide(-1, 1), -1)
        self.assertEqual(self.calc.divide(-1, -1), 1)
        with self.assertRaises(ValueError):
            self.calc.divide(10, 0)

if __name__ == '__main__':
    unittest.main()
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

======================================================================
FAIL: test_multiply (__main__.TestCalculator)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "c:\Users\kln\Desktop\ProiectTSS\test_calculator.py", line 22, in test_multiply
    self.assertEqual(self.calc.multiply(5, 6), 35)
AssertionError: 30 != 35

----------------------------------------------------------------------
Ran 4 tests in 0.001s

FAILED (failures=1)
PS C:\Users\kln>
```

Prin rularea acestui set de teste, putem să ne asigurăm că metodele din clasa `Calculator` funcționează corect conform așteptărilor noastre, ajutând la detectarea și remedierea eventualelor erori sau bug-uri.