Testování softwaru (KI/TSW)

Pavel Beránek

Únor 26, 2025

Seminář 3 - Extremní programování a vývoj softwaru řízený testování

Úvod

Cílem tohoto semináře je seznámit se s typy procesů vývoje, které se nazývají vývoj softwaru řízený testy. Dalším cílem je seznámit se s testovými scénáři a jejich technickou realizací pomocí BDD testu.

Vývoj softwaru řízený testy je přístup k vývoji, ve kterém testy představují prostředek řízení vývoje. Techniky vývoje řízeného testováním zahrnují:

1. Vývoj řízený testy (TDD, Test-driven development) - přístup k vývoji, kde píšeme standardní automatizované testy (ty z minulé lekce) a až poté implementační kód. Využívá se semaforové pomůcky řídící proces vývoje. Cílem tohoto přístup je zajistit od počátku testovatelný kód, zvýšit sebedůvěru programátora v implementaci, zlepšit kvalitu kódu.
2. Vývoj řízený chováním (BDD, Behavior-driven development) - přístup k vývoji, kde se píšou speciální automatizované testy v doménově specifickém jazyce Gherkin. Gherkin popisuje testovací scénář s jeho vstupními podmínkami, akcí a očekávaným následkem. Cílem je zvýšit usnadit komunikaci mezi vývojáři, testery a zástupci businessu.
3. Vývoj řízený akceptačními testy (ATDD, Acceptace test-drivein development) - přístup k vývoji, ve kterém se nejprve definují požadavky na přijetí softwaru klientem ve formě akceptačních testů. Technicky se akceptační testy mohou realizovat jako BDD testy, ale bývají většinou komplikovanější. Pro takové testy je lepší využívat například Selenium pro testování webových stránek a organizovat si testy v Robot frameworku. Cílem je zajistit, že vývoj odpovídá podnikovým požadavkům. Toto testování budeme dělat v kursu mnohem později.

Tyto metodiky jsou používané ve vývojových procesech jako je například Extrémní programování (XP). Extrémní programování je metodika zaměřená na získávání zpětné vazby. TDD je hlavní strategie vývoje a zajišťuje rychlou zpětnou vazbu vývojářům v rámck sekund. Ačkoliv XP se již dnes příliš nepoužívá, tak ostatní metodiky vývoje si vypůjčují jeho techniky (párové programování, společné vlastnictví kódu, TDD atd.). Proto se s TDD setkáme občas i u Scrumu nebo vodopádového procesu vývoje softwaru.

Úkol 1 - Vývoj řízený testy (TDD)

TDD řeší efektivně problém nejasných specifikací testových případů. Specifikace pro kalkulačku s popisem "Kalkulačka musí umět sčítat" je pro programátora velmi nejasné, i když se zadavateli musí zdát zcela jednodušše pochopitelné. Programátor si totiž může klást následující otázky:

1. Kolik čísel má kalkulačka sčítat najednou? Obyčejně to jsou 2, ale může to být i více.
2. Jak se má kalkulačka chovat při zadání jen jednoho čísla do sčítací metody?
3. Jaké datové typy má kalkulačka umět sčítat? Má například při řetězových vstupech provádět připojování a při logických vstupech logický součet?
4. Existuje nějaký speciální formát dat, který bude do kalkulačky vstupovat? Například dvojice hodnota-měna?
5. Jak se má chovat při extrémních hodnotách? Co bude výsledkem hodnot blízkých nekonečnu?
6. Jaká je požadovaná přesnost?
7. Pokud existuje potřeba efektivního sčítání velkého seznam čísel, máme to dělat vektorově (modul numpy)?
8. Má sčítací funkce provádět nějaké logovací operace?

Takových otázek bychom vymysleli ještě více a nejistota jen narůstá.

TDD tento problém efektivně řeší tím, že testy vznikají dříve než samotný programovací kód. Pokud jsou vytvořeny dlouho předem, tak je možné je předat IT analytikovi pro vyjednávání s klientem nebo minimálně vývojáři, která má ponětí o architektuře softwaru a následné integraci třídy do jiných tříd. Pokud vznikají za běhu během kódování, tak alespoň dodávají důvěru v kód samotnému programátorovi, že plní jasně sepsané chování. TDD mantra zní: "Testy prvně, kód následně.", což lze také překládat do business jazyka jako "Cíl prvně, řešení následně.". Tuto mantru dodrží i jiné obory, jako je třeba marketing, který využívá termín KPI (key performance indicator, klíčové ukazatele výkonnosti). KPI je například: "Počet organických nových nálezů naší Instagramové stránky pomocí kampaně bude minimálně 1000 za jeden týden".

Mějme následující testovací případ (test case) realizovaný jednotkových testem (unit test) napsaným pomocí rámce AAA (arrange-act-assert):

```
def test_sum_two_pos_integers(calc):
    a, b, expected = 5, 3, 8
    outcome = calc.add(a, b)
    assert outcome == expected
```

Tento test zkoumá chováná objektu kalkulačky při sčítání. Jasně definuje, že při pokusu o součet 5 a 3 se má vrátit 8. Testo test může existovat dříve než samotný kód kalkulačky.

Podobně mohu napsat další test aniž bych stále psal kód kalkulačky (to je i důkaz toho, že testovací aktivity mohou probíhat i v raných dobách vývoje (analýza požadavků, návrh architektury).

Pokud využíváme TDD tak, že testy vyvíjí programátor během fáze programování, pak TDD má svůj vlastní proces psaní testů a následného kódu. Využívá k tomu model semaforu s barvami:

- Red = první fáze TDD procesu, napíšeme test specifikující funkční požadavek. Test po spuštění neprojde, protože neexistuje kód, který by v Act fázi AAA navrátil správný výsledek.
- Green = druhá fáze TDD procesu je napsaní kódu do testované jednotky tak, aby test prošel. Cílem není napsat komplexní kód, cílem je napsat kód, který projde testem (i triviální).
- Refakctor = poslední fáze TDD procesu, ve které se napsaná kód, který plní test, může upravit (generalizace, lepší algoritmizace apod). Pokud refaktorizací přestaneme plnit funkční požadavek testu, tak test to zachytí.

Tento cyklus se opakuje. Píšeme buď další test ke stejné testované jednotce testovaného objektu nebo píšeme kód pro novou testovanou jednotku. Typicky budete pokračovat v testování stejné jednotky do té doby, dokud jste nepokryli Vámi identifikované domény ekvivalence.

Pojďme si vyzkoušet tento proces. Nejprve napíšeme test, který selže (červená barva na semaforu):

```
def test_list_sum(calc):
    a, expected = [2, 3, 5], 10
    outcome = calc.list_sum(a)
    assert outcome == expected
```

Kód spustíme a zjistíme, že selže. Dostali jsme se úspěšně do červené barvy na semaforu.

Následně napíšeme kód, který test splňuje (i triviální), abychom se dostali z červené do zelené barvy.

```
class Calc:
    def list_sum(self, vals):
        return 10
```

Spustíme test a zjistíme, že úspěšně prošel. Dostali jsme se úspěšně z červené do zelené barvy. To, že funkce vrací číslo 10, je v pořádku. V praxi byste takto hloupý kód nenapsali. Chci však demonstrovat možné triviální řešení, které je podle TDD procesu v pořádku.

Teď je čas na refaktorizaci. Ta není povinná, proto ji vynechávám.

Jdeme dále vyzkoušet nový testový případ.

```
def test_list_sum_strings(calc):
    a, expected = ["2", "5", "7"], 14
    outcome = calc.list_sum(a)
    assert outcome == expected
```

Spustíme testy, které nás z barvy refactor přivedou do červené barvy. Naším cílem je dostat se do barvy zelené.

```
def list_sum(self, vals):
    return sum(list(map(int, vals)))
```

Po spuštění jsme se dostali do barvy zelené a můžeme refaktorovat. Tentokrát refaktorovat budu:

```
def list_sum(self, vals):
    int_vals = map(int, vals)
    sum_result = sum(int_vals)
    return sum_result
```

S refaktorizací jsem již spokojen a mohu začít psát další testovací případ pro stejnou testovanou jednotku (sčítací funkce) nebo můžu začít testovat jinou jednotku na stejném testovaném objektu nebo novém testovaném objektu.

```
def test_list_mul(calc):
    a, expected = [2, 3, 4], 24
    outcome = calc.list_mul(a)
    assert outcome == expected
```

V minulé lekci jste se učili psát parametrizované testy, takže testové případy můžete psát jako parametry testovací funkce.

Vyzkoušejte si proces TDD:

1. Napište si testovaný objekt, který bude třída Kalkulačka.
2. Vytvořte první test z mých příkladů pro hromadné násobení a dejte instanci třídy kalkulačka jako parametr testovací funkce ve formě fixture.
3. Spusťte test a dostaňte se do červené barvy na semaforu.
4. Upravte kód metody pro hromadné násobení tak, aby prošel mým testem a dostaňte se tak do zelené barvy.
5. Refaktorujte nebo nerefaktorujte. Refakrotizační úprava musí projít existujícím testem.
6. Napište testový případ pro testování násobení záporných čísel. Zkuste, zda se dostanete do zelených světel.
7. Pokud jste v zelených světlech, můžete refaktorovat.
8. Až budete spokojeni, napište test pro násobení čísel s tím, že v číslech se objeví nekonečno (math.inf). Definujte si cílové chování testem.
9. Upravte kód tak, aby prošel testy. Refaktorujte.
10. Přidejte další testovací metodu. Nechám na Vaší kreativitě.

Řešení

Bude dodáno.

Úkol 2

Vyzkoušejte celý proces bez mého procesního návodu:

1. Vytvořte třídu BankAccount, která bude mít tyto metody. Nepište implementaci metod, nechte je prázdné:

- `deposit(amount: float) -> None`
- `withdraw(amount: float) -> bool`
- `get_balance() -> float`

2. Pomocí TDD napište postupně jeden test za druhým, doplňujte kód a refaktorujte pomocí semaforového principu.

Řešení

```
# vy budete mít více testů, uvádím jen pár pro technickou kontrolu z důvody úspory místa
import pytest
from app import BankAccount

@pytest.fixture
def account():
    return BankAccount()

def test_initial_balance(account):
    expected = 0
    outcome = account.get_balance()
    assert outcome == expected

def test_deposit(account):
    deposit_amount, expected_balance = 100, 100
    account.deposit(deposit_amount)
    outcome = account.get_balance()
    assert outcome == expected_balance

def test_withdraw_success(account):
    deposit_amount = 200
    withdraw_amount = 100
    expected_balance = 100
    expected_state = True
    account.deposit(deposit_amount)
    outcome_state = account.withdraw(withdraw_amount)
    outcome_balance = account.get_balance()
    assert outcome_state == expected_state
    assert outcome_balance == expected_balance

def test_withdraw_fail(account):
    withdraw_amount = 50
    expected_state = False
    expected_balance = 0
    outcome_state = account.withdraw(withdraw_amount)
    outcome_balance = account.get_balance()
    assert outcome_state == expected_state
    assert outcome_balance == expected_balance
```

Úkol 3 - Testování případů pomocí TDD pro webové stránky

Kalkulačka je triviální příklad testovacího objektu se kterým se příliš v ostré praxi nesetkáte. Python se používá primárně pro datovou analýzu, strojové učení a webové stránky. Pojďme si tedy zkusit testovat pomocí TDD webový server napsaný v pracovním rámci Flask. Jelikož není v tomto kurzu prostor vysvětlovat vývoj webových aplikací, omezíme se na triviální případy webových stránek a mechanismu jejich návratu. S takovým způsobem psaní webových serverů se doufám v praxi nikdy nesetkáte.

Prvně si do virtuálního prostředí nainstalujte modul flask: `pip install flask`. Tento modul nám poslouží pro psaní webových portálů v jazyce Python. Následně si nainstalujte modul `beautifulsoup4` pro dotazování se nad obsahem webových stránek. Dále si vytvořte v projektovém adresáři soubor app.py, ve kterém bude následující kód:

```
from flask import Flask, render_template, jsonify


app = Flask(__name__)


@app.route('/home')
def homepage():
    return render_template('home.html')


@app.route('/api/author')
def api_author():
    return jsonify({'author': 'John Doe'})


if __name__ == '__main__':
  app.run(port=8050, debug=True)
```

Dále si ve stejném pracovním adresáři vytvořte adresář templates a v něm vytvořte soubor s názvem 'home.html'. Vložte do něj následující kód:

```
<!DOCTYPE html>
<html>
  <head>
    <title>UJEP Web</title>
  </head>
  <body>
    <h1>Best Website</h1>
    <p>This is the best website ever!</p>
  </body>
</html>
```

Do webového prohlížeče zadejte URL adresu `localhost:8050/home`, což zobrazí webovou stránku s nadpisem první úrovně (HTML značka h1) "Best website".

Testovací případ můžeme jednotkovým testem napsat tímto způsobem (třeba do souboru test.py do pracovního adresáře nebo si to můžete chytře organizovat do adresářů):

```
import pytest
import json
from app import app as flask_app


@pytest.fixture
def app():
    yield flask_app
```

```
@pytest.fixture
def client(app):
    return app.test_client()


def test_home_rescode(app, client):
    expected = 200
    outcome = client.get('/home').status_code
    assert outcome == expected


def test_home_content(app, client):
    expected == ""


def test_api_author(app, client):
    expected == {'author': 'John Doe'}
    outcome = json.load()
```

Řešení

lorem

Úkol 4 - Vývoj řízený chováním (BDD)

Pojďme si shrnout vlastnosti testů v předchozím příkladu.

- Formát zadání: testovací případ (vstup, očekávaný výstup)
- Testovaný objekt: testovací jednotka (funkce nebo třída)
- Typ návrhu testu: testování bílou skříňkou (navrhuji test podle struktury kódu)
- Důvod testování: verifikace (ověřuji, že programuji správně)

Tyto testy tedy pomáhají primárně programátorům při psaní kódu zajistit důvěru v naprogramovanou jednotku, tedy, že minimalizují bugy a snižují nejistotu specifikací. Z toho důvodu se verifikační testy jednotek berou jako slušnost každého programátora a málokdy to přenecháváme na dedikovaného testera. Ten se stará o složitější testy.

Testeři typicky pracují s testovacími scénáři, které představují složitější situace, ve kterých uživatel v nějaké situaci koná akci na testovaném objektu a očekává výsledek. Scénáře představují vysokoúrovňové popisy, které se pak převádí na testovací případy a ty tester testuje, čímž potvrzuje splněný scénář. Tyto testy mají za cíl ověřit, že software se chová podle zadání a nemá tedy testovat, zda programátor dobře programuje. Vznikají často na základě Dokumentace specifikací požadavků z fáze Inženýrství požadavků ve vývojovém procesu softwaru.

Ideální technologií pro testování scénářů jsou BDD testy. BDD testy mohou pokrýt testové případy pro testování jednotek, ale i komplikovanější případy včetně celých scénářů. Propojují svět podnikových potřeb se světem programování. Takové testy umožňují jak verifikaci testovaných jednotek, tak validace testovaných scénářů. Není to tedy tak, že BDD slouží jen pro scénáře. Chápejme je jako komplexnější a silnější testovací nástroj. Vzhledem k tomu trvá jejich napsání i podstatně delší dobu a proto je málokdy píšou programátoři pro verifikaci.

Pytest má plugin `pytest-bdd`, který si musíte nainstalovat. Tento plugin využívá pro psaní BDD testů doménově-specifický jazyk Gherkin, který strukturuje testovací scénář do 3 kroků

1. `Given` = kontextu testu, co vše musí platit, než test začne
2. `When` = specifikací akcí, které se provedou
3. `Then` = očekáváný výsledek provedení akce

Výsledků může být více a propojují se logickým operátorem `And`. Stejně tak lze kontext rozdělit do více `Given` struktur a propojit je `And` prvkem testu a stejně tak akční prvek.

```
Given (kontext),
And (dodatečný kontext),
When (prvotní akce),
And (následná akce),
Then (očekávaný výsledek)
And (další očekávaný výsledek)
```

SEM JESTE NECO DOPISU

Vytvořte si soubor s názvem `bank_account.feature` a vložte do něj následující test v jazyce Gherkin:

```
Feature: Bank Account

    Scenario: Vklad peněz na účet
        Given nový bankovní účet
        When vložím 200 Kč
        Then zůstatek je 200 Kč
```

Následně vytvořte soubor s názvem `test_bank_account_bdd.py` a vložte do něj následující BDD test využívající předchozí scénář:

```
import pytest
from pytest_bdd import scenarios, given, when, then
from app import BankAccount

# Načtení scénářů
scenarios('bank_account.feature')
```

```
@pytest.fixture
def account():
    return BankAccount()

@given('nový bankovní účet')
def new_account():
    pass #již realizováno pomocí fixture

@when('vložím 200 Kč')
def deposit_money(account):
    account.deposit(200)

@then('zůstatek je 200 Kč')
def check_balance_200(account):
    assert account.get_balance() == 200
```

Vyřešte následující úkoly:

1. Přidejte scénář pro úspěšný výběr peněz. Počáteční předpoklad je existující bankovní účet se zůstatkem 300 Kč, akcí bude výběr 100 Kč a očekávaný stav bude zůstatek 200 Kč.
2. Napište BDD test realizující takový scénář.
3. Přidejte scénář pro neúspěšný výběr peněz. Počáteční předpoklad je nový bankovní účet, akcí bude výběr 500 Kč a očekávaný stav bude zůstatek 0 Kč.
4. Napište BDD test realizující takový scénář.

Řešení

```
Feature: Bank Account

    Scenario: Vklad peněz na účet
        Given nový bankovní účet
        When vložím 200 Kč
        Then zůstatek je 200 Kč

    Scenario: Úspěšný výběr peněz
        Given bankovní účet se zůstatkem 300 Kč
        When vyberu 100 Kč
        Then zůstatek je 200 Kč

    Scenario: Neúspěšný výběr peněz
        Given nový bankovní účet
        When vyberu 500 Kč
        Then zůstatek je 0 Kč
```

```
import pytest
from pytest_bdd import scenarios, given, when, then
from app import BankAccount

# Načtení scénářů
scenarios('bank_account.feature')

@pytest.fixture
def account():
    return BankAccount()

@given('nový bankovní účet')
def new_account():
    pass #již realizováno pomocí fixture
```

```
@given('bankovní účet se zůstatkem 300 Kč')
def account_with_balance(account):
    account.deposit(300)

@when('vložím 200 Kč')
def deposit_money(account):
    account.deposit(200)

@when('vyberu 100 Kč')
def withdraw_money_100(account):
    account.withdraw(100)

@when('vyberu 500 Kč')
def withdraw_money_500(account):
    account.withdraw(500)

@then('zůstatek je 200 Kč')
def check_balance_200(account):
    assert account.get_balance() == 200

@then('zůstatek je 0 Kč')
def check_balance_0(account):
    assert account.get_balance() == 0
```

Domácí cvičení

Cílem domácí cvičení je vyzkoušet si BDD testy pro webové stránky pro testování tradičních scénářů z průmyslové praxe.

1. Vymyslete alespoň 3 scénáře pro BDD testování webové stránky (jak obsahu HTML, tak json dat pro API testování).
2. Předepište tyto scénáře do feature souborů pomocí jazyka Gherkin. Vytvořte zvlášť soubor pro testování HTML, tak API.
3. Napište ke scénářům testy pomocí pytest-bdd modulu.
4. Pro všechny následující úkoly budete využívat tuto stránku jako zdroj: ZDROJ1 a ZDROJ2i
5. Na základě tutoriálů z webových stránek parametrizujte Vaše BDD testy.
6. Na základě tutoriálu vytvořte datové tabulky parametrických hodnot (scenario outline) pro Vaše BDD testy (druhý odkaz má i více tabulek v jednom feature souboru).
7. Přidejte značky pro selektivní spouštění Vašich BDD testů (testy HTML, testy API).
8. Podívejte se na plugin pytest-splinter ZDE pro efektivní testování webových stránek. Zkuste si napsat jednoduchý test.
9. Přepište jednoduchý splinter test do BDD testu.