# Testování softwaru (KI/TSW)

Pavel Beránek

11 Března, 2025

## 3.1. Integrační testování

Integrační testování je druh testovací techniky při které se různé komponenty softwaru testují jako jedna propojená komponenta. V jednotkovém testování se komponenty testovaly izolovaně a to primárně za účelem verifikace, tedy zjištění, zda programujeme správně (bez chyb). Integrační testování se provádí za účelem validace, tedy zjištění, zda program plní požadovanou funkcionalitu nebo má požadované mimofunkční požadavky (spolehlivost, zabezpečení atd.). Kromě validace je ještě cílem zjistit, zda jsou kompatibilní rozhraní samotných komponent a spolupracují podle očekávání.

Teorie integračního testování se stále rozvíjí. Nejznámější jsou dva přístupy k integračnímu testování:

1. Velký třesk

- Všechny naprogramované komponenty se spustí jako jeden celek a otestuje se funkčnost systému. Tohle je Vaše tradiční programování, kdy přidáváte nový kód a pak to celý spouštíte naráz.
- Výhodou je to, že všechny komponenty jsou testovány naráz a lze tak simulovat komplexní chování.
- Nevýhodou je obtížná izoloce problémových komponent, které vyvolávají selhání systému při komplexních interakcích.

2. Inkrementální integrační testování

- Přístup k integračnímu testování, kde se postupně přidávají komponenty do testování komplexního chování. Nejčastěji začneme s dvěmi, pak přidáme třetí, čtvrtou atd.
- Výhodou je snadné nalezení problémových komponent.
- Nevýhoda vychází z pořadí vývoje komponent. Pro otestování integrace dvou komponent musíte čekat na to, až budou obě vyvinuté. Řešením je využít tzv. testové dvojníky.

Z pohledu procesu integračního testování lze provádět integraci komponent pro testování dvěma procesními směry:

1. Top-Down přístup

- Testování se nazývá také Top-Down z toho důvodu, že vrcholné (typicky UI) komponenty si volají nižší komponenty pro konkrétní úkony.

2. Bottom-Up přístup

- Začnete od komponent nízké úrovně (funkce/třídy) a poté je zapojíte do vyšší jednotky (funkce/třídy využívající výsledky funkcí/třídy).

Naivní integrační testování, které jste dělali bez znalosti teorie bylo Top-Down Big Bang přístup, tedy spustíte celou aplikaci a klikáte na UI webovky, které buď funguje (správně si volá funkce) nebo nefunguje a něco je špatně. Buď nefunguje nějaká z volaných jednotek nebo jejich integrace.

Korektní automatizované testování pro GitHub Workflows od automatizačního testera by bylo opačně. Bottom-Up přístupem inkrementálně testuji postupné propojování komponent až do nejvyšší úrovně. To ovšem vyžaduje velké množství času a v malých týmech nejste toho schopni.

Nepleťte si to s End-to-end testováním. Cílem integračního testování je zjistit korektní integraci voláním vybraných funkcí nebo interakcí s vybranými prvky. End-to-end simuluje cílovou práci uživatele podle typických úkonů (scénáře).

## 3.2. Testoví dvojníci

Testový dvojník je termín pro objekt, který se chová zdánlivě jako pravý objekt, ale není jím. Příklad je objekt, který reaguje na SQL dotazy, ale nejedná se o pravou databázi.

Testových dvojníků existuje celá řada:

1. Atrapa (Dummy)

- Objekt, který je potřeba vytvořit pro vyhovění rozhraní (protokolu), ale nebude využíván. Využíváte ho jen, aby Vám interpretr/překladač nevracel chybovou hlášku a kód šlo spustit.

2. Padělek (Fake)

- Objekt, který předstírá chování reálného objektu (jiné komponenty). Příkladem může být simulace odezvy webové odpovědi, která vrací kód 200, nebo přihlašovací objekt, který vrací potvrzení o správném přihlášení do systému pomocí přihlašovacích údajů. Padělky se rozdělují na Pahýly (Stubs) a Napodobeniny (Mocks).
- V některé literatuře je Fake brán ne jako kategorie testovacích objektů, ale jako sama o sobě kategorie. Autoři tím myslí velmi přesnou napodobeninu skutečného objektu, který poskytuje minimální potřebné chování. Vlastně takový prototyp komponenty, který už lze použít pro integraci, ale je nutné ho dodělat pro funkční software.

3. Pahýl (Stub)

- Padělek, který vrací při žádosti konstantní hodnotu. Slouží zejména pro vyhovění závislostem. Pokud naše aplikace vyžaduje interakci s databází nebo webovými REST API službami, pak je možné izolovat naše testované komponenty od těchto externích pomocí Stubu. Používáme také tehdy, pokud ještě externí komponenta neexistuje (typicky databáze).

4. Napodobenina (Mock)

- Padělek, který vrací při žádosti různorodé hodnoty podle kontextu. Jedná se tedy o nekonstantní chytrý Stub.

5. Špeh (Spy)

- Objekt, který monitoruje interakci mezi propojenými komponentami a vede o ní záznamy (logy). Loguje se typicky: názvy volaných metod, počet volaných metod, argumenty metody, pořadí volání metod. Tato vnitřní komunikace je často modelována pomocí Sekvenčních diagramů jazyka UML.

## 3.3. Aplikace s externí službou

Budeme testovat integraci našeho kódu s externí službou. Nejprve se registrujte na stránce: ODKAZ. Po registraci si vygenerujte API klíč. Pro rychlejší orientaci: po přihlášení API klíče naleznete ZDE. API klíč si zkopírujte do schránky.

Ve Vašem pracovním adresáři si vytvořte soubor s názvem `.env` a uložte do něj API klíč v následujícím formátu:

```
API_KEY=zde_vložte_váš_api_klíč
```

Nainstalujte si následující moduly do virtuálního prostředí Pythonu:

```
python-dotnev
pytest
requests
```

Poté si vytvořte soubor s kódem, který se bude testovat:

```
import os
import requests
from dotenv import load_dotenv

# Načtení proměnných z .env souboru
load_dotenv()

API_KEY = os.getenv("API_KEY")

class WeatherService:
    """Služba pro získání počasí pomocí API klíče uloženého v .env."""

    def get_weather(self, city):
        """Vrátí počasí pro dané město na základě API volání."""
        if not API_KEY:
            raise ValueError("API klíč není nastaven.")

        url = url = f"http://api.openweathermap.org/data/2.5/weather?q={city}&appid={API_KEY}&units=metric
        response = requests.get(url)

        if response.status_code == 200:
            return response.json()
        else:
            return {"error": f"Chyba: {response.status_code}, {response.text}"}

# Testovací běh
if __name__ == "__main__":
    print(f"API_KEY: {API_KEY}")  # Ujistěte se, že se načetl API klíč
    city = "Prague"
    service = WeatherService()
    response = service.get_weather(city)
    temperature = response["main"]["temp"]
    print(temperature)
```

Otestujte, že Vám služba funguje (normálním spuštěním, ne automatizovaným testem). Tento kód vlastně představuje (podle jedné z definice) objekt typu Fake. Určitě už funguje, jen není nejvhodnější do produkce a chtělo by to jeho řádné vylepšení.

## 3.4. Komponenta pro ukládání získaných dat do JSON souboru

Vytvořte komponentu WeatherDataLogger, která bude ukládat získaná data ze služby WeatherService do JSON souboru. Pokud se na to necítíte, tak zkopírujte můj kód, jelikož to není jádrem dnešního semináře.

Řešení:

```
import json
import os
from datetime import datetime

class WeatherDataLogger:
    """Ukládá získaná data o počasí do JSON souboru."""

    FILE_NAME = "weather_data.json"

    def __init__(self):
        """Načte existující soubor nebo vytvoří prázdný JSON, pokud neexistuje."""
        if not os.path.exists(self.FILE_NAME):
            with open(self.FILE_NAME, "w") as file:
                json.dump({}, file)

    def update_data(self, city, temperature):
        """Přidá nebo aktualizuje záznam pro dané město s časem."""
        now = datetime.now().strftime("%Y-%m-%d %H:00")  # Zaokrouhlení na celou hodinu

        # Načtení existujících dat
        with open(self.FILE_NAME, "r") as file:
            data = json.load(file)

        # Přidání/aktualizace města a teploty
        if city not in data:
            data[city] = {}  # Vytvoříme záznam pro město
        data[city][now] = temperature

        # Uložení zpět do souboru
        with open(self.FILE_NAME, "w") as file:
            json.dump(data, file, indent=2)

        print(f"Data for {city} updated: {temperature}°C at {now}")
```

Implementujte logger do Vašeho hlavního kódu:

```
if __name__ == "__main__":
    city = "Prague"
    service = WeatherService()
    response = service.get_weather(city)
    temperature = response["main"]["temp"]
    print(temperature)

    #ZDE PŘIDÁN LOGGER
    logger = WeatherDataLogger()
    logger.update_data(city, temperature)
```

## 3.5. Používání Pahýlu (Stub)

Vytvořte Stub, který simuluje chování služby WeatherService a vrací do našeho mainu vždy konstantní odpověď. Tím můžete otestovat integraci logovací komponenty s mainem (trošku nevhodná komponenta, ale nechci testování komplikovat). Popřemýšlete o tom, jak si dobře organizovat soubory a adresáře, abyste mohli snadno nahradit reálnou službu Stubem.

Ukázka implementace Stub v pytest:

```
import pytest

class MathOperations:
    def add(self, a, b):
        raise NotImplementedError("Ještě není naprogramováno.")

class StubMathOperations:
    def add(self, a, b):
        return 10

def test_stub_add():
    math_stub = StubMathOperations()
    assert math_stub.add(5, 5) == 10
```

Řešení:

```
# bude dodáno
```

## 3.6. Používání Napodobeniny (Mock)

Vytvořte Mock, který simuluje chování služby WeatherService a vrací do našeho mainu různorodé odpovědi na základě kontextu. Mě osobně napadají následující kontexty:

1. Dodaný vs. nedodaný API klíč.
2. Různá města - např.: podle prvního písmenka v abecedě vracím předdefinovanou hodnotu (ať testujete variabilitu odpovědí).
3. Různý čas - můžete napsat funkci (třeba lineární nebo kvadratickou), která podle hodiny (u Vás pro tohle cvičení minuty) vrací různé hodnoty teploty podle funkcí předdefinovaného trendu.

Můžete si napsat Mock svépomocí nebo použít unittest.mock:

```
import unittest
from unittest.mock import MagicMock

class MathOperations:
    def add(self, a, b):
        return a + b  # Normální implementace

class TestMock(unittest.TestCase):
    def test_mock_add(self):
        math_mock = MathOperations()
        math_mock.add = MagicMock(return_value=10)  # Přepíšeme metodu mockem

        result = math_mock.add(3, 7)

        math_mock.add.assert_called_once_with(3, 7)  # Ověříme volání s argumenty
        self.assertEqual(result, 10)

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

Řešení:

```
# bude dodáno
```

## Domácí cvičení

1. Zkuste si naimplementovat komponentu pro opakované hromadné volání API pro vybraná města (třeba z konfiguračního souboru TOML).
2. Vytvořte si integrační logovací test pomocí Spy objektu.
3. Podívejte se na testovací techniku s názvem Monkey Patching. Jak ji lze implementovat do Vašich integračních testů? Vyzkoušejte.