VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA PODNIKATELSKÁ
FACULTY OF BUSINESS AND MANAGEMENT

ÚSTAV INFORMATIKY
INSTITUTE OF INFORMATICS

NÁVRH NÁSTROJE PRO AUTOMATIZOVANÉ INTEGRAČNÍ TESTOVÁNÍ

DESIGN OF AUTOMATED INTEGRATION TESTING TOOL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Lucia Odrobinová

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. Jan Luhan, Ph.D., MSc

BRNO 2021

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA PODNIKATELSKÁ

Zadání diplomové práce

Ústav:	Ústav informatiky
Studentka:	**Bc. Lucia Odrobinová**
Studijní program:	Systémové inženýrství a informatika
Studijní obor:	Informační management
Vedoucí práce:	**Ing. Jan Luhan, Ph.D., MSc**
Akademický rok:	2020/21

Ředitel ústavu Vám v souladu se zákonem č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů a se Studijním a zkušebním řádem VUT v Brně zadává diplomovou práci s názvem:

Návrh nástroje pro automatizované integrační testování

Charakteristika problematiky úkolu:

Úvod
Cíle práce, metody a postupy zpracování
Teoretická východiska práce
Analýza současného stavu
Vlastní návrh řešení
Závěr
Seznam použité literatury
Přílohy

Cíle, kterých má být dosaženo:

Navrhnout nástroj pro automatizované integrační testování databázového schématu. Účelem nástroje je detekce datových změn mezi databázovou a aplikační vrstvou při provádění úprav databázového schématu. Záměrem je tímto usnadnit testování klientských aplikací a předcházet obtížně detekovatelným datovým změnám. Nástroj bude implementován v jazyce C# a bude pracovat nad SQL Server databází.

Základní literární prameny:

CLARK, D. Beginning C# Object-Oriented Programming. 2nd ed. USA: Apress, 2013. 384 p. ISBN 978-1-4302-4936-8.

OLSEN, K., M. POSTHUMA a S. ULRICH. Certified Tester: Foundation Level Syllabus. 8th ed. Belgium: ISTQB, 2019. 93 p.

SCHWICHTENBERG, H. Modern Data Access with Entity Framework Core. 1st ed. Germany: Apress, 2018. 644 p. ISBN 978-1-4842-3552-2.

SPILLNER, A., T. LINZ a H. SCHAEFER. Software Testing Foundations. 4th ed. USA: Sheridan, 2014. 305 p. ISBN 978-1-937538-42-2.

TROELSEN, A. a P. JAPIKSE. Pro C# 7 With .NET and .NET Core. 8th ed. USA: Apress, 2017. 1372 p. ISBN 978-1-4842-3017-6.

Termín odevzdání diplomové práce je stanoven časovým plánem akademického roku 2020/21

V Brně dne 28.2.2021

L. S.

Mgr. Veronika Novotná, Ph.D.
ředitel

doc. Ing. Vojtěch Bartoš, Ph.D.
děkan

Abstrakt

Diplomová práca sa zameriava na tvorbu nástroja pre automatizované integračné testovanie pre vybranú spoločnosť. Cieľom tohto nástroja je detekcia a reportovanie dátových zmien, ktoré mohli nastať počas vývoja softvéru pri upravovaní jeho databázovej schémy. Takáto automatizácia detekcie dátových zmien prináša spoločnosti množstvo benefitov.

Nástroj je implementovaný v programovacom jazyku C#, ktorý umožňuje objektovo orientované programovanie, s použitím softvéru Microsoft Visual Studio Community 2019. Vzhľadom na prácu nástroja nad Microsoft SQL Server databázou bol pri jeho implementácii využitý Entity Framework, ktorý zabezpečuje objektovo-relačné mapovanie.

Kľúčové slová

automatizované integračné testovanie, objektovo orientované programovanie, programovací jazyk C#, Entity Framework

Abstract

The diploma thesis focuses on the development of an automated integration testing tool for a company. The tool is employed to detect and report data changes that may have been caused by the modification of a database schema during the development of a software. This automation of recognizing data changes brings many advantages to the company.

The tool has been developed using the C# programming language, which supports object-oriented programming, using the Microsoft Visual Studio Community 2019 software. Since the tool works with the Microsoft SQL Server database, Entity Framework, an object-relational mapping framework, has been used during the implementation of the tool.

Key words

automated integration testing, object-oriented programming, C# programming language, Entity Framework

Bibliografická citácia

ODROBINOVÁ, Lucia. *Návrh nástroje pro automatizované integrační testování* [online]. Brno, 2021 [cit. 2021-05-15]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/135339. Diplomová práce. Vysoké učení technické v Brně, Fakulta podnikatelská, Ústav informatiky. Vedoucí práce Jan Luhan.

Čestné prehlásenie

Prehlasujem, že predložená diplomová práca je pôvodná a spracovala som ju samostatne. Prehlasujem, že citácia použitých prameňov je úplná, že som vo svojej práci neporušila autorské práva (v zmysle Zákona č. 121/2000 Sb., o právu autorském a o právech souvisejících s právem autorským).

V Brne dňa 15. mája 2021

……..……

podpis autora

Poďakovanie

Týmto by som sa chcela poďakovať vedúcemu práce Ing. Janovi Luhanovi, Ph.D. za jeho cenné rady pri tvorbe tejto diplomovej práce. Zároveň by som rada poďakovala firme, pre ktorú bol výstup diplomovej práce vyhotovený, za spoluprácu a poskytnutie potrebných podkladov.

Obsah

ÚVOD

Pod pojmom testovanie softvéru rozumieme súbor činností vykonávaných v snahe zabezpečiť jeho čo najvyššiu kvalitu, resp. zmenšiť riziko zlyhania softvéru počas jeho užívania. Testovanie softvéru je jednou zo základných fáz životného cyklu vývoja softvéru, pričom jeho pozícia v rámci neho sa odvíja od zvoleného modelu.

Testovanie softvéru je fundamentálnou súčasťou v rámci tohto životného cyklu z dôvodu množstva benefitov, ktoré prináša. Tieto benefity vychádzajú nepriamo z cieľov testovania, ktoré sú stanovované individuálne vzhľadom ku kontextu testovaného komponentu, resp. systému, ale i v závislosti od úrovne testovania či vybraného modelu životného cyklu vývoja softvéru.

Pre proces testovania je charakteristické jeho rozdelenie na úrovne, za ktoré považujeme skupiny testovacích aktivít, ktoré sa realizujú súbežne. Každá z týchto úrovní má svoje špecifické ciele, testovaciu bázu, testovaný objekt i zlyhania, resp. defekty. Typicky rozlišujeme úroveň jednotkového testovania, úroveň integračného testovania, systémovú úroveň testovania a akceptačnú úroveň testovania. Tá integračná sa sústredí na vzájomné pôsobenie medzi komponentmi, resp. systémami. Medzi bežné defekty, resp. zlyhania plynúce z testov na tejto úrovni sa radia napríklad nesprávne či chýbajúce dáta.

Testovanie softvéru môže vo všeobecnosti prebiehať manuálne i automaticky. Automatizovanie testovania je obzvlášť vhodné v prípadoch, ktoré sa vykonávajú vo vysokej frekvencii a zároveň v rámci nich nedochádza často k zásadným zmenám. Automatizácia potom prináša množstvo výhod, akými je napríklad časová úspora oproti manuálnemu testovaniu zamestnancami, ale aj vyššia presnosť v odhaľovaní defektov, resp. zlyhaní, a to aj z dôvodu možnosti pretestovania viacerých prípadov.

CIELE PRÁCE, METÓDY A POSTUPY SPRACOVANIA

Cieľom diplomovej práce je poskytnúť návrh testovacieho nástroja pre vybranú spoločnosť. Táto spoločnosť má záujem automatizovať testovania, ktoré sú aktuálne vykonávané manuálne a zároveň sú vhodnými kandidátmi pre ich zautomatizovanie. Ideálnymi kandidátmi sú tie testovania, ktoré sú realizované často a s opakovaním využitia tých istých mechanizmov. Takou oblasťou, teda oblasťou hodnou pokrytia týmto nástrojom, je aj detekcia dátových zmien, ktoré mohli byť zapríčinené modifikáciou databázovej schémy počas vývoja softvéru. Práca sa teda bude zameriavať na tvorbu návrhu backend časti automatizovaného integračného testovacieho nástroja zaznamenávajúceho zmeny na dátovej úrovni spôsobenými úpravou databázovej schémy.

Práca bude rozdelená do troch základných častí, a to do časti poskytujúcej teoretické východiská, do časti analyzujúcej súčasnú situáciu vo zvolenej firme a v neposlednom rade do časti venujúcej sa vlastnému návrhu riešenia.

Teoretická časť pokryje jednak oblasť testovania softvéru, a jednak oblasť technológií využiteľných v návrhovej časti práce. Oblasť testovania softvéru vysvetlí, čo testovanie vlastne je, a kde ho je možné zaradiť v rámci životného cyklu vývoja softvéru. Taktiež budú uvedené jeho možné ciele, čiže nepriamo prínosy, jeho typické členenie na rôzne úrovne, ale i základných sedem princípov testovania. V neposlednom rade bude uvedené, kedy a prečo je vhodné manuálne testovanie automatizovať.

Ďalej budú v teoretickom úseku záverečnej práce spomenuté technológie, ktoré by mohli byť využiteľné pri navrhovaní nástroja. Tieto technológie vychádzajú do značnej miery z tých technológií, ktoré sa vo firme používajú, čím sa myslí objektovo orientované programovanie pomocou programovacieho jazyka C#, ktorý je možné využiť pri tvorbe aplikácií využitím softvéru, akým je Microsoft Visual Studio. Taktiež dôjde k objasneniu problematiky relačných databáz a ich prepojeniu s Microsoft Visual Studio za pomoci objektovo-relačného mapovania, a to technológiou Entity Framework.

Druhou hlavnou kapitolou práce je analýza súčasného stavu. Tá najprv predstaví vybranú spoločnosť, pre ktorú je nástroj navrhovaný, a to primárne z hľadiska jej miest pôsobenia a predmetu podnikania. Následne dôjde k preskúmaniu jej testovacích spôsobov, z ktorého vyplynú požiadavky pre návrhovú časť práce.

Návrhová časť práce sa bude venovať tvorbe samotného nástroja. Bude vysvetlený proces návrhu a implementácie jeho backend strany a následne dôjde k prezentácii týchto funkcionalít na vzorových dátach. V rámci tejto kapitoly bude aj stručne vysvetlená integrácia výsledného nástroja do prostredia firmy, a to spolu s názornými ukážkami demonštrujúcimi ako užívateľ môže s nástrojom pracovať. Na záver tejto kapitoly, čiže na záver celej práce budú zhrnuté kľúčové prínosy nástroja pre danú spoločnosť.

1 TEORETICKÉ VÝCHODISKÁ PRÁCE

V tejto časti diplomovej práce dôjde k vysvetleniu teoretických východísk úzko súvisiacich s tvorbou samotného nástroja, demonštrovaného v návrhovej časti diplomovej práce. Najprv budú vysvetlené súvislosti týkajúce sa samotného testovania softvérov pre pochopenie jeho podstaty, po ktorých budú odprezentované podklady potrebné pre technologickú stránku nástroja.

1.1 Testovanie softvéru

Pre lepšie pochopenie aspektov nástroja určeného pre automatizované integračné testovanie dôjde v nasledujúcich odstavcoch, resp. podkapitolách k zadefinovaniu pojmu testovanie softvéru, uvedeniu jeho možnej pozície v rámci životného cyklu vývoja softvéru a taktiež vysvetleniu, prečo je testovanie nevyhnutné. Na záver sa zmienia jeho zvyčajné úrovne, typy a zároveň možnosti jeho automatizácie.

Testovanie softvéru možno vo všeobecnosti **definovať** ako spôsob zhodnotenia úrovne kvality softvéru, resp. spôsob ako zmenšiť riziko zlyhania softvéru počas jeho užívania (1, s. 13).

1.1.1 Pozícia testovania v rámci životného cyklu vývoja softvéru

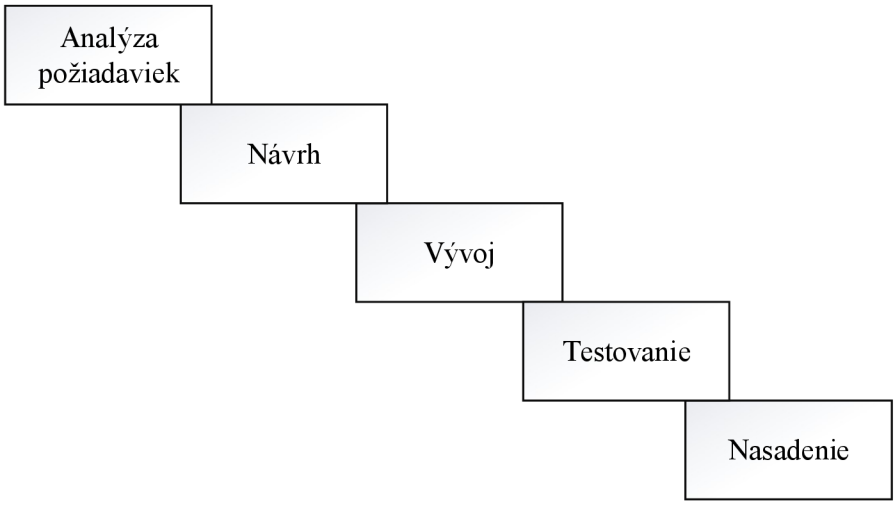
Pozícia testovania v rámci životného cyklu vývoja softvéru závisí od zvoleného modelu. Modely životného cyklu vývoja softvéru možno rozdeliť na sekvenčné a iteratívne. Oba typy zahŕňajú rovnaký sled činností, a to analýzu požiadaviek, návrh, vývoj, testovanie a nasadenie (2). Stručne si tieto modely popíšme.

Sekvenčný vývojový model popisuje proces vývoja softvéru ako lineárne za sebou idúci sled aktivít, čo znamená, že každá fáza začne, až keď tá tesne predchádzajúca skončí. V rámci teórie sa tieto fázy neprelínajú, avšak v praxi je užitočné disponovať spätnou väzbou pre nasledujúcu fázu ešte pred ukončením tej tesne predchádzajúcej (1, s. 28).

Sekvenčné vývojové modely dodávajú zainteresovaným stranám a užívateľom softvér, ktorý obsahuje kompletný set vlastností. K tomuto dodaniu však zväčša dochádza až po niekoľkých mesiacoch či rokoch od počiatku projektu (1, s. 28).

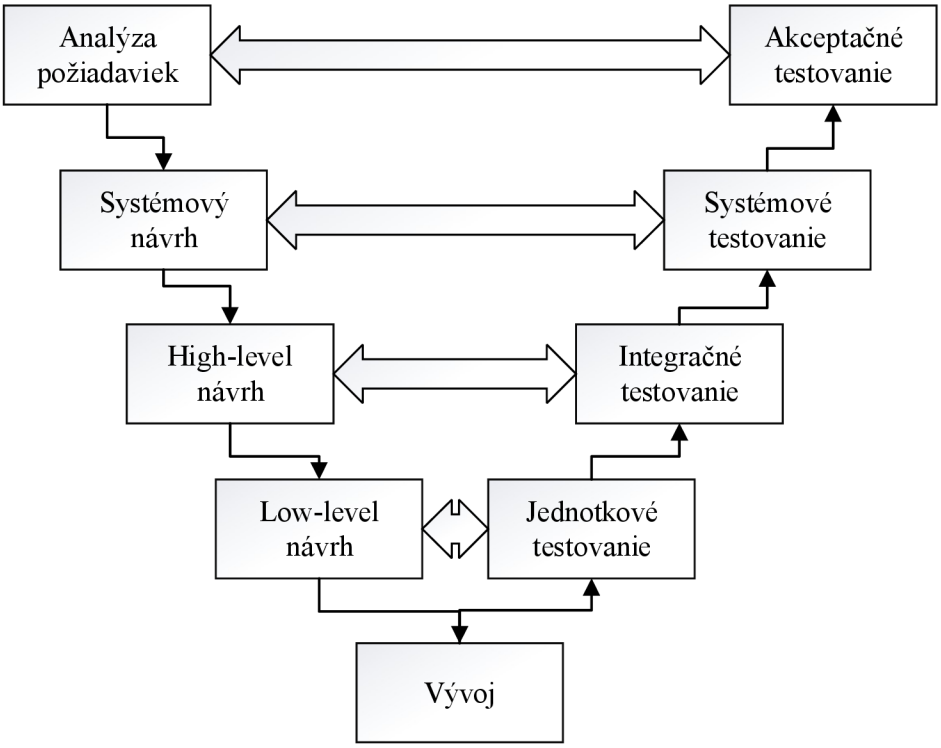
Medzi sekvenčné modely životného cyklu vývoja softvéru možno zaradiť napríklad vodopádový model alebo V-model (2).

Vodopádový model je najjednoduchší a najstarší explicitný model životného cyklu. Skladá sa zo sledu fáz s malým alebo žiadnym prekrytím medzi susediacimi fázami (3, s. 81). To znamená, že k testovaniu dochádza viac-menej až po ukončení fázy vývoja (1, s. 28).



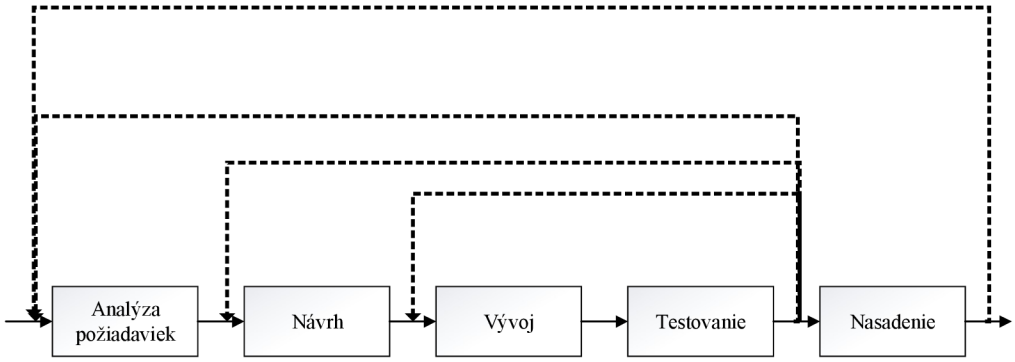
Obrázok 1: Model životných cyklov vývoja softvéru - vodopádový model s malým prekrytím fáz
(Zdroj: Vlastné spracovanie podľa 2)

S cieľom klásť väčší dôraz na verifikáciu a validáciu v porovnaní s vodopádovým modelom vzniká tzv. V-model, ktorý rozdeľuje lineárny životný cyklus na dve vetvy. Ľavá vetva obsahuje rôzne úlohy týkajúce sa analýzy požiadaviek a návrhu, ktoré vedú k fáze vývoja. V pravej vetve je systém integrovaný, testovaný a verifikovaný, pričom každá z týchto fáz sa dotýka jednej z fáz ľavej vetvy a jej úlohou je overiť, či vyvinuté produkty správne implementujú výsledok príslušnej fázy (3, s. 83). Týmto dochádza vo V-modeli k uplatneniu základného princípu testovania, ktorý tvrdí, že s testovaním treba začať čo najskôr (1, s. 28).



Obrázok 2: Model životných cyklov vývoja softvéru - V-model
(Zdroj: Vlastné spracovanie podľa 2)

O **iteratívnom** vývoji hovoríme, keď sú skupiny vlastností špecifikované, nadizajnované, implementované a testované súbežne v rámci sérií cyklov, ktoré majú zväčša fixne stanovenú dĺžku trvania. Hovoríme o iteráciách. V rámci iterácie môže dôjsť k pridaniu nových vlastností (inkrementov) softvéru, ale aj k modifikácii už tých predtým vyvinutých. Každá iterácia dodáva funkčný softvér, ktorý postupne rastie, až kým nie je dosiahnutý cieľový stav z hľadiska jeho funkcionalít. Vtedy dôjde k zastaveniu vývoja (1, s. 29).



Obrázok 3: Model životných cyklov vývoja softvéru - iteratívny model znázorňujúci rôzne možnosti iterácie

(Zdroj: Vlastné spracovanie podľa 3, s. 90)

Je vhodné zmieniť, že takýto postupný rast softvéru zvyšuje mieru dôležitosti regresného testovania overujúceho, či nové prírastky nepriaznivo neovplyvnili doposiaľ existujúcu funkcionalitu (1, s. 29).

Výhodou iteratívnych modelov životného cyklu vývoja softvéru je, že použiteľný softvér je k dispozícii už za niekoľko dní či týždňov. Táto verzia softvéru však ešte neobsahuje všetku požadovanú funkcionalitu. Tá môže byť dodaná až v priebehu niekoľkých mesiacov, resp. rokov (1, s. 29).

Medzi typické iteratívne modely možno zaradiť napríklad model RUP (skratka z anglického Rational Unified Process), špirálový model, ale aj agilné modely, a to napríklad Scrum (2).

V rámci metódy Scrum sa obvykle používa relatívne krátka doba iterácie, ktorá môže byť v hodinách, dňoch, prípadne pár týždňoch (1, s. 29). Táto iterácia sa v rámci Scrum označuje termínom sprint (4). V tomto úseku je vyvinuté malé množstvo nových vlastností, prípadne sa len doladia, resp. opravia tie už existujúce. Scrum metóda pevne definuje role v rámci tímu (2). Scrum tím je tvorený vývojármi, product ownerom a Scrum masterom. Product owner má na starosti maximalizáciu hodnoty vyvíjaného produktu. Tvorí a komunikuje cieľ produktu a tvorí, komunikuje a priraďuje prioritu položkám z product backlogu, t.j. zjednodušene povedané, úlohám, ktoré je potrebné zrealizovať. Scrum master má na starosti zabezpečenie korektného chodu Scrum metódy, a to jej tlmočením členom tímu i organizácii. Taktiež zodpovedá za efektívnosť Scrum tímu, a to napríklad koučovaním jeho členov v ich samo organizovaní sa, napomáhaním im sústrediť sa

na tvorbu inkrementov majúcich najvyššiu hodnotu, či odstraňovaním prípadných prekážok brániacich progresu tímu. Ďalej zabezpečuje, aby Scrum mítingy boli efektívne a realizované v stanovenom časovom limite. Poslednou súčasťou Scrum tímu sú vývojári, ktorí sa podieľajú na vytvorení plánu pre daný sprint a pracujú na jeho naplnení (4).

1.1.2 Činnosti testovania softvéru

Proces testovania pozostáva z niekoľkých činností. Neexistuje univerzálne použiteľný set činností testovacieho procesu, pretože jeho potreby sa odvíjajú od množstva kontextových faktorov. Medzi tieto činitele možno zaradiť napríklad model životného cyklu vývoja softvéru, resp. metódy použité v projekte, zvažované úrovne a typy testovania, odvetvie podnikania, požadované štandardy, ktoré majú byť splnené, rozpočet, zmluvné požiadavky a iné. Dá sa však zostaviť typická sada činností, bez ktorých by bol testovací proces menej schopný dosiahnutia cieľov testovania (1, s. 17, 28). Tieto aktivity budú v nasledujúcich odsekoch stručne zmienené.

V prvom rade je potrebné testovanie naplánovať, čo znamená, že dôjde k definovaniu cieľov, ktoré má testovanie naplniť, a to s ohľadom na obmedzenia plynúce z daného kontextu (1, s. 18). Taktiež sa definujú potrebné ľudské zdroje, čas na realizáciu i nutné vybavenie potrebné pre výkon testovania (5, s. 20). Tento vzniknutý plán môže byť neskôr modifikovaný na základe spätnej väzby získanej z monitoringu a riadenia testovacích aktivít (1, s. 18-19).

Ďalšou typickou činnosťou pre testovací proces je testovacia analýza. Tá, zjednodušene povedané, skúma, čo treba testovať. Po jej absolvovaní dôjde k návrhu testovania, ktorého cieľom je určiť, ako testovať (1, s. 19-20).

Následne môže dôjsť k implementácii testov. Tá spočíva vo vytvorení testvéru potrebného pre vykonanie testov. Po nej je možno tieto testy vykonať, a to buď manuálne, alebo použitím nástrojov na vykonanie testov. V prípade nájdenia defektov dôjde k ich reportu (1, s. 21).

V neposlednom rade možno uviesť kompletizáciu testovania spočívajúcej v snahe zozbierať dáta z uskutočnených testovacích aktivít za účelom zjednotenia nadobudnutých skúseností, testvéru a iných relevantných informácií (1, s. 22).

1.1.3 Ciele testovania softvéru

Spomeňme si niektoré všeobecné ciele, a teda nepriamo i prínosy, platné pre testovanie akéhokoľvek softvéru. Za jeden z nich možno považovať snahu predísť defektom prostredníctvom revízie užívateľských scenárov či zdrojového kódu. Je však pochopiteľné, že nedôjde k ich 100% eliminácii, a preto je ďalším cieľom testovania ich odhalenie a následné napravenie, čím sa dospeje k zníženiu rizika nedostatočnej kvality softvéru. Za cieľ možno považovať taktiež aj verifikáciu, či boli všetky kladené požiadavky, štandardy (zmluvné, právne, regulatórne) splnené, ale aj zistenie, či je testovaný objekt dokončený a či funguje tak, ako zainteresované strany očakávajú (1, s. 13).

Treba však brať na vedomie, že ciele testovania sa môžu vzájomne líšiť v závislosti od kontextu testovaného komponentu, resp. systému, či v závislosti od úrovne testovania i modelu životného cyklu vývoja softvéru (1, s. 13).

1.1.4 Princípy testovania softvéru

V rámci tejto podkapitoly považujeme za vhodné zmieniť existenciu siedmich základných testovacích princípov.

Prvý princíp tvrdí, že vďaka testovaniu možno odhaliť prítomnosť defektov, ale nemožno dokázať ich absolútnu absenciu, t.j. testovanie nedokazuje stopercentnú správnosť softvéru (1, s. 16).

Druhý princíp spočíva v tom, že nie je možné otestovať všetko v zmysle všetkých kombinácií vstupov a vstupných podmienok. Výnimkou však môžu byť triviálne prípady (1, s. 16).

Tretí princíp uvádza, že pre skoré zistenie defektov je potrebné s testovaním začať čo najskôr v rámci vývoja softvéru. Hlavnou motiváciou je snaha predísť prípadným nákladným zmenám v budúcnosti (1, s. 16).

Štvrtý princíp tvrdí, že defekty majú tendenciu sa zhlukovať, t.j. častokrát je väčšina defektov nájdená v malom počte modulov (1, s. 16).

Piaty princíp hovorí o vyhnutiu sa tzv. pesticídnemu paradoxu, teda opakovaniu stále tých istých testov napriek tomu, že tieto testy nenachádzajú žiadne nové defekty. Pre detekciu nových defektov je častokrát potrebná modifikácia, resp. tvorba nových testov či setov testovacích dát (1, s. 17).

Šiesty princíp konštatuje, že proces testovania je vykonávaný v závislosti od kontextu, čo znamená, že naň vplýva to, o aký typ softvéru sa jedná, to, akým spôsobom je daný projekt riadený a mnoho ďalších aspektov (1, s. 17).

Posledný, teda siedmy, princíp upozorňuje na to, že objavenie a korekcia veľkého množstva defektov nie je záruka úspešnosti daného systému. Táto úspešnosť môže byť myslená napríklad z hľadiska jeho schopnosti naplniť očakávania či potreby užívateľov, resp. jeho dostačujúcej konkurencieschopnosti (1, s. 17).

1.1.5 Typy a úrovne testovania softvéru

Za úrovne, teda levely testovania, považujeme skupiny testovacích činností, ktoré sa realizujú súbežne. Každá úroveň je inštanciou procesu testovania, t.j. pozostáva z aktivít bližšie uvedených v predchádzajúcej podkapitole, a to na danej úrovni vývoja, počnúc individuálnymi jednotkami, resp. komponentmi až po hotové systémy. My si zmienime jednotkové testovanie, integračné testovanie, systémové testovanie a akceptačné testovanie (1, s. 30).

Každá z týchto úrovní je charakteristická špecifickými cieľmi, testovacou bázou, testovaným objektom, typickými zlyhaniami, defektami i špecifickými prístupmi. Pre každú úroveň testovania je nevyhnutné vhodné testovacie prostredie. Ako príklad možno uviesť, že pri testovaní komponentov je využívané prostredie samotných vývojárov, zatiaľ čo pri akceptačnom testovaní by malo ísť o prostredie odpovedajúce tomu produkčnému (1, s. 30). V nasledujúcich odsekoch dôjde k stručnému popisu jednotlivých úrovní.

Jednotkové testovanie je úroveň častokrát nazývané aj ako úroveň testovania komponentov či testovania modulov. Vykonáva ju zväčša samotný vývojár. Na tejto úrovni sa testujú objekty ako sú napríklad triedy, kód a dátové štruktúry, či databázové moduly. Ak dôjde k defektu, resp. zlyhaniu, ktorým môže byť nesprávna funkčnosť (napr. odlišujúca sa od tej definovanej v špecifikácii návrhu) alebo chybný kód či logika, tak častokrát dôjde k okamžitej oprave, t.j. bez akéhokoľvek formálneho manažmentu defektov (1, s. 31-32).

Považujeme za vhodné spomenúť, že v prípade, že životný cyklus vývoja softvéru je inkrementálneho alebo iteratívneho typu (napr. pri agilnom riadení projektov), kde dochádza k častým zmenám kódu, dôležitou súčasťou testovania sú automatizované regresné

testy komponentov, ktoré kontrolujú, či tieto nové zmeny nenarušili existujúce komponenty (1, s. 31).

Ďalšou úrovňou je **úroveň integračného testovania**. Tá sa sústredí na vzájomné pôsobenie medzi komponentmi, resp. systémami. Túto úroveň teda možno rozlíšiť na dve úrovne, a to úroveň integračného testovania komponentov a integračného testovania systémov. Typickými testovanými objektmi tejto úrovne sú databázy, infraštruktúry či rozhrania. Medzi bežné defekty, či zlyhania plynúce z testov na tejto úrovni možno zaradiť napríklad nesprávne, či chýbajúce dáta, resp. nesprávne kódovanie dát, zlyhania v komunikácii medzi komponentmi alebo systémami (1, s. 32-33).

Úroveň integračného testovania komponentov sa zameriava na interakcie a rozhrania medzi integrovanými komponentmi. Toto testovanie je vykonávané po testovaní komponentov a je spravidla automatizovaného charakteru. Vo všeobecnosti tento typ testovania majú na zodpovednosť vývojári (1, s. 32, 34).

Úroveň integračného testovania systémov sa zameriava na interakcie a rozhrania medzi systémami. Toto testovanie môže prebehnúť po, resp. súbežne so systémovým testovaním. Vo všeobecnosti tento typ testovania majú na zodpovednosti testeri (1, s. 32, 34).

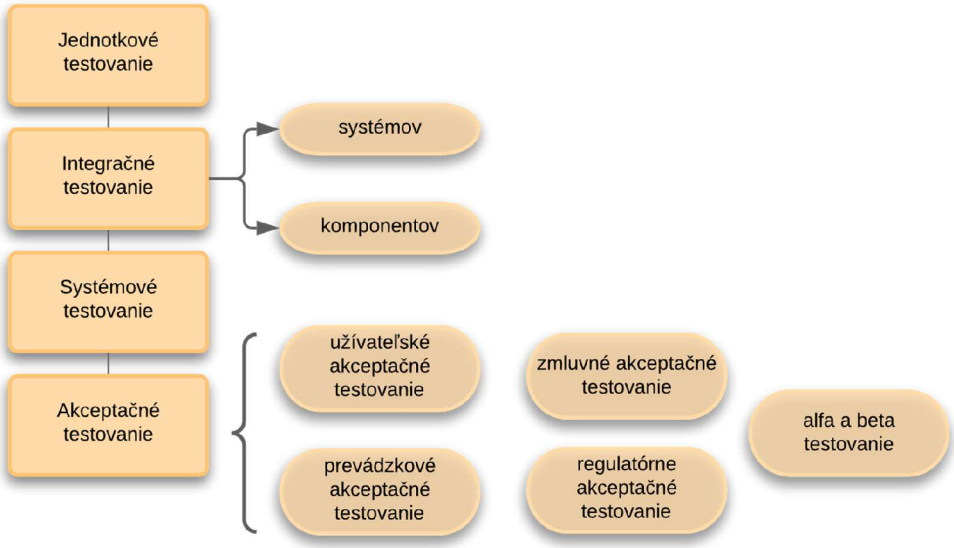
Systémová úroveň testovania sa zameriava na správanie a schopnosti celého systému, resp. produktu. Testovacie prostredie by malo v ideálnom prípade odpovedať tomu konečnému cieľovému prostrediu. Charakteristickými testovacími objektmi sú teda napríklad aplikácie, hardvérové, softvérové systémy, či operačné systémy. Štandardným defektom, resp. zlyhaním na tejto úrovni je napríklad nekorektné, resp. iné ako očakávané správanie systému. Toto testovanie je typicky realizované nezávislými testermi, ktorí vychádzajú z poskytnutých špecifikácií (1, s. 34-36).

Akceptačná úroveň testovania sa, taktiež ako systémové testovanie, sústredí na správanie a schopnosti celkového systému, resp. produktu. Cieľom tejto úrovne testovania však nie je hľadanie defektov (avšak môžu byť nájdené), ale overiť si, či dochádza k splneniu všetkých požiadaviek. Akceptačné testovanie môže napomôcť k zisteniu, či je systém pripravený na nasadenie a užívanie zákazníkom, resp. koncovým užívateľom (1, s. 36).

Akceptačné testovanie má tradične niekoľko foriem, a to užívateľské akceptačné testovanie, prevádzkové akceptačné testovanie, zmluvné a regulatórne akceptačné testovanie a alfa a beta testovanie (1, s. 36).

Užívateľské akceptačné testovanie je štandardne orientované na overenie vhodnosti systému na používanie plánovanými užívateľmi, zatiaľ čo prevádzkové akceptačné testovanie sa sústredí na správcov systému z hľadiska, či budú schopní zaistiť správny chod systému pre užívateľov. Zmluvné akceptačné testovanie je vykonávané v snahe zistiť, či boli naplnené všetky zmluvné akceptačné kritériá, zatiaľ čo regulatórne skúma, či došlo k splneniu potrebných vládnych, právnych, bezpečnostných, či iných predpisov, ktoré má systém spĺňať. Alfa a beta testovanie je typické pre bežne komerčne dostupné riešenia v prípade, že je žiaduca spätná väzba na systém od potenciálnych, resp. existujúcich zákazníkov predtým, než bude produkt oficiálne uvedený na trh (1, s. 36-37).

Zmienené úrovne testovania a ich delenia, resp. druhy zhŕňa nasledovný obrázok.



Obrázok 4: Úrovne testovania softvéru
(Zdroj: Vlastné spracovanie podľa 1, s. 31-37)

Na každej úrovni testovania je možné vykonať rôzne **typy testov**. Typ testu je set testovacích aktivít orientovaných na testovanie špecifických charakteristík systému, resp. jeho časti vychádzajúc z definovaných cieľov testovania (1, s. 39-41). Ako príklad si možno uviesť funkcionálny a nefunkcionálny typ testovania.

Funkcionálny typ testovania zahŕňa také testy, ktoré hodnotia funkcie, ktoré by mal systém vykonávať. Tieto funkcie môžu byť popísané napríklad v špecifikáciách biznis

požiadaviek, v užívateľských scenároch či prípadoch použitia, ale taktiež nemusia byť vôbec zdokumentované. Cieľom týchto testov je zhodnotenie funkcionálnych charakteristík kvality, medzi ktoré možno radiť kompletnosť, korektnosť a vhodnosť. Tento typ testov by mal byť vykonávaný v každej úrovni testovania (1, s. 39).

Nefunkcionálny typ testovania vyhodnocuje charakteristiky kvality, akými sú spoľahlivosť, výkonnosť, bezpečnosť, kompatibilita a využiteľnosť. Aj tento typ testov by sa mal vykonávať na všetkých zmienených úrovniach testovania (1, s. 39-40). Tieto testy zisťujú, aké je správanie systému napríklad v prípade, že dochádza k jeho zvýšenému zaťaženiu spôsobeného väčším počtom súčasne ho používajúcich užívateľov, ale taktiež môžu pozorovať správanie systému v prípade, že spracováva veľké množstvo rozsiahlych súborov. Môže ísť aj o zisťovanie zabezpečenia pred neoprávneným prístupom do systému, resp. k údajom, ale aj o preskúmanie kompatibility systému s tými už jestvujúcimi, či importu a exportu dát (5, s. 73).

1.1.6 Kedy testovanie automatizovať

Testovanie môže prebiehať manuálne a automaticky. Automatizácia testov prináša množstvo benefitov, akými je napríklad skutočnosť, že tieto testy je možno realizovať častejšie ako pri manuálnej forme testovania, čím sa zmenšuje riziko neodhalenia defektov. Taktiež je tieto testy možné spúšťať na rôznych platformách a sme pomocou nich schopní otestovať väčšie množstvo kombinácií vstupných dát, než by bolo možné pri manuálnom teste (6, s. 180).

Automatizáciu testov je však vhodné realizovať iba v niektorých prípadoch, ktoré spĺňajú určité predpoklady. Jedným z týchto predpokladov je, že testovacie scenáre sa často opakujú, pričom tieto scenáre musia byť triviálne automatizovateľné, aby bol tento proces automatizácie ekonomicky výhodný. Taktiež je predpokladom, že nie je v danom procese potrebné množstvo manuálnych prerušení, resp. situácií, v ktorých je nutné, aby užívateľ prepol medzi jednotlivými systémami (6, s. 195).

1.2 Technológie použiteľné pri návrhu nástroja

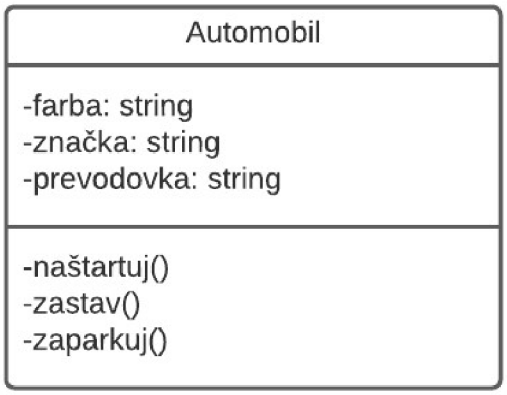
Táto podkapitola ozrejmí teoretické hľadisko technológií, s ktorými bude vhodné pracovať v rámci návrhovej časti práce. Výber týchto technológií vychádza primárne z tých firmou zaužívaných, pre ktorú bude cieľový nástroj navrhnutý.

Ako bude aj neskôr v analytickej časti práce spomenuté, firma pri tvorbe automatizovaných testovacích nástrojov využíva programovací jazyk C#, umožňujúci objektovo orientované programovanie, v prostredí Microsoft Visual Studio (7, s. 161; 8). Preto je vhodné objasniť jednak pojmy s touto programovacou paradigmou súvisiace, ale aj niektoré záležitosti týkajúce sa vyslovene jazyka C#. Zároveň vzhľadom na očakávanú prácu s databázami dôjde k ujasneniu aj tejto problematiky. V neposlednom rade sa uvedú možné spôsoby práce s databázami s objektovo orientovaným programovacím jazykom v rámci softvéru Microsoft Visual Studio.

1.2.1 Objektovo orientované programovanie (OOP)

Objektovo orientované programovanie, ktoré sa skrátene zvykne uvádzať ako OOP, je prístup k vývoju softvéru, ktorého štruktúra je založená na objektoch navzájom interagujúcich s cieľom splniť určitú úlohu (9, s. 1).

OOP predpokladá, že program je simuláciou reálneho, resp. virtuálneho prostredia. Toto prostredie je spravidla tvorené objektmi, ktoré na seba navzájom vplývajú, a s ktorými by mal tento objektovo orientovaný program vedieť pracovať. Objekty možno rozdeliť do skupín objektov s podobnými vlastnosťami, ktoré nazývame triedy. Triedy potom popisujú vlastnosti a metódy týchto objektov. Objekt patriaci do triedy nazývame ako inštanciou danej triedy (10, s. 42). Za triedu je možné považovať napríklad triedu *Automobil*, ktorej inštanciou, teda objekt je konkrétne auto. Tento objekt môže mať naplnené triedou deklarované vlastnosti, ako je napríklad farba, značka alebo typ prevodovky. Medzi metódy triedy, teda činnosti vykonateľné objektom, by sme mohli zaradiť napríklad naštartovanie, zastavenie či parkovanie. Takúto triedu zaznamenáva aj nasledujúca ukážka vyjadrená pomocou grafického modelovacieho jazyka UML (skratka z anglického Unified Modeling Language).



Obrázok 5: OOP – príklad triedy – vyjadrenie pomocou UML
(Zdroj: Vlastné spracovanie podľa 11)

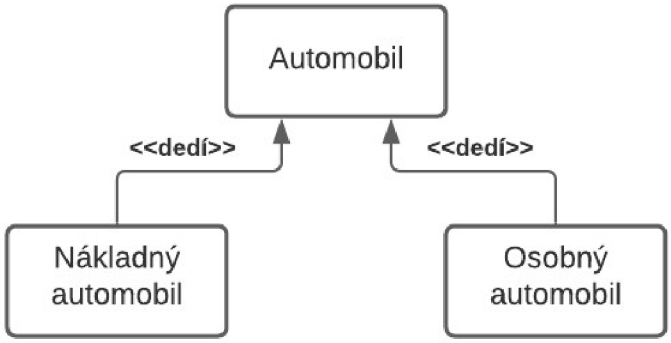
1.2.1.1 Typy vzťahov medzi objektmi v OOP

Keď sa objektovo orientovaný program spustí, dochádza pri vykonávaní úloh k spolupráci medzi objektmi. To znamená, že tieto objekty majú medzi sebou určitý typ vzťahu. Uveďme bežne vyskytujúce sa typy vzťahov medzi objektmi, resp. ich triedami (9, s. 17). Pre lepšie pochopenie každý vzťah vyjadríme aj využitím stručným diagramom tried v modelovacom jazyku UML (9, s. 8).

Asociácia je typ vzťahu, keď jedna trieda referuje na inú triedu. Môže dôjsť k referencii jednej inštancie jednej triedy na inú inštanciu, resp. viacero inštancií inej triedy, ale aj referencii jednej inštancie jednej triedy na inú inštanciu z tej istej triedy (9, s. 17-18). Prípad referencie jednej inštancie jednej triedy na jednu, resp. viacero inštancií inej triedy znázorňuje aj nasledujúca ukážka. Tá reprezentuje situáciu, keď jedna inštancia triedy *Človek* môže referovať na viacero inštancií triedy *Automobil*, t.j. jeden konkrétny človek môže vlastniť niekoľko automobilov, ale automobil môže byť vlastnený, t.j. evidovaný len na jedného človeka.

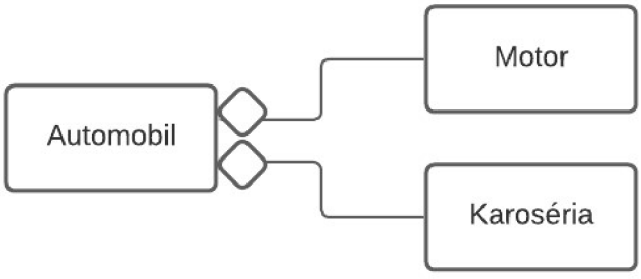
Obrázok 6: OOP – typy vzťahov – príklad asociácie – vyjadrenie pomocou UML
(Zdroj: Vlastné spracovanie podľa 9, s. 17)

Iným typom vzťahu je **dedenie**. To znamená, že viacero tried zdieľa rovnaké atribúty, teda vlastnosti a metódy základnej triedy, ktorú môžeme značiť ako rodičovskú. Zároveň platí, že tieto dediace triedy majú každá svoje vlastné špecifiká. Tento princíp umožňuje ľahšiu a intuitívnejšiu prácu s objektami, pretože umožňuje kombinovať všeobecné charakteristiky do rodičovskej triedy, ktoré sú následne dedené jej dedičmi (9, s. 5-6, 18). Príkladom môže byť trieda *Automobil*, ktorá je triedou rodičovskou, teda z ktorej sú dedené vlastnosti a metódy jej dedičmi. Jej dedičmi môžu byť napríklad trieda *Nákladný automobil* a trieda *Osobný automobil*, pričom každá má svoje vlastné špecifiká.



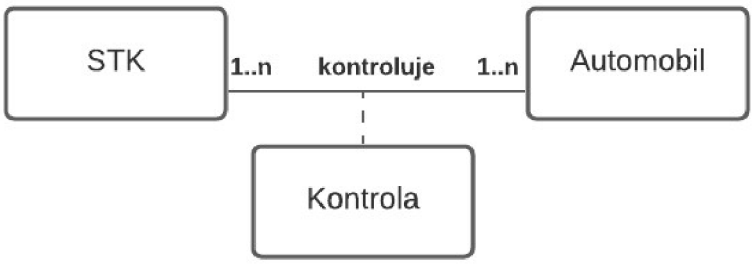
Obrázok 7: OOP – typy vzťahov – príklad dedenia – vyjadrenie pomocou UML
(Zdroj: Vlastné spracovanie podľa 9, s. 18)

Tretím typom vzťahu, ktorý si uvedieme, je **agregácia**, resp. kompozícia. Ide o prípady, keď je trieda zložená z viacerých tried (9, s. 6). Pre lepšiu predstavu tohto vzťahu si môžeme uviesť príklad triedy *Automobil*, ktorá pozostáva napríklad z triedy *Motor* a triedy *Karoséria*.



Obrázok 8: OOP – typy vzťahov – príklad agregácie – vyjadrenie pomocou UML
(Zdroj: Vlastné spracovanie podľa 9, s. 18)

Posledným typom vzťahu, ktorý bude v rámci tejto práce zmienený, je prípad, keď atribút nie je priraditeľný k žiadnej z existujúcich tried, ale je výsledkom ich vzájomnej asociácie. Takúto **triedu** možno značiť ako **asociačnú** (9, s. 19). Ako príklad si môžeme uviesť stanicu technickej kontroly, ktorá posudzuje rôzne automobily, pričom automobil môže byť posúdený v rôznych staniciach technickej kontroly. Potom pre vyjadrenie atribútu ako je napríklad cena za kontrolu je potrebný vznik asociačnej triedy.



Obrázok 9: OOP – typy vzťahov – príklad asociačnej triedy – vyjadrenie pomocou UML
(Zdroj: Vlastné spracovanie podľa 9, s. 19)

1.2.1.2 Základné princípy OOP

V nasledujúcich odsekoch dôjde k predstaveniu niektorých charakteristík, resp. základných princípov typických pre všetky objektovo orientované programovacie jazyky. Medzi tieto princípy možno zaradiť aj už zmienený princíp **dedenia** (9, s. 3-5).

Ďalším z princípov je **abstrakcia**. Abstrakcia značí, že triedy zahŕňajú len tie informácie, ktoré sú relevantné v kontexte daného programu (9, s. 3). Pre príklad môžeme uviesť, že síce každý automobil disponuje vlastnosťou týkajúcou sa počtu dverí, no nie v každom kontexte je táto informácia hodná zaznamenania.

Ďalším základným princípom je **zapuzdrenie**, ktorý spočíva v neposkytovaní priameho prístupu k údajom. To znamená, že pre ich získanie je potrebné interagovať s objektom súvisiacim s týmito údajmi. Objektu je potrebné zaslať správu žiadajúcu o poskytnutie týchto údajov, pričom on, pokiaľ má takúto možnosť operácie v sebe definovanú, tieto údaje následne poskytne (9, s. 4).

Posledný OOP princíp, ktorý bude v tejto práci spomenutý, je **polymorfizmus**. Tento pojem značí, že rôzne triedy môžu reagovať na rovnakú správu odlišne, teda v ich

vlastnom jedinečnom tvare implementácie. Zjednodušeným príkladom z reálneho života môže byť príkaz vydaj zvuk, ktorý bude mať v prípade psa implementáciu v podobe štekania, v prípade mačky v podobe mňaukania. Je teda možné zaviesť viaceré metódy rovnakého značenia avšak iného obsahu v závislosti od ich kontextu, teda objektu, ku ktorému sa vzťahujú (9, s. 4).

1.2.1.3 Výhody OOP oproti iným programovacím paradigmám

Objektovo orientovaný prístup k programovaniu prináša vo všeobecnosti množstvo benefitov, a to najmä pri programoch väčšieho rozsahu. Medzi tieto výhody možno zaradiť napríklad schopnosť udržiavať a implementovať zmeny v programoch efektívnejšie a rýchlejšie, schopnosť vytvárať pre používateľov intuitívnejšie graficko-užívateľské rozhranie, či schopnosť opakovane používať komponenty kódu v iných programoch (9, s. 2-3).

1.2.1.4 Programovací jazyk C#

Ako sme už v úvode podkapitoly „Technológie použiteľné pri návrhu nástroja“ uviedli, medzi programovacie jazyky umožňujúce objektovo orientované programovanie možno zaradiť aj jazyk C# (7, s. 161; 8). Práve ten je vo firme využívaný, a preto bude použitý aj v návrhovej časti tejto práce. Pre jeho lepšie pochopenie považujeme za vhodné objasnenie niektorých jeho základných aspektov.

Premenné a ich dátové typy v C#

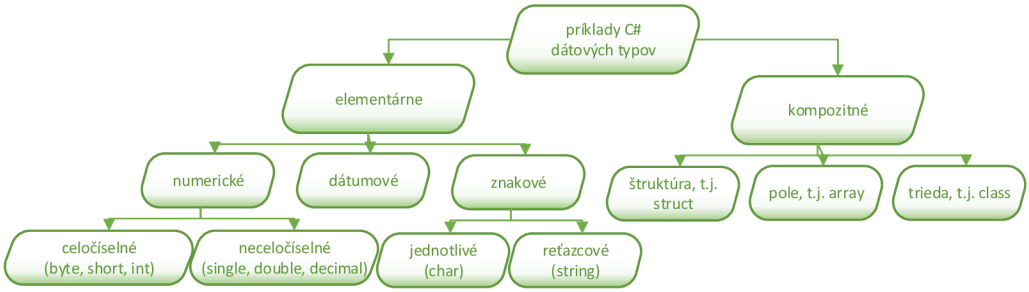
Ako každý programovací jazyk aj C# pracuje s premennými. Pri ich deklarácii a následnom použití je potrebné brať na vedomie skutočnosť, že C# je programovací jazyk, ktorý rozlišuje veľké a malé písmená, čiže je tzv. case-sensitive. To znamená, že premenná označená ako *počítadlo* a premenná *Počítadlo* sú dve rozličné premenné (7, s. 56). V súvislosti s deklaráciou premenných možno uviesť, že premenné môžu byť rozličného dátového typu. Tie možno rozdeliť na elementárne a kompozitné (9, s. 325).

V rámci elementárnych dátových typov možno diskutovať numerické (napr. byte, short, int, long, single, double, decimal), znakové (napr. char, string) dátové typy a dátové typy súvisiace s dátumom (9, s. 325-327).

Kompozitné dátové typy sú dátové typy zložené z tých elementárnych, prípadne iných kompozitných. Medzi ne možno zaradiť štruktúry (struct), ale taktiež polia (array)

slúžiace pre prácu so skupinami dát rovnakého typu, a triedy (class) majúce rôzne vlastnosti (anglicky fields) či metódy (9, s. 327-329). Tieto vlastnosti tried môžu mať rôzne modifikátory prístupu, a to napríklad verejný (t.j. public) alebo privátny (t.j. private). Tie určujú, či daná vlastnosť môže byť viditeľná aj v rámci inej triedy. V prípade, že je modifikátor prístupu verejný, t.j. public, tak táto vlastnosť môže byť viditeľná aj v inej triede. Ak je privátny, čiže private, potom ju iná trieda vidieť nemôže. Vďaka privátnym modifikátorom prístupu je možné docieliť zapuzdrenie, čo je, ako sme už v predchádzajúcich častiach práce uviedli, jednou zo základných charakteristík objektovo orientovaného programovania (11).

Zmienené dátové typy možno zhrnúť do nasledujúceho zobrazenia.



Obrázok 10: Členenie dátových typov v C# s príkladmi
(Zdroj: Vlastné spracovanie podľa 9, s. 325-329)

SOLID princípy

V súvislosti s návrhom nástroja použitím jazyka C# je užitočné zmieniť SOLID princípy poskytujúce odporúčania, ktorými by sa mal vývojár pri práci s objektovo orientovaným programovaním v ideálnom prípade riadiť. Termín SOLID je akronymom odvodeným od počiatočných písmen názvov jednotlivých princípov pomenovaných v anglickom jazyku (12). Týmito princípmi máme na mysli:

- S ako single responsibility principle preložiteľný aj ako princíp jednej zodpovednosti. Tento princíp znamená, že každá trieda by mala mať len jeden konkrétny význam, t.j. mala by existovať len pre jednu úlohu, ktorú má splniť. To znamená, že by nemal jestvovať viac ako jeden dôvod, prečo by malo dôjsť k jej zmene. Zároveň platí, že trieda by mala byť primeraného rozsahu. V opačnom prípade je potrebné jej rozdelenie na niekoľko menších tried (12).

- O ako open closed principle je princíp tvrdiaci, že kód by mal byť otvorený, teda prístupný pre jeho rozšírenie, avšak mal by byť uzavretý, čože neprístupný pre jeho modifikáciu (12).
- L ako Liskov substitution principle je princíp spočívajúci v existencii možnosti nahradenia triedy jej podradenou triedou bez potreby akýchkoľvek úprav (12).
- I ako interface segregation principle je princíp rozdelenia rozhrania na niekoľko menších (12).
- D ako dependency inversion principle je princíp vychádzajúci z používania abstrakcie. To znamená, že moduly by nemali byť priamo na sebe závislé, ale mali by byť závislé na nimi zdieľanej abstrakcii, čím môže byť napríklad abstraktná trieda, resp. rozhranie (anglicky interface) (12).

Konvencie pri pomenovávaní v C#

Práca so C# má určité konvencie týkajúce sa pomenovávania jednotlivých súčastí kódu. Uveďme niektoré z nich (11):

- názov triedy sa začína veľkým písmenom
- značenie inštancií tried, teda objektov sa zvykne písať malým písmenom
- názvy vlastností v rámci tried sa začínajú veľkým písmenom, avšak v prípade privátneho modifikátoru prístupu začínajú podčiarkovníkom
- označenie metódy začína veľkým písmenom, zatiaľ čo jej argumenty začínajú malým písmenom
- názvy lokálnych premenných sa značia malým písmenom
- pomenovanie rozhrania sa značí veľkým písmenom, pričom pred tento názov sa vkladá písmeno „I“ vychádzajúceho z anglického označenia rozhrania

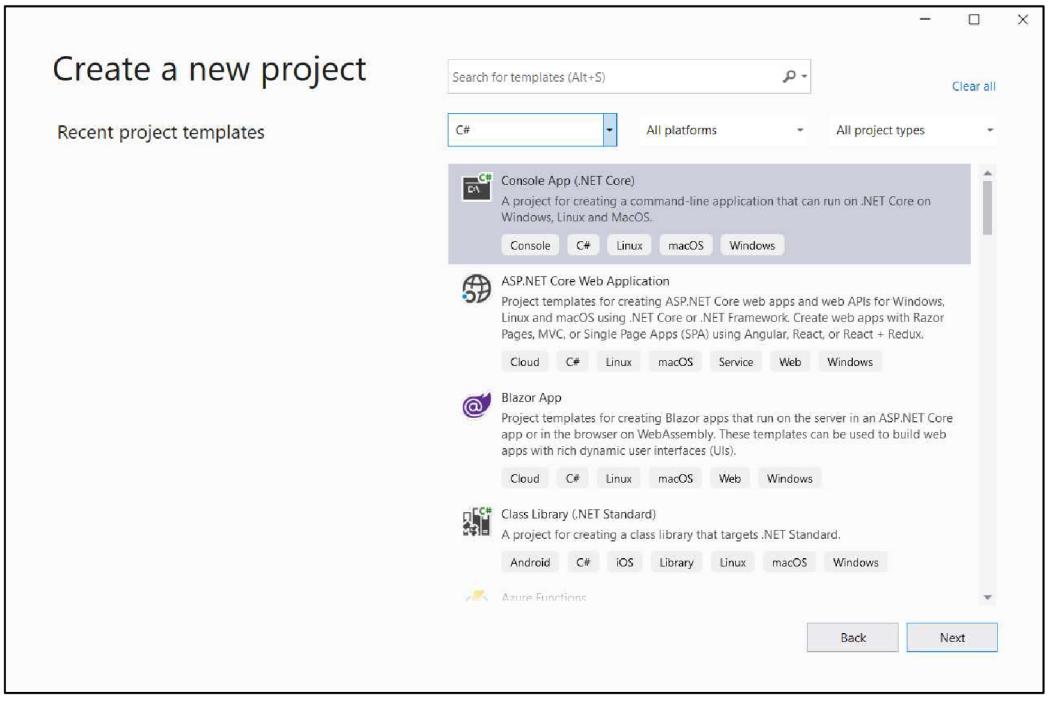
Pre návrh aplikácie využitím programovacieho jazyka C# je vhodný napríklad **softvér Microsoft Visual Studio** (7, s. 35).

1.2.1.5 Softvér Microsoft Visual Studio

Microsoft Visual Studio je IDE (z anglického Integrated Development Environment), čiže integrované vývojové prostredie, čo znamená softvér slúžiaci na vytváranie aplikácií, ktorý kombinuje bežné vývojárske nástroje do jedného grafického používateľského rozhrania, skrátene aj ako GUI (z anglického Graphical User Interface). IDE zvyčajne

pozostáva z troch častí. Jednou z nich je editor zdrojového kódu pomáhajúci pri písaní kódu, a to napríklad zvýraznením syntaxe pomocou vizuálnych podnetov alebo kontrolovaním chýb pri písaní kódu. Druhou súčasťou je automatizácia lokálneho zostavenia (anglicky build) spočívajúca aj v kompilácii zdrojového kódu počítača do binárneho kódu. Poslednou typickou súčasťou IDE je tzv. debugger (13).

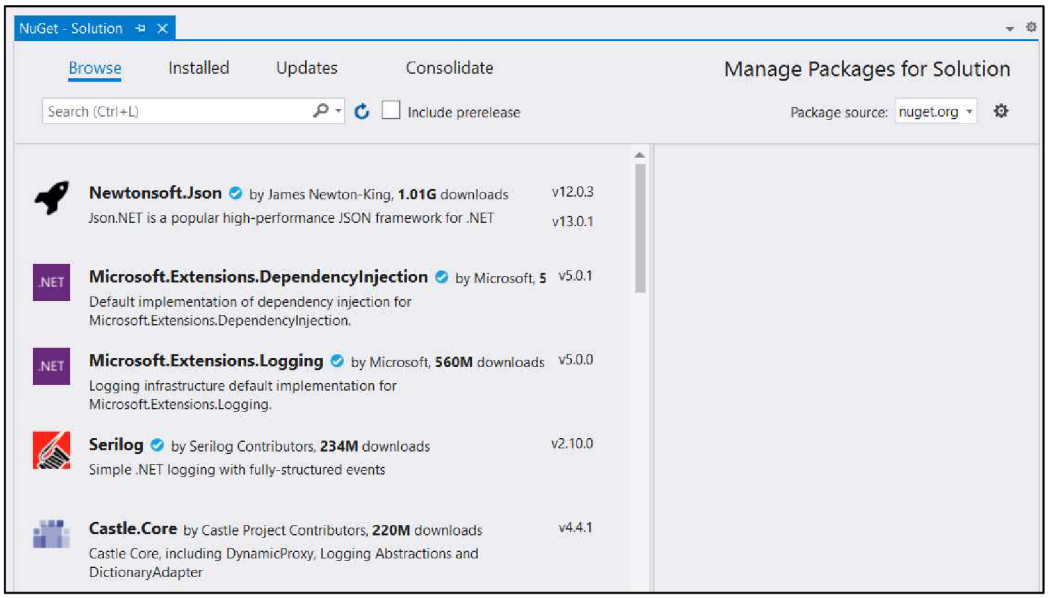
Toto IDE umožňuje pracovať v rámci riešenia (anglicky solution) s jedným, ale i viacerými projektmi (14, s. 9). Tieto projekty môžu byť rôzneho typu, ktorý si užívateľ softvéru Microsoft Visual Studio zvolí na základe svojich potrieb (14, s. 67). Niektoré z nich vidno v rámci nižšie umiestnenej ukážky, u ktorej došlo k využitiu možnosti filtrácie na nami diskutovaný programovací jazyk C#. Jedná sa konkrétne o prostredie Microsoft Visual Studio Community 2019.



Obrázok 11: Microsoft Visual Studio – zakladanie nového projektu
(Zdroj: Vlastné spracovanie podľa 14, s. 65)

Považujeme za hodné spomenutia skutočnosť, že Microsoft Visual Studio umožňuje prácu s tzv. NuGet Packages. Ide o ZIP súbory s .nupkg príponou, ktoré umožňujú znovu použitie už skompilovaného kódu vďaka jeho zdieľaniu s vývojármi zo širokej verejnosti,

resp. s vývojármi v rámci určitej organizácie. Ide vlastne o knižnice kódov, ktoré umožňujú pridať k vyvíjanému riešeniu už niekým spracovanú funkcionalitu. Zoznam týchto balíčkov spolu s rozličnými informáciami a možnosť ich pridania je k dispozícii priamo v rámci Microsoft Visual Studio (14, s. 73-77). Na nasledujúcej snímke prostredia softvéru Microsoft Visual Studio vidíme, ako vyzerá zoznam dostupných balíčkov s možnosťou filtrovania medzi nimi. Spomedzi nich si môže užívateľ softvéru Microsoft Visual Studio vybrať a následne ho nainštalovať do jeho riešenia. Jedným z takýchto balíčkov je napríklad Newtonsoft.Json, ktorý umožňuje prácu s údajmi v JSON formáte.



Obrázok 12: Microsoft Visual Studio – možnosť pridania balíka z NuGet Packages k vyvíjanému riešeniu

(Zdroj: Vlastné spracovanie podľa 14, s. 77)

1.2.2 Relačné databázy

Je očakávané, že nástroj bude pracovať nad relačnou databázou, a preto je súce objasniť niektoré základné pojmy týkajúce sa tejto problematiky.

Databázu môžeme definovať ako usporiadanú sústavu dát. Jedným z databázových modelov je vyššie spomenutý **relačný model**, ktorého cieľom je umožniť konzistentnú reprezentáciu dát s minimálnou, resp. žiadnou redundanciu, a to bez úkoru na kompletnosť. Tento model spočíva v uložení dát do tabuliek pozostávajúcich z riadkov a stĺpcov. Stĺpec, značený aj ako atribút, je množinou údajov určitého dátového typu. Riadok,

značený aj ako záznam, je kombináciou stĺpcových hodnôt v rámci tabuľky (15, s. 11-12; 16, s. 46; 17, s. 5).

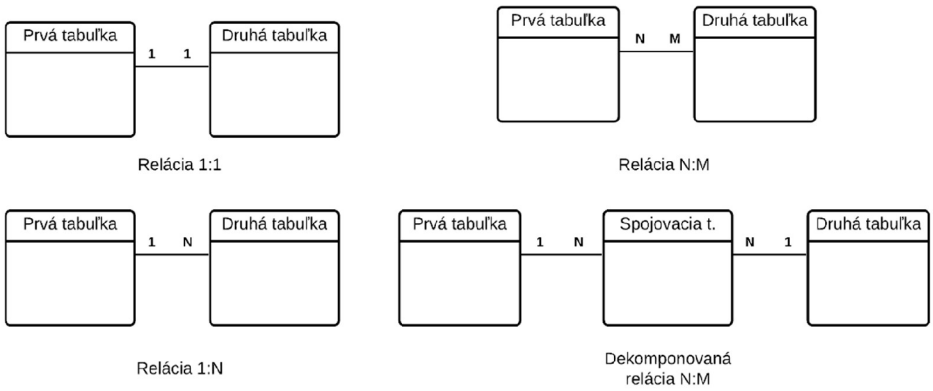
Každý záznam v tabuľke by mal byť jednoznačne identifikovateľný, a to za pomoci **primárneho kľúča** (anglicky primary key), vybraného spomedzi **kandidátnych kľúčov** (anglicky candidate key), teda kľúčov, ktoré by mohli zastávať funkciu toho primárneho. Primárny kľúč môže byť tvorený jedným (vtedy hovoríme o jednoduchom primárnom kľúči), resp. viacerými (vtedy hovoríme o kompozitnom primárnom kľúči) stĺpcami tabuľky, pričom hodnota poľa primárneho kľúča musí byť jedinečná. Primárny kľúč umožňuje vytvárať medzi tabuľkami väzby, s čím úzko súvisí **cudzí kľúč** (anglicky foreign key). Cudzí kľúč je stĺpec, resp. kombinácia stĺpcov, ktoré sú prepojené na primárny kľúč z inej tabuľky (16, s. 46-49; 17, s. 7).

O nutnosti jednoznačnej identifikácie každého záznamu v rámci relačného modelu hovorí aj jedno z integritných obmedzení. **Integritné obmedzenia** sú pravidlá pre zaistenie správnosti a konzistencie uložených dát. Poznáme tri typy, a to entitnú integritu, doménovú integritu a referenčnú integritu. Entitná spočíva v možnosti jednoznačnej identifikácie každého riadku, a to využitím kandidátneho, resp. primárneho kľúča. Doménová spočíva v zaistení, aby každá hodnota atribútu bola z množiny hodnôt prípustných pre daný atribút. Referenčná integrita diskutuje cudzie kľúče tvrdiac, že tieto kľúče nemôžu nadobúdať iné hodnoty, ako sú hodnoty povolené pre primárny kľúč, s ktorými tvoria väzbu (16, s. 47-48; 17, s. 6).

S relačným modelom súvisí **entitno-relačné modelovanie**, v rámci ktorého dochádza k návrhu štruktúry databázy. Pri vypracovávaní entitno-relačného diagramu v prvom rade dochádza k identifikácii entít, teda objektov reálneho sveta schopných samostatnej existencie a zároveň schopných oddelenia od ostatných objektov. Ďalej sa identifikujú vzťahy medzi týmito entitami a priraďujú sa im atribúty umožňujúce popísanie vlastností týchto entít a vzťahov medzi nimi (16, s. 52).

Pri návrhu entitno-relačného diagramu môžeme pracovať s niekoľkými typmi vzťahov medzi entitami, čiže tabuľkami navrhovanej databázy z hľadiska ich **kardinality**. Tieto relácie medzi tabuľkami môžu byť buď typu 1:1, typu 1:N alebo typu N:M. V prípade 1:1 je možné záznamu prvej tabuľky priradiť maximálne jeden záznam druhej tabuľky. To platí aj opačne. V prípade 1:N môže záznamu prvej tabuľky odpovedať viacero záznamov druhej tabuľky, avšak záznamu druhej tabuľky môže odpovedať najviac jeden

záznam prvej tabuľky. Prípad M:N znamená, že záznamom z prvej tabuľky môže odpovedať viacero záznamov z druhej tabuľky, pričom toto tvrdenie platí aj opačne. S takouto situáciou však nevie pracovať väčšina databázových systémov, a preto je potrebná dekompozícia, čiže implementácia tohto vzťahu pomocou spojovacej tabuľky, čím dôjde k rozloženiu tohto vzťahu na dva vzťahy typu 1:N (16, s. 53-55). Tieto vzťahy si môžeme zhrnúť do nasledujúcej ukážky.



Obrázok 13: Kardinalita vzťahov medzi tabuľkami v entitno-relačnom modeli
(Zdroj: Vlastné spracovanie podľa 16, s. 53-55)

Pri návrhu entitno-relačného modelu je vhodné brať do úvahy existenciu **normalizačných pravidiel**, ktorých cieľom je ponechanie minimálnej redundancie bez obetovania úplnosti. Stručne si spomeňme prvé tri z nich. Prvá normalizačná forma tvrdí, že každý riadok tabuľky musí byť jedinečný a zároveň, že jeho jednotlivé atribúty musia byť atomické, čiže jednoduché, nie viachodnotové. Pre splnenie druhého pravidla musí byť splnené to prvé a zároveň všetky nekľúčové atribúty musia byť plne závislé na primárnom, resp. kandidátnom kľúči. Tretia normalizačná forma tvrdí, že musia byť dodržané prvé dve normalizačné formy a zároveň platí, že atribúty netvoriace primárny, resp. kandidátny kľúč sú na sebe navzájom nezávislé (17, s. 7-9).

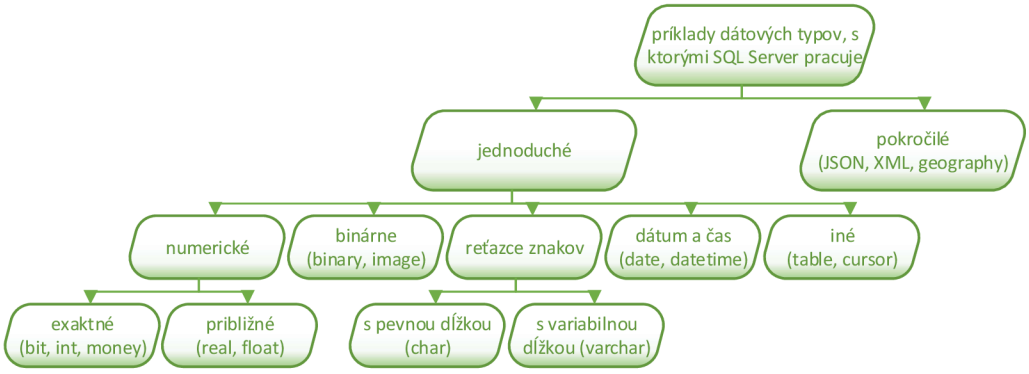
V neposlednom rade si uvedieme, že pri návrhu relačnej databázy je vhodné dodržiavať nasledovné **konvencie** týkajúce sa tvorby názvov. Všetky názvy, t.j. vrátane názvov stĺpcov, tabuliek v rámci databázy, by mali byť regulárnymi identifikátormi, čiže by mali obsahovať len písmená bez diakritiky, číslice, prípadne podčiarkovník. Ďalej platí, že všetky názvy by mali byť charakteristické pre to, čo pomenúvajú. Je vhodné sa vyvarovať

používaniu akronymov, teda skratkám vytvoreným z počiatočných písmen jednotlivých slov (Laurečník, s. 25).

Pre prácu s relačnými databázami bol vyvinutý dotazovací jazyk **SQL** (akronym z anglického Structured Query Language). V zásade však platí, že relačný model je nezávislý na jazyku, ktorý s ním pracuje, čo znamená, že je s ním možné interagovať nielen s jazykom SQL, ale napríklad aj s jazykom C# (15, s. 11-12; 17, s. 2).

Koncepcia relačných databáz sa používa ako základ pre databázové systémy, akým je napríklad aj našou firmou používaný **Microsoft SQL Server**. Pod pojmom databázový systém si možno predstaviť jednak samotnú databázu, a jednak systém riadenia bázy dát. **SQL Server Management Studio** (skrátene ako SSMS) je klientsky nástroj umožňujúci užívateľovi pracovať s databázovými systémami ako je aj zmienený Microsoft SQL Server (15, s. 11-12; 16, s. 45; 17, s. 387).

Microsoft SQL Server je schopný pracovať s množstvom rozličných **dátových typov**. Túto širokú škálu možno rozdeliť na jednoduché a pokročilé dátové typy (18, s. 1, 25). To, čo si možno predstaviť pod jednoduchými a pokročilými dátovými typmi, znázorňuje nasledujúce zobrazenie:



Obrázok 14: Členenie dátových typov v SQL Server s príkladmi
(Zdroj: Vlastné spracovanie podľa 18, s. 1-25)

Považujeme za dôležité zdôrazniť existenciu JSON a XML ich stručným vysvetlením.

XML (z anglického Extensible Markup Language) je značkovací jazyk podobný HTML, ktorý bol navrhnutý za účelom ukladania a prenosu údajov. Rovnako ako HTML aj XML sa skladá zo značiek (anglicky tags) ohraničujúcich jednotlivé elementy, avšak na rozdiel od HTML tieto značky nie sú preddefinované. Namiesto toho ich definuje autor

dokumentu vychádzajúc zo svojich potrieb. Dokument XML má stromovú štruktúru, ktorá začína koreňovým uzlom a obsahuje podradené uzly. Pre lepšie pochopenie tejto štruktúry si zobrazme ukážku, ako môžu údaje v XML formáte napríklad vyzerať. Táto ukážka sa týka príkladu zoskupenia predajcov (*SalesPeople*) obsahujúceho jednotlivých predávajúcich (*SalesPerson*) s informáciami týkajúcimi sa ich identifikátora (*PersonId*), celého mena (*FullName*), telefónneho čísla (*PhoneNumber*) a e-mailovej adresy (*EmailAddress*). Za koreňový uzol považujeme *SalesPeople* (18, s. 29, 188-191).

```
<SalesPeople>
   <SalesPerson>
      <PersonId>1</PersonId>
      <FullName>Kayla Wood</FullName>
      <PhoneNumber>(415) 555-0102</PhoneNumber>
      <EmailAddress>kaylaw@company.com</EmailAddress>
   </SalesPerson>
   <SalesPerson>
      <PersonId>2</PersonId>
      <FullName>Hudson Onslow</FullName>
      <PhoneNumber>(415) 555-0102</PhoneNumber>
      <EmailAddress>hudsono@company.com</EmailAddress>
   </SalesPerson>
</SalesPeople>
```

Obrázok 15: XML formát – príklad
(Zdroj: Vlastné spracovanie podľa 18, s. 188-191)

Za alternatívu k XML možno považovať **JSON** (akronym z JavaScript Object Notation) formát. JSON je podobný XML v možnosti jeho zostavenia podľa svojich potrieb a možnosti hierarchizovania údajov. Na rozdiel od XML sú však JSON dokumenty kratšieho rozmeru, ľahšie čitateľné, a teda rýchlejšie zanalyzovateľné. Vďaka týmto skutočnostiam sa stal JSON formát veľmi populárnym medzi vývojármi aplikácií (18, s. 181, 194).

Syntax u JSON formátu spočíva v tvorbe párov pozostávajúcich z názvu ohraničeného úvodzovkami a príslušnej hodnoty. Ohraničenosť príslušnej hodnoty úvodzovkami sa odvíja od jej dátového typu. Pre ich zrejmé oddelenie od seba je využívaná dvojbodka. JSON objekt, ktorý môže pozostávať z viacerých takýchto párov deliacich sa od seba čiarkou, je ohraničený množinovými zátvorkami. Viacero objektov tvorí pole. To sa ohraničuje hranatými zátvorkami. Príkladom tohto poľa môžu byť riadky tabuľky, pričom jeho objektami budú jednotlivé riadky. Pármi v rámci týchto objektov potom budú

hodnoty v stĺpcoch (18, s. 181-184). Ukážka nižšie znázorňuje príklad, na ktorom bol demonštrovaný aj XML formát, aby bolo možné lepšie vidieť rozdiely medzi nimi. Objekt *SalesPeople*, teda predávajúcich, vytvára pole objektov jednotlivých predajcov. Tieto objekty sú tvorené pármi pozostávajúcimi z názvu (ako je napríklad identifikátor predajcu *PersonId*, jeho celé meno *FullName*, telefónne číslo *PhoneNumber* či e-mailová adresa *EmailAddress*) a k ním príslušnej hodnoty.

```
{
   "SalesPeople":[
      {
         "PersonId":1,
         "FullName":"Kayla Wood",
         "PhoneNumber":"(415) 555-0102",
         "EmailAddress":"kaylaw@company.com"
      },
      {
         "PersonId":1,
         "FullName":"Hudson Onslow",
         "PhoneNumber":"(415) 555-0102",
         "EmailAddress":"hudsono@wcompany.com"
      }
   ]
}
```

Obrázok 16: JSON formát – príklad
(Zdroj: Vlastné spracovanie podľa 18, s. 191-194)

Vidíme, že táto ukážka, t.j. ukážka v JSON formáte je kratšieho rozsahu a lepšie čitateľná ako tá, uvedená v XML formáte.

1.2.3 Objektovo-relačné mapovanie

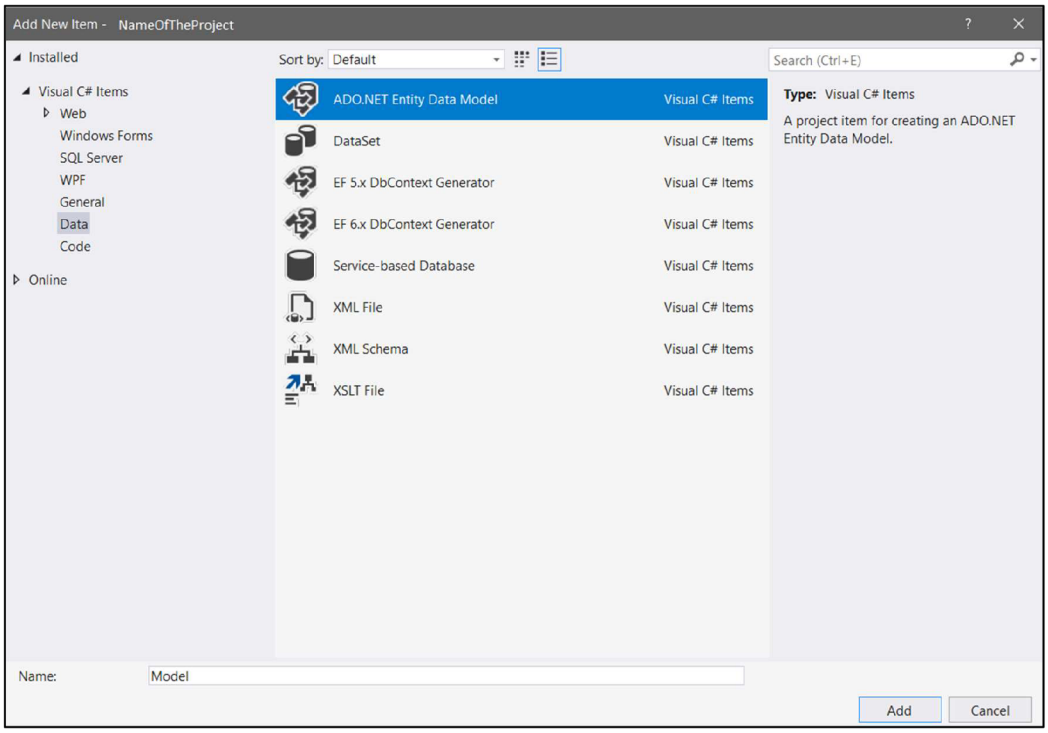
Relačný svet a objektovo orientovaný svet sa od seba značne sémanticky a syntakticky líšia. Pre možnosť ich vzájomného prepojenia bolo vyvinuté objektovo-relačné mapovanie. Toto mapovanie umožňuje preklad tried, atribútov a vzťahov medzi triedami z objektovo orientovaného sveta do tabuliek, stĺpcov a cudzích kľúčov z relačného sveta. Objektovo-relačné mapovanie teda umožňuje vývojárom písať objektovo orientované aplikácie, ktoré interagujú s údajmi uloženými vo forme záznamov v tabuľkách relačnej databázy. Vývojár využitím tejto technológie nemusí pri práci využívať akékoľvek SQL príkazy (19, s. 1-2).

Medzi tieto technológie patrí napríklad klasický ADO.NET Entity Framework a Entity Framework Core. Poslednou verziou klasického Entity Framework je verzia 6 (19, s. 3-4, 9).

Technológia Entity Framework Core má veľmi podobné možnosti a výhody ako technológia Entity Framework 6. Platí však, že Entity Framework Core nemusí obsahovať všetky schopnosti mapovania figurujúcich v rámci technológie Entity Framework 6 (19, s. 8-9; 20). Z týchto dôvodov sa v rámci práce zameriame na Entity Framework 6.

To, ako Entity Framework 6 ukladá informácie o mapovaní tried aplikácie k tabuľkám v databáze, sa odvíja od dvoch základných vecí. Prvou z nich je skutočnosť, či je potrebné vytvoriť novú databázu alebo sa bude pristupovať k už existujúcej. Druhým vplyvom je preferencia vývojára, či uprednostňuje tvorbu vlastného kódu (tzv. Code First prístup) alebo chce využiť nástroj Entity Framework Designer (21).

Pre spustenie nástroja Entity Framework Designer v rámci Microsoft Visual Studio Community 2019 je potrebné pridanie novej položky (anglicky add new item) ADO.NET Entity Data Model k projektu a následné zvoliť adekvátnu voľbu vychádzajúc z toho, či je potrebné pripojenie sa k už existujúcej databáze alebo vytvorenie novej. Potom vývojár postupuje podľa jednotlivých výziev (22).



Obrázok 17: Microsoft Visual Studio – Entity Framework - využitie Entity Framework Designer
(Zdroj: Vlastné spracovanie podľa 22)

V prípade tvorby modelu uprednostnením generovania vlastného kódu, t.j. Code First prístupu je možné pri vytváraní doposiaľ nejestvujúcej databázy postupovať nasledovne.

V prvom rade je potrebná inštalácia balíčka Entity Framework spomedzi NuGet Packages. Potom sa môže definovať odvodený *DbContext* predstavujúci reláciu s databázou. V rámci tohto kontextu sa pomocou *DbSet<T>* definujú všetky tabuľky tejto databázy. Dodáme, že existuje možnosť využitia anotácií, ktoré umožňujú popísanie napríklad toho, či sa jedná o primárny jednoduchý, resp. zložený kľúč, či sa jedná o cudzí kľúč, ale aj zadanie názvu, s akým bude tabuľka v rámci databázy figurovať. V neposlednom rade treba pripomenúť vhodnosť nastavenia automatického prejavovania sa vykonaných zmien v modeli do databázy využitím migrácií (23; 24; 25; 7, s. 861).

```
using System.Collections.Generic;
using System.Data.Entity;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace EF
{
        public class BloggingContext : DbContext
        {
            public DbSet<Blog> Blogs { get; set; }
            public DbSet<Post> Posts { get; set; }

            public BloggingContext() : base("ConnectionToOutputDB")
            {
            }
        }

        [Table("dbo.Blogs")]
        public class Blog
        {
            [Key]
            public int BlogId { get; set; }
            public string Name { get; set; }

            public virtual List<Post> Posts { get; set; }
        }

        [Table("dbo.Posts")]
        public class Post
        {
            [Key]
            public int PostId { get; set; }
            public string Title { get; set; }
            public string Content { get; set; }

            public int BlogId { get; set; }
            public virtual Blog Blog { get; set; }
        }
}
```

Obrázok 18: Microsoft Visual Studio – Entity Framework – využitie príklad použitia prístupu Code First

(Zdroj: Vlastné spracovanie podľa 23)

Odvodený databázový kontext referuje na nasledovný úsek v rámci konfiguračného súboru *App.config*, ktorý definuje detaily týkajúce sa danej databázy.

```
<connectionStrings>
  <add
    name="ConnectionToOutputDB"
    providerName="System.Data.SqlClient"
    connectionString=
        "Data Source = localhost;
        Initial Catalog = blogovanie;
        Integrated Security = True;
        MultipleActiveResultSets = True"
  />
</connectionStrings>
```

Obrázok 19: Microsoft Visual Studio – Entity Framework – využitie príklad použitia prístupu Code First – nastavenie pripojenia k databáze

(Zdroj: Vlastné spracovanie)

Po úspešnom zostavení modelu spočívajúceho vo vytvorení prepojenia na databázu, a to použitím Entity Framework, máme možnosť, napríklad v rámci objektovo orientovaného programovania jazykom C#, využitia **LINQ** (z anglického Language Integrated Query). LINQ umožňuje vývojárom písať dotazy v syntaxi C#, ktoré následne tlmočí do syntaxe poskytovateľa dát, v našom prípade relačnej databázy. Po ich získaní sú preložené naspäť do jazyka objektového modelu (9, s. 184).

Príklad jeho využitia zobrazuje nasledujúca ukážka. V rámci nej dochádza v prvom rade k vytvoreniu inštancie triedy databázového kontextu *BloggingContext* značenej ako *context*. Platí, že v takýchto situáciách je vhodné užitie bloku *using*. V rámci ukážky je možno vidieť, že pomocou vytvorenia tejto inštancie a pomocou entity v podobe tabuľky *Blogs* dochádza k zisteniu prostredníctvom LINQ a následnému zobrazeniu v rámci konzolovej aplikácie údaju týkajúceho sa identifikátora (*BlogId*) prvého blogu v poradí v rámci tabuľky, ktorého meno (*Name*) znie *Moj prvy blog*.

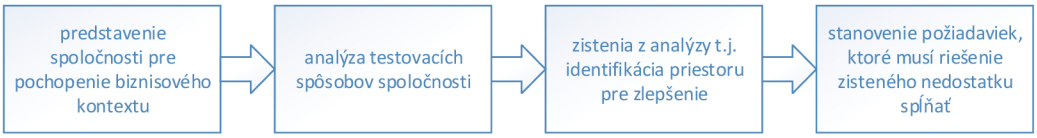
```
using System;
using System.Linq;

namespace EF
{
    class Program
    {
        static void Main()
        {
            using (var context = new BloggingContext())
            {
              var name = "Moj prvy blog";
              var blog = context.Blogs.Where(x => x.Name == name).FirstOrDefault();
              Console.WriteLine(blog.BlogId);
            }
        }
    }
}
```

Obrázok 20: : Microsoft Visual Studio – príklad použitia LINQ

(Zdroj: Vlastné spracovanie)

2 ANALÝZA SÚČASNÉHO STAVU

V rámci analytickej sekcie tejto záverečnej práce dôjde v prvom rade k predstaveniu spoločnosti, pre ktorú bude testovací nástroj tvorený, a to jednak z hľadiska jej predmetu podnikania, ale i súčasného stavu v kontexte testovania. Následne sa identifikujú nedostatky v týchto testovacích spôsoboch firmy a zároveň sa presne pomenuje vybraný, ktorý sa bude návrh automatizovaného integračného testovacieho nástroja snažiť zlepšiť. Taktiež dôjde k stanoveniu požiadaviek, ktoré by mal nástroj spĺňať. Tieto skutočnosti, t.j. obsah analytickej časti práce, si možno vyjadriť pre lepšiu prehľadnosť aj nasledovným spôsobom:



Obrázok 21: Vizualizácia kapitoly Analýza súčasného stavu
(Zdroj: Vlastné spracovanie)

2.1 Predstavenie spoločnosti

Návrh nástroja pre automatizované integračné testovanie je vytváraný na mieru pre LOGEX Solution Center s.r.o., ktorej základné identifikačné údaje možno zhrnúť do nasledujúcej tabuľky:

Tabuľka 1: Základné informácie o LOGEX Solution Center s.r.o.
(Zdroj: 26)

Obchodná firma	LOGEX Solution Center s.r.o
Identifikačné číslo	06111904
Dátum vzniku a zápisu	19.5.2017
Sídlo	Nové sady 996/25, Staré Brno, 602 00 Brno
Predmet podnikania	výroba, obchod a služby neuvedené v prílohách 1 až 3 živnostenského zákona
Spoločník	LOGEX Group B.V., 1077XX Amsterdam, Strawinskylaan 1329, Holandsko, obchodný podiel vo výške 100 %

Z vyššie uvedených informácií je zrejmé, že obchodný podiel 100 % na LOGEX Solution Center s.r.o. má LOGEX Group B.V. sídliaci v Holandsku.

LOGEX Group B.V., teda materská spoločnosť LOGEX Solution Center s.r.o., je taktiež vlastníkom firiem Value2Health, MRDM, IVBAR a Prodacapo. LOGEX Solution Center je hlavné vývojové centrum tohto zoskupenia (27).

Obrázok 22: LOGEX Group B.V. štruktúra
(Zdroj: 28)

Spoločnosť LOGEX Group B.V. vznikla pred viac ako 10 rokmi v Holandsku v nadväznosti na zmenu spôsobu financovania lekárskych zariadení. Tie prestali byť financované štátom a stali sa bežnou súčasťou trhového prostredia. Nemocnice začali teda prirodzene prísnejšie kontrolovať finančnú efektivitu svojej prevádzky. A práve pre tieto účely, t.j. účely napomáhania nemocniciam a lekárskym špecialistom zefektívniť a zlepšiť zdravotnú starostlivosť a finančné riadenie zdravotných zariadení, spoločnosť vyvíja softvérové produkty (27).

Riešenia núkané spoločnosťou sa zameriavajú na tri hlavné oblasti, a to oblasť financií, oblasť výsledkov a oblasť hodnôt. Všetky tieto oblasti si vyžadujú riadenie dát s cieľom zjednodušovať a chrániť ich tok a zabezpečovať ich maximálnu kvalitu, súkromie a bezpečnosť (29).

Vďaka finančnej oblasti získavajú lekári, resp. nemocnice kontrolu nad množstvom pacientov, zdrojov a financií, čo povedie k ich lepšej finančnej výkonnosti. Taktiež dochádza k poskytovaniu zhodnotenia možností liečby a starostlivosti, čo môže napomôcť

k zníženiu nákladov, a to vďaka zvoleniu vhodných liečebných postupov v správny čas (30).

Oblasť orientovaná na výsledky napomáha znížiť administratívnu záťaž lekárov, vďaka čomu sa lekári môžu sústrediť na poskytovanie lepšej zdravotnej starostlivosti. Za pomoci medzinárodných údajov majú lekári a nemocnice možnosť porovnania výkonností s lekármi, resp. nemocničnými zariadeniami z celého sveta. Týmto majú lekári, či nemocnice možnosť odhaliť svoje slabé stránky a zároveň príležitosti na vylepšenie nimi poskytovaných služieb (31).

Hodnotová oblasť sa sústredí na maximalizáciu hodnoty dát lekárov, resp. nemocníc, a to kombináciou informácií o výsledkoch a nákladoch v takmer reálnom čase a integráciou analýz a platieb do jednej platformy. Dochádza k umožneniu preskúmania incidencie, prevalencie, možných liečebných ciest i výsledkov a nákladov počas cyklov starostlivosti, čo umožňuje zvýšenie kvality poskytovanej zdravotnej starostlivosti (32).

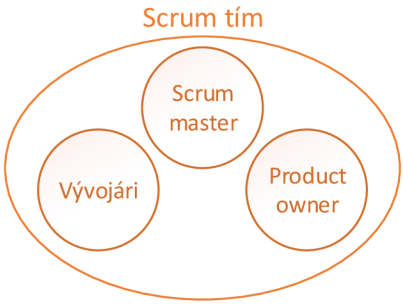
Spoločnosť udáva, že jej služby využíva aktuálne značná väčšina holandských nemocníc. Firma zároveň postupne expanduje do iných krajín. Dnes pôsobí okrem Holandska aj v krajinách, ako je napríklad Veľká Británia, Francúzsko, krajiny Škandinávie či Saudská Arábia (27).

2.2 Analýza testovacích spôsobov spoločnosti

V nasledujúcich odstavcoch priblížime prístup firmy k testovaniu.

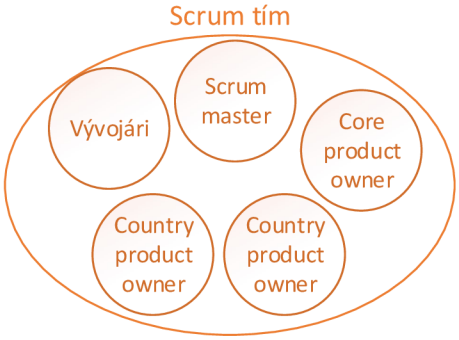
Testovanie firma vníma ako overenie, či sú produkty v dobrom stave a či fungujú tak, ako zákazníci očakávajú. Je to spôsob kontroly kvality (33). Testovanie sa snaží mať zjednotené, t.j. malo by dochádzať k uplatňovaniu tých istých princípov v rámci celej organizácie bez ohľadu na to, o aký produkt sa jedná. Týmito princípmi sú myslené napríklad snaha testovať čo najskôr, ako je to možné, alebo využívať pragmatický prístup (34).

Firma pracuje s iteračným modelom životného cyklu vývoja softvéru, a to konkrétne využitím metódy Scrum, čo znamená, že k vývoju a testovaniu produktov dochádza paralelne. Využívaná Scrum metodika signalizuje, že ku každému projektu je zostavený Scrum tím štandardne pozostávajúci z product ownera, Scrum mastera a vývojárov (34).



Obrázok 23: Scrum tím firmy - bežná skladba členov
(Zdroj: Vlastné spracovanie podľa 34)

V prípade produktov majúcich pre každú krajinu, v ktorej sú predávané, vlastnú inštanciu, a je týchto inštancií mnoho, resp. majú veľké množstvo vlastných špecifík, môže dôjsť k rozdeleniu role product ownera na tzv. core product ownera zodpovedajúceho za celý produkt a na tzv. country product ownerov zodpovedajúcich za príslušnú inštanciu produktu (34).



Obrázok 24: Scrum tím firmy – potenciálna skladba členov v prípade viacerých inštancií produktu
(Zdroj: Vlastné spracovanie podľa 34)

Vzhľadom na možnú existenciu viacerých inštancií jedného produktu firma berie obzvlášť na zreteľ vhodnosť dát, na ktorých bude daná inštancia testovaná (33).

Firma využíva kombináciu automatizovaného a manuálneho testovania, pričom jej snahou je manuálne testy, v prípade, že je to možné, resp. vhodné, zautomatizovať. Pri automatizácii testov sa primárne využíva programovací jazyk C# v Microsoft Visual Studio, a to prípadne spolu s využitím Selenium WebDriver. Selenium poskytuje výhody moderného objektovo orientovaného programovacieho jazyka pre vytváranie automatizovaných

sanity a regresných testov, ktoré je možné aplikovať na rôzne platformy a prostredia, vďaka čomu sa maximalizuje opätovné použitie týchto testov a návratnosť investícií pri ich minimálnej údržbe a podpore. V kontexte používaných technológií v súvislosti s testovaním dodáme, že na správu defektov sa používa Atlassian Jira. Na plánovanie a vykonávanie testov sa využíva TeamCity (33).

Medzi automatizované testy vo firme radíme tzv. sanity testy a regresné testy (33).

Sanity testy sú základné testy užívateľského rozhrania. Ich účelom je overiť, či najdôležitejšie funkcie fungujú tak, ako je očakávané (33). Z hľadiska testovacích úrovní ich môžeme zaradiť niekde medzi úroveň integračného a úroveň systémového testovania (34). Tieto testy prebiehajú každodenne, preto by mali byť navrhnuté tak, aby prebiehali rýchlo (33).

Regresné testy sú testy užívateľského rozhrania kontrolujúce, či nová funkcionalita nenarušila tú pôvodnú. Sú realizované aj v manuálnej podobe, avšak vzhľadom na to, že sa jedná o časté a časovo náročné testovanie, firma sa ho snaží v čo najväčšej miere automatizovať. Ich naplánované vykonávanie je raz za dva týždne, a to pred nasadením hlavných zmien kódu do produkcie, avšak vykonávajú sa aj podľa potreby, čiže častejšie ako je plánované (34).

Uvedené automatizované testy sú zamerané na užívateľské rozhranie — frontend stránku produktu. Síce frontend testami dochádza čiastočne aj k overeniu backend hľadiska, no nejedná sa o kontrolu do úplných detailov. Tieto testy nevedia zachytiť napríklad to, či nedošlo pri zmene databázovej schémy k zmene na úrovni dát. Takýto test pojednáva vyslovene backend stranu produktu, pričom táto kontrola aktuálne prebieha manuálne, a to vizuálne vývojármi na vybranej vzorke dát (8).

2.3 Zistenia z analýzy spôsobov testovania v spoločnosti

Z predchádzajúcej analýzy vyplýva, že je vhodné zautomatizovať vizuálne testy kontrolujúce, či nedochádza k zmenám na dátovej úrovni po zmene databázovej schémy. Vizuálnou kontrolou, aj to len na vybranej vzorke dát, dochádza k relatívne vysokej miere rizika neodhalenia nežiaducej zmeny dát. Zároveň dochádza k veľkým časovým stratám zamestnancov z dôvodu, že sa jedná o činnosť vykonávanú aj z dôvodu agilného spôsobu vývoja veľmi často. Na základe rozhovoru s firmou možno odhadnúť, že takáto činnosť

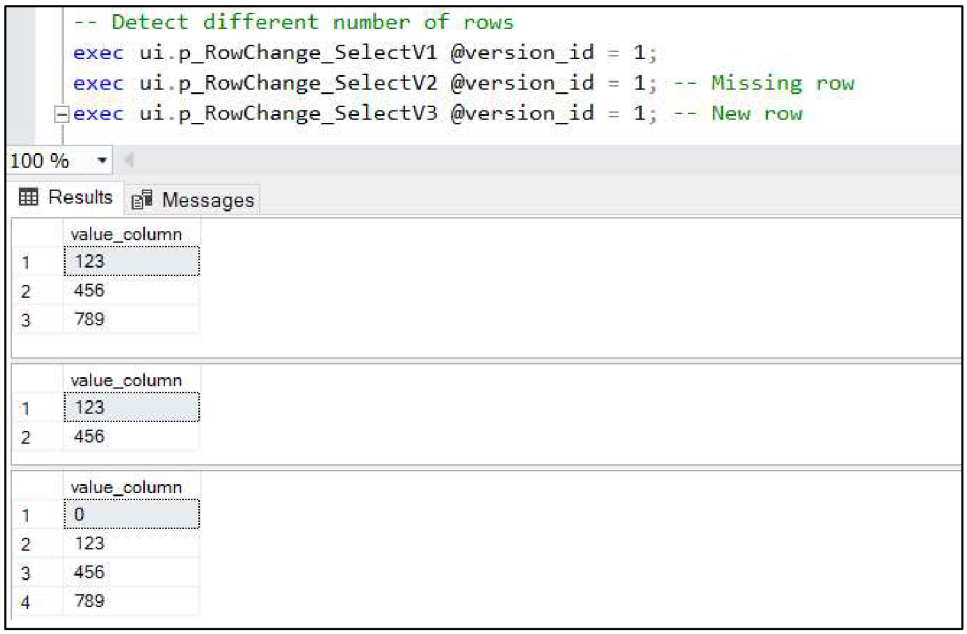
dokáže vývojárom na jednom projekte zabrať približne 160 hodín pracovného času ročne, čo odpovedá jednému mesiacu pracovníka zamestnaného na plný pracovný pomer (34).

Tieto skutočnosti podnecujú k návrhu automatizovaného testovacieho nástroja, ktorý by porovnával dáta získané po zmene databázovej schémy s nejakými predom určenými, t.j. referenčnými dátami. Užívateľ, teda vývojár by bol o výsledku tohto porovnania v prípade nájdenia zmeny nástrojom informovaný. Užívateľ by mal potom možnosť s týmito informáciami adekvátne naložiť.

2.4 Stanovenie požiadaviek na riešenie zisteného nedostatku

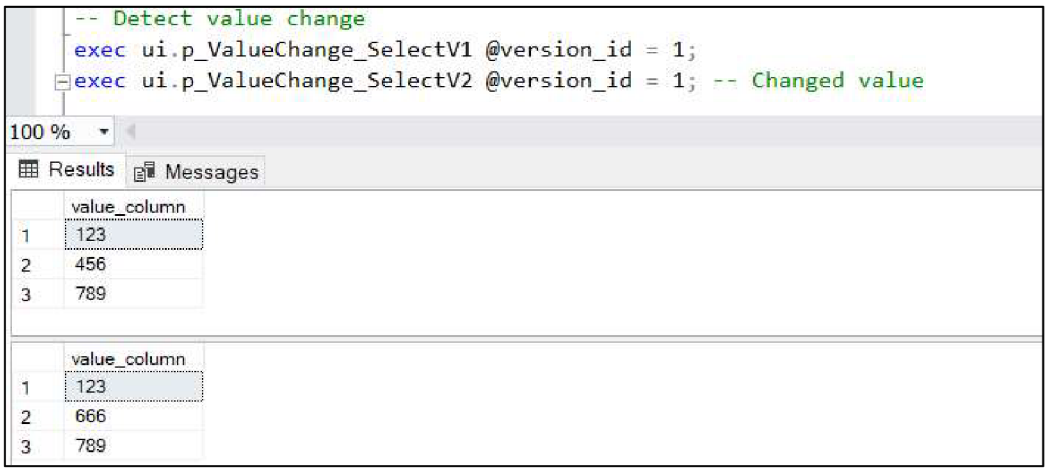
V nasledujúcej časti práce dôjde k zadefinovaniu požiadaviek na nástroj, t.j. pomenovaniu zmien, ktoré by mali byť nástrojom detegované a následne reportované, vychádzajúc z doterajších vizuálnych kontrol. Tieto zmeny budú zadefinované jednak verbálne a zároveň budú pre lepšiu názornosť demonštrované na ukážkach v prostredí SQL Server Management Studio, skrátene značenom ako SSMS.

Prvá zmena, ktorú by mal nástroj identifikovať, sa týka počtu riadkov, teda záznamov. V prípade, že má výstup získaný po databázových úpravách odlišný počet riadkov, ako má referenčný výstup, je potrebné túto skutočnosť zaznamenať. V rámci ukážky v SSMS vidíme, že ak je prvý získaný výstup referenčný, tak zvyšné dva výstupy by mali byť reportované ako zmenené.



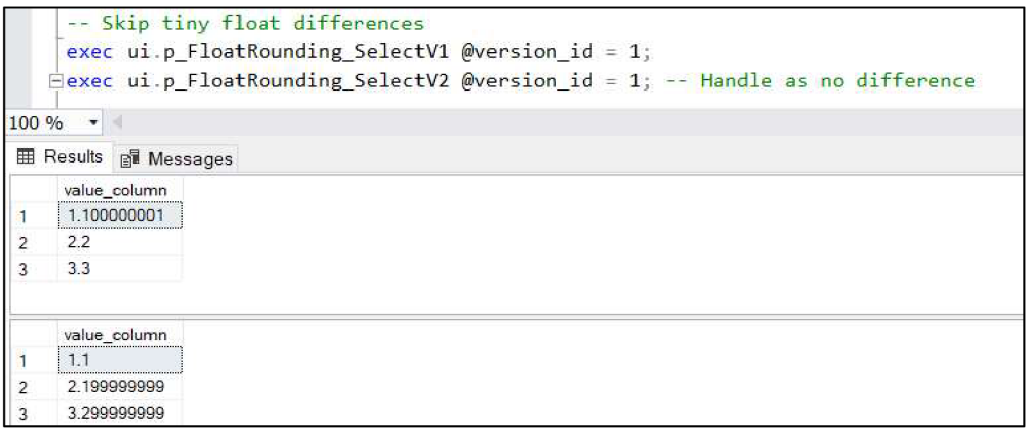
Obrázok 25: Požiadavky na nástroj - detekcia zmeny počtu riadkov
(Zdroj: Vlastné spracovanie podľa 8)

Nástroj by mal taktiež overovať, či nedošlo k zmene jednotlivých hodnôt v bunkách v rámci riadkov. V prípade, že k nejakej zmene došlo, je potrebné ju reportovať v znení, že jedna hodnota chýba, druhá pribudla. Na ukážke v SSMS prostredí vidíme, že pokiaľ prvý výsledok je referenčný a druhý získaný po zmene databázy, tak by malo dôjsť k reportu zmeny tvrdiacej, že pribudla hodnota *666* a zároveň ubudla hodnota *456*.



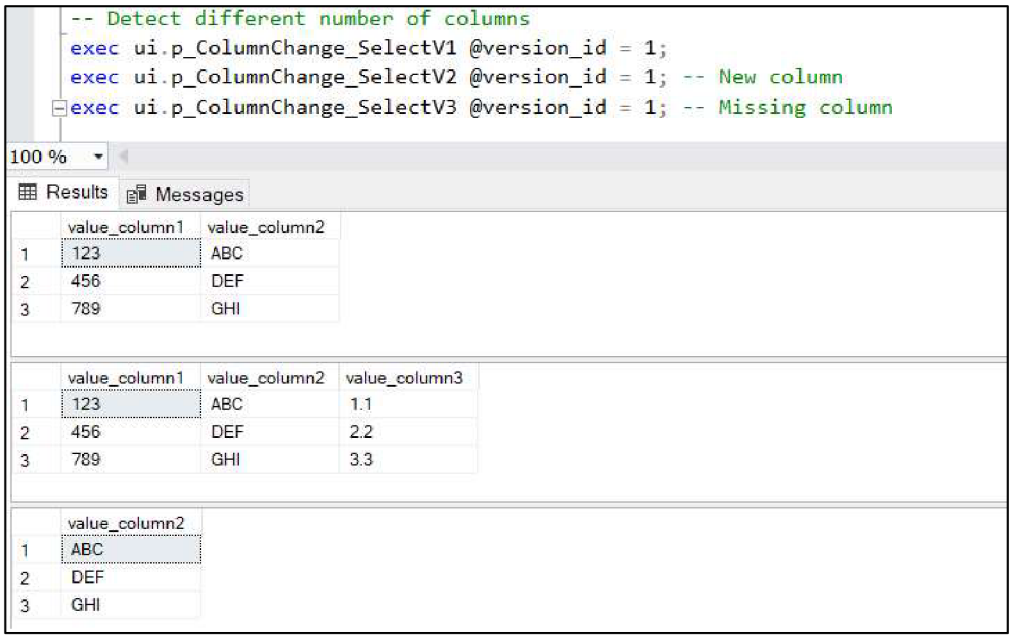
Obrázok 26: Požiadavky na nástroj - detekcia zmeny hodnôt v riadkoch
(Zdroj: Vlastné spracovanie podľa 8)

Čo sa týka hodnôt v bunkách, je potrebné špeciálne pristupovať v prípade, ak sú dátového typu *float*. Za zmenu v bunke typu *float* považujeme iba zmenu hodnoty do ôsmeho desatinného miesta vrátane. Čiže výstupy demonštrované v rámci ukážky budeme považovať za identické.



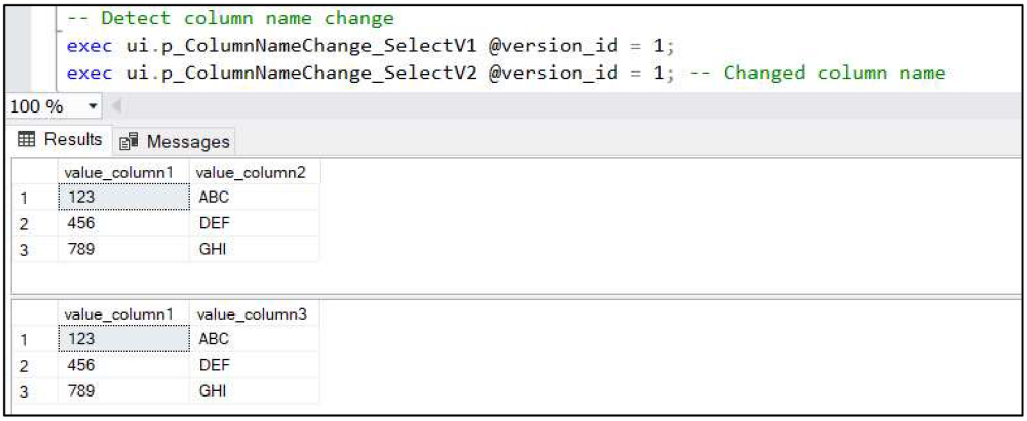
Obrázok 27: Požiadavky na nástroj - detekcia zmeny hodnôt dátového typu float
(Zdroj: Vlastné spracovanie podľa 8)

Z hľadiska počtu je potrebné skúmať nielen riadky, ale i stĺpce. V rámci nižšie zobrazenej ukážky vidíme, že ak je prvý výstup referenčný, tak pri získaní druhého výstupu zmenou databázy by malo byť v reporte uvedené, že došlo k pridaniu stĺpca *value_column3*. V situácii získania výstupu, ktorý máme znázornený v rámci ukážky ako tretí, by malo byť zaevidované, že zmizol stĺpec *value_column2*.



Obrázok 28: Požiadavky na nástroj - detekcia zmeny počtu stĺpcov
(Zdroj: Vlastné spracovanie podľa 8)

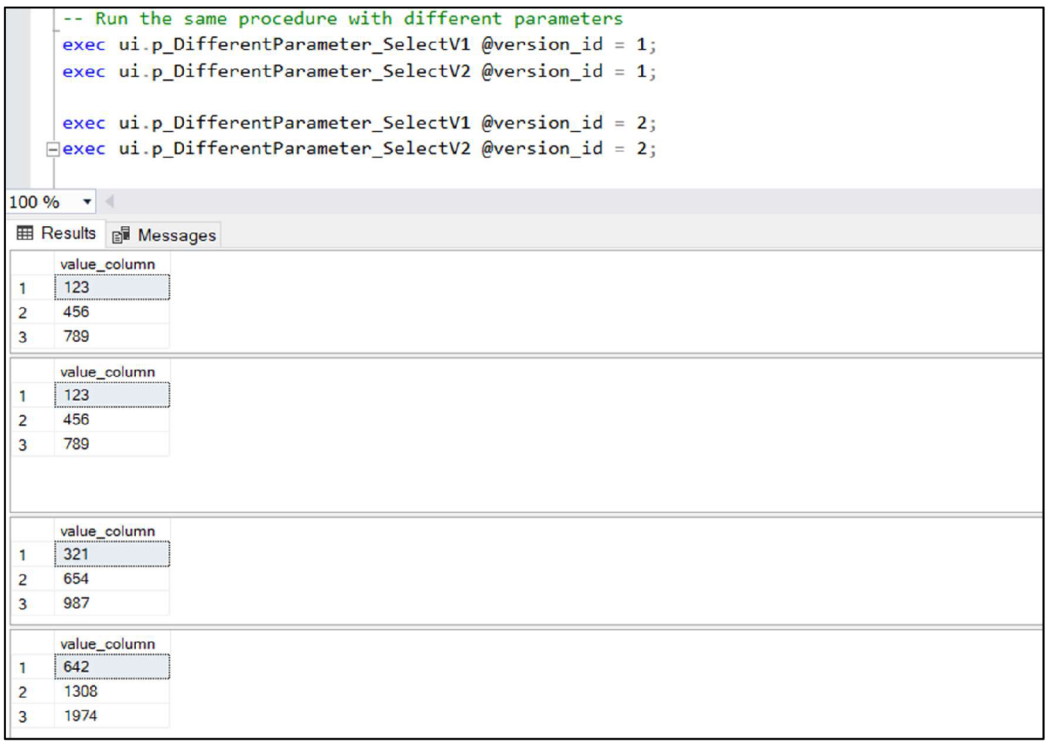
V súvislosti so stĺpcami je nevyhnutné evidovať nielen zmenu týkajúcu sa počtu stĺpcov, ale aj zmenu stĺpca v zmysle zmeny jeho označenia. V prípade, že k takejto zmene dôjde, nahlásenie zmeny prebehne rovnako ako v prípade riadkov — pribudol stĺpec *value_column3* a zmizol stĺpec *value_column2*.



Obrázok 29: Požiadavky na nástroj - detekcia zmeny označenia stĺpcov
(Zdroj: Vlastné spracovanie podľa 8)

V kontexte stĺpcov dodáme, že nástroj by mal taktiež vedieť detegovať zmenu ich dátového typu.

V rámci tohto reportu, kde budú všetky vyššie uvedené typy zmien zaznamenané, by mali byť taktiež evidované informácie ohľadne názvu procedúry a s ňou použitých parametrov. Použité parametre musia byť zaznamenané z dôvodu, že môže nastať situácia, keď pre jeden parameter bude získaný výsledok totožný s tým referenčným, zatiaľ čo výsledok získaný volaním tej istej procedúry len s inými hodnotami parametrov sa od výsledku referenčného bude líšiť. Takúto situáciu demonštruje aj nižšie zobrazená ukážka, v rámci ktorej možno vidieť, že pre procedúru *ui.p.DifferentParameter_SelectV1* je výstup v prípade jej volania s parametrom *@version_id = 1* odlišný ako v prípade jej volania s parametrom *@version_id = 2*. Rovnaký stav nastáva aj v prípade volania procedúry *ui.p.DifferentParameter_SelectV2*.



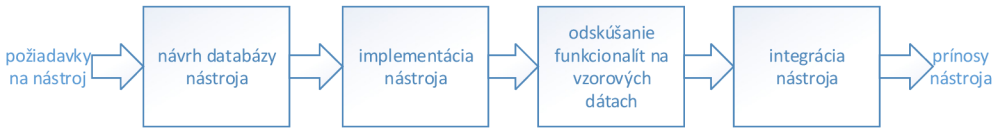
Obrázok 30: Požiadavky na nástroj - rozdielne výstupy pri rôznych hodnotách parametrov procedúry

(Zdroj: Vlastné spracovanie podľa 8)

V neposlednom rade netreba pri návrhu nástroja zabúdať na to, že bude používaný na veľké množstvá dát, a teda je nutné dbať na jeho optimalizáciu. Taktiež platí, že by mal mať ľahko zmeniteľné nastavenie databázy, ku ktorej bude pristupovať pre generovanie výstupov pre ich ďalšie spracovanie.

3 VLASTNÝ NÁVRH RIEŠENIA

V rámci tejto kapitoly záverečnej práce bude vysvetlené, ako sa pri tvorbe nástroja postupovalo, a to vrátane návrhu jeho databázy. Následne bude odprezentovaná funkčnosť nástroja na vzorových dátach. Na záver sa zmieni o tom, ako bol nástroj integrovaný a budú zhrnuté jeho najzásadnejšie prínosy pre firmu. Túto postupnosť možno vyjadriť aj nasledovným spôsobom:



Obrázok 31: Vizualizácia kapitoly Vlastný návrh riešenia
(Zdroj: Vlastné spracovanie)

3.1 Postup pri návrhu nástroja a jeho databázy

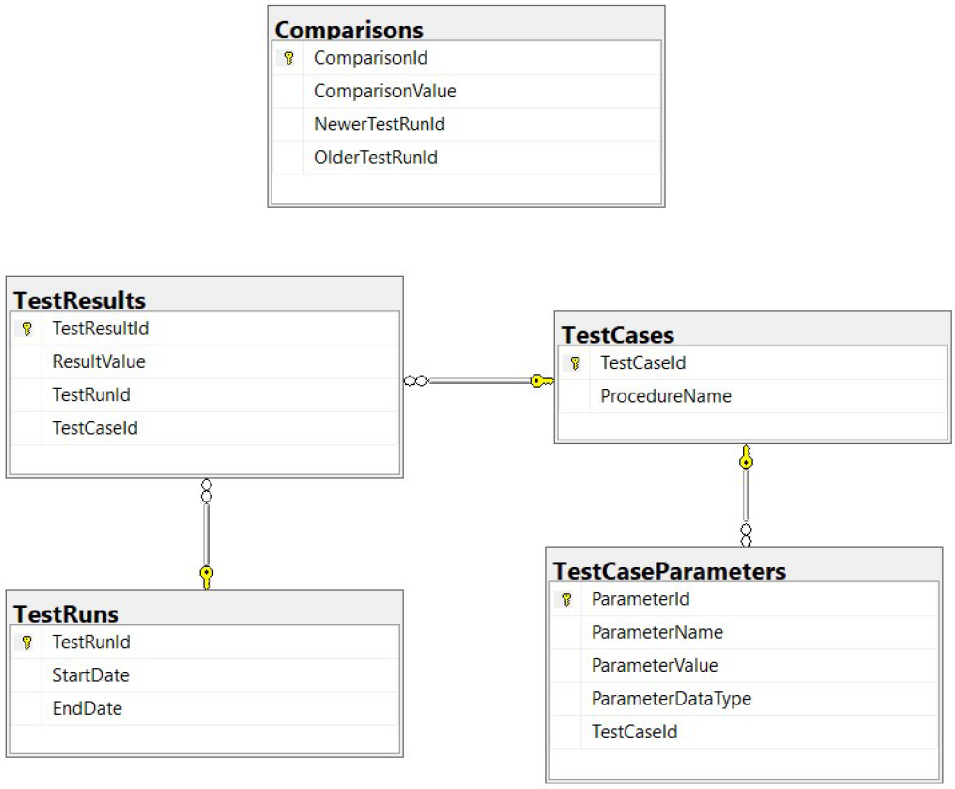
Pri navrhovaní nástroja si bolo v prvom rade potrebné uvedomiť, že nástroj bude pristupovať k dvom databázam. Prvá databáza je databáza poskytujúca dáta a uložené procedúry, z ktorých niektoré budú v rámci nástroja volané a vykonávané nad týmito dátami. V tejto práci bude značená ako vstupná databáza. Získané výsledky volaním týchto uložených procedúr, ktoré sú potrebné pre ďalšiu prácu, je nevyhnutné niekam uložiť. Na to sa využije druhú databázu. Táto databáza bude slúžiť nielen k ukladaniu týchto výsledkov, ale aj k záznamu ďalších informácií. V ďalších častiach tejto práce bude na ňu referované ako na databázu nástroja, alternatívne ako na výstupnú databázu.

Nižšie zobrazený entitno-relačný diagram znázorňuje navrhovanú schému databázy nástroja, čiže databázy určenej primárne na ukladanie výstupov získaných volaním jednotlivých procedúr, a to v tabuľke *TestResults*, a informácií o zmenách, ktoré nastali v týchto výstupoch procedúr vzhľadom na referenčný výstup, a to v tabuľke *Comparisons*. V tejto databáze sú zároveň umiestnené informácie o procedúrach (tabuľka *TestCases*) a parametroch a ich hodnotách (tabuľka *TestCaseParameters*), s akými majú byť volané.

Platí, že všetky procedúry, ktoré sú umiestnené v tabuľke *TestCases*, majú prebehnúť v rámci jedného tzv. TestRunu (tabuľka *TestRuns*). K výkonu TestRunu by malo po integrácii nástroja dôjsť automaticky pri úprave databázovej schémy vstupnej databázy.

Jeden TestRun je teda tvorený výsledkami viacerých procedúr. Tieto výsledky sú však zaznamenané jednotlivo, a to v tabuľke *TestResults*. Preto je potrebné, aby tabuľka *TestResults* odkazovala aj na tabuľku *TestRuns* a aj na tabuľku *TestCases*.

Každý záznam každej tabuľky musí byť jednoznačne identifikovateľný, a to primárnym kľúčom. Vďaka primárnym kľúčom (a cudzím) je možné vytvoriť potrebné väzby medzi tabuľkami.



Obrázok 32: Návrh databázy nástroja
(Zdroj: Vlastné spracovanie)

Celý nástroj bude navrhovaný, resp. implementovaný využitím softvéru Microsoft Visual Studio Community 2019, a to vrátane vytvorenia vyššie diskutovanej databázy nástroja. Tá bude vytvorená za pomoci využitia technológie Entity Framework umožňujúcej objektovo-relačné mapovanie. Možnosť využitia tejto technológie spočíva v inštalácii NuGet balíka v rámci Microsoft Visual Studio značeného ako EntityFramework 6.

Obrázok 33: Microsoft Visual Studio - NuGet balík - Entity Framework
(Zdroj: Vlastné spracovanie)

Po inštalácii tohto NuGet balíka je možné začať s tvorbou databázy, počnúc vytvorením databázového kontextu, čiže triedy značenej ako *EFDBContext*, ktorá je dedičom triedy *DbContext*. Táto trieda predstavuje databázové spojenie, vďaka ktorému vzniká možnosť interagovania s touto databázou.

```
using System.Data.Entity;

namespace AutomatedIntegrationTestingTool
{
    public class EFDBContext : DbContext
    {
        public DbSet<Entities.TestCase> TestCases { get; set; }
        public DbSet<Entities.TestRun> TestRuns { get; set; }
        public DbSet<Entities.Comparison> Comparisons { get; set; }
        public DbSet<Entities.TestResult> TestResults { get; set; }
        public DbSet<Entities.TestCaseParameter> TestCaseParameters { get; set; }

        public EFDBContext() : base("ConnectionToOutputDB")
        {
        }
    }
}
```

Obrázok 34: Microsoft Visual Studio - vytvorenie databázového kontextu – trieda EFDBContext
(Zdroj: Vlastné spracovanie)

To, s ktorou databázou táto trieda vytvára spojenie, definuje tzv. *ConnectionToOutputDB*. Ide o informáciu o serveri a názve databázy, ku ktorej sa aplikácia pripája, prípadne ktorú vytvára v situácii, že v čase interakcie takáto databáza ešte neexistuje. Táto informácia je uložená v konfiguračnom súbore *App.config*, konkrétne v jeho sekcii *ConnectionStrings*. Vloženie informácií týkajúcich sa databázy, ku ktorej sa pripájame, do konfiguračného súboru aplikácie je považované za vhodnejšie riešenie v porovnaní s ich vložením priamo do kódu aplikácie. Vkladanie tohto typu informácií priamo do zdrojového kódu aplikácie môže okrem iných komplikácií spôsobiť aj zbytočnú nutnosť opakovanej kompilácie programu pri zmene názvu databázy, resp. servera, na ktoré je potrebné sa pripojiť. V rámci demonštrovanej ukážky dochádza k pripojeniu na *localhost* k databáze pomenovanej ako *databaza_nastroja*.

```
<connectionStrings>
  <add
    name="ConnectionToOutputDB"
    providerName="System.Data.SqlClient"
    connectionString=
        "Data Source = localhost;
        Initial Catalog = databaza_nastroja;
        Integrated Security = True;
        MultipleActiveResultSets = True"
  />
</connectionStrings>
```

Obrázok 35: Microsoft Visual Studio - definovanie pripojenia k databáze nástroja v konfiguračnom súbore

(Zdroj: Vlastné spracovanie)

Vďaka tomuto kontextu, t.j. triede *EFDBContext* je teda jednak možné vytvoriť samotnú databázu, a jednak do nej premietnuť prípadné úpravy, resp. čerpať z nej dáta. Každý *DbSet,* viditeľný v triede *EFDBContext*, reprezentuje tabuľku, ktorú je možné vytvoriť, pristupovať k nej, modifikovať jej dáta, ale ju aj zmazať. Atribúty a vzťahy týchto tabuliek s ostatnými tabuľkami sú definované v rámci samostatných tried. Napríklad tabuľka *TestCaseParameters* je definovaná triedou *TestCaseParameter*, ktorá obsahuje údaje o parametroch v podobe vlastností tejto triedy. Taktiež dochádza k vytvoreniu väzby s tabuľkou *TestCases* prezentovanou triedou *TestCase*. Trieda *TestCasteParameter* vyzerá potom nasledovne:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AutomatedIntegrationTestingTool.Entities
{
    [Table("dbo.TestCaseParameters")]
    public class TestCaseParameter
    {
        [Key]
        public int ParameterId { get; set; }
        public string ParameterName { get; set; }
        public string ParameterValue { get; set; }
        public string ParameterDataType { get; set; }
        public int TestCaseId { get; set; }

        public virtual TestCase TestCases { get; set; }
    }
}
```

Obrázok 36: Microsoft Visual Studio - definovanie triedy TestCaseParameter reprezentujúcej tabuľku v databáze

(Zdroj: Vlastné spracovanie)

Zatiaľ čo trieda *TestCase*, v ktorej sa taktiež premietne väzba na tabuľku *TestCasesParameters*, t.j. na triedu *TestCaseParameter*, je definovaná takto:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace AutomatedIntegrationTestingTool.Entities
{
    [Table("dbo.TestCases")]
    public class TestCase
    {
        [Key]
        public int TestCaseId { get; set; }
        public string ProcedureName { get; set; }

        [ForeignKey("TestCaseId")]
        public virtual HashSet<TestResult> TestResults { get; set; }

        [ForeignKey("TestCaseId")]
        public virtual HashSet<TestCaseParameter> TestCaseParameters { get; set; }
    }
}
```

Obrázok 37: Microsoft Visual Studio - definovanie triedy TestCase reprezentujúcu tabuľku v databáze

(Zdroj: Vlastné spracovanie)

Obdobné triedy boli vytvorené aj pre ostatné tabuľky vyzobrazené v entitno-relačnom diagrame navrhujúceho databázovú schému databázy nástroja.

V rámci zhrnutia dosiaľ spomenutých skutočností týkajúcich sa tvorby nástroja je možné konštatovať, že došlo k navrhnutiu a následne využitím tried a konfiguračného súboru zabezpečeniu prvotného vytvorenia databázy nástroja, ku ktorej sa nástroj bude neskôr opakovane pripájať.

V tejto databáze boli naplnené vopred určenými informáciami tabuľky *TestCases* a *TestCaseParameters*, ktoré, ako už bolo uvedené, obsahujú informácie o názvoch procedúr a ich parametroch, ktoré majú byť vykonané nad vstupnou, t.j. vopred danou databázou obsahujúcou dáta a tieto procedúry.

V tomto štádiu je teda možné vykonať tzv. TestRun, t.j. inštanciu triedy *TestRun* odpovedajúcej tabuľke *TestRuns*. TestRun, t.j. vykonanie všetkých procedúr zaznamenaných v tabuľke *TestCases* spolu s príslušnými parametrami z *TestCaseParameters* generuje záznam nielen do tabuľky *TestRuns* o čase začiatku a konca jeho realizácie, ale aj jednotlivé záznamy (čiže inštancie triedy *TestResult*) odpovedajúce výstupom jednotlivých procedúr do tabuľky *TestResults*.

Informácie o pripojení na spomínanú vopred danú databázu, nad ktorou má TestRun prebehnúť, sú aktuálne nástroju dodávané do metódy *Main* v rámci argumentov príkazového

riadku. Takýto spôsob poskytovania názvu serveru a databázy je možné považovať v porovnaní so zadaním týchto údajov v rámci kódu programu za výhodnejší, a to z dôvodu ohľadu na možnosť potreby zmeny vstupnej databázy.

Samotná funkcionalita týkajúca sa vykonania TestRunu je uložená v samostatnej triede nazvanej ako *TestCasesProvider*. Táto trieda obsahuje metódu *MakeTestRun*, ktorá jej volaním v rámci metódy *Main* (po vytvorení inštancie triedy *TestCasesProvider*) vykoná samotný TestRun.

V rámci metódy *MakeTestRun* dôjde v prvom rade k napojeniu sa na databázový kontext vytvorením inštancie triedy *EFDBContext*. Následne metóda vytvorí inštanciu triedy *TestRun*, o ktorej je následne schopná uložiť informácie týkajúce sa napríklad začiatku jej konania. Dôjde k pripojeniu k vstupnej databáze a k privolaniu ďalšej metódy, a to *ProvideAllTestCases*.

Táto metóda, t.j. metóda *ProvideAllTestCases* využitím ďalších vytvorených metód zabezpečí vykonanie všetkých procedúr s príslušnými parametrami nad vstupnou databázou a uloženiu ich výstupov do tabuľky *TestResults*. Je vhodné spomenúť, že dochádza k využitiu nainštalovaného NuGet balíka Newtonsoft.Json, vďaka ktorému je možné ukladať hodnoty *ResultValue,* čo je jeden zo stĺpcov tabuľky *TestResults*, v JSON formáte.

Obrázok 38: Microsoft Visual Studio - NuGet balík - Newtonsoft.Json
(Zdroj: Vlastné spracovanie)

Zároveň možno podotknúť, že dochádza k využitiu príkazov LINQ. Po dokončení metódy *ProvideAllTestCases* sa uzavrie spojenie so vstupnou databázou, zaeviduje sa čas o konci TestRunu, ktorý sa následne pridá do tabuľky *TestRuns*. Všetky zmeny sa do databázy uložia, čím je tento proces, t.j. proces vykonania TestRunu ukončený.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Data;
using System.Data.SqlClient;
using Newtonsoft.Json;

namespace AutomatedIntegrationTestingTool
{
```

```
    public class TestCasesProvider
    {
        public void MakeTestRun(string serverName, string databaseName)
        {
            using (var context = new EFDBContext())
            {
                var testRun = context.TestRuns.Create();

                testRun.StartDate = DateTime.Now;

                using (var connInputDB = new SqlConnection("Data Source =" + serv-
erName + "; Initial Catalog =" + databaseName + "; Integrated Security = True"))
                {
                  connInputDB.Open();
                  ProvideAllTestCases(context, connInputDB, testRun, databaseName);
                  connInputDB.Close();
                }

                testRun.EndDate = DateTime.Now;
                context.TestRuns.Add(testRun);
                context.SaveChanges();
            }
        }

        ...
    }
}
```

Obrázok 39: Microsoft Visual Studio - trieda TestCasesProvider a jej metóda MakeTestRun
(Zdroj: Vlastné spracovanie)

Je vhodné samostatne detailnejšie uviesť navrhnutú štruktúru *ResultValue*, ktorý je vo formáte JSON. Nižšie je znázornená štruktúra obecného výstupu, ktorý má dva stĺpce a dva riadky. Vidíme, že tento JSON formát obsahuje dve polia, a to pole Columns, teda pole riešiace stĺpce, a pole Rows, teda pole riešiace riadky. Pole Columns zachytáva informácie o názvoch a dátových typoch jednotlivých stĺpcov, ktoré boli súčasťou výstupu. Každý stĺpec je samostatným objektom tohto poľa. Pole Rows zachytáva objekty značené ako RowValues. Objekt RowValues prezentuje jeden riadok. Ten je tvorený poľom pozostávajúcim z objektov jednotlivých hodnôt v rámci tohto riadku, pričom jeden objekt nesie informáciu o názve stĺpca a hodnote v tomto stĺpci v rámci tohto objektu, teda riadku.

```
{
   "Columns":[
      {
         "ColumnName":"SomeColumnName1",
         "DataType":"SomeDataType1"
      },
      {
         "ColumnName":"SomeColumnName2",
         "DataType":"SomeDataType2"
      }
```

```
   ],
   "Rows":[
      {
         "RowValues":[
            {
               "ColumnName":"SomeColumnName1",
               "Value":"SomeValue1"
            },
            {
               "ColumnName":"SomeColumnName2",
               "Value":"SomeValue2"
            }
         ]
      },
      {
         "RowValues":[
            {
               "ColumnName":"SomeColumnName1",
               "Value":"SomeValue4"
            },
            {
               "ColumnName":"SomeColumnName2",
               "Value":"SomeValue5"
            }
         ]
      }
   ]
}
```

Obrázok 40: Obecná štruktúra ResultValue v JSON formáte

(Zdroj: Vlastné spracovanie)

Týmto bola vysvetlená jedna z dvoch hlavných funkcionalít nástroja. Druhou, tou kľúčovou, je porovnávanie dvoch TestRunov, teda presnejšie ich TestResultov medzi sebou, pričom jeden TestRun bude predom určený ako referenčný.

Nástroj bude zachytávať zmeny, ktoré boli uvedené v druhej kapitole tejto záverečnej práce, a to v rámci požiadaviek na tento nástroj. Informácie o situácii týkajúcej sa zmien budú zapísané do tabuľky *Comparisons*.

K samotnému porovnaniu dôjde v návrhu volaním metódy *CompareTestRuns* inštancie triedy *TestRunComparer* vytvorenej v rámci metódy *Main*. Metóda *CompareTestRuns* pracuje taktiež s vytvorením databázového kontextu využitím triedy *EFDBContext*. V prvom rade dôjde k uloženiu TestResultov prislúchajúcich k porovnávaným TestRunom do samostatných premenných následne spracovaných metódou *CompareTestResultsIfPossible*, ktorá si v rámci seba privoláva ďalšie metódy. Opäť dochádza k využitiu NuGet balíka Newtonsoft.Json a príkazov LINQ. Získaný výsledok porovnávania sa uloží do databázy pomocou metódy *SaveTestRunsComparisonsToDboComparisons*.

```
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Linq;

namespace AutomatedIntegrationTestingTool
{
    public class TestRunComparer
    {
        public void CompareTestRuns(int olderTestRunId, int newerTestRunId)
        {
            using (var context = new EFDBContext())
            {
                var olderTestRunTestResults = con-
text.TestRuns.Find(olderTestRunId).TestResults.ToList();
                var newerTestRunTestResults = context.TestRuns.Find(new-
erTestRunId).TestResults.ToList();

                var comparisonValueList = CompareTestResultsIfPossible(con-
text, olderTestRunTestResults, newerTestRunTestResults);

                SaveTestRunsComparisonToDboComparisons(new-
erTestRunId, olderTestRunId, comparisonValueList, context);
            }
        }

        ...
    }
}
```

Obrázok 41: Microsoft Visual Studio - trieda TestRunComparer a jej metóda CompareTestRuns
(Zdroj: Vlastné spracovanie)

Štruktúru hodnoty *ComparisonValue* v JSON formáte si pre lepšiu orientáciu možno znázorniť na obecnom príklade. Príklad znázorňuje situáciu, v rámci ktorej došlo k porovnaniu dvoch TestRunov obsahujúcich po dva TestResulty, pričom výsledky týchto dvoch TestRunov sa zhodujú.

Výsledkom záznamu *ComparisonValue* je teda pole objektov týkajúcich konkrétnych procedúr s konkrétnymi parametrami. Tieto objekty nesú informáciu aj o tom, že či bola táto procedúra (*TestCaseId*, *ProcedureName*) s týmito parametrami (*TestCaseParameters*) vykonaná v rámci oboch TestRunov, a v prípade, že nie, špecifikuje, či k ich spusteniu došlo v rámci referenčného alebo nového TestRunu. Túto informáciu možno vidieť evidovanú v rámci *ProvidedBy*. V neposlednom rade tento JSON záznam obsahuje informácie o *ColumnChanges*, teda zmenách na úrovni stĺpcov, a *RowChanges*, teda zmenách na úrovni riadkov, ktoré by boli v prípade, že by došlo k nejakým zmenám obsiahnuté v rámci poľa.

```
[
   {
      "TestCaseId":1,
      "ProcedureName":"SomeProcedureName1",
      "ProvidedBy":"both",
      "TestCaseParameters":[
         {
            "ParameterName":"SomeParameterName1",
            "ParameterValue":"SomeParameterValue1"
         }
      ],
      "ColumnChanges":[],
      "RowChanges":[]
   },
   {
      "TestCaseId":2,
      "ProcedureName":"SomeProcedureName2",
      "ProvidedBy":"both",
      "TestCaseParameters":[
         {
            "ParameterName":"SomeParameterName2",
            "ParameterValue":"SomeParameterValue2"
         }
      ],
      "ColumnChanges":[],
      "RowChanges":[]
   }
]
```


Obrázok 42: Obecná štruktúra ComparisonValue v JSON formáte
(Zdroj: Vlastné spracovanie)

3.2 Demonštrácia funkcionalít nástroja na vzorových dátach

V rámci tejto podkapitoly bude ukázaná funkčnosť nástroja pre lepšiu názornosť na vzorových dátach.

V databáze nástroja v tabuľke *TestCases* sú uložené nasledovné procedúry, ktoré budú nástrojom volané.

	TestCaseId	ProcedureName
1	1	ui.p_GetResult_01
2	2	ui.p_GetResult_02
3	3	ui.p_GetResult_03
4	4	ui.p_GetResult_04
5	5	ui.p_GetResult_05

Obrázok 43: SSMS - databáza nástroja - naplnená tabuľka TestCases
(Zdroj: Vlastné spracovanie)

Príslušné parametre a ich hodnoty k daným procedúram sú umiestnené v tabuľke *TestCaseParameters*.

	ParameterId	ParameterName	ParameterValue	ParameterDataType	TestCaseId
1	1	version_id	1	int	1
2	2	version_id	1	int	2
3	3	version_id	1	int	3
4	4	version_id	1	int	4
5	5	version_id	1	int	5

Obrázok 44: SSMS - databáza nástroja - naplnená tabuľka TestCaseParameters
(Zdroj: Vlastné spracovanie)

Pri volaní týchto procedúr s parametrami ním prislúchajúcimi v rámci SSMS, budú získané nasledovné výsledky.

```
use vstupna_databaza;
exec ui.p_GetResult_01 @version_id = 1;
exec ui.p_GetResult_02 @version_id = 1;
exec ui.p_GetResult_03 @version_id = 1;
exec ui.p_GetResult_04 @version_id = 1;
exec ui.p_GetResult_05 @version_id = 1;
```

100 %

Results | Messages

	value_column
1	123
2	456
3	789

	value_column
1	123
2	456
3	789

	value_column
1	123
2	456
3	789

	value_column1	value_column2
1	123	ABC
2	456	DEF
3	789	GHI

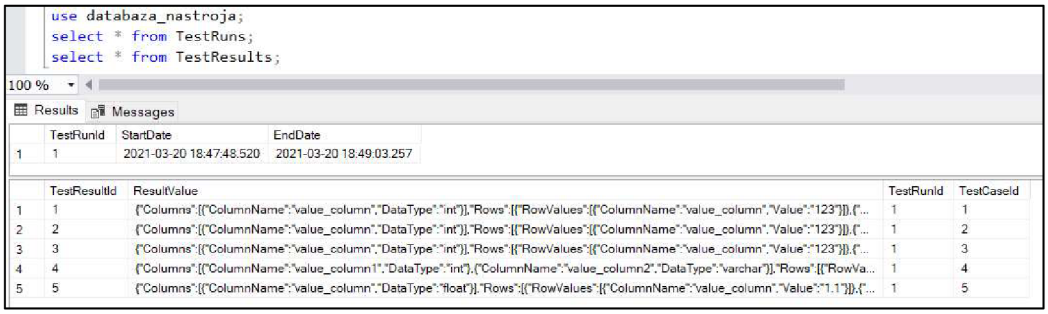
	value_column
1	1.1
2	2.2
3	3.3

Obrázok 45: SSMS - vstupná databáza - volanie procedúr s ich parametrami
(Zdroj: Vlastné spracovanie)

Po vykonaní jedného TestRunu dôjde k tvorbe záznamu do tabuľky *TestRuns* obsahujúceho identifikátor záznamu (atribút *TestRunId*), čas začatia (atribút *StartDate*) a konca (atribút *EndDate*) jeho vykonávania.

Súbežne sa do tabuľky *TestResults* uložia jednotlivé záznamy obsahujúce výstupy jednotlivých vykonaných procedúr. Každý záznam obsahuje vlastný identifikátor (atribút *TestResultId*), referenciu na procedúru (atribút *TestCaseId*), ktorej výstup reprezentuje, referenciu na TestRun (atribút *TestRunId*), v rámci ktorého bol vykonaný, a v neposlednom rade samotný výstup procedúry (atribút *ResultValue*), a to v JSON formáte. Obsah tohto výstupu bolo možné vidieť aj na figúre s názvom „SSMS - vstupná databáza - volanie procedúr s ich parametrami".

Obsah tabuliek vyzerá potom nasledovne:



```
use databaza_nastroja;
select * from TestRuns;
select * from TestResults;
```

100 %

Results | Messages

	TestRunId	StartDate	EndDate
1	1	2021-03-20 18:47:48.520	2021-03-20 18:49:03.257

	TestResultId	ResultValue	TestRunId	TestCaseId
1	1	{"Columns":[{"ColumnName":"value_column","DataType":"int"}],"Rows":[{"RowValues":[{"ColumnName":"value_column","Value":"123"}]},{"...	1	1
2	2	{"Columns":[{"ColumnName":"value_column","DataType":"int"}],"Rows":[{"RowValues":[{"ColumnName":"value_column","Value":"123"}]},{"...	1	2
3	3	{"Columns":[{"ColumnName":"value_column","DataType":"int"}],"Rows":[{"RowValues":[{"ColumnName":"value_column","Value":"123"}]},{"...	1	3
4	4	{"Columns":[{"ColumnName":"value_column1","DataType":"int"},{"ColumnName":"value_column2","DataType":"varchar"}],"Rows":[{"RowVa...	1	4
5	5	{"Columns":[{"ColumnName":"value_column","DataType":"float"}],"Rows":[{"RowValues":[{"ColumnName":"value_column","Value":"1.1"}]},{"...	1	5

Obrázok 46: SSMS - databáza nástroja - tabuľky TestRuns a TestResults po prvom TestRun
(Zdroj: Vlastné spracovanie)

Možno si priblížiť, ako vyzerá hodnota *ResultValue* napríklad v prípade, kde *TestResultId* je rovné *1*, čo v demonštrovanom prípade odpovedá procedúre *ui.p_GetResult_01* s parametrom a hodnotou v tvare *@version_id = 1*.


```
{
   "Columns":[
      {
         "ColumnName":"value_column",
         "DataType":"int"
      }
   ],
   "Rows":[
      {
         "RowValues":[
            {
               "ColumnName":"value_column",
               "Value":"123"
            }
         ]
      },
      {
         "RowValues":[
            {
               "ColumnName":"value_column",
               "Value":"456"
            }
         ]
```

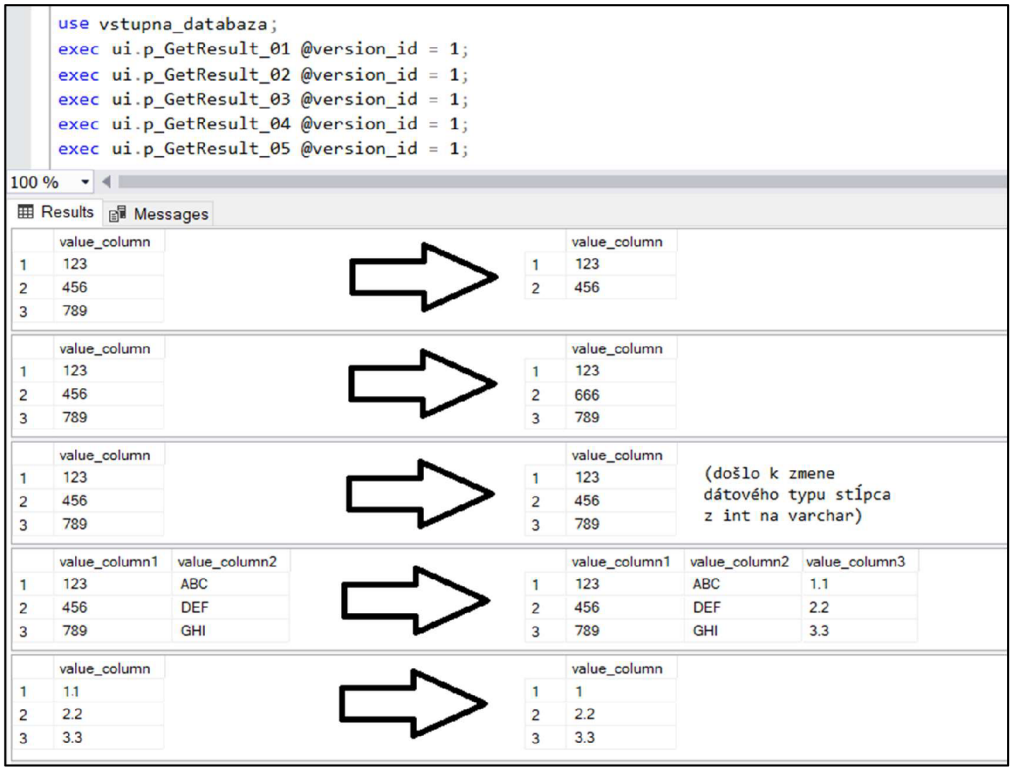
```
      },
      {
         "RowValues":[
            {
               "ColumnName":"value_column",
               "Value":"789"
            }
         ]
      }
   ]
}
```


Obrázok 47: Hodnota ResultValue pri TestResultId rovnému 1

(Zdroj: Vlastné spracovanie)

Je zrejmé, že tento výstup skutočne odpovedá výstupu procedúry *ui.p_GetResult_01*, ktorý bol ukázaný na obrázku s názvom „SSMS - vstupná databáza - volanie procedúr s ich parametrami“.

Pre ďalšie ukážky je vhodné pozmeniť vstupné dáta, ktoré spracovávajú dané procedúry. Pre lepšiu viditeľnosť ich modifikácie sú znázornené v rámci nasledovného obrázku aj pôvodné, aj nové výstupy.

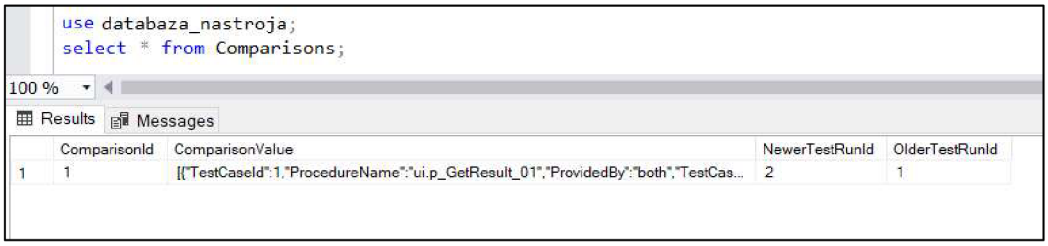


Obrázok 48: SSMS - vstupná databáza - volanie procedúr s ich parametrami pôvodne a po zmene vstupných dát

(Zdroj: Vlastné spracovanie)

Po spustení nástroja dôjde k výkonu tohto TestRun s vygenerovaním príslušných TestResults a ich následným uložením do príslušných tabuliek databázy nástroja.

Teraz, keď je v rámci tejto demonštrácie k dispozícii viac ako jeden TestRun, je možné zameranie na to, aký by bol výstup nástroja v prípade ich porovnania, presnejšie povedané porovnania ich príslušných záznamov v tabuľke *TestResults*. Samozrejme pri tom dochádza k zohľadneniu procedúry, ktorá jednotlivé TestResults vykonala. Informácie získané z tohto porovnania sa uložia do tabuľky *Comparisons*. Atribútmi vzniknutého záznamu v tabuľke sú jeho identifikátor (atribút *ComparisonId*), informácia o tom, ktorý TestRun bol s ktorým porovnávaný (atribút *OlderTestRunId*, atribút *NewerTestRunId*) a výsledok samotného porovnania v JSON formáte (atribút *ComparisonValue*).



Obrázok 49: SSMS - databáza nástroja - tabuľka Comparisons
(Zdroj: Vlastné spracovanie)

V prípade detailnejšieho nahliadnutia na hodnotu *ComparisonValue* tohto vytvoreného záznamu, je viditeľné, že zmeny, ku ktorým došlo boli zachytené korektne.


```
[
   {
      "TestCaseId":1,
      "ProcedureName":"ui.p_GetResult_01",
      "ProvidedBy":"both",
      "TestCaseParameters":[{"ParameterName":"version_id","ParameterValue":"1"}],
      "ColumnChanges":[],
      "RowChanges":[
         {
            "ChangeType":"removed",
            "RowValues":[
               {"ColumnName":"value_column","Value":"789"}
            ]
         }
      ]
   },
   {
      "TestCaseId":2,
      "ProcedureName":"ui.p_GetResult_02",
      "ProvidedBy":"both",
      "TestCaseParameters":[{"ParameterName":"version_id","ParameterValue":"1"}],
      "ColumnChanges":[],
      "RowChanges":[
```

```
        {
           "ChangeType":"removed",
           "RowValues":[
              {"ColumnName":"value_column","Value":"456"}
           ]
        },
        {
           "ChangeType":"added",
           "RowValues":[
              {"ColumnName":"value_column","Value":"666"}
           ]
        }
     ]
  },
  {
     "TestCaseId":3,
     "ProcedureName":"ui.p_GetResult_03",
     "ProvidedBy":"both",
     "TestCaseParameters":[{"ParameterName":"version_id","ParameterValue":"1"}],
     "ColumnChanges":[
        {"ChangeType":"added",ColumnName":"value_column","DataType":"varchar"},
        {"ChangeType":"removed","ColumnName":"value_column","DataType":"int"}
     ],
     "RowChanges":[]
  },
  {
     "TestCaseId":4,
     "ProcedureName":"ui.p_GetResult_04",
     "ProvidedBy":"both",
     "TestCaseParameters":[{"ParameterName":"version_id","ParameterValue":"1"}],
     "ColumnChanges":[
        {"ChangeType":"added","ColumnName":"value_column3","DataType":"float"}
     ],
     "RowChanges":[]
  },
  {
     "TestCaseId":5,
     "ProcedureName":"ui.p_GetResult_05",
     "ProvidedBy":"both",
     "TestCaseParameters":[{"ParameterName":"version_id",ParameterValue":"1"}],
     "ColumnChanges":[],
     "RowChanges":[
        {
           "ChangeType":"removed",
           "RowValues":[
              {"ColumnName":"value_column",Value":"1.1"}
           ]
        },
        {
           "ChangeType":"removed",
           "RowValues":[
              {"ColumnName":"value_column","Value":"2.2"}
           ]
        },
        {
           "ChangeType":"removed",
           "RowValues":[
              {"ColumnName":"value_column",Value":"3.3"}
           ]
        },
        {
           "ChangeType":"added",
           "RowValues":[
```

```
            {"ColumnName":"value_column","Value":"1"}
         ]
      },
      {
         "ChangeType":"added",
         "RowValues":[
            {"ColumnName":"value_column","Value":"2.1"}
         ]
      },
      {
         "ChangeType":"added",
         "RowValues":[
            {"ColumnName":"value_column",Value":"3.2"}
         ]
      }
   ]
 }
]
```


Obrázok 50: Hodnota ComparisonValue pri ComparisonId rovnému 1
(Zdroj: Vlastné spracovanie)

Tento výsledok obsahuje informácie, s ktorými bude užívateľ nástroja interagovať.

3.3 Integrácia nástroja

Návrh tejto záverečnej práce pokryl backend stranu nástroja. K nástroju bola následne dotvorená frontend strana — užívateľské rozhranie. Pre upozornenie užívateľov o prípadnom zistení zmeny v dátach po vykonaní testovania týmto nástrojom, resp. iných súvisiacich informáciách bola využitá aplikácia Slack (35).

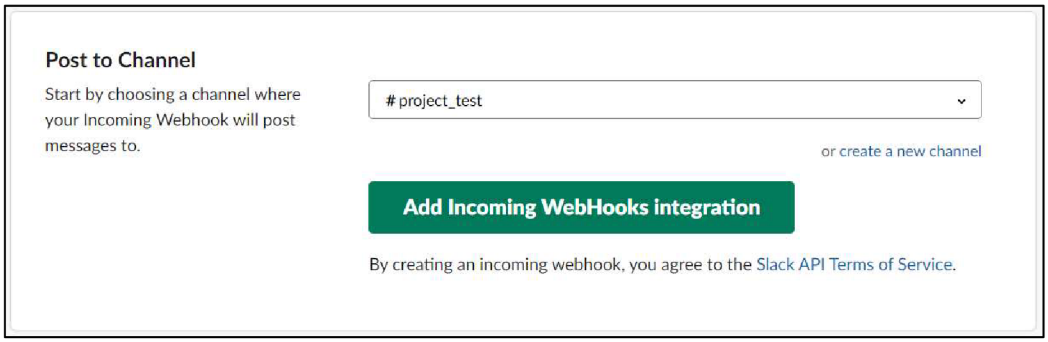
Aplikácia Slack je jednou z možností, ako môžu zamestnanci medzi sebou v rámci firmy komunikovať. Okrem aplikácie Slack používajú aj Microsoft Teams, a to primárne na realizáciu videohovorov, ale aj Microsoft Outlook na e-mailovú korešpondenciu. Pre rýchle, menej formálne sprostredkovanie, resp. zisťovanie informácií primárne písomnou formou je využívaný práve Slack (8).

Slack okrem tradičných četov umožňuje vytvorenie kanálov. Kanály sú vytvárané za účelom pokrytia konkrétnej témy, projektu či tímu ľudí. Táto forma umožňuje lepší prehľad, resp. ľahšie dohľadanie informácií, dokumentov týkajúcich sa konkrétnej problematiky, resp. konkrétneho projektu najmä v prípade, že tá istá skupina ľudí diskutuje viaceré oblasti súbežne (36).

Existencia kanálov v rámci Slacku bola využitá aj pre účely tohto nástroja, a to konkrétne spôsobom, že do kanála vytvoreného výlučne na tieto účely sa zasielajú notifikácie. Teda v prípade, že je zistená nejaká zmena na dátach, resp. nastane iná spomenutiahodná

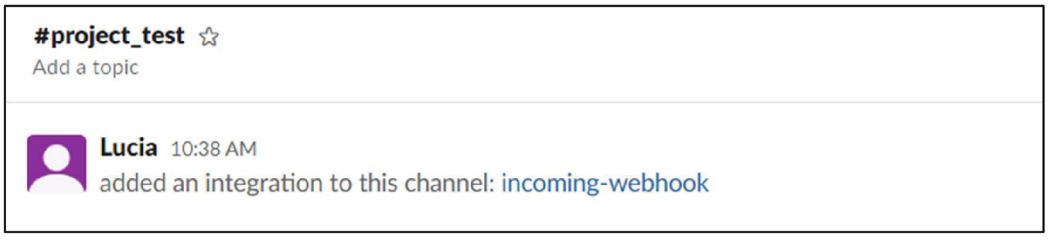
skutočnosť, sa do kanála pošle upozornenie popisujúce danú situáciu. Na existenciu novej správy v rámci kanála, t.j. tejto zaslanej notifikácie sú upozornení všetci jeho členovia. Je vhodné spomenúť, že navrhovaný nástroj sa začal používať na štyroch projektoch firmy, pričom každý z nich má zriadený samostatný kanál.

Pre zasielanie notifikácií plynúcich z výsledku testovania nástroja bolo však potrebné do aplikácie Slack zaintegrovať aplikáciu Incoming Webhooks, ktorá je určená práve na odosielanie správ v reálnom čase z externých zdrojov priamo do aplikácie Slack (37). Táto funkcionalita bola zároveň zahrnutá priamo do kanála, v ktorom bude využívaná. Kanál, ktorý bude v rámci nasledovnej demonštrácie používaný, bude značený ako *#project_test*.



Obrázok 51: Slack - integrácia Incoming Webhooks do kanála
(Zdroj: Vlastné spracovanie)

O tejto skutočnosti, čiže o integrácii Incoming Webhooks do daného kanála boli informovaní jeho členovia nasledovným spôsobom.



Obrázok 52: Slack - premietnutie informácie o integrácii Incoming Webhooks v rámci kanálu
(Zdroj: Vlastné spracovanie)

Aplikácia Incoming Webhooks využíva na prenos metódu POST HTTP protokolu. Prenášané informácie obsahujú samotný text správy a prípadne iné voliteľné detaily, ako je

napríklad meno odosielateľa daného Incoming Webhook, jeho príslušná ikonka či rôzne interaktívne prvky. Tieto informácie sú odosielané v JSON formáte (37).

Pri procese integrácie Incoming Webhooks do Slack kanálu dochádza k priradeniu URL kódu, pomocou ktorého je možné zasielanie daných notifikácií priamo do tohto kanálu. Jeho formát je vo všeobecnosti nasledovný:

```
https://hooks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
```

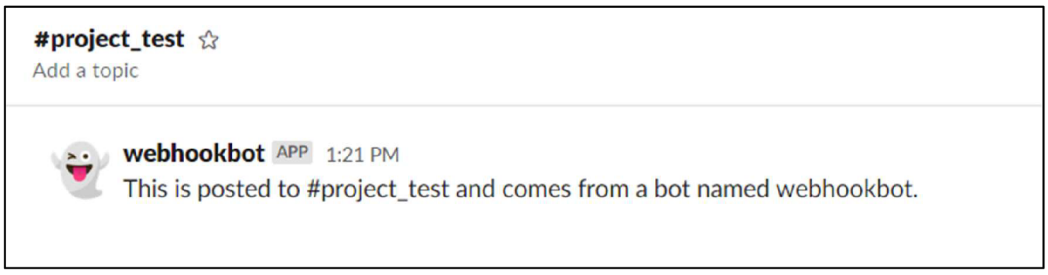
Obrázok 53: Slack - obecný formát URL kódu integrovaného Incoming Webhook
(Zdroj: 37)

Potom v prípade, že sa chce poslať veľmi jednoduchá správa s využitím pomenovania odosielateľa správy a pridelením ikonky pomocou príkazového riadku ako externého zdroja správ, je možné využiť tento pridelený URL kód. Príkaz v rámci príkazového riadku by mohol v prípade dosadenia obecného tvaru URL kódu vyzerať napríklad takto:

```
curl -X POST --data-urlencode "payload={\"channel\": \"#project_test\", \"user-
name\": \"webhookbot\", \"text\": \"This is posted to #project_test and comes from
a bot named webhookbot.\", \"icon_emoji\":\":ghost:\"}" https://ho-
oks.slack.com/services/T00000000/B00000000/XXXXXXXXXXXXXXXXXXXXXXXX
```

Obrázok 54: Slack - zasielanie správy pomocou príkazového riadku ako externého zdroja správ
(Zdroj: Vlastné spracovanie podľa 38)

Výstupom by bola doručená správa v rámci kanálu, a to v nasledovnom tvare:

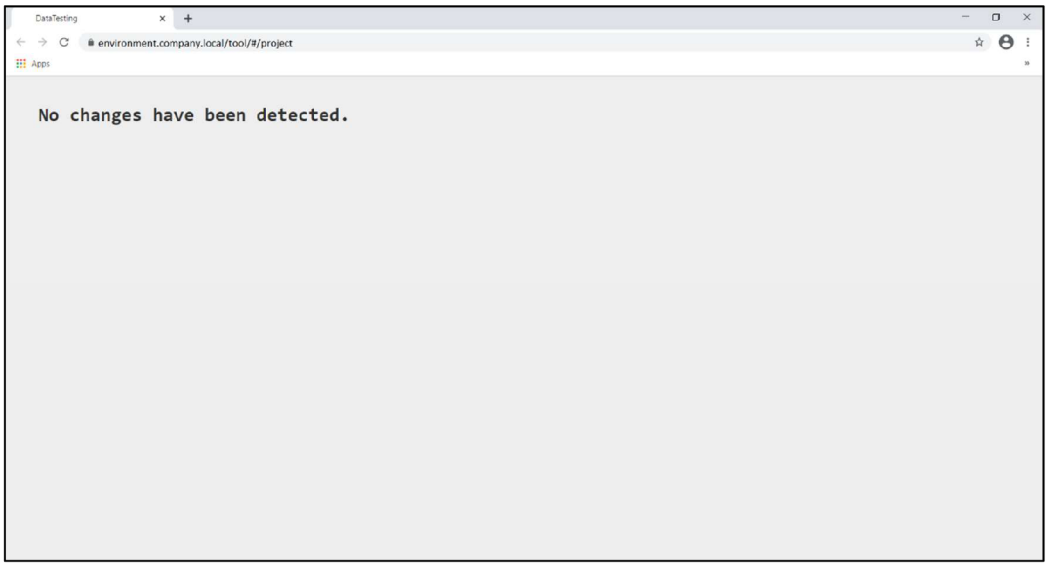


Obrázok 55: Slack - výsledok zaslania správy pomocou príkazového riadku ako externého zdroja správ
(Zdroj: Vlastné spracovanie)

Príkazový riadok ako externý zdroj správ bol použitý len pre demonštratívne účely. Reálne je týmto zdrojom nástroj, ktorý je predmetom tejto záverečnej práce. Teraz dôjde

k ukážkam, s akými rôznymi typmi správ sa reálne možno v rámci kanála stretnúť, t.j. aké typy správ nástroj odosiela.

K užívateľskému rozhraniu nástroja je možné dostať sa pomocou stránky umiestnenej na firemnom intranete. V rámci nasledujúcej demonštrácie bude označovaná ako *https://environment.company.local/tool/#/project*. V prípade, že nie sú v danom momente identifikované akékoľvek zmeny potrebné k ich schváleniu, resp. zamietnutiu, stránka poskytuje nasledujúce hlásenie. To tvrdí, že žiadne zmeny neboli detegované.

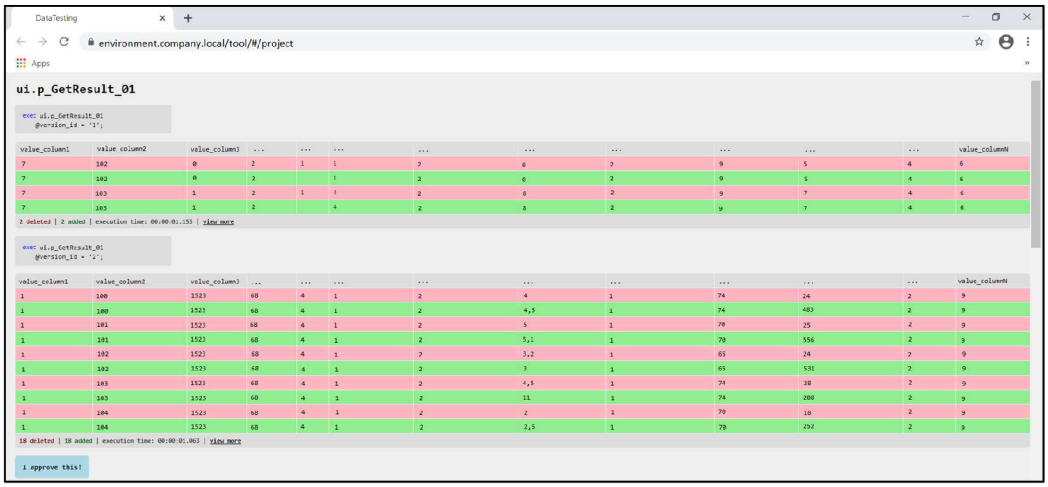


Obrázok 56: Užívateľské rozhranie nástroja - nie sú detegované žiadne zmeny
(Zdroj: Vlastné spracovanie)

Ak však boli po úprave databázovej schémy nájdené nejaké zmeny, tak v kanáli v rámci Slacku pribudne notifikácia nasledovného tvaru. Notifikácia zároveň rovno obsahuje odkaz na danú stránku.

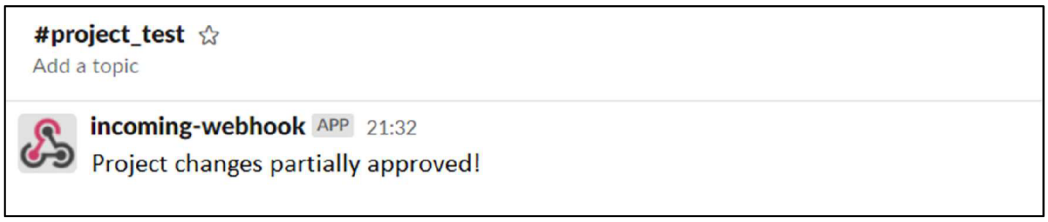
Obrázok 57: Slack - notifikácia v kanáli o zistení zmeny
(Zdroj: Vlastné spracovanie)

Po rozkliknutí odkazu užívateľ vidí, k akým zmenám pri volaní konkrétnych procedúr s konkrétnymi parametrami prišlo. Červené riadky znamenajú, že tento riadok ubudol, zatiaľ čo zelený indikuje, že bol pridaný. V prípade zmeny na úrovni už predtým existujúceho riadku je táto skutočnosť evidovaná tak, ako bolo kladené v požiadavkách firmy, t.j. tak, ako sme to v rámci nášho návrhu zohľadnili, čiže sa zaeviduje, ako keby došlo k jeho výmazu (s tými pôvodnými hodnotami) a zároveň pribudol (s tými novými hodnotami). Pre lepšiu prehľadnosť nie sú na obrazovke automaticky zobrazené všetky zmeny, ku ktorým došlo. Pre ich zobrazenie je potrebné kliknúť na tlačidlo *„View All“*. V prípade, že užívateľ s týmito zmenami súhlasí, môže kliknúť na tlačidlo *„I approve this!“*, ktoré je umiestnené na konci stránky. V opačnom prípade je potrebné nájdenie príčiny ich zmenenia a jej odstránenie.



Obrázok 58: Užívateľské rozhranie nástroja - sú detegované zmeny
(Zdroj: Vlastné spracovanie)

Zmeny môže užívateľ odobriť aj čiastočne, t.j. len pre konkrétnu kombináciu volanej procedúry s konkrétnou hodnotou príslušných parametrov. V takomto prípade sa to v kanáli v aplikácii Slack premietne s adekvátnou hláškou.



Obrázok 59: Slack - notifikácia v kanáli o čiastočnej akceptácii zistenej zmeny
(Zdroj: Vlastné spracovanie)

V prípade, že sú už odobrené všetky vzniknuté zmeny, dôjde k vygenerovaniu nasledovnej notifikácie:

Obrázok 60: Slack - notifikácia v kanáli o akceptácii zistenej zmeny
(Zdroj: Vlastné spracovanie)

A v neposlednom rade je vhodné uviesť, že nástroj v prípade zlyhania vykonávania volanej procedúry s príslušným parametrom odošle tomu odpovedajúcu informáciu na daný kanál v aplikácii Slack.

Obrázok 61: Slack - notifikácia v kanáli o zlyhaní procedúry
(Zdroj: Vlastné spracovanie)

3.4 Prínosy nástroja

Navrhnutie a následné nasadenie nástroja do štyroch projektov prinieslo firme množstvo benefitov. Uveďme niektoré z nich.

Nástroj umožňuje odhalenie zmien na dátovej úrovni. Tieto zmeny môžu byť žiaduceho i nežiaduceho charakteru. V prípade, že zmena chcená bola, užívateľ nástroja zmenu potvrdí. V prípade, že zmena žiadaná nebola, dôjde k adekvátnej náprave na úrovni databázovej schémy. Dá sa povedať, že nástroj poskytuje možnosť určitého auditu.

Nástroj ďalej umožňuje adresovať zodpovednosť. Vďaka vygenerovanému upozorneniu je možné určiť, ktorý commit do repozitára toto vytvorenie notifikácie spôsobil, a to na základe jeho času, t.j. skutočnosti, kedy k nemu došlo. Keďže commit v repozitári eviduje aj údaj o jeho autorovi, je teda možné zistiť, vďaka komu k tejto zmene došlo.

Dôležitým benefitom nástroja je aj zvýšenie kvality projektu. Vzhľadom na skutočnosť, že pred týmto nástrojom dochádzalo len k vizuálnej kontrole dát, a aj to len na vybraných verziách, tento nástroj umožňujúci kontrolu všetkého zabezpečí nižšie riziko chybovosti, teda nekvality produktu.

Pri porovnaní s predchádzajúcim kontrolovaním týchto zmien nemožno opomenúť časové úspory. Hrubým odhadom bolo vypočítané, že v rámci jedného z týchto projektov dôjde vďaka existencii tohto nástroja k časovej úspore 160 hodín ročne, čo odpovedá dĺžke jedného mesiaca pracovníka zamestnaného na plný pracovný úväzok. Netreba zabúdať na to, že nástroj je aktuálne nasadený na štyroch projektoch, a teda môže dochádzať k približne 640 hodinovej úspore času ročne.

V neposlednom rade je vhodné uvedenie skutočnosti, že nástroj je schopný informovať nielen o zmenách, ktoré nastali na dátovej úrovni, ale zároveň podáva informáciu o tom, či všetky nadefinované procedúry na spustenie prebehli korektne, alebo došlo k chybe.

ZÁVER

Na začiatku práce boli podložené teoretické východiská, akými sú vysvetlenie problematiky testovania softvéru i objasnenie použiteľných technológií pre naplnenie cieľa práce. Tieto východiská boli následne využité v návrhovej časti práce.

Po poskytnutí týchto teoretických podkladov došlo k stručnému predstaveniu firmy z hľadiska jej predmetu podnikania, pre ktorú bol cieľový nástroj, teda nástroj pre automatizované integračné testovanie navrhnutý. Taktiež došlo k analýze testovacích spôsobov firmy, u ktorých bol zistený priestor na zlepšenie. Týmto priestorom bolo doterajšie backend testovanie zmien na dátovej úrovni pri zmene databázovej schémy. Toto posudzovanie prípadných zmien bolo vykonávané čisto vizuálnou kontrolou, a to len na vzorke dát. Táto skutočnosť signalizovala potrebu zmeny. Touto zmenou sa stal práve návrh a implementácia automatizovaného integračného testovacieho nástroja. Požiadavky naň kladené, t.j. zmeny, ktoré by mal nástroj skúmať, boli nadefinované vychádzajúc z doterajších vizuálnych kontrol.

V rámci návrhovej časti práce bol odprezentovaný postup návrhu a implementácie tohto nástroja. Pri návrhu, resp. implementácii došlo k použitiu firmou zaužívaného objektovo orientovaného programovacieho jazyka C# v Microsoft Visual Studio.

Tento proces tvorby nástroja začal návrhom vlastnej databázy, ktorej úlohou bolo zaznamenávanie získavaných čiastkových i celkových výsledkov skúmania zmien. Táto databáza bola vytvorená a následne s ňou bolo interagované za pomoci objektovo-relačného mapovania, konkrétne technológiou Entity Framework. V práci odprezentovaný návrh nástroja pokryl jeho backend hľadisko, ktorého funkčnosť bola následne demonštrovaná na vzorových dátach. K nástroju bolo teda potrebné dotvoriť jeho frontend stránku, ktorá však nebola predmetom tejto práce.

Potom mohlo dôjsť k jeho integrácii do prostredia firmy tak, aby užívatelia, v tomto prípade teda primárne vývojári, ktorí doposiaľ tieto kontroly vykonávali manuálne, mohli získavať informácie o výsledku tohto testovania a následne s nimi adekvátne naložiť. Pre tieto účely bola využitá aplikácia Slack, ktorú firma bežne využíva na rýchlu komunikáciu medzi zamestnancami. Táto aplikácia však umožňuje aj tvorbu tzv. kanálov, ktorým môže byť dodaná špeciálna funkcionalita. Tento prípad sa využil aj v kontexte tohto nástroja.

V Slacku bol pre tieto účely vytvorený samostatný kanál so zaintegrovanou funkcionalitou Incoming Webhooks. Tá je určená práve na odosielanie správ v reálnom čase z externých zdrojov, ako je napríklad v práci navrhovaný nástroj, priamo do aplikácie Slack. V rámci práce bolo odprezentované, ako tento kanál umožňuje užívateľom interagovať s výsledkami generovanými nástrojom.

V rámci zhrnutia významu tohto nástroja možno konštatovať, že nástroj výrazne zlepšil zachytávanie chýb a uľahčil vývojárom prácu. Napomohol k lepšiemu mapovaniu prípadných zmien na dátovej úrovni počas vývoja, teda úprav databázovej schémy, a to spôsobom, že vždy pri nahratí zmeny kódu do databázového prostredia je vykonané porovnanie verzií. Zároveň dôjde k upozorneniu, či mala táto zmena vplyv na dáta, a ak áno, tak aký.

Prínos nástroja spočíva primárne v zjednodušení práce databázových vývojárov, ktorí týmto získavajú prehľad o prípadných zmenách na dátovej úrovni počas modifikácie databázovej schémy. Zároveň dochádza k zvýšeniu záruky kvality nástroja, ktorá by inak mohla byť dosahovaná len čiastočne z dôvodu čisto vizuálnych testov realizovaných len na vybranej vzorke prípadov. Tieto vizuálne kontroly boli vykonávané vývojármi, čo znamená, že toto zautomatizovanie zároveň umožnilo časovú úsporu. V prípade zavedenia nástroja v rozsahu jedného projektu (v skutočnosti je aktuálne využívaný v rámci štyroch projektov) dochádza k úspore v podobe jedného pracovného mesiaca človeka zamestnaného na trvalý pracovný pomer ročne.

ZOZNAM POUŽITÝCH ZDROJOV

1. OLSEN, K., M. POSTHUMA a S. ULRICH. *Certified Tester: Foundation Level Syllabus*. 8th ed. Belgium: ISTQB, 2019.
2. DORONINS, Andrejs. ISTQB® Foundation: Testing throughout the Software Development Lifecycle. In: *Pluralsight* [online]. 7.12.2020 [cit. 2021-04-09]. Dostupné z: https://app.pluralsight.com/library/courses/istqb-foundation-testing-software-development-lifecycle/table-of-contents
3. KNEUPER, Ralf. *Software Processes and Life Cycle Models*. Germany: Springer, 2018. ISBN 978-3-319-98845-0.
4. *The 2020 Scrum Guide™*. Scrum Guides [online]. USA: Scrum Guides, ©2020, [cit. 2021-04-10]. Dostupné z: https://scrumguides.org/scrum-guide.html
5. SPILLNER, A., T. LINZ a H. SCHAEFER. *Software Testing Foundations*. 4th ed. USA: Sheridan, 2014. ISBN 978-1-937538-42-2.
6. BUREŠ, Miroslav a kol. *Efektivní testování softwaru*. Česká republika: Grada Publishing, 2016. ISBN 978-80-271-9389-9.
7. TROELSEN, Andrew a Philip JAPIKSE. *Pro C# 7 With .NET and .NET Core.* Eighth Edition. USA: Apress, 2017. ISBN 978-1-4842-3017-6.
8. VÝVOJÁR FIRMY. *Softvér, resp. technológie používané vo firme a testovanie softvéru v podniku z backend hľadiska* [ústna komunikácia]. LOGEX Solution Center. Nové sady 996/25, Brno. Január 2020.
9. CLARK, Dan. *Beginning C# Object-Oriented Programming*. USA: Apress, 2013. ISBN 978-1-4302-4936-8.
10. PECINOVSKÝ, Rudolf. *OOP: naučte se myslet a programovat objektově*. Brno: Computer Press, 2010. ISBN 978-80-251-2126-9.
11. BEŇO, Jaroslav. C# pre mierne pokročilých. In: *Learn2Code*. 4.1.2020 [cit. 2020-09-05]. Dostupné z: https://learn2code.sk/kurzy/c-sharp-pre-mierne-pokrocilych
12. BEŇO, Jaroslav. Princípy objektovo orientovaného programovania SOLID. In: *robime.it* [online]. 28.5.2019 [cit. 2021-05-10]. Dostupné z: https://robime.it/principy-objektovo-orientovaneho-programovania-solid/
13. *What is an IDE?* Red Hat [online]. USA: Red Hat, ©2021, [cit. 2021-05-09]. Dostupné z: https://www.redhat.com/en/topics/middleware/what-is-ide

14. STRAUSS, Dirk. *Getting Started with Visual Studio 2019: Learning and Implementing New Features*. South Africa: Apress, 2020. ISBN 978-1-4842-5449-3.
15. LAURENČÍK, Marek. *SQL - Podrobný průvodce uživatele*. Praha: Grada Publishing, 2018. ISBN 978-80-271-2154-0.
16. LACKO, Luboslav. *1001 tipů a triků pro SQL*. Brno: Computer Press, 2011. ISBN 978-80-251-3010-0.
17. BEN-GAN, Itzik. *Microsoft® SQL Server ® 2012 T-SQL Fundamentals*. USA: Microsoft Press, 2012. ISBN 978-0-735-65814-1.
18. CARTER, Peter A. *SQL Server Advanced Data Types: JSON, XML, and Beyond*. UK: Apress, 2018. ISBN 978-1-4842-3901-8.
19. SCHWICHTENBERG, Holger. *Modern Data Access with Entity Framework Core*. Germany: Apress, 2018. ISBN 978-1-4842-3552-2.
20. *Entity Framework 6*. Microsoft Docs [online]. USA: Microsoft, ©2021, [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/ef/ef6/
21. *Creating a Model*. Microsoft Docs [online]. USA: Microsoft, ©2021, [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/ef/ef6/modeling/
22. *Model First*. Microsoft Docs [online]. USA: Microsoft, ©2021, [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/ef/ef6/modeling/designer/workflows/model-first
23. *Code First to a New Database*. Microsoft Docs [online]. USA: Microsoft, ©2021, [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/workflows/new-database
24. *Defining DbSets*. Microsoft Docs [online]. USA: Microsoft, ©2021, [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/dbsets
25. *Code First Data Annotations*. Microsoft Docs [online]. USA: Microsoft, ©2021, [cit. 2021-05-10]. Dostupné z: https://docs.microsoft.com/en-us/ef/ef6/modeling/code-first/data-annotations
26. *Výpis z obchodního rejstříku.* Veřejný rejstřík a Sbírka listin [online]. Praha: Ministerstvo spravedlnosti České republiky, ©2012-2015, [cit. 2020-11-14].

Dostupné z: https://or.justice.cz/ias/ui/rejstrik-firma.vysledky?subjektId=974520&typ=PLATNY

27. *LOGEX Solution Center s.r.o.* StartupJobs [online]. Praha: StartupJobs.com, ©2012-2021, [cit. 2020-11-14]. Dostupné z: https://www.startupjobs.cz/startup/logex-solution-center-s-r-o
28. *LOGEX, V2H, Prodacapo and Ivbar join forces to turn data into better healthcare in Europe.* LOGEX Healthcare Analytics [online]. Amsterdam: LOGEX, ©2021, [cit. 2020-11-14]. Dostupné z: https://logex.com/logex-offers-the-most-advanced-and-complete-solutions-in-healthcare-analytics/
29. *Healthcare today faces greater challenges than ever.* LOGEX Healthcare Analytics [online]. Amsterdam: LOGEX, ©2021, [cit. 2020-11-15]. Dostupné z: https://logex.com/solutions/
30. *Financial.* LOGEX Healthcare Analytics [online]. Amsterdam: LOGEX, ©2021, [cit. 2020-11-15]. Dostupné z: https://logex.com/solutions/financial/
31. *Outcomes.* LOGEX Healthcare Analytics [online]. Amsterdam: LOGEX, ©2021, [cit. 2020-11-15]. Dostupné z: https://logex.com/solutions/outcomes/
32. *Value.* LOGEX Healthcare Analytics [online]. Amsterdam: LOGEX, ©2021, [cit. 2020-11-15]. Dostupné z: https://logex.com/solutions/value/
33. SYSTÉMOVÝ ANALYTIK FIRMY. *Testovanie softvéru v podniku - obecne* [písomná komunikácia]. Február 2021.
34. SYSTÉMOVÝ ANALYTIK FIRMY. *Testovanie softvéru v podniku – detaily* [telefonická komunikácia]. Apríl 2021.
35. VÝVOJÁR FIRMY. *Integrácia nástroja* [telefonická komunikácia]. Marec 2021.
36. *What is a channel?* Slack [online]. San Francisco: Slack Technologies, ©2021, [cit. 2021-04-06]. Dostupné z: https://slack.com/intl/en-sk/features/channels
37. *Sending messages using Incoming Webhooks*. Slack [online]. San Francisco: Slack Technologies, ©2021, [cit. 2021-04-06]. Dostupné z: https://api.slack.com/messaging/webhooks
38. *Sending your first Slack message using Webhook*. Slack [online]. San Francisco: Slack Technologies, ©2021, [cit. 2021-04-06]. Dostupné z: https://api.slack.com/tutorials/slack-apps-hello-world

ZOZNAM POUŽITÝCH CUDZÍCH POJMOV

Backend je termín označujúci skrytú časť softvéru (užívateľ ju priamo nevidí), ktorá interaguje so serverom a databázou. Možno ju považovať za základ tohto softvéru.

Frontend je termín označujúci časť softvéru, ktorú užívateľ vidí a pracuje s ňou. Vďaka nej je užívateľ schopný využívať služby backend strany softvéru. Častokrát sa označuje aj ako užívateľské rozhranie, prípadne skratkou UI (z anglického User Interface).

Commit je termín označujúci záznam verzie zdrojového kódu.

IDE (z anglického Integrated Development Environment) je skratkou pre integrované vývojové prostredie, čiže softvér slúžiaci na vytváranie aplikácií, ktorý kombinuje bežné vývojárske nástroje do jedného grafického používateľského rozhrania.

Incoming Webhooks je funkcionalita určená na odosielanie správ v reálnom čase z externých zdrojov do aplikácie akou je napríklad Slack.

NuGet Packages sú súbory obsahujúce skomplikovaný kód určený pre opakované použitie. Môžu byť zdieľané v rámci organizácie, ale aj so širokou verejnosťou. Prácu s nimi umožňuje softvér Microsoft Visual Studio.

Product backlog je termín označujúci zoznam úloh, ktoré sa majú postupne, v závislosti od ich priority, pri vývoji softvéru zrealizovať. Ide o pojem z kontextu Scrum metodiky.

Product owner je termín označujúci člena Scrum tímu, ktorý má na starosti maximalizáciu hodnoty vyvíjaného produktu. Tvorí a komunikuje cieľ produktu. Taktiež tvorí, komunikuje a priraďuje prioritu položkám z product backlogu.

V prípade, že je produkt veľkého rozsahu v zmysle veľkého množstva jeho samostatných inštancií, t.j. verzií ponúkaných rozličnej klientele (napr. v závislosti od krajiny predaja), ktoré majú svoje vlastné špecifiká, si niektoré firmy túto rolu delia na:

- **Core product owner** zodpovedajúceho za celý produkt, majúc na starosti základné funkcionality produktu.
- **Country product owners** zodpovedajúcich za jednotlivé inštancie produktu. Majú na starosti špecifiká týchto inštancií produktu.

Scrum master je termín označujúci člena Scrum tímu, ktorý má na starosti zabezpečenie korektného chodu Scrum metódy jej tlmočením ostatným členom Scrum tímu. Taktiež zodpovedá za efektívnosť Scrum tímu.

Scrum metodika je termín označujúci agilnú metódu riadenia iteratívneho modelu životného cyklu vývoja softvéru. Vytvára Scrum tím, ktorý je tvorený product ownerom, Scrum masterom a vývojármi.

Sprint je termín označujúci jednu dobu iterácie vývoja softvéru v terminológii Scrum metodiky. V tomto časovom úseku je vyvinuté malé množstvo nových vlastností, prípadne sa len doladia, resp. opravia tie už jestvujúce.

ZOZNAM TABULIEK

ZOZNAM OBRÁZKOV