Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra obchodu a financí

Bakalářská práce

Problematika VRP aplikace v optimalizaci distribučních cest

Vilém Zelienka

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Vilém Zelienka

Informatika

Název práce

Problematika VRP aplikace v optimalizaci distribučních cest

Název anglicky

The Issue of VRP Applications in Optimizing Distribution Channels

---

Cíle práce

Cílem bakalářské práce je vytvořit aplikaci pro hledání a optimalizaci distribučních cest a zjednodušení činnosti zaměstnanců v oboru obchodní a dopravní logistiky, a to díky zaměření aplikace na přívětivost uživatelského rozhraní a jednoduchost.

Metodika

Bakalářská práce vychází z předpokladu systematického zpracování teoretických východisek pro vytvoření vlastní práce. Teoretická východiska budou zpracována na základě samostatného studia tematicky zaměřené odborné literatury. Na základě zpracování přehledu současného stavu řešené oblasti bude zpřesněn cíl bakalářské práce, jehož dosažení bude předmětem vlastní části práce.

V části vlastní práce bude zaměřena na tvorbu aplikace pro hledání a optimalizaci distribučních cest v oboru obchodní a dopravní logistiky. zaměřena na tvorbu aplikace pro operační systém Windows, pro hledání a optimalizaci distribučních cest v oboru obchodní a dopravní logistiky, kam uživatel zadá parametry, nutné pro výpočet optimální trasy. Aplikace poté spočítá optimální cestu, vzhledem k zadaným informacím, a zobrazí danou trasu na mapě. Aplikace je vytvořena za použití objektově orientovaného přístupu k programování a programovacího jazyka C# pro logickou část programu a XAML pro část uživatelského rozhraní aplikace. Pro tvorbu aplikace bude použit nástroj Visual Studio od společnosti Microsoft, který využívá knihoven .NET Framework pro práci s jazykem C#. Dále budou v této části práce vysvětleny postupy tvorby dané aplikace a výhody použitých přístupu a jazyka C#.

Doporučený rozsah práce

30 – 40 stran

Klíčová slova

VRP aplikace, objektově orientované programování, jazyk C, logistika, dopravní problémy

---

Doporučené zdroje informací

Kubíčková, L. Obchodní logistika. Brno: Mendelova zemědělská a lesnická univerzita, 2006. ISBN 80-7157-952-1.

Merunka, V. Objektové modelování. Praha: Alfa Nakladatelství, 2008. ISBN 978-80-87197-04-2.

Šubrt, T. Ekonomicko-matematické metody. Plzeň: Vydavatelství a nakladatelství Aleš Čeněk, 2011. ISBN 978-80-7380-345-2.

Předběžný termín obhajoby

2017/18 ZS – PEF (únor 2018)

Vedoucí práce

Ing. Helena Čermáková, Ph.D.

Garantující pracoviště

Katedra obchodu a financí

Elektronicky schváleno dne 27. 11. 2017

Ing. Helena Čermáková, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 27. 11. 2017

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 28. 11. 2017

Čestné prohlášení

Prohlašuji, že svou bakalářskou práci " Problematika VRP aplikace v optimalizaci distribučních cest" jsem vypracoval samostatně pod vedením vedoucího bakalářské práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené bakalářské práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28.11.2017 _____

Poděkování

Rád bych touto cestou poděkoval Heleně Čermákové za skvělé vedení a trpělivost při zpracování této práce.

Problematika VRP aplikace v optimalizaci distribučních cest

Souhrn

Práce je zaměřena na návrh a tvorbu Windows Forms aplikace v programovacím jazyce C#. Aplikace je schopná vypočítat optimální trasu mezi zadanými místy, za pomoci nástrojů pro VRP, a následně zobrazit mapu s vypočtenou cestou. K tvorbě aplikace jsou použity přístupy objektově orientovaného programování.

Klíčová slova: VRP aplikace, objektově orientované programování, jazyk C#, logistika, dopravní problémy

The Issue of VRP Applications in Optimizing Distribution Channels

Summary

This thesis is focused on design and implementation of Windows Forms application in C# programming language. Application is able to determine optimal route through places, with the use of tools for VRP, and show found route in a map. Object oriented programming is used for the application development.

Keywords: VRP application, object oriented programming, C# language, logistics, transportation theory

Obsah

1 Úvod 11

2 Cíl práce a metodika 12

2.1 Cíl práce 12

2.2 Metodika 12

3 Teoretická východiska 13

3.1 Objektově orientované programování 13

3.1.1 Historie OOP 13

3.1.2 Porovnání lineárního programování s OOP 14

3.1.3 Kombinovaný přístup 15

3.2 Jazyk C 15

3.3 Datové modelování 16

3.4 Objektová normalizace 16

3.4.1 Objektové normální formy 17

3.5 Logistika 18

3.5.1 Historie logistiky 19

3.5.2 Logistické řízení 19

3.6 Vehicle routing problem 20

3.6.1 Clarke-Wrightova metoda 21

3.6.2 Okružní dopravní problémy 23

3.6.2.1 Jednookruhový okružní dopravní problém 23

3.6.3 Metoda nejbližšího souseda 23

4 Vlastní práce 25

4.1 Vývojové prostředí pro Microsoft Windows aplikace 25

4.1.1 .NET Framework 25

4.1.2 IDE – Microsoft Visual Studio 26

4.1.2.1 Visual Studio Team Services 26

4.2 Konvence při psaní kódu 28

4.3 Základní pojmy 28

4.3.1 Modifikátory 28

4.3.2 Typy 29

4.3.3 Forma 29

4.3.4 Třída 30

4.3.4.1 Deklarace třídy 31

4.3.4.2 Tvorba objektů ... 31
4.3.5 API ... 31
4.3.6 JSON ... 32
4.4 Vytvoření projektu ve Visual Studiu ... 32
4.5 Grafické rozhraní aplikace ... 34
4.6 Programová část ... 37
4.6.1 Třída Zákazník ... 37
4.6.2 Třída Databaze ... 37
4.6.3 Soubor VRP ... 39
4.6.3.1 Třída Google ... 39
4.6.3.2 Metoda ZjistiSouradnice ... 40
4.6.3.3 Metoda SpocitejVzdalenost ... 40
4.6.3.4 Metoda GetMapUri ... 40
4.6.4 Třída NejblizsiSoused ... 42
5 Závěr ... 45
6 Seznam použitých zdrojů ... 46
7 Přílohy ... 47

Seznam obrázků

Obrázek 1: Version control ve VSTS 27
Obrázek 2: Listener tlačítka Spocitej 30
Obrázek 3: Vytvoření instance třídy 31
Obrázek 4: Vytvoření projektu 32
Obrázek 5: Nově vytvořený projekt 33
Obrázek 6: Form1 34
Obrázek 7: FormPridaniZakaznika 35
Obrázek 8: Listener tlačítka Pridat 36
Obrázek 9: FormZobrazeni 36
Obrázek 10: Metoda pro zavedení třídy Databaze 38
Obrázek 11: Přidání výchozího zákazníka do Databaze 38
Obrázek 12: Tvorba URL pro získaní mapy 41
Obrázek 13: Kód pro vytvoření seznamu vzdáleností mezi zákazníky 42
Obrázek 14: Hlavní část metody pro výpočet trasy 43
Obrázek 15: Změna proměnných na konci logické části výpočtu trasy 44

Seznam zkratek

OOP = objektově orientované programování

VRP = vehicle routing problém

ONF = objektové normální formy

TSP = travelling salesman problem

API = Application programming interface

SDK = Software Development Kit

JSON = JavaScript Object Notation

IDE = Integrated Development Environment

1 Úvod

Neustále se rozvíjející internetové a mobilní technologie v posledních letech tvoří poptávku po internetových obchodech a dalších online službách. Nedílnou součásti tohoto odvětví je rozšiřování možností přepravy zboží. Díky tomu je možné v posledních několika letech zaznamenat nárůst soukromých, především zahraničních, přepravních společností. S tímto také vzniká významný společenský nárůst poptávky po aplikacích, které mohou zjednodušit a automatizovat výpočet distribuci zboží.

Právě díky této poptávce vznikl nápad na vytvoření aplikace, který poskytovala alespoň některé z funkcí nutných pro optimalizaci distribučních cest. Cílovou platformou pro aplikaci byl zvolen operační systém Microsoft Windows, který má celosvětově největší podíl na trhu operačních systémů pro osobní počítače a je také nejpoužívanějším operačním systémem ve firemním prostředí.

Cíl práce a metodika

1.1 Cíl práce

Cílem bakalářské práce je vytvořit aplikaci pro hledání a optimalizaci distribučních cest a zjednodušení práce zaměstnanců v oboru obchodní a dopravní logistiky, a to díky zaměření aplikace na přívětivost uživatelského rozhraní a jeho jednoduchost.

1.2 Metodika

Bakalářská práce vychází z předpokladu systematického zpracování teoretických východisek pro vytvoření vlastní práce. Teoretická východiska budou zpracována na základě samostatného studia tematicky zaměřené odborné literatury. Na základě zpracování přehledu současného stavu řešené oblasti bude zpřesněn cíl bakalářské práce, jehož dosažení bude předmětem vlastní části práce.

V části vlastní práce bude zaměřena na tvorbu aplikace pro hledání a optimalizaci distribučních cest v oboru obchodní a dopravní logistiky. zaměřena na tvorbu aplikace pro operační systém Windows, pro hledání a optimalizaci distribučních cest v oboru obchodní a dopravní logistiky, kam uživatel zadá parametry, nutné pro výpočet optimální trasy. Aplikace poté spočítá optimální cestu, vzhledem k zadaným informacím, a zobrazí danou trasu na mapě. Aplikace je vytvořena za použití objektově orientovaného přístupu k programování a programovacího jazyka C#. Pro tvorbu aplikace bude použit nástroj Visual Studio od společnosti Microsoft, který využívá knihoven .NET Framework pro práci s jazykem C#. Dále budou v této části práce vysvětleny postupy tvorby dané aplikace a výhody použitých přístupu a jazyka C#.

Teoretickou částí práce bude studium online a off-line informačních zdrojů a článků, v návaznosti na to budou ony zdroje použity k prezentaci získaných vědomostí o historii a využití použitých problematik logistiky, jazyka C# a s ním spojený .NET Framework a objektově orientovaného programování.

2 Teoretická východiska

Tato část práce je zaměřena na průzkum a shrnutí základních témat potřebných pro pochopení problematiky této bakalářské práce.

2.1 Objektově orientované programování

Objektově orientované programování (OOP) je způsob vytváření softwarové architektury, který umožnuje flexibilitu díky modulárnímu designu. Lidé, kteří využívají objektově orientovaného programování nemusí být nutně lepšími programátory, zvolili si však mnohem strategičtější a čistší cestu k programování. OOP není jazyk, je to přístup k architektuře SW a myšlenkové pochody, které vytvářejí objektově orientované aplikace a jazyky (Stefanov, 2008).

2.1.1 Historie OOP

Většina odborníků se mylně domnívá, že OOP je produkt 80. let a práce Bjarna Stroustrupa, který přesunul programovací jazyk C do objektově orientovaného světa vytvořením jazyka C++. Nicméně, prvními objektově orientovanými jazyky jsou SIMULA 1 (1962) a Simula 67 (1967). Jazyky Simula byly vytvořeny Ole-Johnem Dahlem a Kristen Nygaard v Norském výpočetním centru v Oslu. I přes to, že většina kladů OOP již byla obsažena v jazycích Simula, až nástupem C++, v 90. letech, se začalo OOP rozvíjet (Nygaard a Dahl, 1978).

Vývoj OOP je datován roku 1991, kdy James Gosling, Bill Joy, Patrick Naughton, Mike Sheradin pracovali na projektu Stealth. Cílem tohoto projektu bylo vytvoření inteligentního elektronického zařízení, které by bylo možné centrálně ovládat a programovat z přenosného zařízení. Autoři se rozhodli, že OOP je správný směr, kterým se budou ubírat, nicméně C++ se projevil jako příliš robustní jazyk pro jejich projekt. Tak vznikl programovací jazyk Oak, který byl později přejmenován na jazyk Java. Jazyk Oak rychle získával popularitu, která byla umocněna nástupem World Wide Webu (WWW). Tento nárůst byl také zapříčiněn integrací modulů do internetových prohlížečů, které

umožnily spouštět Java aplikace v internetovém prohlížeči přímo z internetu. S takto rozšířenou funkcionalitou se World Wide Web, díky Javě, rychle rozrostl.

OOP, jako přístup k programování, se zrodilo v 70. letech v USA v Palo Alto Research Center (PARC) v Kalifornii. Společnost Xerox v té době vyvíjela projekt osobního počítače budoucnosti Dynabook, u kterého se předpokládalo, že bude obsahovat jednotné softwarové prostředí, které bude plnit jak úkol operačního systému, tak zároveň i programovacího jazyka. Tento systém byl pojmenován Smalltalk. Odrazily se v něm prvky jazyka LISP a Simula (Merunka, 2008).

Od vzniku OOP dochází k soutěži mezi dvěma koncepty. Na jedné straně stojí čistý koncept OOP a na straně druhé koncept smíšený. Jak z hlediska tvorby softwaru, tak z hlediska modelování dat, je čisté OOP ideální. Bohužel, v dnešní době je, díky jeho vysoké úrovni abstrakce, jeho implementace poměrně složitá. Na druhou stranu smíšený koncept, kde je úroveň abstrakce mnohem nižší, se víc přibližuje tomu, jak počítače pracují (Merunka, 2008).

2.1.2 Porovnání lineárního programování s OOP

Procedurální programování neboli lineární metoda programování, často ústí v existenci řádek kódu, ve kterých nejsou nijak odděleny různé chování aplikace. Takto se z programovacího jazyka nestává nic jiného než série rutinních úkonů. Procedurální programování může dobře fungovat, pokud na projektu pracuje pouze jeden programátor, který je dobře obeznámen se svým kódem. Jakmile se však projektu účastní více lidí, je nutné a časově náročné neustále se seznamovat s kódem ostatních a vyhledávat, ve které části programu je, v případě potřeby, nutné kód změnit. V OOP je každé chování v aplikaci popsáno ve vlastní unikátní třídě, což poskytuje elegantnější a jednodušší náhled na spolupráci různých objektů a chování. Díky tomu, že každá třída má své vlastní unikátní jméno a obsahuje pouze jeden typ chování aplikace, je tak velice jednoduché nalézt požadovanou třídu a změnit ji (Merunka, 2008).

2.1.3 **Kombinovaný přístup**

Je možné konstatovat, že ve světě počítačů platí něco, co by se dalo nazvat jako „zákon o zachování složitosti“. Tedy pokud je vytvářen systém, který má být jednoduchý pro uživatele, bude velice složité ho implementovat, tak aby mu počítač rozuměl. Na druhou stranu, věci jako je strojový kód, který je pro normální lidi velice složitý, právě díky nízké úrovni abstrakce, zvládne počítač implementovat bez problémů (Merunka, 2008).

Z těchto důvodů bylo přistoupeno ke kompromisu, kterým je právě zmíněný smíšený koncept, ve kterém bylo docíleno jednodušší implementace pro počítače za cenu ztráty abstraktních vlastností OOP. Tímto se ulehčí práce tvůrců programovacích jazyků a systémů, naopak se velice ztíží práce programátorů a běžného uživatele těchto jazyků a systémů. Většina moderních jazyků se vydává smíšenou cestou, včetně jazyků jako jsou C#, C++ nebo Java. V dnešní době, kdy jsou výkony počítačů nepředstavitelně vyšší, než byly dříve, byl by teoreticky čistý přístup možný. Bohužel, díky zpětné kompatibilitě se stále používá systém smíšený (Merunka, 2008).

Objekt je základní stavební jednotkou OOP. Prostřednictvím objektů se objektově orientovaný programátor snaží, co možná nejvěrněji, napodobit reálný svět. Objekt je vlastně jakýmsi zjednodušením reálného objektu jako například člověk. V reálném světě má každý člověk charakteristické rysy jako je např. barva očí, barva a délka vlasů, věk, jméno. Tyto rysy jsou v OOP nazývány parametry nebo atributy. A dále má každý člověk nějaké chování. Toto chování je nazýváno metoda. Pokud by existoval objekt, který by reprezentoval člověka s atributy jméno a příjmení, mohla by existovat například metoda „rekniJmeno“, která by vrátila jméno a příjmení dohromady (Purdum, 2008).

2.2 **Jazyk C**

Jazyk C byl mateřským jazykem C++. Spousta programátorů tvrdí, že C++ je velmi silný jazyk a dodnes je velice rozšířený, bohužel s takovou silou přichází také značná komplexita. Proto chtěli vývojáři programovacích jazyků vytvořit jednodušší a méně komplexní jazyk pro OOP, tak vznikl jazyk C# od společnosti Microsoft (Smith, 2014).

Pro mnohé programátory je jazyk C# odpovědí Microsoftu na Javu. Někteří dokonce tvrdí, že C# je výsledkem „tvrdohlavosti“ Microsoftu v podpoře jiných programovacích jazyků než těch, které sám vytvořil (Smith, 2014).

2.3 Datové modelování

Datové modelování slouží k vytvoření objektového modelu budoucí aplikace. Objektový model by měl sloužit jako koncept pro odborníky, na kterém mohou postavit funkční aplikaci. Datové modelování se však vůbec nezabývá funkční stránkou aplikace, modelovány jsou pouze objekty, které slouží pouze k ukládání dat a přístupu k nim, dále je také určeny vazby mezi objekty a jejich vzájemná komunikace (Merunka, 2008).

Většina analytiků se domnívá, že objekty v OOP by měly být více méně pevně určeny, např. auta mají vypadat takto..., osoby takto... apod. Načež by už měly být jen vytvářeny vazby mezi těmito objekty. Takový přístup by fungoval, pokud by byla snaha vytvořit model světa, jako takového, nebo nějaké jeho velké části. Většinou jsou však tvořeny pouze menší projekty, kde by bylo mnoho, ne-li většina atributů nadbytečných. Z těchto důvodů jsou většinou třídy objektů vytvářeny podle potřeby daného modelu (Merunka, 2008).

Datové modelování je poměrně složitý proces, avšak existují metody a postupy, které mohou jejich tvorbu usnadnit.

2.4 Objektová normalizace

Od objektové normalizace jsou očekávány následující vlastnosti:

- **jednoduchost** – normalizace by měla být, pokud možno co nejjednodušší s ohledem na daný problém, neměla by obsahovat zbytečné definice, vazby apod., které jsou mimo zadaný rozsah projektu,
- **konkrétnost** – neměla by se zaměřovat na objekty odpovídající za funkcionalitu a chod aplikací. Měla by být zaměřena pouze na návrh struktur objektů, sloužící k

ukládání dat a manipulaci s nimi, k tomuto jsou používány návrhové vzory pro různé problematiky tvorby aplikací,

- **univerzálnost** – měla by být směrována k tomu, aby byla kompatibilní s modelem entitně-relačního přístupu, popř. aby se z objektové normalizace dala relační normalizace odvodit.

Pro pochopení datového modelu je nutné definovat, co je to datový objekt a jeho atributy. Je to takový objekt, který slouží pouze k ukládání a přeměně dat. Datový objekt by ve svých metodách neměl přímo obsahovat kód aplikace. Bohužel, toto bývá často porušováno, obzvláště pokud mají analytici zkušenosti s programováním. Datový objekt by měl mít uvedeny pouze své atributy, které slouží jako nosiče dat daného objektu (Merunka, 2008).

2.4.1 Objektové normální formy

Objektové normální formy (dále jen ONF) jsou postupy, které popisují správné použití dědění, vazeb mezi objekty a jejich skládání.

Dle Merunky (2008) jsou ONF definovány takto:

- ONF: „třída je v první objektové normální formě, jestliže její objekty neobsahují skupinu opakujících se atributů. Takové atributy je třeba vyčlenit do objektů nové třídy a skupinu opakujících se atributů nahradit jednou vazbou na kolekci objektů této nové třídy. Schéma je v 1. ONF, jestliže všechny třídy objektů v něm jsou v 1. ONF.“
- ONF: „Třída je v druhé objektové normální formě, jestliže její objekty neobsahují atribut nebo skupinu atributů, které by byly sdílené s nějakým jiným objektem. Sdílené atributy je třeba vyčlenit do objektu nové třídy a ve všech objektech, kde se vyskytovaly, nahradit vazbou na tento objekt nové třídy. Schéma je v 2. ONF, jestliže všechny třídy objektů v něm jsou v 2. ONF.“
- ONF: „třída je ve třetí objektové normální formě, jestliže její objekty neobsahují atribut nebo skupinu atributů, které mají samostatný význam nezávislý na objektu, ve kterém jsou obsaženy. Pokud takové atributy existují, je třeba je vyčlenit do objektu nové třídy, a v objektu, kde byly obsaženy, nahradit vazbou na tento objekt

nové třídy. Schéma je ve 3. ONF, jestliže všechny třídy objektů v něm jsou ve 3. ONF.“

2.5 Logistika

Distribuční logistika (z anglického slova „to distribute“ = šířit, rozdat, rozdělit, roznést, rozšířit, rozprostřít) z hlediska výrobního podniku představuje spojovací článek mezi výrobou a zákazníkem. Cílem distribuční logistiky je, dodání zboží ve správný čas na správné místo a ve správném množství i kvalitě, a to vše s ohledem na co nejoptimálnější poměr kvality služeb a jejich cenu (Malindžák, 2014).

Snaha o vytvoření společného trhu v rámci obchodu v Evropské unii, politické změny, rozvoj vědy a technologií, všechny tyto výše zmiňované věci vytvářejí nové nároky a požadavky na obchod a výrobu. Díky tomu jsou také neustále vytvářeny nové strategie a nové struktury, což má přímý dopad i na distribuční logistiku (Malindžák, 2014).

Aby mohly společnosti dodávat své zboží zákazníkům, musí být jejich nabídka velmi pružná vzhledem ke konkurenci. Pokud jsou totiž dva produkty stejné kvality a ceny, často si pak zákazníci vybírají podle rychlosti a kvality dodání. Společnosti v moderním prostředí se dostávají do jakéhosi, dříve trojúhelníku, v dnešní době spíše čtyřúhelníku, kde jako vrcholy figurují: snížení nákladů, zvýšení kvality a zvýšení pružnosti nabídky, v poslední době je ještě přidáváno „dělat věci jinak“, se zaměřením především na individuální vztah k zákazníkovi. Tyto faktory jsou ovlivněny úrovní technologií a techniky, úrovní pracovníků a podnikové organizace ve společnosti. V posledních letech je zřejmé, že se společnosti zaměřují právě na poslední dva vrcholy, a ty společnosti, které se nepřizpůsobují, jsou pomalu vytlačovány z trhu (Kubíčková, 2006).

Během samotného procesu přepravy, distribuce, výroby nebo zásobování není příliš účinné racionalizovat samotné dílčí články, ale je potřeba řídit celý proces centrálně, se zaměřením na jednotlivé odvětví, které jsou pak řetězově propojené mezi sebou. Takový přístup potom může být nazván logistickým. Toky surovin a materiálů jsou poté nazývány logistické řetězce. Obor, který se zabývá vytvářením a řízením těchto řetězců se nazývá logistika (Kubíčková, 2006).

2.5.1 **Historie logistiky**

První definice hospodářské logistiky byla zformulována v 60. letech v USA. Podle Národního výboru pro řízení distribuce v USA z roku 1964 zní definice logistiky takto: „Metoda řízení, zabývající se pohybem surovin od zdrojů k místu finální výroby a distribuce výrobků, a to z hlediska dopravy, zásobování, služeb spotřebitelům, skladování, manipulace, balení, ale i projektování výroby a rozmisťování kapacit“ (Malindžák, 2014).

Ze začátku se logistika v praxi ujala pouze jako nástroj podnikového řízení, který byl využíván pro operativního řízení a plánování na úseku distribuce v návaznosti na marketing. Následně bylo zjištěno, že distribuce prolíná základní funkce – zásobování, výrobu a distribuci. Tak se logistika stala zabezpečovací (obslužnou) funkcí podniku. Později, především u větších společností se vyčlenila do samostatného podnikového útvaru (Kubíčková, 2006).

2.5.2 **Logistické řízení**

Logistický management má za úkol vytvářet a řídit logistický systém. Logistický systém se dělí na ekonomické subjekty vytvářející hmotný tok, logistickou infrastrukturu a na logistické procesy a operace obsluhující hmotný tok (Kubíčková, 2006).

Logistický systém můžeme být představen jako množina uzlů a hran. Uzly představují pevná zařízení (např. továrny) a hrany – cesty pro fyzické toky a toky informací.

V podniku logistické řízení obsahuje plánování, organizaci, koordinaci a kontrolu logistických procesů a operací. Mezi hlavní logistické procesy patří nákup, výroba a prodej. Dále na podporu hlavních procesů existují další operace jako balení a doprava. K celému procesu je zapotřebí mít odpovídající informace, které zajišťuje logistický informační subsystém, který provádí například zpracování objednávek a řízení zásob. Logistický informační systém má za úkol přijmout a zpracovat objednávky a zajistit, aby došlo k včasné a bezproblémové dodávce. Tento tok informací je protisměrný k toku materiálnímu (Kubíčková, 2006).

Klíčový pojem pro logistický přístup k řízení hmotných toků je logistický řetězec, který lze vyjádřit jako posloupnost na sebe navazujících operací, které vedou k uspokojení požadavků zákazníka. Jednotlivé ekonomické subjekty mohou v logistickém řetězci vystupovat například jako dodavatelé a výrobci. Jeden subjekt může mít více funkcí (Kubíčková, 2006).

Logistické řízení má za úkol zabezpečit, aby hmotný tok byl co nejplynulejší a bez zbytečných přerušení. Zásoby jsou tradičním řešením různých problémů. Efektivní logistické řízení podniku se snaží docílit nejkratší doby uspokojení zákazníka a aby logistické náklady byly minimální. Při hodnocení efektivnosti nestačí jen kontrola nákladů, je zapotřebí hodnotit i dosažený výkon hodnocený pomocí několika ukazatelů (Kubíčková, 2006).

2.6 Vehicle routing problem

Je problémem kombinační optimalizace a celočíselného programování, který je zaměřený na řadu zákazníku s flotilu vozidel. Model byl navrhnut Dantzigem a Remserem v roce 1959 a od té doby hraje velkou roli v oborech přepravy, distribuce a logistiky (Paschos, 2013).

Vehicle routing problem může být popsán jako problém návrhu distribučních cest od jednoho nebo více dep k několika různě geograficky rozmístěným bodům, městům nebo zákazníkům, a to s co nejnižšími možnými náklady (Paschos, 2013).

VRP úloha se definuje na obecné dopravní síti, tj. S = (V,H), jako V je označena množina uzlů sítě a H značí množinu hran tyto uzly spojující. Uzlem V_0 bývá označováno středisko sítě a uzly $V_1,...,V_n$ označují místa odběru. Do míst odběru je dopravováno určité množství zboží, za pomoci vozidel, jejichž trasa má počátek a konec ve středisku V_0 a jejichž počet je omezen shora (Šubrt, 2011).

Úlohou je pak sestavení trasy nebo více tras vozidel tak, aby byly obslouženy všechny uzly s ohledem na co nejnižší náklady na přepravu (ať už z hlediska času nebo délky). Z takovéhoto zadání vyplývají dvě podmínky a to 1. každý zákazník má být obsloužen právě jednou a 2. nesmí být překročena kapacita obsluhujících vozidel (Šubrt, 2011).

K těmto základním podmínkám mohou být přidány ještě další globální (např. množství převáženého zboží, doba převozu), nebo lokální (dosažení určitého uzlu v určitý čas, limit spotřeby pohonných hmot) podmínky pro modifikaci dané úlohy, či trasy (Šubrt, 2011).

2.6.1 **Clarke-Wrightova metoda**

Mezi nejzákladnější heuristické metody pro řešení úloh VRP je Clark-Wrightova, zveřejněná roku 1964 jejími autory G. Clarkem a J.W. Wrightem.

Použití této metody vychází ze situace, kdy jsou v každém jejím opakování podle určitého kritéria vybírány dvě možné trasy V_0-V_i-V_0 a V_0-V_j-V_0, které jsou poté sdruženy do tzv. sdružené trasy V_0-V_i-V_j-V_0. Dvě trasy mohou být spojeny pouze, pokud jsou splněny dvě, výše uvedené, podmínky přípustnosti. Při použití této metody se dá jednoduše kontrolovat plnění ostatních, předem zadaných, globálních podmínek, např. počet navštívených uzlů nebo maximální délku trasy (Šubrt, 2011).

Zda je spojení dvou tras výhodné nebo není je určena úsporou, která vznikla jejich sdružením. Úspora je změřena výhodnostním koeficientem $z_{ij} = d_{0i} + d_{0j} – d_{ij}$, kde jako d_{0i}, d_{0j} a d_{ij} jsou označeny délky hran (V_0, V_i), (V_0, V_j) a (V_i, V_j). Výhodnostní koeficient vyjadřuje tedy rozdíl součtu obou tras a délkou sdružené trasy. Při každé iteraci postupu sdruží metoda právě ty dva uzly, u kterých výhodnostní koeficient z_{ij} dosahuje nejvyšších hodnot. Výhodou této metody je, že díky závislosti pouze na vzájemných vzdálenostech uzlů, se po spojení dvou zkoumaných uzlů koeficient z_{ij} již nemění. (Šubrt, 2011).

V praxi by se dala výše popsaná metoda provést takto:

1. Pro danou dopravní síť S = (V,H) je sestavena matice vzdáleností D = {d(i,j)}, kde i,j = 0,1,...,n; n = |V|. Obecně nemusí být síť S úplná, to znamená, že prvky matice D mohou vyjadřovat jak délky úseků, tak i vzdálenosti mezi jednotlivými uzly. Dále jsou zadány následující hodnoty:
   c.....průměrná rychlost pohybu vozidla v síti,
   t.....doba potřebná k výkladu,
   T.....maximální doba pohybu vozidla mimo V_0,
   K.....kapacita vozidla a

q_i.....množství zboží přepravovaných z uzlu V_0 do V_i

2. Je vytvořeno počáteční řešení, které představuje soubor elementárních tras ($V_0 – V_i – V_0$) pro všechny uzly sítě i = 1,…,n s uvedeným množstvím elementů a dobami přepravy:
   Pro trasu $V_0 – V_n – V_0$ je použit vzorec pro dobu přepravy ((2d_0n)/c)+q_n t
3. Z matice D je odvozena matice výhodnostních koeficientů $Z = \{z_{ij}\}$, kde i,j = 1,...,n podle vztahu $z_{ij} = d_{0i} + d_{0j} – d_{ij}$, kde z_{ij} vyjadřuje rozdíl mezi součtem tras ($V_0 – V_i – V_0$) a ($V_0 – V_j – V_0$) a délkou sdružené trasy ($V_0 – V_i – V_j – V_0$).
4. V matici Z je nalezen největší kladný prvek z_{ij} a je-li to možné, jsou sdruženy trasy ($V_0 – V_i – V_0$) a ($V_0 – V_j – V_0$) do trasy ($V_0 – V_i – V_j – V_0$). Pokud takový prvek neexistuje, postup je ukončen. Aktuální množina okružních tras je výsledkem algoritmu. V opačném případě se přejde na krok 5).
5. Je zkontrolováno, zda sdružením tras ($V_0 – V_i – V_0$) a ($V_0 – V_j – V_0$) vznikne přípustná trasa. Pokud přípustná trasa nevznikne, je z_{ij} položeno 0 a postoupeno na krok 4). V opačném případě je pokračováno krokem 6).
6. Množina uzlů V je aktualizována vyjmutím uzlů i a j, dokud sdružením tras přestanou být krajními uzly trasy. Konstanta z_{ij} je položeno 0. Množina tras je aktualizována vyjmutím sdružených tras a vložením nové trasy. Současně jsou také aktualizovány ostatní sledované parametry (doba přepravy, množství elementů, délka trasy aj.). Není-li krok 4 a 5 možný, je nalezen nejbližší menší nebo stejně velký prvek z_{st} a jsou sdruženy trasy obsahující uzly V_s a V_t; mohou to být elementární trasy nebo trasy, vzniklé předchozím sdružováním. Pro krajní uzly V_s a V_t nově vzniklé trasy je položeno $z_{st} = 0$ a přesunuto na krok 4).
7. Postup je opakován, dokud není matice Z vyčerpána, nebo dokud není zřejmé, že všechny kapacity vozidel jsou vyčerpány a další řešení nemá smysl. Výsledné řešení nemusí být optimální, často bude jen suboptimální

(Šubrt, 2011).

2.6.2 **Okružní dopravní problémy**

V praxi je možné se s okružními problémy setkat relativně často. A to v případech, kdy je potřeba rozvézt zboží od několika málo dodavatelů k většímu počtu spotřebitelů, nebo, na druhou stranu, od většího počtu dodavatelů k malému množství spotřebitelů. Za pomoci implementace okružních tras dodavatel šetří své náklady, než když by měl vyjíždět ke každému spotřebiteli zvlášť (Bazaraa, Jarvis a Sherali, 2010).

Okružních úloh existuje mnoho typů. Základním a zároveň nejjednodušším typem je jednookruhový okružní dopravní problém, u kterého probíhá přeprava jen jedním okruhem, tzn. že je potřeba obsáhnout všechny spotřebitele pouze jednou vyjížďkou. Dalším problémem je více okruhový okružní dopravní problém. Většinou jde o problém, u kterého jsou časová nebo jiná omezení, díky kterým není problém realizovatelný s použitím pouze jediného okruhu. Dále je možné okružní problémy rozdělit např. na problémy s úplnou nebo neúplnou sítí cest, kdy ve druhém případě, během cesty, mezi některými dvojicemi, neexistuje přímé spojení (Bazaraa, Jarvis a Sherali, 2010).

2.6.2.1 Jednookruhový okružní dopravní problém

Jednookruhový dopravní problém, nebo také problém obchodního cestujícího je nejjednodušším typem okružních dopravních problémů. Obecně je tento problém formulován jako množina n uzlů a sazba c_{ij}, která představuje např. vzdálenost, čas, náklady apod., pro každou dvojici uzlů. Cílem úlohy je vytvořit trasu, která bude obsahovat všechny uzly právě jednou, kromě počátečního uzlu, který se znovu objeví na konci a aby součet sazeb c_{ij} byl minimální (Šubrt, 2011).

2.6.3 **Metoda nejbližšího souseda**

VRP (viz str. 12) je velice složitý problém, který stále oslovuje velký počet odborníků. VRP sestává ze základního modelu TSP (travelling salesman problem), kde jde o to, aby obchodní cestující zvládl zabezpečit všechny své zákazníky co nejkratší možnou cestou a žádným místem neprošel dvakrát. I když jsou VRP a TSP velice podobné, z

praktického hlediska jsou však VRP o stejné velikosti jako TSP mnohem složitější k řešení. Většina algoritmů pro VRP je heuristická, což samo o sobě napovídá větší složitosti problému (Gutin a Punnen, 2002).

Cesta je vlastně graf, kde hranami grafu jsou cesty-vzdálenosti a vrcholy a uzly onoho grafu jsou pak distribučními místy. Úkolem je najít cestu k požadovaným uzlům tak, aby byla co nejkratší a nejefektivnější (Gutin a Punnen, 2002).

Technik pro výpočet VRP je několik, většina z nich je heuristická a meta-heuristická, protože, díky tomu, že tato problematika je NP-úplná, je téměř nemožné najít algoritmus pro úlohy s větším počtem uzlů tak, aby takováto úloha byla spočítatelná v rozumném časovém intervalu (Gutin a Punnen, 2002).

Je používáno několik přístupů:

- **exaktní metody** – jak název napovídá, tento přístup počítá veškeré možné řešení, ze kterého potom vybere to nejlepší. Patří mezi ně metoda větví a mezí, a větví a řezů,
- **heuristické** – hledají řešení v poměrně limitovaném okruhu vyhledávaného spektra a většinou jsou poměrně účinné se slušnými výpočetními časy,
- **konstruktivní metody** – postupně buduje proveditelné řešení s kontrolou nákladů. Používané metody jsou např. multi-route improvement heuristics, matching based savings algorithm, savings algorithm,
- **2-fázové algoritmy** – algoritmy složené ze dvou kroků. Prvním je shlukování vrcholů do možných řešení, druhým je aktuální tvorba cest. Používané algoritmy: Fisher and Jaikumar, The Petal Agorithm, The Sweep Algorithm, Taillard a
- **metaheuristické** – Ant algorithm, Contraint Programming, Deterministic Annealing, Genetic Algorithms, Simulated Annealing, Tabu Search.

(http://neo.lcc.uma.es/vrp/solution-methods/)

3 Vlastní práce

V této části je popsáno vývojové prostředí a způsob vývoje aplikací pro systém Microsoft Windows. Dále je popsána programová a grafická část vyvíjené aplikace.

3.1 Vývojové prostředí pro Microsoft Windows aplikace

V následující části jsou popsány koncepty potřebné pro tvorbu aplikací pro operační systém Microsoft Windows.

3.1.1 .NET Framework

Pro vývoj aplikací pro operační systém Microsoft Windows je nutné používat vývojovou sadu .NET Framework. Tato vývojová sada je součástí operačního systému Windows od jeho verze 6.2 (Windows 8). Pro starší verze operačního systému musí být .NET Framework nainstalován samostatně. Stejně tak existuje možnost nainstalovat starší verzi rámce.

.NET Framework sestává ze dvou hlavních komponent:

- **Common Language Runtime (CLR)** – run-time engine pro .NET obsahující JIT (just-in-time) kompilátor, který překládá instrukce Common Intermediate Language CIL do procesorem srozumitelného strojového kódu, garbage collector, typové verifikace, code access security a další. Je implementován jako COM server v procesu (dll) a používá různé funkce poskytované Windows API,
- **.NET Framework Class Library (FCL)** - rozsáhlá kolekce typů jako služby uživatelského rozhraní, síťové nástroje, přistup do databází a mnohé další typy, typicky používané klientem a serverovými aplikacemi.

.NET Framework společně s dalšími nástroji a vyššími programovacími jazyky (jako např. C#, Visual Basic, F#) byly vytvořeny za účelem zjednodušení práce vývojářů Windows aplikací a zvýšení jejich produktivity, tím pádem také zlepšení kvality, spolehlivosti a bezpečnosti aplikací.

3.1.2 **IDE – Microsoft Visual Studio**

IDE neboli Integrated development environment = integrované vývojové prostředí je velmi důležitým nástrojem pro vývoj aplikací. Správnou volbou prostředí může být značně usnadněn, nebo naopak ztížen vývoj aplikace.

Vývojové prostředí je softwarová aplikace, poskytující podporu při vývoji různých aplikací. Různá vývojová prostředí existují pro řady programovacích jazyků a frameworků. Většina moderních IDE poskytuje inteligentní doplnění kódu.

Autor si zvolil pro vývoj prostředí Microsoft Visual Studio, nástroj vyvíjen přímo společností Microsoft, díky tomu jsou v něm vždy dostupné nejnovější verze a vylepšení vývojových rámců a nástrojů pro jazyky jako C#, Visual Basic, F#, které také vyvíjí společnost Microsoft.

Velmi důležitou součástí IDE je také debugger. Tento nástroj je vývojáři používán pro hledání chyb v kódu. Jednou z nejdůležitějších funkcí debuggeru je možnost zastavit běh programu. Kde bude program zastaven si definuje programátor takzvaným breakpointem, nebo jednoduše zmáčkne tlačítko zastavit během běhu programu. Výhodou debuggeru ve Visual Studiu je analýza výkonu, kde je debugger schopný vykreslit do grafu využití paměti a procesoru při běhu aplikace. Díky tomu je jednoduché nalézt části kódu potřebující optimalizaci, nebo např. zda někde neuniká paměť.

3.1.2.1 Visual Studio Team Services

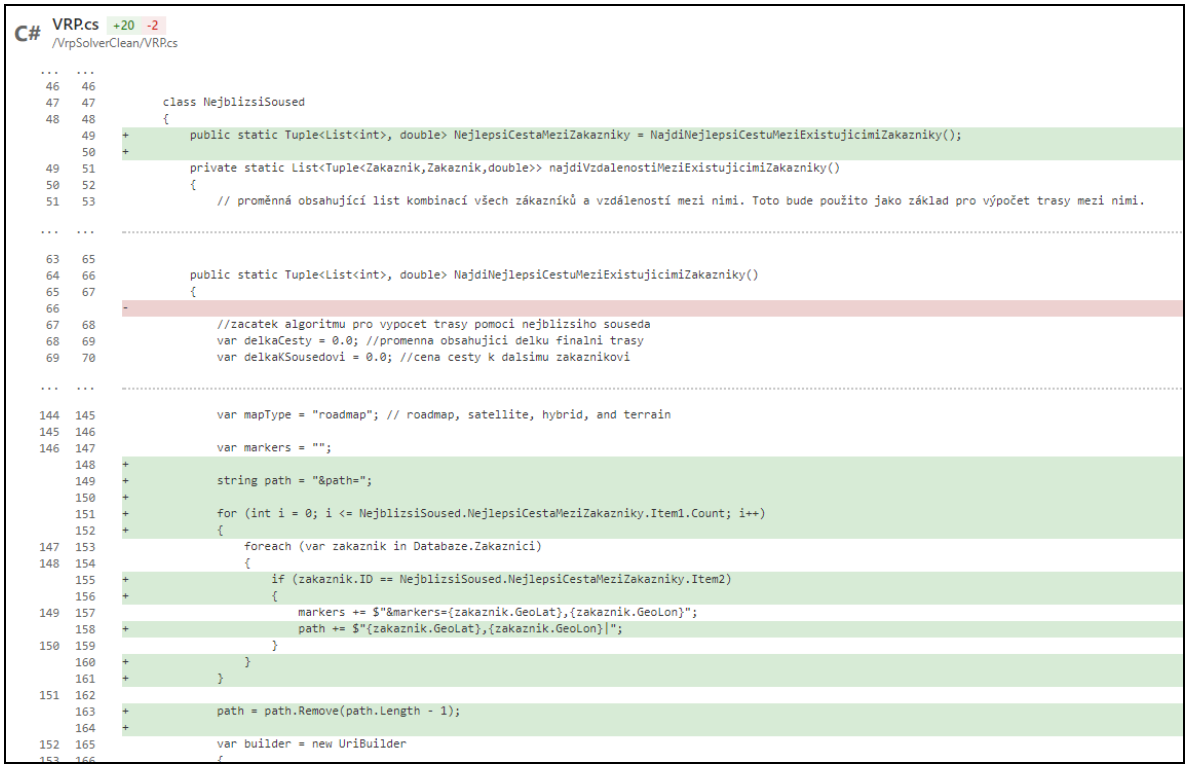
Další výhodou použití Microsoft Visual Studia je možnost využití již integrovaného nástroje pro verzování. Existuje řada verzovacích nástrojů jako Git, SVK, Mercurial a další, Visual Studio používá proprietární verzovací systém společnosti Microsoft – Visual Studio Team Services. Díky integraci s Visual Studiem není nutnost instalovat aplikace třetích stran pro komunikaci se verzovacím serverem.

Version control nástroje jsou používány při práci více lidí na více zařízeních a pro verzování kódu. Pro nahrání na server se používá kombinace metod commit a push. Commit zkontroluje a zaznamená všechny změny mezi serverem a klientem, push odešle změny na server. Většinou je vývojářem zanechán komentář o provedených změnách.

Výhodou tohoto přístupu k vývoji aplikací je možnost návratu ke kterékoliv starší verzi kódu, v případě nečekané chyby.

Na Obrázku 1 lze vidět, jak jsou zobrazeny změny u jednotlivých commitů ve VSTS. Červené řádky začínající mínusem byly oproti předchozí verzi odebrány, naopak zelené řádky začínající plusem byly přidány.

Obrázek 1: Version control ve VSTS



Zdroj: vlastní zpracování

VSTS má také webovou verzi, ve které lze zobrazit počet změn podle uživatelů – přísun pro projekt. Dále obsahuje nástroj pro sledování issues, chyb apod. Tato služba může být velice užitečná pro vývojářské týmy používající takzvaný Agile přístup, kde je celý projekt rozplánován do částí a poté jsou každému členu přiřazeny určité úkoly na dva týdny dopředu. Díky tomu, že je tento nástroj integrovaný přímo ve VSTS, pak tyto týmy nemusí používat nástroje třetích stran, což opět usnadňuje a zrychluje vývoj.

V případě této bakalářské práce byl VSTS použit pro synchronizaci kódu mezi různými zařízeními.

3.2 Konvence při psaní kódu

Při psaní kódu je důležité dodržovat určité základní konvence. Ve firmách to mohou být interní konvence pro programátory, pro případy, že by se projekt dostal do rukou někoho jiného, aby mohl v kódování pokračovat bez nutnosti učit se programovací styl někoho jiného.

Jak již bylo řečeno tyto konvence mohou být vytvářeny nebo upravovány interně ve společnostech, existují však programovací jazyky, které mají určité základní konvence definovány globálně někde na internetu.

- Kódové konvence by měly sloužit k následujícímu:
- Vytvářejí konzistentní náhled na kód, čtenář je tedy schopný se soustředit na obsah, spíše než na strukturu,
- Umožnují čtenáři porozumět kódu rychleji tím, že používají již známé konvence,
- Umožňují kopírování, změny a údržbu kódu,
- Předvádí osvědčené postupy pro psaní kódu.

Pro programovací jazyk C# byly určité konvence vytvořeny přímo společností Microsoft, podle nichž tyto konvence nedefinují standart pro programování, spíše popisují interní standart společnosti, který byl použit při psaní ukázek a dokumentace.

3.3 Základní pojmy

V této kapitole budou pospány základní pojmy programovacího jazyka C# a ostatní pojmy spojené s tvorbou tohoto projektu. Znalost vysvětlených pojmů je nezbytná pro úspěšné a plné pochopení tvorby této práce.

3.3.1 Modifikátory

Modifikátory jsou důležitou součástí OOP. Podporují koncept zapouzdření, který podporuje myšlenku skrývání funkcí. Přístupové modifikátory umožnují definovat kdo má nebo nemá přístup k určitým částem kódu.

- Public (veřejný) - tento přístupový modifikátor je možné použít pro typy a jejich členy. Public je nejméně omezující modifikátor. Pro přistup k veřejným objektům není žádné omezení,
- Private (soukromý) - nejvíce restriktivní přístupový modifikátor. Soukromé objekty jsou přístupné pouze v objektu, ve kterém jsou definované. Pokud není žádný modifikátor uveden, je automaticky použit tento.

V jazyce C# existuje víc modifikátoru než pouze veřejný a soukromý, v této práci se však objevují pouze tyto dva.

3.3.2 **Typy**

Jazyk C# je staticky typovaný jazyk. To znamená, že u každé proměnné musí být deklarován její datový typ. Výhodou tohoto přístupu je, že při kompilaci proběhne u všech proměnných kontrola, zda obsah proměnné odpovídá deklarovanému datovému typu proměnné, čímž je možné předejít pozdějším chybám při běhu aplikace.

Jedny ze základních typů datových proměnných jsou např.

- celá čísla: int,
- desetinná čísla: float,
- textový řetězec: string.

3.3.3 **Forma**

Do forem je možné vložit ovládací prvky, čímž vzniká grafické rozhraní aplikace. Aby byla aplikace schopná něco dělat, je potřeba později ke grafickým ovládacím prvkům napsat programový kód, který řekne, co se má po interakci s ovládacím prvkem stát.

K tomuto účelu existují metody - takzvané *listenery*, které čekají, například, na kliknutí na tlačítko (*názevTlačítka_Click())* a poté provedou to, co je v dané metodě uvedeno.

Tyto metody si může programátor napsat sám, nebo například v případě tlačítka, může dvakrát poklepat myší na tlačítko v grafickém návrhu formy a Visual Studio

automaticky vytvoří *listener* pro kliknutí na tlačítko, ve kterém může programátor následně definovat požadovanou akci.

V této bakalářské práci je možné vidět několik takovýchto *listenerů*, převážně právě na objektech tlačítek.

Obrázek 2: Listener tlačítka Spocitej

```
private void ButtonSpocitej_Click(object sender, EventArgs e)
{
    // Zobrazi chybovou hlasku, pokud nemame alespon dva zakazniky. Jinak pokracuje dale.
    if (Databaze.Zakaznici.Count < 2)
    {
        MessageBox.Show(
                    "Pridejte zakazniky pro pokracovani",
                    "Chyba",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Warning);
    }
    else
    {
        Form f = new FormZobrazeni();
        f.Show();
    }
}
```

Zdroj: vlastní zpracování

Na Obrázku 2 je vidět *listener* pro kliknutí na tlačítko pro výpočet ideální cesty. Pro vytvoření cesty je potřeba mít alespoň dva zákazníky, tím pádem bylo nutné definovat kontrolu pro počet zákazníků. Když je zákazníků méně než 2, zobrazí aplikace chybové hlášení o nedostatečném počtu zákazníků. Jakmile je však zaznamenáno kliknutí při počtu zákazníků větším než 1 - je zavolána nová forma *Zobrazení* a následovně je zobrazena.

3.3.4 Třída

Třída je základním konstruktorem umožňujícím vytváření vlastních objektů tím, že definuje a seskupuje různé proměnné, metody a eventy. Třída je jako plán, který definuje data a chování objektu. Třídy podporují dědění a základní charakteristiky OOP.

3.3.4.1 Deklarace třídy

Třída je deklarována slovem „*class*“, před kterým je většinou uveden přístupový modifikátor (viz výše). V těle třídy jsou pak definována data a chování objektů. Pole – fields, vlastnosti – properties, metody a eventy ve třídě jsou nazývány *členy třídy*.

3.3.4.2 Tvorba objektů

Třída a objekt jsou rozdílné pojmy. Třídou je definován typ objektu, není to však sám objekt. Objekt je konkrétní entita založená právě na třídě. Někdy jsou objekty také nazývány instancemi třídy.

Instance třídy je vytvořena použitím slova *new*, jak je možné vidět na Obrázku 3. V metodě *NovyZakaznik* lze vidět zavolání *new Zakaznik*, takto je vytvořena nová instance třídy *zákazník*, které jsou předány proměnné v závorkách.

Obrázek 3: Vytvoření instance třídy

```
public static void NovyZakaznik(string ulice, string cp, string mesto)
{
    Zakaznici.Add(new Zakaznik(ulice, mesto, cp, pocetZakazniku++));
}
```

Zdroj: vlastní zpracování

3.3.5 **API**

Application programming interface neboli Programovatelné aplikační rozhraní je souborem nástrojů a metod pro komunikaci mezi různými součásti aplikací, nebo aplikacemi samotnými. Existují různé druhy API, ať už pro webové aplikace nebo třeba operační systémy. Účelem tohoto rozhraní je vytvořit pro programátora možnost komunikovat s pro něj neznámým kódem aplikace.

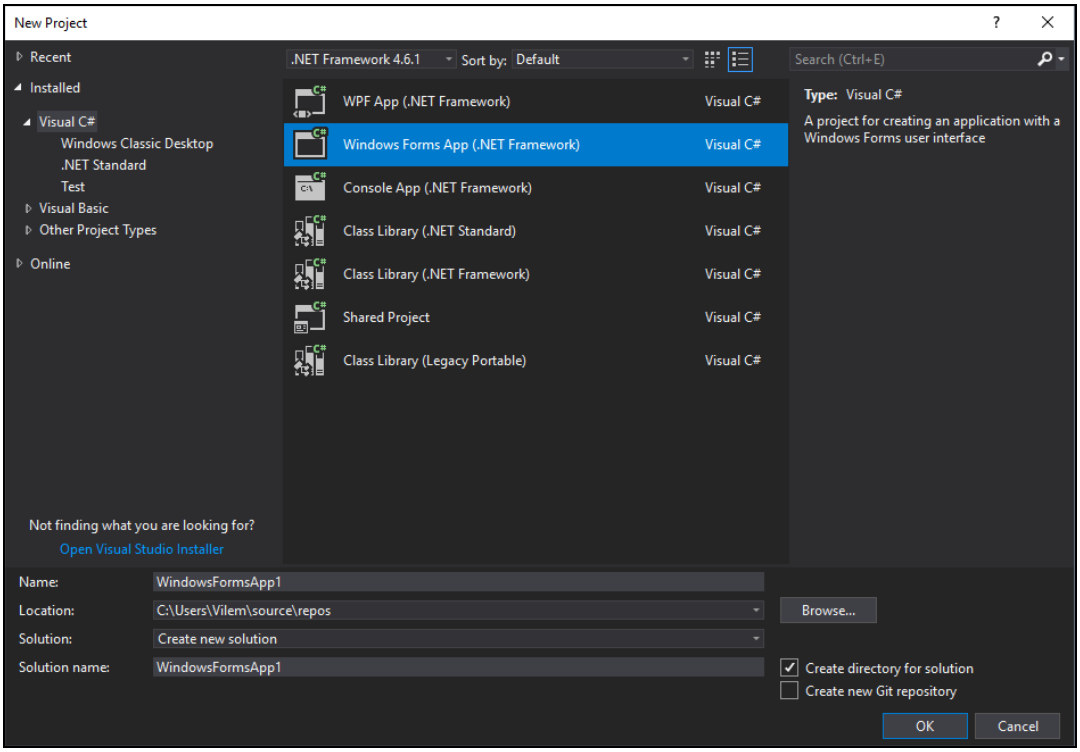
3.3.6 JSON

Neboli JavaScript Object Notation je Celosvětově používaný datový formát pro asynchronní komiunikaci mezi serverem a klientem. JSON byl vytvořen na základě jazyku JavaScript, není však na něm závislý, proto je možné ho použít v jakémkoliv jiném programovacím jazyce.

3.4 Vytvoření projektu ve Visual Studiu

Pro vytvoření nového projektu je nutné zvolit File – New – Project. V zobrazeném okně si uživatel zvolí, v jakém programovacím jazyce bude projekt napsán a následně typ aplikace. Autor zvolil jazyk Visual C# a typ aplikace Windows Forms App (.NET Framework). Dále je nutné zvolit jméno a umístění projektu na disku (Obrázek 4).

Obrázek 4: Vytvoření projektu

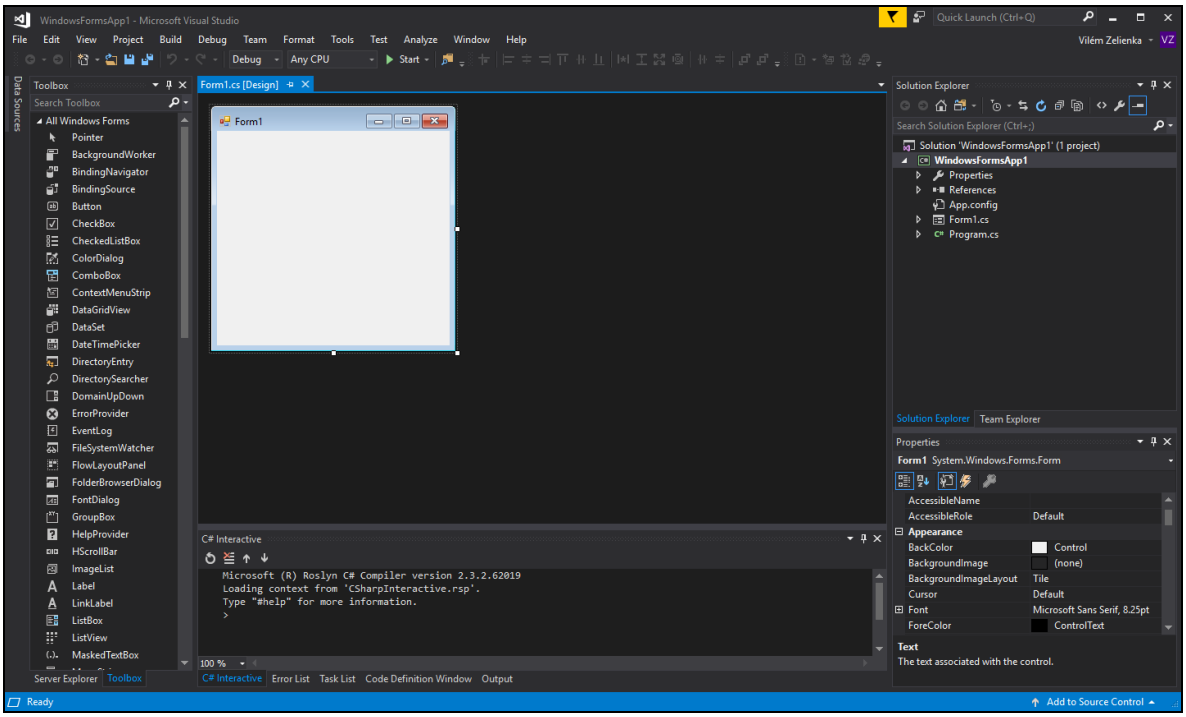


Zdroj: vlastní zpracování

Následně je Visual Studiem automaticky vytvořen funkční, ale velice základní program, který po spuštění otevře prázdné okno.

Důležitým prvkem pro tvorbu grafického rozhraní je panel Toolbox, který je možné vidět na Obrázku 5 po levé straně. V tomto menu je seznam všech možných ovládacích prvků pro aplikace typu Windows Forms. Tyto ovládací prvky jsou použity pro interakci koncového uživatele s aplikací, ať už k zadání dat, nebo jejich zobrazování.

Obrázek 5: Nově vytvořený projekt



Zdroj: vlastní zpracování

Nejpoužívanějšími prvky v této aplikaci jsou například:

- **Form** – dialogové okno nebo okno, které je hlavní součástí aplikačního uživatelského prostředí. Je možné ho použít k vytvoření modálních oken jako například dialogového okna. Je možné nastavit vzhled, velikost, barvu a další proměnné,
- **Button** – na button (česky tlačítko) je možné kliknout myší, klávesou ENTER nebo MEZERNÍK, pokud je systém zaměřen na zvolený ovládací

prvek. Je možné zvolit vzhled tlačítka. Po kliknutí uživatelem je vyvolána událost,

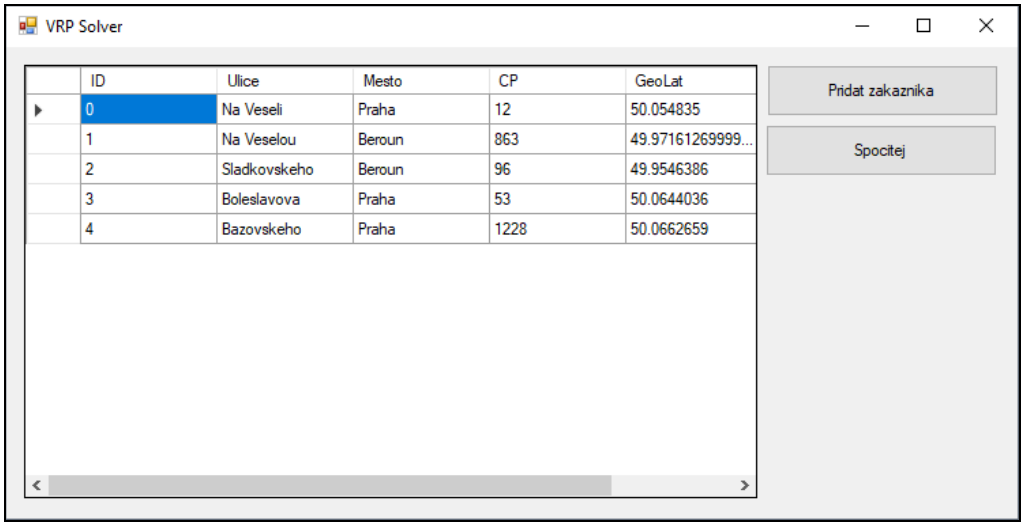
- **Label** – typicky používán pro popisné texty ostatních ovládacích prvků. Např. je možné ho použít pro přidání popisu TextBoxu (viz níže), aby informoval uživatele o jeho účelu. Dalším možným využitím je popis nebo nápověda k celé Formě, opět pro zjednodušení práce uživatele,
- **TextBox** – používán pro zobrazení dat ze *záznamového* zdroje nebo pro zobrazení výsledku výpočtů. Dále poskytuje uživateli možnost zadat text, víceřádkové úpravy a možnost skrytí znaků hesla,
- **PictureBox** – typicky používán pro zobrazení grafiky ve formě bitmap, metafile, ikony, JPEG, GIF a PNG. Obrázek je automaticky oříznut, pokud není ovládací prvek dostatečně velký.

3.5 Grafické rozhraní aplikace

Grafické rozhraní je tvořeno třemi formami.

Form1 (Obrázek 6), který je inicializován při startu aplikace obsahuje seznam zákazníků, tlačítko pro přidání zákazníka a tlačítko pro zahájení výpočtu trasy.

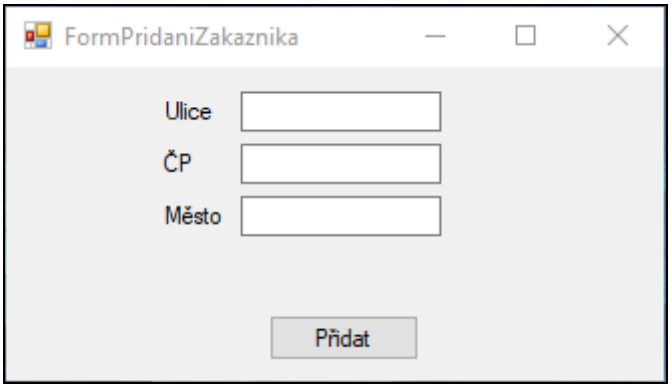
Obrázek 6: Form1



Zdroj: vlastní zpracování

FormPridaniZakaznika (Obrázek 7) – tato forma je zobrazena po kliknutí na tlačítko „*Pridat zakaznika*“. Obsahuje tři *TextBoxy*, do kterých by měl uživatel vyplnit základní údaje o zákazníkovi. Zákazník je přidán do databáze kliknutím na tlačítko „*Přidat*“.

Obrázek 7: FormPridaniZakaznika



Zdroj: vlastní zpracování

Na tlačítku „*Přidat*“ je implementována kontrola vyplnění všech polí (Obrázek 8). Je to funkce, která je složená z několika logických podmínek. Funkce kontroluje každý objekt typu *Control* v aktuální formě, pokud se jedná o *TextBox*, který má prázdnou nebo žádnou hodnotu – je uživateli zobrazen *MessageBox* typu *Warning*, který žádá o vyplnění všech polí. Poté je funkce zastavena a spustí se znovu až po opětovném kliknutí na tlačítko „*Přidat*“. Pokud jsou všechna pole vyplněna, kontrola je vyhodnocena úspěšně a nová instance třídy zákazník je přidána do *Databáze*, forma je zavřena.

Obrázek 8: Listener tlačítka Pridat

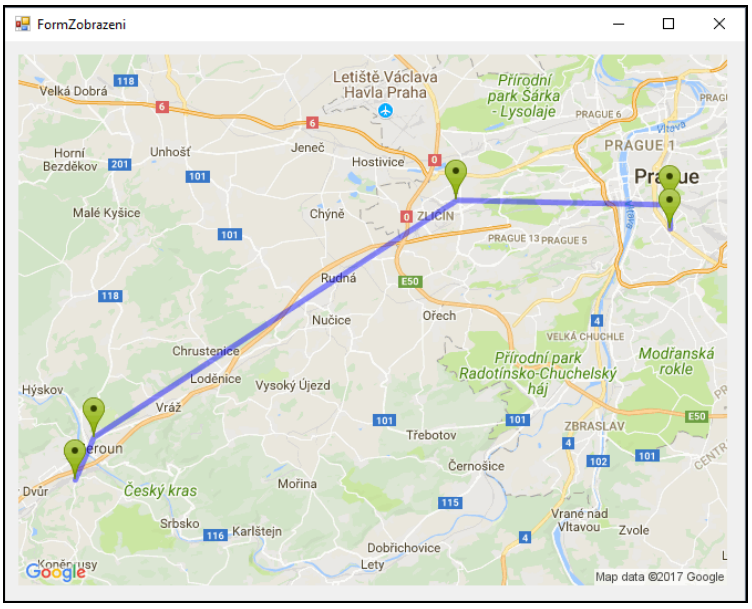
```
private void PridatButton_Click(object sender, EventArgs e)
{
    // Pomocna promenna pro zjisteni, jestli nam program nahlasil chybu nebo ne
    bool error = false;
    // Skupina logickych pravidel, kontrolujici vsechny textboxy v aktualni forme, jestli obsahuji nejake data. Kdyz ne, nahlasi chybu.
    foreach (Control child in this.Controls)
    {
        if (child is TextBox textBox)
        {
            if (string.IsNullOrWhiteSpace(textBox.Text))
            {
                MessageBox.Show(
                    "Vyplnte prosim vsechna pole",
                    "Chyba",
                    MessageBoxButtons.OK,
                    MessageBoxIcon.Warning);
                error = true;
                break;
            }
        }
    }

    // Pokud vse probehlo bez chyby - vse je vyplneno, je vytvoren novy zakaznik s daty z textboxu, pote je aktualni okno zavreno.
    if (!error)
    {
        Databaze.NovyZakaznik(tbUlice.Text, tbCP.Text, tbMesto.Text);
        this.Close();
    }
}
```

Zdroj: vlastní zpracování

FormZobrazeni (Obrázek 9) – velice jednoduchá forma, která obsahuje pouze *PictureBox*, do kterého je, po kliknutí na tlačítko „*Spocitej*“ z *Form1*, vykreslena mapa se všemi zákazníky a ideální trasou mezi nimi.

Obrázek 9: FormZobrazeni



Zdroj: vlastní zpracování

3.6 Programová část

Tato kapitola je zaměřena na popis částí kódu, které mají na starost funkce aplikace.

3.6.1 Třída Zákazník

Pro uložení informací o jednotlivých místech, která jsou potřeba navštívit, byla vytvořena třída *Zákazník*. V této třídě jsou uloženy základní informace potřebné pro výpočet souřadnic a nalezení ideální trasy.

Tato třída definuje zákazníka jako objekt, obsahující následující vlastnosti (properties):

- ***ID*** – veřejné *číslo* začínající nulou, které je vždy navýšeno o 1 oproti předchozímu zákazníkovi, automaticky přiřazené při vytvoření každého zákazníka,
- ***Ulice*** – veřejný *string* obsahující název ulice,
- ***Mesto*** – veřejný *string* obsahující název města,
- ***CP*** – veřejný *string* obsahující číslo popisné,
- ***GeoLat*** – veřejný *double* obsahující zeměpisnou šířku,
- ***GeoLon*** – veřejný *double* obsahující zeměpisnou délku.

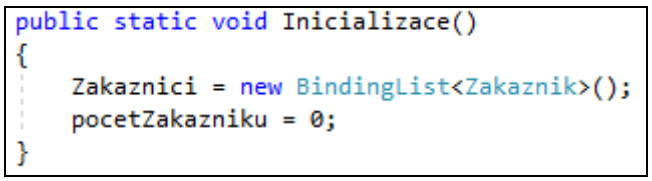
Dále obsahuje veřejný konstruktor, kterým je možné vytvořit *Zákazníka* kdekoliv v kódu. Na obrázku č.3 je možné vidět, že pro vytvoření zákazníka jsou potřeba tři věci – *ulice, město* a *ID*. Z těchto proměnných jsou poté zjištěny souřadnice a uloženy v globálních proměnných *GeoLat* a *GeoLon*, aby bylo možné je použít jinde v kódu.

3.6.2 Třída Databaze

Třída databáze slouží k přidávání a udržování seznamu zákazníků. Základní součástí této třídy je *BindingList* objektů *Zákazník*, ve kterém je uložen seznam všech zákazníku, jako objektů. Ve třídě je obsažena metoda *Inicializace* (Obrázek 10), která zavadí třídu *Databaze*. Třída musí být zavedena ihned po spuštění programu, protože je v ní vytvořen

nový *BindingList*, obsahující seznam všech zákazníků, jako objektů. Následovně je při zavedení vynulována proměnná, ve které je uložen aktuální počet zákazníků v databázi – *pocetZakazniku*.

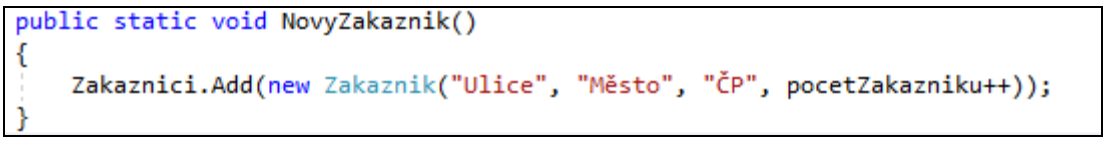
Obrázek 10: Metoda pro zavedení třídy Databaze



Zdroj: vlastní zpracování

Třída *Databaze* dále obsahuje přetíženou metodu *NovyZakaznik*. V jednom případě je možné metodu zavolat bez parametru, tím je do *BindingListu* přidána nová instance třídy *Zakaznik*, kde jsou místo aktuálních informací o přesném místě uvedeny výchozí hodnoty, jak lze vidět na Obrázku 11.

Obrázek 11: Přidání výchozího zákazníka do Databaze



Zdroj: vlastní zpracování

Díky přetížení je možné použit stejný název metody, která však očekává tři parametry – *ulice, cp, mesto*. Po zavolání této přetížené metody je do *BindingListu* přidána nová instance třídy zákazník s informacemi, které byly metodě při zavolání předány jako parametry – tímto je vytvořen konkrétní zákazník.

U obou metod *NovyZakaznik* je předána třídě *Zakaznik* proměnná *pocetZakazniku++*, to znamená, že nově vytvořený zákazník bude mít *ID* o jedna vyšší než poslední počet zákazníků.

3.6.3 **Soubor *VRP***

V souboru *VRP* je obsaženo několik tříd, které se všechny podílejí na výpočtu a zobrazení trasy.

3.6.3.1 Třída *Google*

Třída *Google* používá API od společnosti Google. Obsahuje několik metod a proměnných.

Všechny proměnné jsou definovány jako soukromé a statické, aby bylo možné je použít v ostatních metodách této třídy. Proměnné jsou celkem čtyři:

- ***googleApiKey*** – API klíč od Google, lze použít pouze pro tento projekt. V případě další distribuce program by bylo nutné zakoupit nový API klíč, protože na tomto klíči je omezený počet žádostí za dne,
- ***outputFormat*** – formát, v jakém Google vrací výsledky na dotazy. V případě této práce byl použit formát JSON, další možností je formát XML,
- ***scheme*** – schéma protokolu HTML – v případě této práce je použita zabezpečená verze HTTPS,
- ***host*** – host, na který jsou posílány dotazy. Protože se jedná o dotazy na mapy z Google API je hodnota této proměnné nastavena na "maps.googleapis.com".

Všechny výše zmíněné proměnné jsou použity pro tvorbu URL, která je posílána do Google API. Protože je ve třídě více metod, které Google API používají a tyto parametry jsou pro všechny metody společné, tím, že jsou definovány přímo v těle třídy, mizí nutnost opětovné definice stejných parametrů v každé metodě.

Ve většině metod z této třídy je použit *UriBuilder* a *Serializer*. Tyto dva objekty jsou použity pro sestavení URL, která je odesílaná na Google API. Jelikož API vrací hodnoty ve formátu JSON, je zde nutnost použít *serializer*, který JSON hodnoty zpracuje a rozdělí do proměnných.

3.6.3.2 Metoda ZjistiSouradnice

V této metodě je použita Google Maps Geocoding API. Tato metoda přijímá parametry ulice, číslo popisné a město – po dokončení vrátí zeměpisné souřadnice jako *Dictionary* (slovník) s klíčovou hodnotou *string* a hodnotou *double*.

3.6.3.3 Metoda SpocitejVzdalenost

V této metodě je použita Google Maps Distance Matrix API. Parametry metody jsou souřadnice dvou míst. Po zavolání metody následuje pokus o dotaz ke Google API pro výpočet vzdálenosti. Pokud je spojení úspěšné, Google API vrátí vzdálenost mezi místy, která byla předána metodě ve formátu zeměpisných souřadnic jako parametry metody.

3.6.3.4 Metoda GetMapUri

V této metodě je použita Google Static Maps API. Tato metoda vrátí *string* s URL ke k obrázku mapy, na které je vykreslena ideálni trasa mezi zákazníky. Nemá žádný vstup, pracuje přímo s daty ze třídy *NejblizsiSoused*. API je schopná přijmout dotazy s různými parametry. V tomto případě jsou definovány následující parametry:

- ***markers*** – tento parametr definuje, jak budou na mapě označeny zadané body. V případě této práce byl zvolen výchozí typ se zelenou barvou. Tento parametr je volitelný,
- ***size*** – v parametru size je definováno rozlišení mapy. V případě této práce bylo zvoleno rozlišení 640x480 pixelů. Takto velká mapa by měla být čitelná, zároveň by však měla být dostatečně kompaktní, pro rychlejší stažení z internetu,
- ***type*** – v parametru type je definován typ mapy. Možnosti jsou *roadmap, satellite, hybrid* a *terrain*. Změna tohoto parametru by měla za následek pouze kosmetickou změnu mapy,

- ***path*** – v případě této práce je nejdůležitějším parametrem *path*, ve kterém je definován seznam zeměpisných souřadnic cesty, která má být vykreslena do mapy. Jednotlivé zeměpisné souřadnice jsou od sebe odděleny znakem „|“.

Hlavní částí metody *GetMapUri* je část, která získává seznam zeměpisných souřadnic pro vykreslení cesty (Obrázek 12). V prvních *foreach* cyklu jsou načteni všichni zákazníci z předem vypočtené ideální trasy, v druhém *foreach* cyklu jsou pak srovnáni se seznamem existujících zákazníků, kde, pokud je ID zákazníka stejné jako zákazník z ideální cesty – jsou jeho zeměpisné souřadnice uloženy do proměnných *path* a *markers*, následně je druhý *foreach* cyklus přerušen a pokračuje první cyklus.

Obrázek 12: Tvorba URL pro získaní mapy

```
var markers = "markers=color:green";

string path = "path=color:blue";

foreach (var cesta in NejblizsiSoused.NejlepsiCestaMeziZakazniky.Item1)
{
    foreach (var zakaznik in Databaze.Zakaznici)
    {
        if (zakaznik.ID == cesta)
        {
            path += $"|{zakaznik.GeoLat},{zakaznik.GeoLon}";
            markers += $"|{zakaznik.GeoLat},{zakaznik.GeoLon}";
            break;
        }
    }
}
```

Zdroj: vlastní zpracování

Takto je zkontrolován celý seznam obsahující ideální trasu a postupně sestavena URL, obsahující zeměpisné souřadnice ideální trasy. Následně je URL odeslána na Google API, která pošle zpět URL adresu obsahující obrázek s výslednou mapou.

3.6.4 **Třída NejblizsiSoused**

V této třídě jsou sjednoceny všechny potřebné informace výpočtu nejkratší trasy. Třída obsahuje jednu proměnnou a dvě metody. Proměnná se jmenuje *NejlepsiCestaMeziZakazniky*, je veřejná, typu *Tuple<List<int>, double>*. List obsahuje seznam ID zákazníků a hodnota double celkovou vzdálenost mezi nimi v kilometrech.

Soukromá metoda *najdiVzdalenostiMeziExistujicimiZakazniky* používa LINQ příkaz k nalezení vzdáleností mezi všemi zákazníky a vytvoření seznamu nalezených vzdáleností. Tyto informace jsou uloženy v proměnné *vzdalenosti* (Obrázek 13). V příkazu je použit LINQ, kde se vyberou dva zákazníci a pokud nejsou stejní je vytvořen nový *Tuple*, který obsahuje instance obou tříd *Zakaznik* a vzdálenost mezi nimi. Ta je zjištěna zavoláním veřejné metody *SpocitejVzdalenost* ze třídy *Google*. Následně je nový *Tuple* přidán do listu a proces se opakuje pro zbývající zákazníky.

Obrázek 13: Kód pro vytvoření seznamu vzdáleností mezi zákazníky

```
// proměnná obsahující list kombinací všech zákazníků a vzdáleností mezi nimi. Toto bude použito jako základ pro výpočet trasy mezi nimi.
var vzdalenosti = (from zakaznik1 in Databaze.Zakaznici
                   from zakaznik2 in Databaze.Zakaznici
                   where zakaznik1.ID < zakaznik2.ID
                   select new Tuple<Zakaznik, Zakaznik, double>(
                       zakaznik1,
                       zakaznik2,
                       Google.SpocitejVzdalenost(zakaznik1.GeoLat, zakaznik1.GeoLon, zakaznik2.GeoLat, zakaznik2.GeoLon)))
                    .ToList();
```

Zdroj: vlastní zpracování

Hlavní součástí třídy je metoda pro výpočet trasy – *NajdiNejlepsiCestuMeziExistujicimiZakazniky*. Výstupem této metody je jeden *Tuple*, který obsahuje list *ID* zákazníků a celkovou délku cesty ve formátu *double*. Metoda obsahuje několik proměnných, které jsou používány na různých místech v metodě. Jsou definovány proměnné pro délku cesty, vzdálenost k sousedovi, aktuální místo, souseda, seznam navštívených míst a seznam vzdáleností.

Výše zmíněné proměnné obsahují výchozí hodnoty. *Délka cesty* je na začátku rovna nule, stejně tak jsou *stavajiciMisto* a *soused* rovni nule, protože zákazník s *ID* = 0 je výchozí místo pro výpočet trasy. Výchozí hodnota vzdálenosti k sousedovi je nastavena na 100000.0 km, protože později ve výpočtu trasy existuje podmínka, kterou je ověřováno, jestli by byla vzdálenost k novému sousedovi kratší než vzdálenost k aktuálnímu

sousedovi. Volbou takto velkého čísla máme garantováno, že vzdálenost od souseda bude vždy menší, než toto výchozí číslo, protože vzdálenost mezi dvěma body na zemi nemůže být nikdy takto velká.

Tato metoda je implementací algoritmu pro nalezení nejkratší cesty za pomocí nejbližších sousedů.

Obrázek 14: Hlavní část metody pro výpočet trasy

```
while (navstiveno.Count() < Databaze.Zakaznici.Count - 1)
{
    //pro kazdy spoj ze seznamu vsech spoju
    foreach (var spoj in vzdalenosti)
    {
        //pokud je stavajici zakaznik na prvnim miste v listu spoju,
        if (spoj.Item1.ID == stavajiciMisto &&
            !navstiveno.Contains(spoj.Item2.ID) &&
            delkaKSousedovi > spoj.Item3)
        {
            soused = spoj.Item2.ID;
            delkaKSousedovi = spoj.Item3;
        }
        else if (spoj.Item2.ID == stavajiciMisto &&
            !navstiveno.Contains(spoj.Item1.ID) &&
            delkaKSousedovi > spoj.Item3)
        {
            soused = spoj.Item1.ID;
            delkaKSousedovi = spoj.Item3;
        }
    }

    //po nalezeni nejblizsiho souseda je cesta prodlouzena o cestu k
    navstiveno.Add(stavajiciMisto);
    stavajiciMisto = soused;
    delkaCesty += delkaKSousedovi;
    delkaKSousedovi = 100000.0;
    if (navstiveno.Count() == Databaze.Zakaznici.Count - 1)
    {
        navstiveno.Add(soused);
    }
}
```

Zdroj: vlastní zpracování

Jelikož je nutné nalézt spoje mezi zákazníky, je jejich počet o jedna menší, než je celkový počet zákazníků. Funkce tedy začíná kontrolou, jestli dosáhl počet navštívených míst součtu zákazníků mínus jeden. Dokud toto platí, je načítán seznam zákazníků a vzdáleností mezi nimi. V dalším kroku je nalezena první cesta vedoucí od právě zvoleného

zákazníka. Cesta však nesmí vést k již navštívenému zákazníkovi a zároveň musí být cesta kratší než aktuální vzdálenost k sousedovi. Pokud soused ještě nebyl navštíven a vzdálenost k němu je kratší než vzdálenost k aktuálně zvolenému sousedovi, je do proměnné *soused* dosazen nově nalezený soused a do proměnné *délkaKSousedovi* – vzdálenost k nově zvolenému sousedovi. Poté se proces hledání opakuje.

Na konci tohoto procesu je nalezen nejbližší soused aktuálně zvoleného místa. Po jeho nalezení je aktuální místo přidáno do seznamu již navštívených míst, aktuální místo je změněno na místo nejbližšího souseda, k celkové délce cesty se přidá vzdálenost k nalezenému sousedovi a vzdálenost k sousedovi je opět nastavena na 100000.

Obrázek 15: Změna proměnných na konci logické části výpočtu trasy

```
navstiveno.Add(stavajiciMisto);
stavajiciMisto = soused;
delkaCesty += delkaKSousedovi;
delkaKSousedovi = 100000.0;
if (navstiveno.Count() == Databaze.Zakaznici.Count - 1)
{
    navstiveno.Add(soused);
}
```

Zdroj: vlastní zpracování

Jakmile je ideální cesta nalezena, je vytvořen nový *Tuple*, který obsahuje seznam *ID* navštívených zákazníků, v pořadí, v němž byli navštíveni a celkovou *délku cesty*.

4 Závěr

Tato bakalářská práce byla zaměřena na problematiku vývoje Windows Forms aplikace pro výpočet optimální trasy mezi zákazníky za pomoci objektově orientovaného programování v jazyce C#.

V teoretické části byla popsána historie a principy OOP a datového modelování, historie a vývoj programovacího jazyka C. Dále byla objasněna historie logistiky a následně popsány principy VRP a vysvětlení několika algoritmů používaných v optimalizaci distribučních cest.

V praktické části jsou popsány nutné nástroje a pojmy potřebné pro pochopení vývoje aplikací v prostředí Microsoft Windows. Následně jsou popsány části kódu vyvíjené aplikace s vysvětlením jejich použití.

Výsledkem této práce je aplikace, do které může uživatel napsat adresy zákazníků a následně je aplikací nalezena optimální trasa, s použitím algoritmu nejbližšího souseda, která je na konci zobrazena na mapě. Uživatelské prostředí aplikace se snaží být jednoduché, bez matoucích grafických prvků.

Dalším krokem vývoje aplikace by mohlo být přidání více algoritmů pro optimalizaci trasy, nebo např. implementace možnosti importu seznamu zákazníků z externího souboru.

Pro větší podporu ve firemním prostředí by z dlouhodobého hlediska mohla být aplikace přetvořena do podoby webové aplikace, která by byla uložena v Cloudu. Tím by byla eliminována nutnost firem povolit spouštění nepodepsané aplikace třetí strany.

5 Seznam použitých zdrojů

KUBÍČKOVÁ, Lea. Obchodní logistika. Brno: Mendelova zemědělská a lesnická univerzita, 2006. ISBN 80-7157-952-1.

MERUNKA, Vojtěch. Objektové modelování. Praha: Alfa Nakladatelství, 2008. Informatika studium (Alfa Nakladatelství). ISBN 978-80-87197-04-2.

PURDUM, Jack. Beginning C# 3.0: An Introduction To Object Oriented Programming. WroxPress, 2008, 556s. ISBN 978-0-470-26129-3.

SMITH, Ben. Advanced ActionScript 3: design patterns. Second edition. New York, NY: Apress, 2015. Expert's voice in Web development. ISBN 978-1-484206-72-0.

ŠUBRT, Tomáš. Ekonomicko-matematické metody. Plzeň: Vydavatelství a nakladatelství Aleš Čeněk, 2011. ISBN 978-80-7380-345-2.

STEFANOV, Stoyan. Object-oriented JavaScript: create scalable, reusable high-quality JavaScript applications and libraries. 1. Birmingham, UK: Packt Publishing, 2008. ISBN 978-1-847194-14-5.

NYGAARD, Kristen a DAHL, Ole-Johan. The Development of the SIMULA Languages [online]. Norwegian Computing Center and University of Oslo [cit. 2016-11-04]. Dostupné z: http://cs-exhibitions.uni-klu.ac.at/fileadmin/template/documents/text/The_development_of_the_simula_languages.pdf

Solution Methods for VRP. NEO: Networking and Emerging Optimization [online]. Espaňa: NEO, 2013 [cit. 2016-11-04]. Dostupné z: http://neo.lcc.uma.es/vrp/solution-methods/

BAZARAA, M. S., JARVIS, John J. a SHERALI, Hanif D.. Linear programming and network flows. 4th ed. Hoboken, N.J.: John Wiley & Sons, 2010. ISBN 978-0470462720.

GUTIN, Gregory, PUNNEN Abraham P. The traveling salesman problem and its variations. 12. vyd. Boston: Kluwer Academic Publishers, 2002. ISBN 978-1-4020-0664-7.

6 Přílohy

Přílohy jsou k dispozici na optickém disku, který je přiložen k této bakalářské práci.

Příloha 1 – zdrojový kód aplikace

Příloha 2 – spustitelný soubor aplikace