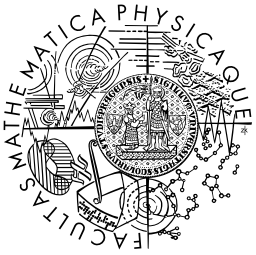


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Matej Kukurugya

Trasy v mapách pro orientační běh

Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: RNDr. Ondřej Pangrác, Ph.D.

Studijní program: Informatika (B0613A140006)

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ....

Podpis autora

Rád by som poďakoval RNDr. Ondřejovi Pangrácovi, Ph.D. za ochotu a pomoc pri vedení bakalárskej práce. Taktiež by som rád poďakoval svojím rodičom za ich podporu a vytvorenie podmienok, vďaka ktorým bolo možné tvorbu baklárskej práce započať a dokončiť.

Název práce: Trasy v mapách pro orientační běh

Autor: Matej Kukurugya

ústav: Informatický ústav Univerzity Karlovy

Vedoucí bakalářské práce: RNDr. Ondřej Pangrác, Ph.D., Informatický ústav Univerzity Karlovy

Abstrakt: Původním záměrem této bakalářské práce bylo vytvořit aplikaci, která bude na základě mapových dat hledat vhodnou trasu v prostředí orientačního běhu. Tento záměr však nebyl plně naplněn, neboť až příliš času zabralo vytvoření návrhu a naprogramování samotné aplikace. Z toho důvodu nezbyl čas na vytvoření vyhledávacích alogirtmů ani mechanizmů na zpracováni map, na kterých by se cesty hledaly. Z tohoto důvodu vznikla jenom kostra, která v této chvíli sama o sobě neobsahuje žádnou konrkétní funkcionalitu. Oproti tomu objektový návrh je detailně promyšlen a propracován. Tato bakalářská práce tedy v konečném důsledku pojednává právě o objektovém návrhu vytovřené aplikace a myšlenkách aplikovaných v průběhu jeho tvorby.

Klíčová slova: návrh aplikace, MVVM, hledáni tras v otevřeném terénu

Title: Navigation in orienteering maps

Author: Matej Kukurugya

institute: Computer science institute of Charles University

Supervisor: RNDr. Ondřej Pangrác, Ph.D., Computer science institute of Charles University

Abstract: The prior intention of this bachelor thesis was creating and application which based on map data will look for suitable route in orienteering environment. This intention was not completly fulfilled though. Creating and programming of application itsel took to much time so there was left none for creating of usable searching algorithms nor mechanisms for processing of maps on which routes would be looked for. For this reason the skelet of application was created which currently does contain none specific functionality. On the other side design of application is thought out and elaborated in detail. This bachelor thesis talks in the end specifically about design of created application and ideas applied during its creation process.

Keywords: application desing, MVVM, path finding in open terrain

Obsah

Úvod

Veľmi často v živote narážame na situácie, kedy sa potrebujeme dostať z bodu A do bodu B čo najrýchlejším spôsobom. V mnohých prípadoch máme k dispozícii sieť cestných komunikácii či chodníkov, na základe ktorých sa môžeme rozhodovať, ktorá trasa je pre nás vyhovujúca. Pre takéto prípady nám veľmi dobre poslúžia už existujúce navigačné systémy, ktoré majú podrobne zmapovanú cestné siete a vedia nám na základe parametrov týchto ciest určiť najrýchlejšiu alebo najúspornejšiu trasu.

Avšak môžu nastať aj situácie, kedy je cestná sieť priveľmi riedka, až takmer neprítomná. Väčšinou takéto situácie nastávajú vo voľnej prírode a oblastiach ďaleko od civilizácie. V takých prípadoch nám konvenčné navigácie veľmi dobre neposlúžia. Ak máme k dispozícii kvalitnú mapu, môžeme sa pokúsiť v nej nájsť vyhovujúcu trasu vlastnými silami, ale často existuje príliš mnoho možností, ktorými sa môžeme vydať. Preto by sa niekedy hodilo mať nástroj, ktorý na základe zadaných parametrov nájde na predloženej mape najrýchlejšiu trasu, po ktorej sa človek môže vydať k vytýčenému cieľu.

Príklady využitia takéhoto software-u by sme mohli nájsť v špecifických profesiách ako sú napríklad lesníctvo, záchranné služby či ozbrojené sily. V týchto profesiách je potrebné z času na čas sa dostať na odľahlé miesto, aby bolo možné vykonať ich zámery.

Takýto software by však našiel uplatnenie taktiež v rekreačných aktivitách. Či už turizmus alebo športové aktivity sa často odohrávajú vo voľnej prírode. Z rôznych dôvodov sa potom môže zísť schopnosť nájdenia najrýchlejšej trasy späť do najbližšej obývanej oblasti, či už za alebo bez pomoci ciest.

Rád by som vyzdvihol špecificky jedno športové odvetvie, ktoré veľmi úzko súvisí s hľadaním ciest v otvorenom teréne a to *orientačný beh*. „Orientačný beh (skratka OB) je športové odvetvie vytrvalostného charakteru, pri ktorom je úlohou prejsť alebo prebehnúť podľa mapy a buzoly trať vyznačenú na mape za čo najkratší čas. V teréne nie je trať vyznačená, sú tam umiestnené iba kontrolné stanovišťa (Kontroly)“[1]. Tento šport bol hlavnou motiváciou pre vytvorenie aplikácie, v ktorej by si užívateľ mohol nechať vykresliť na mape najrýchlejší postup pre ním zadanú trať.

V tejto práci sú popísané myšlienky použité pri vytváraní spomínanej aplikácie, jej architektúry a implementácie.

V prvej kapitole je možné nájsť abstrakciu nad samotnou problematikou hľadania najrýchlejšej trasy v otvorenom teréne podľa konkrétnej mapy. Táto abstrakcia je poňatá z pohľadu orientačného behu. Ďalej je v nej možné nájsť informácie o využitých prostriedkoch pri implementácii aplikácie. Týmito prostriedkami sú napríklad využívaný jazyk a knižnice, ale aj použitý návrhový vzor architektúry aplikácie.

V následujúcej kapitole je predostretý bližší pohľad na spomínanú architektúru aplikácie, jej „horizontálne“ a „vertikálne“ členenie na vrstvy. Funkcie a správanie jednotlivých vrstiev sú v tejto kapitole dopodrobna vysvetlené.

V tretej kapitole sa pozrieme na aktuálne existujúce vertikálne vrstvy aplikácie. Ich funkciu a architektúru popíšeme po jednotlivých častiach. Popíšeme špecifické

vlastnosti a spôsoby použitia týchto častí.

Vo štvrtej a zároveň poslednej kapitole bude možný k nahliadnutiu popis horizontálnej vrstvy zvanej *Model*. Táto vrstva sa delí na mnoho oblastí, z ktorých každá spravuje určitý typ dát a mechanizmov využívaných vo zvyšných vrstvách aplikácie. Štruktúra implementácií jednotlivých oblastí bude v tejto kapitole detailne rozobraná.

1 Obecné informácie o aplikácii

1.1 Aspekty hľadania najrýchlejšej trasy (v OB)

Mapy pre orientačný beh sú veľmi detailné a bežec má preto mnoho informácií na to, aby si mohol zvoliť ideálnu trasu. Za predpokladu, že bežec nerobí chyby a beží presne podľa svojho zámeru, sú pre výber najrýchlejšieho postupu dôležité dva druhy objektov: líniové (cesty, potoky, prieseky, ...) a plošné (lúky, kroviská, vodné objekty, močiare, ...). Tie určujú typ terénu nachádzajúci sa v danej časti mapy a tým pádom aj veľkosť odporu, ktorý je kladený rýchlosti behu pretekára. Táto veličina sa dá použiť ako hlavný parameter, ktorý bude určovať preferencie výberu postupu.

Proces hľadania cesty sa skladá z dvoch hlavných častí:

- **vytvorenie mapovej reprezentácie** - Na začiatku je potrené na základe mapového súboru vygenerovať mapovú reprezentáciu, na ktorej bude možné cestu vyhľadávať. Mapovou reprezentáciou by mal byť objekt, ktorý dobre vystihne topografiu mapy a umožní hľadanie najideálnejšej trasy. V našej aplikácii budú týmito objektmi *ohodnotené grafy*.
- **aplikácia vyhľadávacieho algoritmu** - mapová reprezentácia je predaná algoritmu a on za použitia jej interface-u v nej vyhľadá najkratšiu cestu. V našom prípade najkratšia cesta znamená tá najrýchlejšia.

V následujúcich podsekciách spomeniem koncepty, ktoré budú v procese hľadania najrýchlejších ciest vystupovať. Medzi jednotlivými konceptmi sú tvorené rôzne závislosti. Grafické znázornenie týchto závislostí je možné nahliadnuť na konci sekcie v obrázku 1.1.

1.1.1 Mapy

Na začiatok je potrebné spomenúť koncept mapy. V procese hľadania ciest bude na viacerých miestach potrebné agregovať dáta z užívateľom vybraného mapového súboru. Aby sme nemuseli neustále čítať priamo zo súboru, budeme si udržovať jeho obsah v pamäti prívetivejším spôsobom.

Touto formou bude práve *mapa*. Mapa si bude udržovať všetky objekty definované v mapovom súbore a keď bude potrebné agregovať z daného súboru nejakú informáciu (mapovú reprezentáciu, mapovú grafiku, ...), použije sa namiesto súboru odpovedajúci mapový objekt.

Je vhodné zmieniť, že koncept mapy je odlišný od konceptu *mapovej reprezentácie*. Mapa, na rozdiel od jej reprezentácie, neobsahuje žiadne zložité prepojenia medzi objektami ktoré v sebe drží. Jej vytvorenie by malo byť rýchle, s lineárnou časovou zložitosťou v závislosti na veľkosti mapového súboru.

1.1.2 Mapové reprezentácie

Mapové reprezentácie sú jednou z dôležitých zložiek procesu hľadania ciest. Sú to jednotky, na ktorých sa vyhľadávanie uskutočňuje. Mapové objekty sú oproti

mapám zložitejšie objekty, ktoré už v sebe zahrňujú plno závislostí a prepojení. Ich generovanie môže zabrať oveľa viac času ako generovanie mapových objektov.

Ako už bolo spomenuté vyššie, v aplikácii budú mapovými reprezentáciami *ohodnotené grafy* (naďalej iba grafy). Napriek tomu, že mapová reprezentácia a graf budú v aplikácii reprezentovať rovnaký objekt, ich významy sú odlišné:

- **Mapová reprezentácia** hovorí o tom, ako daný objekt pracuje interne. Popisuje jeho vlastnosti, mechanizmy a spôsoby akými generuje výsledný *graf*, v ktorom sa následne hľadá cesta. Objekt, ktorý je agregovaný z mapy, je v prvom rade mapovou reprezentáciou a až v druhom rade grafom.

  Príkladom myšlienky, ktorú môže mapová reprezentácia vyjadrovať je „samozahusťujúci“ sa graf. Takáto mapová reprezentácia počas behu algoritmu zahusťuje predom pripravený graf o ďalšie vrcholy na miestach, v ktorých sa prehľadávanie aktuálne uskutočňuje. Redukuje sa tým veľkosť vygenerovaného grafu.

- **Graf** na druhej strane hovorí o vlastnostiach objektu, ktoré sú viditeľné navonok. Bude informovať o „grafových“ službách objektu, ktoré dokáže poskytnúť. Tieto služby môžu byť obmedzené práve vnútornou štruktúrou ktorá je definovaná *mapovou reprezentáciou*. Graf sa berie ako druhotný produkt spracovania mapy.

  Príkladom vlastnosti, ktorú môže graf prezentovať vonkajšiemu svetu, je možnosť poskytnutia kompletne vygenerovaného grafu, pri ktorom sa vonkajší užívateľ nemusí obávať, že by sa počas práce s ním graf nejakým spôsobom modifikoval. Túto vlastnosť napríklad nevie zaručiť graf ktorý zodpovedá vyššie zmienenej mapovej reprezentácii, ktorá stav grafu počas práce neustále modifikuje, zahusťuje ho.

Pojmy *mapová reprezentácia* a *graf* budú naďalej v práci brané ako zameniteľné a ich použitie bude závisieť od okolitého kontextu a ich špecifického významu.

1.1.3 Užívateľské modely

Je potrebné si uvedomiť, že rôzni bežci majú rôzne schopnosti a preto aj ich preferencie na výber trasy nemusia byť rovnaké. Niekto sa dokáže rýchlejšie predierať cez husté pasáže, inému ide rýchlejšie beh cez močiar a ďalšiemu vyhovujú dlhšie postupy po cestách.

Preto by bolo vhodné, aby užívateľ aplikácie mal možnosť aplikovať svoje preferencie do procesu vyhľadávania. K tomuto účelu boli vytvorené tzv. „užívateľské modely“. Užívateľ si pomocou nich môže vytvoriť vlastný „profil“, na základe ktorého bude vyhľadávanie cesty prispôsobené.

Užívateľské modely vďaka svojej informovanosti nadobúdajú zodpovednosť za výpočty hodnôt, ktoré sú závislé na preferenciách užívateľa. Stávajú sa teda dôležitou zložkou procesu vyhľadávania ciest. Vyhľadávacie algoritmy stávajú závislé na nimi vykonávaných výpočtoch.

1.1.4 Výškové dáta

Jedným z dôležitých a neopomenuteľných faktorov voľby najrýchlejšieho postupu je prevýšenie, ktoré je potrebné pri jeho prevedení zdolať. Mnoho typov máp popisuje reliéf krajiny pomocou takzvaných *vrstevníc*. Vrstevnica je krivka na mape, ktorá spája body rovnakej nadmorskej výšky. Pre človeka je vrstevnicová abstrakcia výšky terénu veľmi ľahko pochopiteľná a spracovateľná.

Pre strojové spracovanie mapy však vrstevnice predstavujú veľmi neprirodzený spôsob reprezentácie nadmorskej výšky. Vypracovanie reliéfneho obrazu za pomoci vrstevníc samotných je veľmi ťažká úloha. V niektorých prípadoch (mapy pre OB napríklad) dokonca jednotlivé vrstevnice nie sú v mapovom súbore reprezentované jedným objektom, nakoľko kvôli dobrej čitateľnosti máp sú na viacerých miestach prerušované.

Z vyššie uvedených dôvodov je v aplikácii zahrnutý systém, ktorý sprostredkováva užívateľom možnosť stiahnutia a spravovania digitálnych výškových dát, ktoré následne môžu byť použité ako pomocný zdroj v procese tvorby mapových reprezentácií. Pre konštrukciu mapových reprezentácií pre niektoré mapové formáty bude nutná prítomnosť zodpovedajúcich výškových dát a pri ich absencii jednoducho nebude možné mapové reprezentácie vytvoriť.

Výškové dáta môžu byť sťahované z viacerých zdrojov. Každý zdroj môže definovať viacero dátových distribúcií, ktoré dokáže ponúknuť. Tieto distribúcie sa väčšinou líšia kvalitou a dostupnosťou sprostredkovaných dát (v niektorých prípadoch je ku možnosti stiahnutia výškových dát potrebná užívateľova autorizácia).

1.1.5 Atribútové template-y

Ďalšia vec, nad ktorou je potrebné sa zamyslieť je, akým spôsobom sa bude v grafoch mapových reprezentácií uchovávať informácia o mapových vlastnostiach a atribútoch, ktoré konkrétne vrcholy a hrany grafu reprezentujú. Zároveň je potrebné aby dotyčné vlastnosti dokázal príslušný užívateľský model spracovať a dopočítať z nich hodnoty, potrebné pre beh vyhľadávacích algoritmov. Používané atribúty, ktoré agregujeme z máp do mapových reprezentácií preto musia byť jednotné v celom procese hľadania cesty, od vytvárania mapovej reprezentácie, po spustenie vyhľadávacieho algoritmu.

V aplikácii nám definíciu a jednotnosť atribútov budú zabezpečovať tzv. *template*-y. Každý template bude definovať jedinečnú sadu vrcholových a hranových atribútov. Príkladom pre takúto kolekciu pre potreby OB môže byť napríklad:

- vrcholové atribúty - pozícia, výška, indikátory reprezentovaných terénnych objektov (či sa daný vrchol nachádza na ceste, v húštine, na lúke, v dobre priebežnom lese,...)
- hranové atribúty - či daná hrana reprezentuje úsek nejakej cesty, hranu nejakého objektu(lúky, húštiny, močiaru,...), sklon terénu

Na template-och ako takých bude teda závisieť:

- **výber užívateľských modelov** - model musí vedieť spracovať atribúty definované daným template-om a vrátiť od neho požadované výsledky.

- **formát vybraného mapového súboru** - musí existovať konvertor mapy špecifického formátu na odpovedajúcu mapovú reprezentáciu, v ktorej vrcholoch a hranách sú obsiahnuté atribúty definované daným template-om. Túto závislosť môžeme brať aj opačným smerom - pre vybraný mapový formát môžeme zvoliť len použiteľný template.

1.1.6 Vyhľadávacie algoritmy

Nakoniec nemôžeme opomenúť koncept samotných vyhľadávacích algoritmov, poslednú neodmysliteľnú súčasť procesu hľadania ciest. Vyhľadávací algoritmus, podobne ako *mapová reprezentácia*, reprezentuje koncept vnútorného mechanizmu ktorým je najkratšia cesta v grafe hľadaná. Príkladom takéhoto algoritmu môže byť napríklad algoritmus *A**.

Vstupom do každého algoritmu sú:

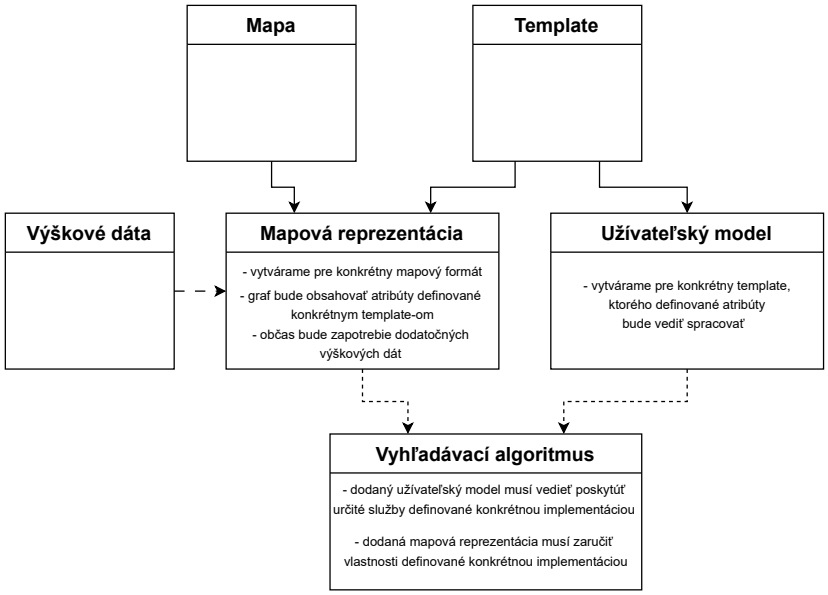
- **graf**, na ktorom je hľadanie najkratšej cesty vykonané a
- **užívateľský model**, ktorý algoritmus nutne potrebuje ku výpočtom váh grafových hrán a iných hodnôt potrebných k jeho správnemu chodu.

Každý algoritmus následne ponúka množinu jeho implementácií. Každá implementácia definuje množinu vlastností, ktoré vložený graf a užívateľský model musia spĺňať. Napríklad implementácie A* algoritmu budú požadovať, aby užívateľský model bol schopný, popri výpočte váh pre grafové hrany, dodať ešte aj výpočet heuristiky využívanej v týmto algoritmom.

Stav používaného grafu sa môže počas behu algoritmu meniť. Teda je potrebné ho vždy na konci vyhľadávania opäť vrátiť do pôvodného stavu a zaručiť, že graf je v jednu chvíľu súčasťou len jedného vyhľadávacieho procesu.

Vyhľadávanie samotné je sprostredkované dvomi spôsobmi. Buď sa požiada algoritmus priamo o nájdenie cesty na danej trati alebo sa využije takzvaný *executor* algoritmu.

Executor nám sprostredkuje vybraný algoritmus, ktorý bude pracovať s dodanou mapovou reprezentáciou a užívateľským modelom. Executor si pri inicializácii pre seba uzamkne dodanú mapovú reprezentáciu a až do jeho uvolnenia ju neodblokuje. Do tejto doby môže prijímať rôzne vstupné trate a vracať pre ne nájdené cesty. Hlavná výhoda executor-ov je práve ich schopnosť uzamknutia mapovej reprezentácie na dlhšiu dobu. Vďaka tomu ju dokážu využívať pre viacero separátnych vyhľadávaní, bez potreby jej neustáleho uvádzania do konzistentného stavu.



Obrázek 1.1 Diagram závislostí jednotlivých konceptov

1.2 Zvolené implementačné prostriedky

1.2.1 Použitý programovací jazyk

Pre implementáciu aplikácie bol vybraný jazyk C#. Jazyk bol vybraný pre jeho jednoduchosť a bezpečnosť použitia. Alternatívnou volbou by mohol byť jazyk Java, ktorý má blízko práve ku C#. Jeho nedostatočná znalosť v dobe začiatku práce ho však vyradila z použiteľných možností.

Ďalšími možnosťami by mohol byť typicky vysoko-úrovňový jazyk Python alebo nízko úrovňový jazyk C++. Nakoľko v aplikácii bude prebiehať mnoho výpočtov, python by nebol vhodnou voľbou pre nedostatočnú rýchlosť ním vytvoreného strojového kódu. Na druhú stranu, C++ by bol dobrým kandidátom z hladiska výpočtovej sily. V tomto prípade však narážame opäť na nie veľmi dobrú znalosť tohto jazyka a na jeho programátorskú neprívetivosť. Pri väčšom projekte sme považovali za potrebné mať istotu, že nás použitý jazyk v programovaní podrží.

Vhodnou úpravou implementácie by bolo implementovať výpočtovo náročné procesy v jazyku typu C++ a následne túto implementáciu volať externe z C# jadra.

1.2.2 Užívateľské rozhranie

Pre implementáciu užívateľského rozhrania som sa rozhodol pre *Avalonia UI* framework. Užívateľské rozhranie je veľmi jednoduché. Bolo vytvorené tak, aby zabezpečilo všetku funkcionalitu.

V C# existuje viacero možných knižníc, z ktorých bolo možné si vybrať:

- GUI knižnice, ktoré sú súčasťou samotného .NET framework-u:
  - **Windows Forms** je klasická GUI knižnica. Je jednoducho použiteľná a vďaka jej dlhoročnej podpore aj robustná a spoľahlivá. Jej vek je však aj jej nevýhodou, nakoľko vzhľad aplikácií vytvorených za pomoci

Windows Forms príde človeku pomerne zastaralý. Ďalšou nevýhodou je rastrová povaha jeho renderovcieho engine-u. Táto vlastnosť sa nehodí pre aplikáciu, ktorej jedným z hlavných účelov je vykreslovanie mapových, vektorových objektov.

  - **Windows Presentation Foundation (WPF)** je modernejší nástupca Windows Forms. Aplikácie tvorené touto knižnicou majú modernejší vzhľad a sú generované renderovacím enginom založeným na vektorovej grafike. Ku programovaniu sa tu využíva popri C# aj XAML, v ktorom sa definuje layout užívateľského rozhrania. Avšak spoločne s Windows Forms je ich spoločnou nevýhodou platformová závislosť na operačnom systéme Windows.
  - **.NET MAUI** je nástupcom *Xamarin.Forms*. Je to open-source-ový cross-platformný framework s množinou UI nástrojových balíčkov pre jednotlivé platformy. Podobne ako vo WPF sa pre vytváranie UI využíva kombinácia C# a XAML.

- Alternatívou ku vstavaným .NET GUI knižniciam je práve nezávislá, open-source knižnica **Avalonia UI**. Čo do vlastností je veľmi dobre porovnateľná s *.NET MAUI*. Hlavný rozdiel medzi týmito dvomi knižnicami je v spôsobe, akým vykresľujú užívateľské rozhranie. Avalonia zapojuje kresliaci engine poháňaný knižnicou pre 2D grafiku *Skia*. Na druhú stranu MAUI využíva natívne nástrojové balíčky pre každú platformu zvlášť. Ďalším rozdielom je, že Avalonia, na rozdiel od MAUI, podporuje aj niektoré distribúcie Linuxových systémov.

Rozhodnutie nakoniec padlo na využitie framework-u Avalonia UI. Nakoľko je veľmi podobný natívnemu .NET MAUI, rozhodla podpora pre Linuxové systémy a aj kvalitne spravená dokumentácia, z ktorej sa ľahko dalo vyčítať, ako sa s knižnicou má pracovať.

Informované rozhodnutie pre výber GUI knižnice bolo učinené na základe zdrojov [2, 3, 4]. Zároveň väčšinu informácií, ktoré som o Avalonia UI počas tvorby programu čerpal, pochádzali z jej dokumentácie [5].

1.2.3 Architektúra Model-View-ViewModel (MVVM)

Jeden z ďalších dôležitých aspektov, ktorý hral úlohu vo výbere knižnice pre užívateľské rozhranie, bola podpora *MVVM* návrhového vzoru. Dokumentácia Avalonia UI popisuje architektúru MVVM následovne: „The Model-View-View Model (MVVM) pattern is a common way of structuring a UI application. It uses a data binding system that helps move data between its view and view model parts. This means it achieves separation of application logic (view model) from the display of the UI (view). Separation between the application logic and the business services (model) is commonly achieved by a Dependency Injection (DI) system.“[6].

MVVM architektúra je vhodná pre našu aplikáciu, nakoľko pre jej rozsah by klasická *event-driven code-behind* architektúra nemusela postačovať. Týmto spôsobom zaručíme lepšiu separáciu a dostatočnú vnútornú nezávislosť kódu.

V našej aplikácii bude tento návrhový vzor uplatnený s jemnou obmenou. Vrstva View model, ktorá v pôvodnom návrhovom vzore zastávala úlohu logiky

aplikácie, bude rozdelená na dve časti: View model a novo vzniknutý *Model view*. Viac informácií o tejto úprave je možné nájsť v sekcii 2.1.

1.2.4 Reaktívne programovanie

Ďalším dôležitým aspektom aplikácie je Avaloniou a architektúrou MVVM iniciované využitie *reaktívneho programovania.* Avalonia pre aplikáciu tohto paradigma využíva framework **Reactive UI**.

Tento framework definuje reaktívne programovanie ako „Reactive programming is programming with asynchronous data streams.“ a dodáva „Event buses or your typical click events are really an asynchronous event stream, on which you can observe and do some side effects. Reactive programming is that idea on steroids. You are able to create data streams of anything, not just from click and hover events. Streams are cheap and ubiquitous and anything can be a stream: variables, user inputs, properties, caches, data structures, etc.“[7].

V aplikácii sa bude tento framework využívať prevažne vo vrstvách View a View model, medzi ktorými prebieha väčšina reaktívnej komunikácie.

2 Architektúra ako celok

Architektúra aplikácie má mriežkovú štruktúru. Je tvorená horizontálnymi (MVVM návrhový vzor) a vertikálnymi (*session*-y + hlavné okno) vrstvami. V následujúcich sekciách popíšeme, ako jednotlivé vrstvy vyzerajú, aké su ich úlohy a ako medzi sebou komunikujú. Na konci kapitoly je následne k nahliadnutiu diagram 2.1 znázorňujúci príklad možnej architektúry aplikácie.

2.1 MVVM(MV) návrhový vzor

Ako už bolo spomenuté v podsekcii 1.2.3, v aplikácii je využívaný návrhový vzor MVVM s drobnou obmenou. Táto obmena sa týka rozdelenia originálnej vrstvy View model do dvoch častí: View model a Model view. Toto rozdelenie zaručí ešte o niečo lepšiu separáciu kódu a odľahčí tým úlohy View model vrstvy.

MVVM(MV) architektúra teda rozdeluje aplikáciu na 4 vrstvy: View, ViewModel, ModelView a Model. Popis jednotlivých vrstiev je k nahliadnutiu v následujúcich podsekciách.

2.1.1 View

View je vrstva, ktorá popisuje a implementuje grafickú stránku aplikácie a určuje, akým spôsobom sa dáta dodané vrstvou View model zobrazia užívateľovi. View-y sú objekty viazané na odpovedajúce view model-y za pomoci reaktívneho programovania. Spracovávajú akcie užívateľa a iniciujú reakcie zvyšných vrstiev architektúry prostredníctvom príslušného view model-u. Následne zabezpečujú zobrazovanie dodaných výsledkov reakcie.

View vrstva je implementovaná za pomoci Avalonia UI framework-u. V náväznosti na tento fakt sú v aplikácii použité tri hlavné typy view-ov:

- **Window** - Reprezentujú špecifické okná aplikácie, „top-level“ kontajnery, ktoré v sebe držia nejaký obsah. Okná sami o sebe nedefinujú vzhľad aplikácie. Slúžia predovšetkým ako rám, v ktorom sa striedajú jednotlivé view-y. Každé okno má naviazaný svoj vlastný view model, ktorý drží informáciu o tom, aký view je v danej chvíli v okne zobrazovaný. Naviazaný view model taktiež obsahuje vlastnosti, ktoré priamo súvisia s vlastnosťami daného okna.

- **View** - Sú to hlavné zložky, ktoré nesú grafiku zobrazujúcu sa oknách. Reprezentujú „pohľady“, ktoré sú užívateľovi vykresľované. Každý view je naviazaný na špecifický view model. Viaže sa na jeho vlastnosti a reaktívne vykresluje dáta, ktoré tieto vlastnosti nadobúdajú.

  Býva zvykom že sa konkrétny view zobrazuje práve v jednom konkrétnom okne. V takom prípade si na jeho view model drží referenciu view model daného okna. Za pomoci tejto referencie potom dokáže oknu jeho view model oznámiť, že sa v ňom má daný view zobraziť.

- **DataTemplate** - Definuje grafickú reprezentáciu dát dodávaných z View model vrstvy do vrstvy View. Pre každý dátový typ, ktorý chce byť správne vykreslený pre užívateľa, musí existovať špecifický dátový template, pomocou

ktorého sa daný údaj vykreslí. Dátové template-y sú špecifické tým, že sú raz definované pre celú aplikáciu, aby sa zachovala konzistencia vykreslovania jednotlivých dátových typov.

Je potrebné podotknúť, že na to aby údaj vygenerovaný vo vrstve Model bolo možné vykresliť, musí preňho najprv existovať tzv. *data view model* do ktorého sú jeho informácie zabalené a až v takejto podobe predávané nejakému view model-u, ktorý ich následne pomocou svojich vlastností odovzdá view-u na vykreslenie. Pre každý dátový view model sa následne hľadá príslušný DataTemplate, pomocu ktorého v ňom držané informácie zobrazia užívateľovi.

View vrstva je implementovaná pomocou dvoch jazykov: C# a XAML. Pomocu XAML definujeme všetky polohy a tvary grafických objektov a bind-ujeme vlastnosti týchto objektov na vlastnosti z vrstvy View model. Pre každý view máme obecne jeden špecifický XAML súbor ktorým ho implementujeme. Ku väčšine XAML súborov je priradený C# zdrojový súbor v ktorom sa implementuje takzvaný *code-behind*. V tomto zdrojovom súbore môžeme doplniť všetku funkcionalitu view-u, ktorú nebolo možné vyjadriť jazykom XAML.

2.1.2 View model

Druhou v poradí je vrstva View model. Táto vrstva je zodpovedná za logiku spracovania akcií užívateľa oznámených reaktívnym spôsobom View vrstvou. Disponuje vedomím toho, aké akcie a v akom poradí sa majú vykonať pre zabezpečenie správnej reakcie na detegovaný impulz. K tomuto účelu využíva služby nižších vrstiev prostredníctvom volania metód na vrstve Model view. Tá na základe svojej vnútornej logiky vráti odpoveď s požadovanými údajmi. View model následne spracuje dodané dáta a predostrie ich vrstve View, aby ich mohla zobraziť užívateľovi.

View model na základe aplikačnej logiky koriguje a obmedzuje akcie užívateľa a tým zabraňuje vzniku nekonzistentných stavov aplikácie. Taktiež v niektorých prípadoch iniciuje *interakcie* s inými view model-mi za účelom získania ich doplňujúcich služieb do jeho vlastného procesu.

Podobne ako vo View vrstve sa view model-y delia na tri základné typy:

- **Session view model** + *Main window view model* - Odpovedajú jednotlivým *session*-om, ktoré sú základným kameňom vertikálnej štruktúry aplikácie. Viac informácií o session-och je možné nájsť v sekcii **??**. Výnimkou je práve *Main window view model*, ktorý je naviazaný na hlavné okno aplikácie a zabezpečuje preň aplikačnú logiku. Klasické session view model-y sú taktiež naviazané zvyčajne na jedno okno z vrstvy View, pre ktoré zabezpečujú aplikačnú logiku.

  Každý session view model obsahuje kolekciu príslušných view model-ov, ktoré spoločne implementujú mechanizmus daného session-u. Je zvykom, že v jednej chvíli je aktívny iba jeden view model. O aktívnom view model-e je informované okno viazané na daný session view model. To následne v sebe zobrazuje odpovedajúci view aktívneho view model-u. Informovanie o aktívnom view model-e je hlavnou pracovnou náplňou session view model-u.

K ďalším jeho povinnostiam patrí spracovávanie užívateľom vyvolaných akcií, ktoré sa týkajú samotného naviazaného okna. Príkladom takej akcie môže byť požiadavka o zatvorenie dotyčného okna.

- **View model** - Reprezentujú zložky, ktorých funkcionalita sa najväčšmi ponáša na obecnú, vyššie popísanú funkcionalitu vrstvy View model. Väčšina view model-ov spadá pod réžiu konkrétneho session-u (alebo hlavného okna), pre ktorý implementuje určitú časť jeho mechanizmu. Klasicky sú view model-y naviazané na príslušné view-y z vrstvy View. S tými následne reaktívne komunikujú a reagujú na ich podnety. View model-y sú navzájom nezávislé. To znamená, že medzi nimi neprebieha takmer žiadna komunikácia ani presun dát. Túto funkciu na seba berie Model view vrstva.

  Väčšina view model-ov by mala byť zahrnutá v zodpovedajúcom session view model-e (poprípade Main window view model-e). Ten potom zabezpečuje správu toho, ktorý view model je v danej chvíli aktívny.

  Výnimkou sú view model-y, ktoré sú výhradne používane pre interakcie z iných view model-ov, ktorým týmto spôsobom doručujú svoje služby. Tieto view model-y sú väčšinou vytvorené na mieste interakcie a po jej skončení zanikajú. Pri ich vytváraní im sú zvyčajne predané nejaké vstupné parametre a keď je ich práca dokončená, vracajú jej výsledok.

- **Data view model** - slúžia ako kontajnery pre informácie abstrahované z dát vygenerovaných v Model vrstve. Dáta sú klasicky konvertované do ich zodpovedajúcich view model-ov v Model view vrstve a už takto zabalené informácie sú predávané do View model vrstvy, kde sú spracované a pomocou vlastností odovzdané vrstve View na zobrazenie. Data view model-y môžu dodané informácie drobne upravovať takým spôsobom, aby ich bolo jednoduchšie zobrazovať vo View vrstve.

  Z toho vyplýva, že na to, aby nejaký údaj vygenerovaný v Model vrstve mohol byť prezentovaný užívateľovi, musí preňho existovať odpovedajúci data view model. Zároveň na to, aby informácie obsiahnuté v dátovom view model-e mohli byť vykreslené pre užívateľa, musí vo View vrstve existovať zodpovedajúci *dátový template*, ktorý sa postará o ich správne grafické znázornenie.

  Niektoré data view model-y nielen že obsahujú informácie príslušných dát, ale obsahujú aj dáta samotné. Takéto data view model-y označujeme pomocou slova *wrapping*. (tvoria akýsi obal okolo daných objektov). Táto funkcionalita je dôležitá hlavne v prípadoch, kedy data view model slúži taktiež pre spätnú komunikáciu s Model view vrstvou. V takých prípadoch musí byť možné identifikovať, ktorú dátovú inštanciu „obaluje“. Wrapping data view model-y sú stotožnené s ich dátovým objektom a tiež sa pomocou neho identifikujú.

View model je prvá z vrstiev, ktorá je kompletne písaná v jazyku C#. Komunikácia medzi View model a View vrstvami funguje čisto na báze reaktívneho programovania za pomoci štruktúr z *Reactive UI* framework-u.

2.1.3 Model view

Treťou v poradí je, do klasickej MVVM architektúry pridaná, Model view vrstva. Táto vrstva je zodpovedná za „vnútornú“ logiku aplikácie. Priamo komunikuje s Model vrstvou a využíva jej zdroje pre zabezpečenie svojich služieb pre View model vrstvu. Dá sa povedať, že nedisponuje vlastným „vedomím“. Medzi jej hlavné úlohy patria:

- prijímanie a spracovávanie požiadaviek od vrstvy View model a dodávanie očakávaných výsledkov.
- zabezpečovať *vnútro-session-ovú* komunikáciu. Model view-y v rámci jedného session-u si na seba držia referencie a predávajú si medzi sebou držané dáta. Táto komunikácia by mala byť vyšším vrstvam skrytá a na povrch by mal byť vidieť iba interface, pomocou ktorého prebieha komunikácia s View model-om.
- konverzia dát, získaných od Model vrstvy, do odpovedajúcich data view model-ov pri ich posielaní do vyšších vrstiev.

Na druhú stranu medzi jej povinnosti nepatrí kontrola konzistentnosti jej vlastného stavu. O konzistenciu stavu aplikácie sa má starať View model.

V aplikácii používame model view-y dvoch typov:

- **Session model view** + *Main window model view* - Odpovedajú jednotlivým session-om. (Viac informácií o session-och je možné nájsť v sekcii **??**). Ich hlavnou úlohou je vytvoriť a distribuovať model view-y odpovedajúce danému session-u. Pri ich inicializácii sa medzi model view-mi vytvoria väzby, ktoré sú následne počas behu aplikácie využívané na spomínanú vnútro-session-ovú komunikáciu.

  Ďalej zabezpečujú spracovávanie požiadaviek pre odpovedajúce session view model-y. Tieto požiadavky sa typicky týkajú akcií, ktoré súvisia so session-om ako takým. Môže sa jednať napríklad o reakcie na zatváranie odpovedajúceho session-ového okna.

  Výnimkou je práve *main window model view*, ktorý je viazaný na view model hlavného okna a spracováva jeho požiadavky. V ostatných aspektoch je ale identický s klasickým session model view-om.

  Každému session model view-u odpovedá jeden konkrétny session view model. Ten pri svojej inicializácii predá model view-y, dodané v session model view-e, svojím odpovedajúcim view model-om.

- **Model view** - typ, ktorý nesie vyššie popísanú funkcionalitu Model view vrstvy. Zvyčajne pre každý view model existuje dedikovaný model view, ktorý sa stará o zabezpečenie view model-om požadovaných služieb. Nie je to však pravidlo, view model môže obsahovať odkazy na viacero model view-ov, ktorých služby následne využíva alebo sú predané novo vytvoreným, v interakciách využívaným, view model-om.

  Je zvykom, že každý model view spadá pod nejaký konkrétny session alebo hlavné okno. V takom prípade je daný model view vytváraný a distribuovaný odpovedajúcim session model view-om/main window model view-om.

2.1.4 Model

Poslednou „horizontálnou“ vrstvou je Model. Je zodpovedná za doručovanie dát vyšším vrstvám a za doručenie mechanizmov pracujúcich s týmito dátami. Model sa svojou štruktúrou diametrálne odlišuje od predchádzajúcich vrstiev. Je tvorený jednotlivými oblasťami, ktoré spravujú dedikovaní *manažéri*. Manažéri sú pristupovaný z jednotlivých model view-ov a doručujú im svoje služby, či už informatívne alebo výpočtové. Predstavujú interface-y ponúkajúce prívetivejší spôsob práce s vnútornými mechanizmami model-ov.

Model je predstaviteľom jedinej perzistentnej „horizontálnej“ vrstvy. Manažéri sú väčšinou singleton triedy, ktoré ponúkajú služby všetkým session-om počas celej doby behu aplikácie. Vďaka tomuto spôsobu obsluhy je Model jediná vrstva, ktorej oblasti niesu viazané na žiadnu vertikálnu vrstvu (session/hlavné okno). Z návrhu Model vrstvy vyplýva, že model-y musia byť schopné svoje služby dodávať paralelne pre viacero session-ov naraz.

Model vrstva je jednoducho rozšíriteľná o nové oblasti. Vďaka *singleton* štruktúre sú oblastní manažéri dosiahnuteľní v podstate z akéhokoľvek miesta v programe a teda nemusia byť nikde zahrnutí. Manažéri by nemali mať problém prijímať nové, primerane vytvorené implementácie typov mechanizmov a dát z ich oblastí. Napríklad by nemalo byť zložité dodať nový vyhľadávací algoritmus odpovedajúcemu manažérovi, ktorý ho následne bude ponúkať zvyšku aplikácie. Viac informácií o návrhu architektúry jednotlivých oblastí vrstvy Model je možné nájsť v kapitole 4.

Špecifickým znakom komunikácie medzi Model a Model view vrstvami je ten, že pri nej dochádza ku strate typovej informácie dodávaných dát. Táto vlastnosť je motivovaná jednoduchým faktom, ktorým je udržanie vrstvy Model view jednoduchou. V Model vrstve sa totiž vo veľkom využívajú generiká pre jednoduché prenášanie typovej informácie v ich mechanizmoch.

Využívanie generických typov v Model view vrstve by však prinieslo značné komplikácie v jej implementácii a tomu odpovedajúce zneprehladnenie kódu. Už len Model samotný trpí jemnou, generikami spôsobenou neprehľadnosťou. Z tohto dôvodu bolo rozhodnuté zabezpečiť jednoduchosť vrstvy Model view za cenu straty typovej informácie dát tečúcich z model-ov do model view-ov.

Pri opačnom smere komunikácie je manažérmi typová informácia dodaných parametrov opäť testovaná a získavaná (väčšinou za pomoci tzv. *generic visitor pattern*. Viac informácii o tejto modifikácii klasického *visitor pattern* návrhového vzoru nájdete v kapitole 4.

Vo vrstve Model existujú popri manažéroch ešte aj tzv. *sub-manažéri*. Tieto entity sú ale určené pre využitie z vrstvy Model. Podporujú generickú komunikáciu, na ktorej báze model-y fungujú, bez straty typovej informácie a teda sú príjemnejšie pre model-ovú komunikáciu než klasickí manažéri.

2.2 Session-y + *hlavné okno*

2.2.1 Session-y

Aplikácia, či už z vizuálneho, logického alebo implementačného hladiska, je rozdelená do tzv. *session*-ov. Session-y sú najväčšie stavebné jednotky, z ktorých

každá predstavuje jedinečný mechanizmus dodávaný aplikáciou. Existencia ich inštancií je pominuteľná - vznikajú a zanikajú na popud užívateľa. Session-ov (aj rovnakého druhu) môže byť v aplikácii spustených viacero naraz. Každý session je klasicky tvorený odpovedajúcimi objektmi z prvých troch vrstiev horizontálnej štruktúry.

Dodatočne môžu session-y využívať cudzie objekty z prvých troch vrstiev horizontálnej štruktúry, ktoré patria buď hlavnému oknu alebo inému typu session-u. V takom prípade by malo byť ale poriadne rozmyslené, či takéto „postranné“ využitie dáva zmysel a či sa ním neporušujú zasady používania daného view-u, view model-u alebo model view-u.

Poprípade je možné využívať špecifické model view-y a view model-y prostredníctvom *interakcií*. V takom prípade by ale dané objekty mali byť na daný účel prispôsobené (viac info v podsekcii 2.1.2, bod **View model**, 3. odsek).

Session-y sú klasicky rozdelené na logické časti, ktoré spolupracujú na dodaní požadovaného mechanizmu. Môžu napríklad reprezentovať konkrétne fázy jeho procesu. Každá časť je väčšinou tvorená jednou triedou z každej vrstvy View, View model a Model view.

Jednotlivé session-y by medzi sebou nemali navzájom komunikovať ani zdielať svoje dáta. Mohlo by to viesť ku problémom s paralelizáciou služieb vykonávaných vrstvou Model.

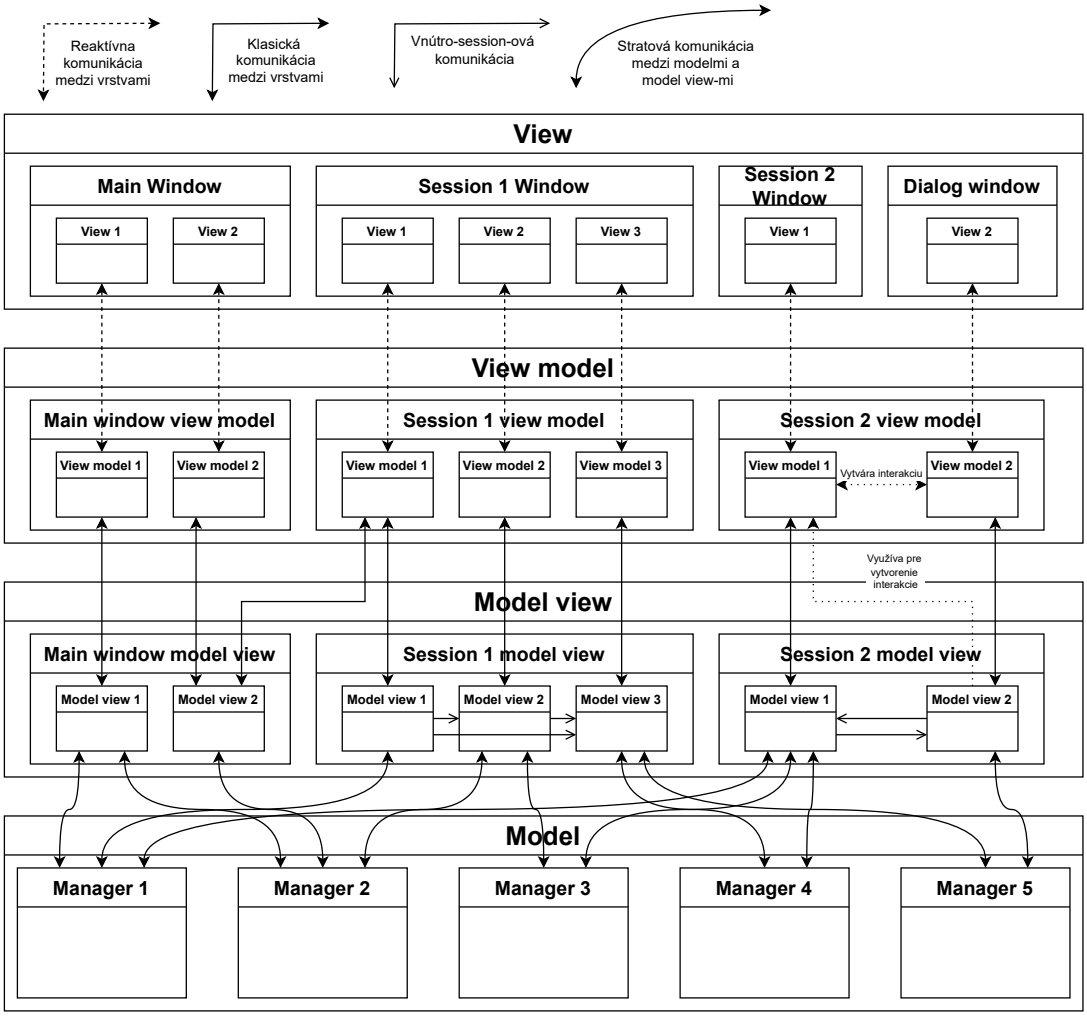
Session-y ako také reprezentujú cestu ku všeobecnej rozšíriteľnosti aplikácie o novú aplikačnú logiku. Ak by vznikla potreba, aby aplikácia obsahovala nejaký nový mechanizmus, stačí preňho vytvoriť odpovedajúci session a upraviť hlavné okno tak, aby ho vedelo ponúknuť užívateľovi.

2.2.2 Hlavné okno

Session-y sú vytvárané a spravované takzvaným *hlavným oknom*. Je to jediná perzistentná „vertikálna“ vrstva aplikácie. Beh aplikácie začína otvorením tohto okna a končí jeho zatvorením. Hlavné okno môže byť používané počas celého behu aplikácie. Pokiaľ je vydaný pokyn na uzavretie hlavného okna, pričom sú stále živé niektoré session-y, užívateľ môže byť na tento fakt upozornený. Ak mu to ale neprekáža, zatvorením hlavného okna sa zavrú aj všetky ostatné a aplikácia sa ukončí. Logika session-ov by mala túto skutočnosť brať do úvahy.

Horizontálna štruktúra hlavného okna je veľmi podobná tej session-ovej. Taktiež využíva MVVM(MV) architektúru. Jej časti sú ale z podstaty hlavného okna vytvárané len raz pri štarte aplikácie a zanikajú pri jej ukončení. *Session view model* a *Session model view* sú nahradené za funkčne identické *Main window view model* a *Main window model view*. Podobne ako pri session-och, aj hlavné okno je rozdelené do niekoľkých častí. Tie si medzi sebou rozdeľujú jeho funkcionalitu. Niektoré z jeho častí môžu byť sprístupnené rôznym session-om, aby si z nich mohli vytiahnuť potrebné parametre platiace pre celú aplikáciu.

Ako bolo spomenuté na začiatku tejto podsekcie, hlavné okno vytvára, eviduje a spravuje inštancie všetkých session-ov. Definuje maximálny počet aktívnych session-ov, spravuje session-y pri ukončovaní aplikácie, poskytuje *dodávateľa* hlavných parametrov, inicializuje pri vytváraní session-u jeho session view model a session model view.



Obrázek 2.1 Príklad možnej „mriežkovej“ štruktúry aplikácie.

3 Súčasná podoba vertikálnej štruktúry aplikácie

V tejto kapitole uvádzame bližší pohľad na súčasnú podobu hlavného okna a session-ov aplikácie. Aktuálne aplikácia disponuje len jedným typom session-u určeným pre samotné vyhľadávanie ciest v mapách (*Path finding session*). Viac informácií o vertikálnej štruktúre aplikácie je možné nájsť v sekcii 2.2.

3.1 Hlavné okno

O obecných úlohách hlavného okne sme sa zmienili už v podsekcii 2.2.2. V tejto sekcii sa pozrieme na aktuálne implementovanú architektúru, ako aj funkcionalitu hlavného okna.

Hlavné okno je tvorené dvomi časťami: hlavným menu a hlavnými nastaveniami. Popri tom interaktívne využíva v nastaveniach služby mechanizmu pre správu výškových dát.

Ako už bolo spomenuté v podsekcii 2.2.2, po zatvorení hlavného okna sa ukončuje aj aplikácia. Tým pádom je úlohou hlavného okna zaistiť uloženie všetkých parametrov aplikácie pre využitie v budúcich behoch.

V následujúcich podsekciách popíšeme časti, z ktorých je aktuálne hlavné okno tvorené. Na konci tejto sekcie je zobrazený diagram 3.1 reprezentujúci jeho architektúru.

3.1.1 Hlavné menu

Hlavné menu je prvá časť, ktorá sa v hlavnom okne užívateľovi po zapnutí aplikácie zobrazí. Obsahuje možnosti vytvorenia inštancií session-ov a možnosť otvorenia hlavných nastavení aplikácie.

View model hlavného menu si vedie evidenciu všetkých otvorených session-ov. Pri detekcii zatvorenia hlavného okna sa stará o ich správne ukončenie. Taktiež si drží referenciu na sprostredkovateľa hlavných nastavení. Toho následne môže predať session-om, ktorý ho môžu použiť pre získanie parametrov platných pre celú aplikáciu.

Zvláštnosťou tejto časti je absencia vlastného model view-u. V aktuálnej podobe aplikácie totiž nie je potrebný, nakoľko hlavné menu nepotrebuje komunikovať so žiadnym modelom ani inou časťou hlavného okna. Ak by táto potreba v budúcnosti vznikla, nemal by byť problém model view pre hlavné menu vytvoriť.

Možné vylepšenie tejto časti by mohlo zahrňovať vypísanie všetkých aktuálne otvorených session-ov pre užívateľa. Zaistilo by to preňho jednoduchšiu orientáciu v aplikácii.

3.1.2 Hlavné nastavenia

Hlavné nastavenia sú druhou časťou hlavného okna. Zabezpečujú možnosť pre užívateľa nastaviť parametre, ktoré sú využívané celou aplikáciou.

V aktuálnom stave sú to len dve možnosti konfigurácie: možnosť zmeny lokalizácie aplikácie a možnosť nastavenia implicitne používaného zdroja výškových dát (presnejšie konkrétnej distribúcie výškových dát daného zdroja).

Lokalizácia je implementovaná na základe lokalizačných .resx zdrojových súborov. Avalonia je schopná takéto zdroje využívať aj na zmenu pevných nápisov aplikácie.

Zdroj výškových dát je nastavovaný za pomoci interakcie s mechanizmom konfigurácie výškových dát. Po stlačení príslušného tlačidla sa v hlavnom okne zobrazí view zodpovedajúci tomuto mechanizmu. Užívateľ si v ňom môže vybrať, ktorú distribúciu dát chce implicitne v aplikácii využívať. Viac informácií ohľadom konfigurácie výškových dat nájdete v následujúcej podsekcii.

Do budúcna sa počíta s rozšírením hlavných nastavení do takej miery, aby sa z nich dali konfigurovať aj rôzne preferencie vo vrstve Model. Pre toto rozšírenie by sa však museli upraviť oblasti vrstvy Model, aby tieto konfigurácie dokázali prijímať.

3.1.3 Konfigurácia výškových dát

Konfigurácia výškových dát má špecifický spôsob využitia. Je určená na to byť otváraná pomocou interakcie, vytvorenej inou časťou aplikácie. Jej hlavnou funkciou je dodávanie mechanizmu sťahovania a mazania výškových dát, ktoré je možné využiť ako dodatočný informačný zdroj pri vytváraní mapových reprezentácií.

Výškové dáta sú sťahované zo špecifických zdrojov. Zdroje výškových dát môžu obsahovať viacero dátových distribúcií. Tie sa môžu líšiť v kvalite, presnosti či dostupnosti dodávaných výškových dát.

Manipulácia s dátami je vykonávaná po oblastiach zvaných *regióny*. Veľkosť a tvar regiónov si každá distribúcia dát definuje sama.

V niektorých prípadoch zdroj výškových dát môže vyžadovať k ich sprístupneniu autorizáciu užívateľa pomocou mena a hesla. Táto autorizácia nemusí byť požadovaná pre všetky jeho dátových distribúciach. V prípade potreby autorizácie mechanizmus zabezpečuje možnosť užívateľovi dané údaje poskytnúť. Následne sa aplikácia pokúsi na ich základe požadované dáta získať.

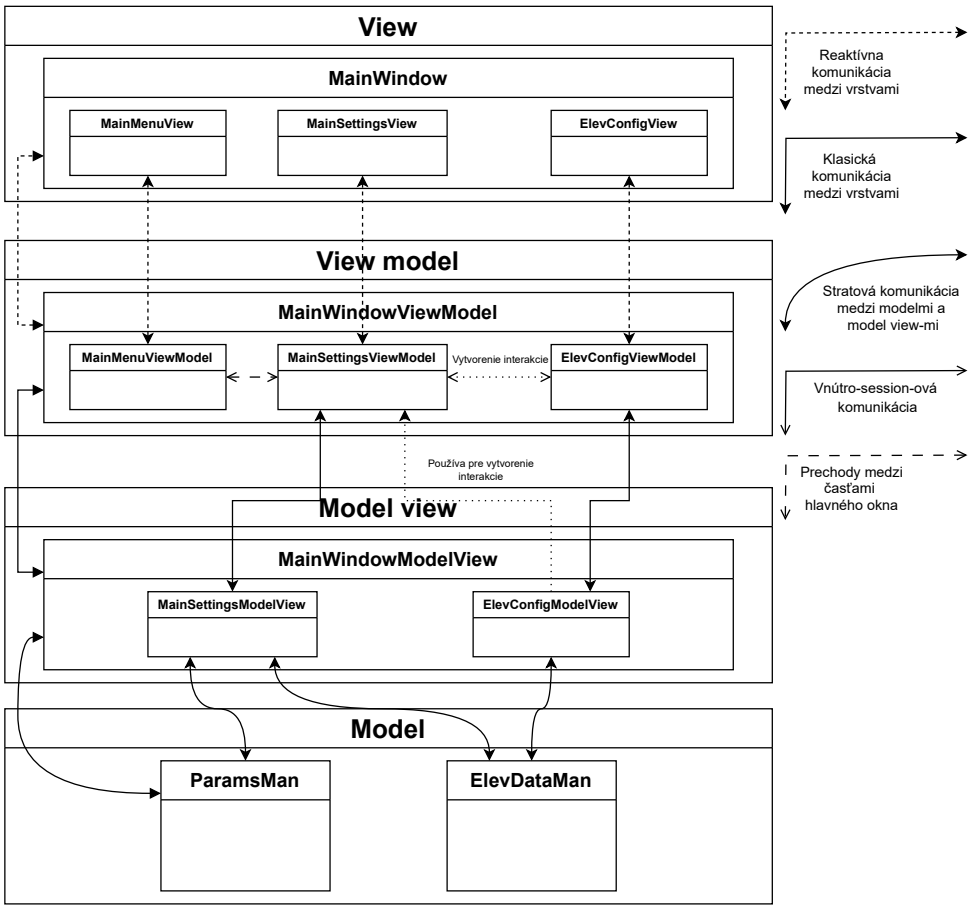
Sťahovanie a mazanie dát prebieha asynchrónne. Užívateľ je informovaný o tom, ktoré regióny su aktuálne stiahnuté, sťahované, neprítomné a odstraňované. Viacero regiónov môže byť sťahovaných/odstraňovaných naraz, či už z jedného alebo rôznych dátových zdrojov. Správne asynchrónne fungovanie manipulácie s dátami zaručujú implementácie zdrojov výškových dát v Model vrstve.

Špecifickou vlastnosťou konfigurácie výškových dát je možnosť ju používať súčasne z viacerých miest v aplikácii. Je navrhnutá tak, aby zvládala korektne akceptovať pokyny z mnohých interakcií naraz. Vďačí za to návrhu model view-u, ktorý poskytuje informácie o tom, v akom štádiu sťahovania sú jednotlivé regióny. Vďaka tomu všetky naviazané view model-y získajú aktuálne informácie a môžu správne korigovať manipuláciu s výškovými dátami.

Spôsob používania konfigurácie výškových dát je následovný:

- Pri inicializácii interakcie je novo vytvorenému view model-u(následne použitému v interakcii) predaná aktuálne využívaná distribúcia výškových dát.

- Tá je v konfigurácii nastavená ako aktuálne konfigurovaná a ukázaná pomocou view-u užívateľovi.
- Následne prichádza fáza samotného konfigurovania výškových dát. Užívateľ môže:
  - Zmeniť aktuálne konfigurovanú distribúciu výškových dát.
  - Sťahovať a odstraňovať výškové dáta aktuálne vybranej distribúcie. Tieto úkony sú uskutočňované na základe regiónov, definovaných konfigurovanou distribúciou.
  - Zadať autorizačné údaje pre možnosť využitia dát zo špecifických distribúcií, ktoré autorizáciu vyžadujú.
  - Ukončiť konfiguračnú interakciu.
- Pri ukončení interakcie sa posledne konfigurovaná distribúcia vráti ako novo zvolená na používanie.



Obrázek 3.1 Diagram popisujúci architektúru hlavného okna aplikácie.

3.2 Session pre vyhľadávanie ciest v mapách

Vyhľadávanie ciest v mapách je hlavnou náplňou aplikácie. V tejto sekcii popíšeme typ session-u, ktorý zabezpečuje logiku za doručovaním tejto služby. Mechanizmus hľadania cesty v mapách zahrnuje:

- výber vstupných parametrov tak, aby ich kombinácia bola platná,
- vytvorenie grafickej reprezentácie mapy,
- vytvorenie mapovej reprezentácie, v ktorej sa bude vyhľadávať cesta,
- umožnenie užívateľovi zadať trať, na ktorej sa má cesta vyhľadať,
- spustenie implementácie vybraného vyhľadávacieho algoritmu na zvolenej trati a vykreslenie nájdenej cesty.

V aktuálnej podobe sa mechanizmus vyhľadávania cesty skladá z troch častí:

- Nastavenie parametrov hľadania cesty (template, mapa, užívateľský model a vyhľadávací algoritmus).
- Vytvorenie mapovej reprezentácie na základe vybranej mapy a template-u. Prípadne za pomoci prítomných výškových dát. Táto časť je navrhnutá pre použitie v interakcii. Interakciu iniciuje časť nastavovania parametrov.
- Spúšťanie zvoleného vyhľadávacieho algoritmu na vytvorenej mapovej reprezentácii za využitia vybraného užívateľského modelu. Zahrňuje prijímanie trate od užívateľa a vykreslovanie nájdenej cesty.

Časť, ktorá sa z časových dôvodov nedostala do mechanizmu hľadania cesty je tzv. *relevance-feedback* mechanizmus. Ten by slúžil pre užívateľa na dodatočné nastavenie hodnôt vybraného užívateľského modelu na základe jeho preferencií vzhľadom ku aktuálne vybranej mape. Z tejto časti zostal v aplikácii len odpovedajúci model view, ktorý v tejto chvíli nenesie žiadnu užitočnú funkcionalitu. Slúži iba pre prenos dát v Model view vrstve. Bol v aplikácii ponechaný z dôvodu plánovaného budúceho rozšírenia aplikácie o relevance-feedback mechanizmus.

V nasledujúcich podsekciách popíšeme časti, z ktorých je aktuálne session tvorený. Na konci tejto sekcie je zobrazený diagram 3.2 reprezentujúci jeho architektúru.

3.2.1 Nastavenia parametrov pre vyhľadávanie cesty

Nastavenia parametrov sú prvou časťou, ktorá je užívateľovi po vytvorení session-u predostretá. Ten si za jej pomoci zvolí:

- template atribútov, ktoré budú extrahované do mapovej reprezentácie,
- mapový súbor, na základe ktorého sa bude vytvárať mapová reprezentácia. Teda súbor s mapou, na ktorej bude prebiehať vyhľadávanie ciest.
- súbor s užívateľským modelom, ktorý bude používaný algoritmom na agregáciu hodnôt z atribútov uložených v mapovej reprezentácii

- vyhľadávací algoritmus použitý na hľadanie ciest v mapovej reprezentácii

Na začiatku sa predvolia parametre na tie, ktoré boli použité v predchádzajúcej inštancii session-u. Uložené parametre prežívajú aj samotné behy aplikácie.

Vyberanie parametrov sa musí riadiť istými pravidlami z dôvodu závislostí jednotlivých typov objektov, popísaných v podsekciách sekcie 1.1. Danými pravidlami sú:

- Jednotlivé parametre sa nastavujú postupne. Najprv je nutné, aby bol vybraný mapový súbor a template. Následne môže byť vybraný aj súbor s užívateľským modelom. Keď sú všetky tri predchádzajúce položky vybrané, je možné vybrať vyhľadávací algoritmus.

- Pre zvolenú kombináciu template-u a formátu mapového súboru musí existovať mapová reprezentácia, ktorá túto kombináciu dokáže spracovať. Taktiež pre zvolený template-u musí existovať typ užívateľského modelu, ktorý dokáže spracovávať atribúty definované týmto template-om. Nakoniec musí existovať aspoň jedna kombinácia takto definovanej mapovej reprezentácie a užívateľského modelu, ktorú dokáže využiť aspoň jedna implementácie ľubovoľného vyhľadávacieho algoritmu.

  Keď je vybraný buďto template alebo mapový súbor a výber druhej položky by spôsobil neplatnú kombináciu, prvá položka sa opäť vynuluje. Možným vylepšením by bolo zabezpečenie indikácie platných kombinácií template-ov a formátov mapových súborov.

- Súbor s užívateľským modelom môže byť vybraný len takého typu, ktorý dokáže spracovávať atribúty definované zvoleným template-om a ktorý spolu s ľubovoľnou mapovou reprezentáciou, ktorá dokáže spracovať aktuálne zvolenú kombináciu template-u a mapového formátu, je vhodnou kombináciou pre aspoň jednu implementáciu nejakého vyhľadávacieho algoritmu.

- Vyhľadávací algoritmus následne môže byť zvolený len taký, ktorý podporuje typ vybraného užívateľského modelu a niektorej mapovej reprezentácie, ktorú je možné vytvoriť z vybraného template-u a mapového súboru.

Po výbere mapového súboru sa ihneď z agregovanej mapy vytvori jej grafická reprezentácia a jej náhľad sa zobrazí užívateľovi.

Po dosadení všetkých parametrov môže užívateľ pokračovať do ďalšej časti mechanizmu, ktorou je vytváranie mapovej reprezentácie. V okamžiku prechodu do tejto časti sa taktiež uložia aktuálne nastavené parametre, aby mohli byť znovu použité ako predvolené v následujúcich inštanciách session-u.

Proces vytvárania mapovej reprezentácie je vykonaný za pomoci interakcie z parametre-nastavovacej časti. Na základe jej výsledku sa následne buď presunieme v mechanizme ďalej do cestu-vyhľadávacej časti (úspešné vytvorenie) alebo zostaneme v nastaveniach.(neúspešné vytvorenie).

3.2.2 Vytváranie mapovej reprezentácie

Vytváranie mapovej reprezentácie je akási prechodová časť medzi nastaveniami a samotným vyhľadávaním ciest v mape. Je prevedená za pomoci interakcie

iniciovanej v nastaveniach, po dosadení všetkých potrebných parametrov. Táto interakcia je spracovaná pomocou dialógového okna. Pomocou neho môže užívateľ pozorovať priebeh tvorby mapovej reprezentácie a aktívne sa zapájať pri riešení vzniknutých problémov.

Priebeh tvorby mapovej reprezentácie je nasledovný:

- Ihneď po inicializácii tejto časti sa spustí proces kontroly podmienok tvorby mapovej reprezentácie. Aktuálne je využívaná iba kontrola prítomnosti potrebných výškových dát pri procese tvorenia mapovej reprezentácie. (Táto kontrola zahrňuje aj schopnosť mapy dodať informácie o svojej pozícii a rozlohe, ktoré sú nevyhnutné pre získanie odpovedajúcich výškových dát).

  Pokiaľ tvorená mapová reprezentácia indikuje potrebu výškových dát, skontroluje sa, či sú odpovedajúce dáta prítomné. Pokiaľ dáta prítomné niesu alebo mapa nie je schopná definovať svoju geografickú polohu/rozlohu, vytváranie mapovej reprezentácie zlyhá a užívateľ sa môže vrátiť do nastavení.

  Do budúcna je v pláne umožniť užívateľovi riešenie problému nedostatku výškových dát za pomoci interakcie s konfiguráciou výškových dát. Viac informácii o konfigurácii výškových dát je možné nájsť v podsekcii 3.1.3.

- Pokiaľ kontrola podmienok dobehne úspešne, je automaticky spustený proces vytvárania mapovej reprezentácie. Tento proces môže trvať dlhšiu dobu a teda je užívateľovi umožnené sledovať jeho vývoj alebo ho dokonca prerušiť. Po prerušení je užívateľ vrátený naspäť do nastavení parametrov.

3.2.3 Vyhľadávanie cesty v mape

Po úspešnom vytvorení mapovej reprezentácie sa môžeme presunúť k samotnému vyhľadávaniu ciest v mape. V tejto chvíli máme k dispozícii všetky potrebné dátové zdroje k správnemu vykonávaniu tejto činnosti.

Na začiatku je vykreslená grafika mapy, ktorá bola vytvorená v nastaveniach pri výbere mapového súboru.

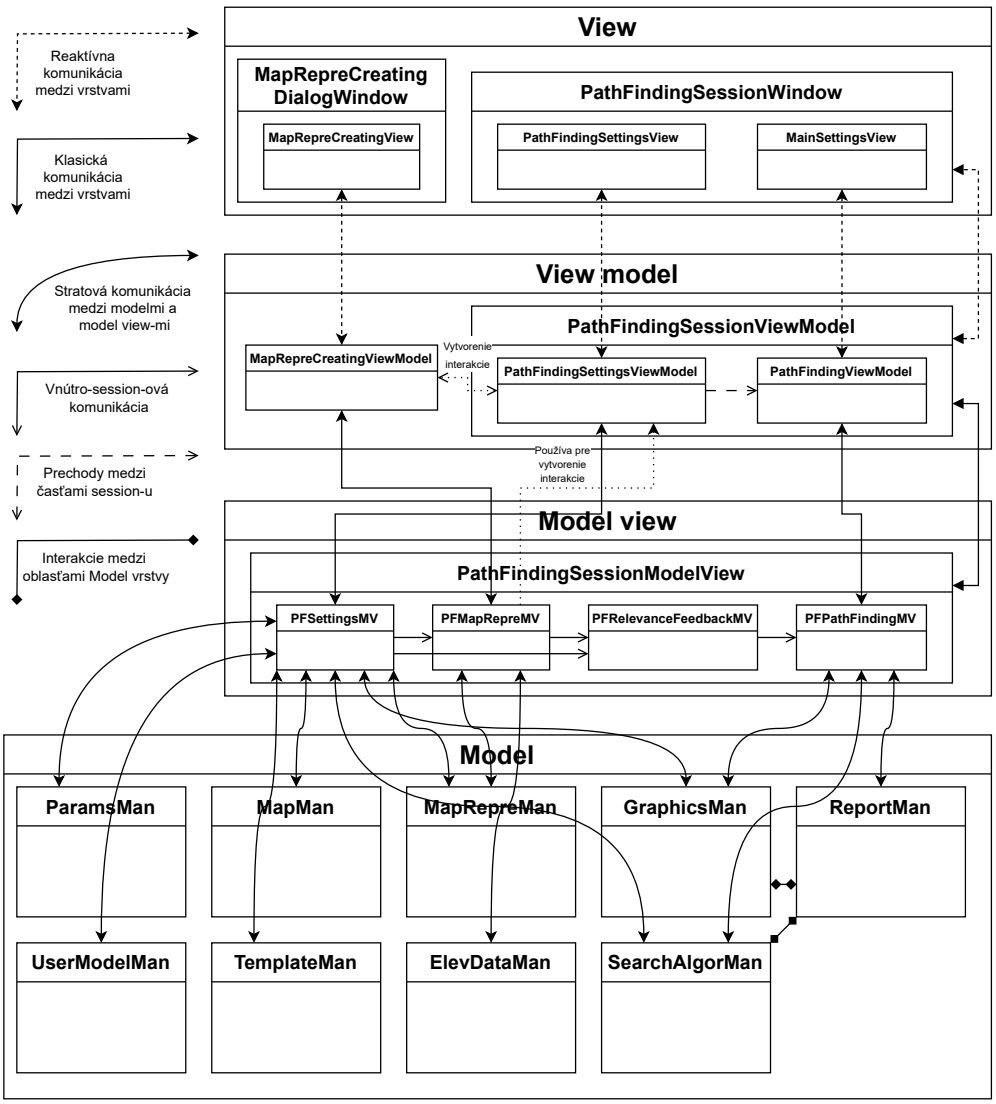
Kolobeh vyhľadávania je rozdelený do troch fáz:

- Prvou fázou je umožnenie výberu trate užívateľovi. Trať sa skladá z postupností bodov, medzi ktorými je následne vyhľadávaná cesta. Užívateľ môže pridávať a odoberať koncové body trate.

- Po zadaní cesty prichádza na rad fáza samotného vyhľadávanie cesty. Algoritmus má možnosť podávať správy o postupe vyhľadávania, či už pomocou textovej informácie alebo grafického znázorňovania. Užívateľ mám možnosť proces vyhľadávania predčasne ukončiť. V takom prípade sa kolobeh vráti do prvej fázy výberu trate.

- Pokiaľ vyhľadávania cesty dobehne úspešne, príde na rad tretia fáza, ktorou je vykreslenie nájdenej cesty. Zároveň sa po strane môžu vypísať informácie o nájdenej trase (napríklad jej dĺžka, prekonané prevýšenie, atď.). V tejto fáze je užívateľovi umožnené interagovať s nájdenými cestami (s takými, ktoré takúto funkcionalitu zabezpečujú). Po dokončení prezerania nájdenej cesty sa opäť vrátime do prvej fázy výberu trate.

Počas ktorejkoľvek fázy je užívateľ schopný približovať, odďalovať a hýbať so zobrazenou grafikou vyhľadávania.



Obrázek 3.2 Diagram popisujúci architektúru session-u pre vyhľadávanie cesty v mapách

4 Súčasná podoba vrstvy Model

Model vrstva je štvrtou vrstvou MVVM(MV) architektúry. Táto vrstva je pre zvyšné vrstvy zdrojom dát a mechanizmov, ktoré tieto dáta spracovávajú. Delí sa do mnohých oblastí. Pre komunikáciu s týmito oblasťami sú vytvorení takzvaní *manažéri*. Manažéri sú pristupovaný z Model view vrstvy a doručujú jej svoje služby, či už informatívne alebo výpočtové. Viac informácií o vrstve Model je možné nájsť v podsekcii 2.1.4.

Implementáciu Model vrstvy z veľkej časti poznačila už spomínaná strata typovej informácie pri komunikácii s Model view vrstvou. Väčšina oblastí sa s touto stratou vysporadúva podobným spôsobom:

- **Komunikácia smerujúca z vrstvy Model** - Pre všetky dátové štruktúry, ktoré sa využívajú mimo vrstvy Model, sú vytvorení predkovia, ktorí sú buďto negenerickí alebo obsahujú kovariantné generické parametre. Tým pádom vedia byť prenášaní negenerickým spôsobom mimo vrstvy Model.
- **Komunikácia smerujúca do vrstvy Model** - Pre dátové štruktúry, ktorých typovú informáciu je potrebné v Model vrstve opäť nadobudnúť, bol vytvorený tzv. *Generic visitor pattern*. V prípadoch, kedy nie je potrebné poznať ich presný typ, sa využíva klasické typové testovanie cez operátor *is*.

Definice 1. ***Generic visitor pattern*** *je obdoba klasického návrhového vzoru* Visitor pattern. *Pracuje na podobnom princípe, kedy na navštevovanej inštancii je volaná metóda* `Visit` *ktorej sa predá inštancia volajúcej triedy. Následne navštívená inštancia odpovie zavolaním metódy* `Accept` *na dodanej volajúcej inštancii a predaním samej seba v argumente indikuje, ktorý overload metódy* `Accept` *sa má zavolať.*

V prípade generic visitor pattern-u však volajúca trieda neimplementuje overload metódy `Accept` *pre každý možný typ navštevovanej inštancie ale len jednu generickú metódu* `Accept<T>`*. V typovom parametri* `T` *je následne predaná informácia o type navštívenej inštancie.*

V nasledujúcich sekciách sa pozrieme detailnejšie na aktuálne existujúce oblasti vrstvy Model a ich vnútorné mechanizmy.

4.1 Template-y

Táto oblasť sa stará o správu atribútových template-ov (len *template-ov* pre jednoduchosť). Viac informácií o myšlienke a funkcii template-ov je možné nájsť v podsekcii 1.1.5. Čo do obsahu je to jedna z najmenších oblastí.

Hlavným uzlom pre komunikáciu z Model view vrstvy je singleton trieda `TemplateManager`. Ten ponúka kolekciu všetkých template-ov, ktoré je možné v aplikácii použiť.

Template-y sú reprezentované pomocou tried, ktoré implementujú rozhranie `ITemplate<TVertexAttributes, TEdgeAttributes>`. Tento generický interface núti svoje implementácie, aby dosadením jeho typových parametrov indikovali typy vrcholových a hranových atribútov ktoré reprezentujú.

Ďalej v aplikácii existuje ešte aj negenerický predok `ITemplate` spomenutého rozhrania, ktorý slúži na komunikáciu mimo vrstvy Model. Definuje vlastnosti, ktoré sú potrebné pri práci s template-ami vo vonkajšom prostredí. Tento interface by nemal byť nikdy priamo implementovaný.

Template-ové triedy by mali byť implementované ako singleton-y. V aplikácii je totiž vždy využívaná len jedna inštancia každého template-ového typu a to tá zahrnutá v kolekcii template-ov v triede `TemplateManager`.

Template-y podporujú návrhový vzor *generic visitor pattern*. Ten sa svojou funkcionalitou trocha odlišuje od ostatných implementácií. Metóda `Visit` je totiž definovaná v rozhraní `ITemplate`, ale metóda `Accept` vracia typový parameter, ktorý je obmedzený na rozhranie `ITemplate<TVertexAttributes, TEdgeAttributes>`. Tento trik slúži k tomu, aby aj na premennej typu `ITemplate` bolo možné ihneď získať typy vrcholových a hranových atribútov daného template-u. Tento trik funguje na základe predpokladu, že všetky template-ové triedy implementujú rozhranie `ITemplate<TVertexAttributes, TEdgeAttributes>`.

4.2 Mapy

Táto oblasť sa stará o vytváranie a správu mapových objektov. Viac informácií o koncepte Máp je možné nájsť v podsekcii 1.1.1.

Hlavným uzlom pre komunikáciu z Model view vrstvy je singleton trieda `TemplateManager`. Tá ponúka kolekciu mapových formátov, ktorá obsahuje zástupcov všetkých mapových typov, ktoré je možné v aplikácii využiť. Každý zástupca zastupuje jeden typ mapy, jeden formát. Popri tom ponúka metódy pre vytváranie mapových inštancií a metódy pre identifikáciu mapových formátov.

Mapy sú reprezentované pomocou tried, ktoré implementujú rozhranie `IMap`. Toto rozhranie nedefinuje žiadnu zaujímavú funkcionalitu. Obsahuje pár vlastností využívaných mimo vrstvy Model pre identifikáciu mapy.

Triedy máp následne môžu implementovať niektoré z ďalších definovaných rozhraní, ktoré pridávajú svoje kontrakty. Tieto kontrakty sú zamerané na geografické lokalizovanie a rozlohu máp. Sú využívané predovšetkým pri získavaní dodatočných výškových dát odpovedajúcich konkrétnej mape. Ak si je mapový typ vedomí toho, že na vytvorenie jeho mapovej reprezentácie bude potrebná výpomoc výškových dát, mal by implementovať aspoň jeden z interface-ov, ktorý definuje informácie o geo-lokalite a rozlohe mapy.

Ako už bolo naznačené, pri mapách je potrebné, aby v aplikácii niekto zastupoval ich formáty. K tomuto slúžia triedy implementujúce trojicu rozhraní:

- `IMapFormat<out TMap>` - Je určený pre komunikáciu mimo vrstvy Model. Definuje vlastnosti, ktoré sú potrebné pri práci s mapovým zástupcom vo vonkajšom prostredí a metódu na vytváranie mapy zastupovaného typu. Tento interface by nemal byť priamo implementovaný.

- `IMapIdentifier<in TMap>` - Slúži na identifikáciu odpovedajúceho mapového formátu pre konkrétny typ mapy. Vďaka kontravariantnej povahe jeho typového parametru bude táto identifikácia fungovať správne aj pre potomkov typu `TMap`. Tento interface by nemal byť priamo implementovaný.

- **`IMapRepresentative<TMap>`** - Zastupuje jeden konkrétny typ/formát mapy. Je potomkom predošlých dvoch interface-ov - spája ich funkcionality. Tento interface je určený k tomu, aby bol priamo implementovaný mapovými zástupcami.

Na to, aby bolo možné mapový typ využiť v aplikácii, musí byť jeho zástupca zahrnutý v príslušnej kolekcii v triede `MapManager`. Z tohto dôvodu je na mieste, aby títo zástupcovia boli implementovaný ako singleton triedy.

Mapy podporujú návrhový vzor *Generic visitor pattern*. Ten je definovaný ako pre rozhranie `IMap`, tak aj pre rozhranie `IGeoLocatedMap` (pridáva kontrakt o geografickej lokalite mapy).

4.3 Mapové reprezentácie

Ďalšia z oblastí sa stará o vytváranie a spravovanie mapových reprezentácii/-grafov. Čo do obsahu aj komplexnosti sa jedná o jednu z najväčších oblastí. Viac informácií o koncepte mapových reprezentácií je možné nájsť v podsekcii 1.1.2.

Hlavným uzlom pre prácu s touto oblasťou z vonkajšieho prostredia je singleton trieda `MapRepreManager`. Ten ponúka kolekciu obsahujúcu zástupcov všetkých typov mapových reprezentácií, ktoré je možné v aplikácii využiť. Popri tom obsahuje metódy určené na

- vytváranie mapových reprezentácií,
- identifikáciu reprezentácií vytvoriteľných pre konkrétnu kombináciu typov template-u a mapy,
- detekciu potreby výškových dát pri konštrukcii mapovej reprezentácie.

Štruktúra dát, ktoré súvisia s mapovými reprezentáciami odzrkadluje dvojakosť významov mapovej reprezentácie a grafu ako bolo popísané v podsekcii 1.1.2. Každému typu mapovej reprezentácie je prisúdený konkrétny typ grafu.

Mapové reprezentácie/grafy sú v aplikácii reprezentované triedami, ktoré implementujú rozhranie **`IGraph<TVertexAttributes, TEdgeAttributes>`**. Toto rozhranie reprezentuje myšlienku grafu, ktorý zastupuje konkrétnu mapovú reprezentáciu. Jeho typové parametre definujú atribúty, ktoré sú používané v jeho vrcholoch a hranách. Obsahuje kontrakty, ktoré musia grafy všetkých typov napĺňovať. Aktuálne je to len jedna metóda vracajúca graf do jeho základného stavu.

`IGraph<TVertexAttributes, TEdgeAttributes>` má za predchodcu rozhranie **`IMapRepre`** reprezentujúce myšlienku samotnej mapovej reprezentácie. Cez toto rozhranie sú mapové reprezentácie/grafy distribuované vonkajšiemu svetu. K tomuto účelu rozhranie definuje vlastnosti využívané mimo vrstvy Model. Grafová podstata mapových reprezentácií je tým vonkajšiemu svetu ukrytá a len špecifické oblasti Model vrstvy ju znova nadobúdajú a využívajú.

Každá mapová reprezentácia/graf má množinu svojich implementácií. Implementácie sú tvorené pre konkrétne kombinácie template-ov a mapových formátov. Konkrétny typ mapovej reprezentácie môže byť vytvorený len pre takú kombináciu template-u a mapového formátu, pre ktorý je vytvorená príslušná implementácia.

V aplikácii sú naďalej prítomné ďalšie rozhrania, ktoré môžu (a mali by v čo najväčšom rozsahu) jednotlivé grafy implementovať. Tieto rozhrania definujú kontrakty týkajúce sa vlastností a funkcií generovaných grafov.

Na to aby mohla byť mapová reprezentácia, graf či implementácia v aplikácii použitá, musí pre ne existovať vhodný zástupca. Tento zástupca je následne poskytnutý na patričnom mieste.

- Zástupcovia typov mapových reprezentácií sú reprezentovaní triedami, ktoré implementujú rozhranie **`IMapRepreRepresentative<out TMapRepre>`**. Tieto triedy sú určené (podobne ako typy mapových reprezentácií) pre komunikáciu mimo vrstvy Model. Definujú vlastnosti, ktoré sú využívané vonkajším svetom pre získanie informácií o zastupovanej mapovej reprezentácii.

  Taktiež si držia indikátorovú kolekciu zástupcov všetkých implementácií zastupovanej mapovej reprezentácie. Táto kolekcia sa následne využíva pri identifikácii použiteľných kombinácií template-ov a mapových formátov a pri samotnom vytváraní mapových reprezentácií/grafov.

  Nakoľko typ mapovej reprezentácia je vždy asociovaný s nejakým typom grafu, zástupca typu mapovej reprezentácia disponuje aj referenciou na zástupcu typu daného grafu. Tento zástupca sa následne využíva v procese vytvárania mapovej reprezentácie/grafu. Taktiež špecifické oblasti vrstvy Model ho môžu využiť ku otestovaniu vlastností zastupovaného grafu.

  Taktiež toto rozhranie definuje metódy pre vytváranie zastupovaného typu mapovej reprezentácie.

  Na to, aby bolo možné typ mapovej reprezentácie/grafu využiť v aplikácii, musí byť jeho zástupca zahrnutý v príslušnej kolekcii manažéra mapových reprezentácií.

- Zástupcovia typov grafov sú reprezentovaní triedami, ktoré implementujú rozhranie **`IGraphRepresentative`**<out TGraph, TVertexAttributes, TEdgeAttributes>. Typové parametre definujú typ zastupovaného grafu a typy atribútov, ktoré sú použité v jeho vrcholoch a hranách. Tento interface nedefinuje žiadny kontrakt pre grafových zástupcov. Implementuje iba metódy, ktoré sú využívané v procese vytvárania mapovej reprezentácie/grafu. Je využívaný špecifickými oblasťami vrstvy Model ku otestovaniu vlastností zastupovaného grafu.

  Každý zástupca grafu je asociovaný s konkrétnym zástupcom mapovej reprezentácie. Ten si na jeho inštanciu drží referenciu a využíva ho v procese vytvárania mapovej reprezentácie/grafu.

- Ku reprezentácii zástupcov jednotlivých implementácií slúžia triedy, ktoré dedia od abstraktnej triedy **`ElevDataIndepImplementationRep`** alebo od abstraktnej triedy **`ElevDataDepImplementationRep`**. Tieto rozhrania disponujú dvojitou funkcionalitou:

  - Obsahujú vlastnosti indikujúce template a mapový formát, na základe ktorých je implementácia mapovej reprezentácie/grafu agregovaná. Tieto vlastnosti sú naplnením kontraktu definovaného ich predchodcom **`IImplementationIndicator`**.

– Je schopná skonštruovať (alebo nechať skonštruovať) zastúpenú implementáciu mapovej reprezentácie. Tieto triedy sa líšia predovšetkým v potrebe dodatočných výškových dát v procese tvorby danej implementácie. Schopnosť skonštruovať danú implementáciu je naplnením kontraktu jedného z rozhraní **`IImplementationElevDataIndepConstr`**, respektíve **`IImplementationElevDataDepConstr`**.

Taktiež disponujú množinou typových parametrov, z ktorých každý má svoj špecifický význam:

– `TTemplate` - definuje typ template-u, pre ktorý je zastupovaná implementácia vytvorená

– `TGraph` - definuje typ mapovej reprezentácie/grafu, pre ktorý je zastupovaná implementácia vytvorená,

– `TVertexAttributes, TEdgeAttributes` - definujú typy atribútov použitých v implementovanom grafe

– `TMap, TUsableSubMap` - tieto parametre sú jemne zavádzajúce. Prvý z nich hovorí o tom, aký typ mapy je navonok indikovaný pomocou vlastnosti mapového formátu. Druhý hovorí o tom, ktorý typ potomok typu `TMap` je v skutočnosti potrebný na vytvorenie mapovej reprezentácie. Malo by byť zaručené okolitým prostredím, že pokiaľ je nejaká mapa správneho formátu, tak je určite možné ju použiť pre vytvorenie mapovej reprezentácie.

Zástupcovia jednotlivých implementácií mapovej reprezentácie sú zahrnutí v kolekcii zástupcu tejto mapovej reprezentácie.

Bolo by vhodné, aby všetci spomenutí zástupcovia boli implementovaní ako singleton triedy. Od každého z nich je totiž za celú dobu behu programu potrebné vytvoriť iba jedinú inštanciu, ktorá je poskytovaná zvyšku aplikácie na patričnom mieste.

Poslednou súčasťou oblasti mapových reprezentácií sú triedy, ktoré reprezentujú vrcholy a hrany používané v grafoch. Jednotlivé typy sa potom líšia dodávanými vlastnosťami.

4.4 Užívateľské modely

V tejto oblasti sú vytvárané a spravované užívateľské modely. Viac informácií o koncepte užívateľských modelov je možné nájsť v podsekcii 1.1.3.

Hlavným komunikačným uzlom s touto oblasťou pre Model view vrstvu je singleton trieda `UserModelManager`. Ten ponúka kolekciu všetkých typov užívateľských modelov, ktoré je možné v aplikácii využiť. Mimo to implementuje metódy pre:

- Serializáciu a deserializáciu užívateľských modelov do/z súborov.
- Vytváranie nových inštancií užívateľských modelov.
- Všeobecnú identifikáciu typov užívateľských modelov na základe rôznych typov argumentov.

Užívateľské modely sú v aplikácii reprezentované triedami, ktoré implementujú rozhranie `IUserModel<out TTemplate>`. Prostredníctvom tohto rozhrania sú užívateľské modely využívané mimo vrstvy Model. Z tohto dôvodu definuje informatívne vlastnosti, ktoré sú potrebné prevažne vo vonkajšom svete. Popri tom definuje ešte aj kontrakty zaručujúce schopnosť serializácie užívateľského modelu. Každý užívateľský model je viazaný na konkrétny template-u, typu `TTemplate`. K tomu si každý užívateľský model nesie aj referenciu na inštanciu daného template-u.

Na to aby bol užívateľský model použiteľný vo vyhľadávacích algoritmoch, nestačí aby implementoval predošlé základné rozhranie. Je potrebné aby implementoval jeho následníka `IComputingUserModel`<out TTemplate, in TVertexAttributes, in TEdgeAttributes>. Tento interface sám o sebe nedefinuje žiadny kontrakt. Až jeho následníci definujú funkcionalitu, ktorú implementujúci užívateľský model vie ponúknuť napríklad vyhľadávaciemu algoritmu.

Jedným z týchto následníkov je interface `IWeightComputingUserModel`<out TTemplate, in TVertexAttributes, in TEdgeAttributes>. Toto rozhranie zaisťuje schopnosť užívateľského modelu, na základe dodaných vrcholových a hranových atribútov, vypočítať váhu odpovedajúcej hrany. Túto funkcionalitu by mali spĺňať všetky užívateľské modely, nakoľko veľká časť vyhľadávacích algoritmov potrebuje poznať váhu jednotlivých hrán pre správny výber postupu.

Oproti výpočtovým rozhraniam existuje v tejto oblasti aj špecifické rozhranie `ISettableUserModel`. Toto rozhranie musia implementovať všetky typy užívateľských modelov, ktoré sú určené na to, aby v nich užívateľ mohol konfigurovať nastaviteľné hodnoty (tzv.*Adjustables*) vzhľadom na svoje preferencie. Kontrakt ktorý definuje zaručuje dodanie kolekcie týchto nastaviteľných hodnôt, aby mohli byť dodané užívateľovi a ten s nimi mohol pracovať. Mechanizmus vytvárania a nastavovania užívateľských modelov zatiaľ v aplikácii nie je implementovaný, a teda toto rozhranie aktuálne čaká na svoje využitie.

V aplikácii je potrebné, aby existencia jednotlivých typov užívateľských modelov bola nejakým spôsobom zastúpená. Týmito zástupcami sú triedy implementujúce následujúcu trojicu rozhraní:

- `IUserModelType<out TUserModel, out TTemplate>` - Je určený pre komunikáciu mimo vrstvy Model. Definuje vlastnosti, ktoré sú potrebné pri práci so zástupcom užívateľského modelu vo vonkajšom prostredí.

  Taktiež obsahuje referenciu na template, na ktorý je zastupovaný užívateľský model viazaný a definuje metódy slúžiace na deserializáciu a vytváranie nových užívateľských modelov. Implementácie deserializácie zástupcu a serializácie zastupovaného užívateľského modelu sa musia zhodovať.

  Tento interface by nikdy nemal byť priamo implementovaný.

- `IUserModelTemplateBond<in TTemplate>` - Reprezentuje väzbu užívateľského modelu a template-u. Vďaka kontravariantnej povahe jeho templateového typového parametru bude táto identifikácia fungovať správne aj pre prípadných potomkov typu `TTemplate`. Tento interface by nikdy nemal byť priamo implementovaný.

- `IUserModelRepresentative<TUserModel,TTemplate>` - Zastupuje jeden konkrétny typ užívateľského modelu viazaného na jeden konkrétny typ

template-u. Je potomkom predošlých dvoch interface-ov - spája ich funkcionality. Tento interface je určený k tomu, aby bol priamo implementovaný zástupcami užívateľských modelov.

Na to, aby bolo možné typ užívateľského modelu využiť v aplikácii, musí byť jeho zástupca zahrnutý v príslušnej kolekcii triedy `UserModelManager`. Z tohto dôvodu je na mieste, aby títo zástupcovia boli implementovaný ako singleton-y.

4.5 Vyhľadávacie algoritmy

Táto oblasť zahrnuje mechanizmy spravujúce algoritmy pre vyhľadávanie ciest v mapových reprezentáciách. Viac informácií o vyhľadávacích algoritmoch samotných je možné nájsť v podsekcii 1.1.6.

Hlavným komunikačným uzlom s touto oblasťou pre Model view vrstvu je singleton trieda `SearchingAlgorithmMan`. Táto trieda zverejňuje v kolekcii všetky použiteľné vyhľadávacie algoritmy aplikácie. Popri tom obsahuje metódy zabezpečujúce:

- spúšťanie procesu vyhľadávania cesty na základe dodaného užívateľského modelu a mapovej reprezentácie
- dodávanie *executor*-u vyhľadávacieho algoritmu
- identifikáciu vyhľadávacích algoritmov spustiteľných pre konkrétne kombinácie mapových reprezentácií a užívateľských modelov

Vyhľadávacie algoritmy sú v aplikácii reprezentované pomocou tried, ktoré implementujú rozhranie **`ISearchingAlgorithm`**. Každý algoritmus môže byť implementovaný niekoľkými spôsobmi. Rozhranie preto definuje kolekciu, v ktorej by mali byť všetky použiteľné implementácie daného algoritmu zverejnené. Ďalej definuje a zároveň implementuje metódy, ktoré slúžia na:

- testovanie, či zástupcovia typov mapovej reprezentácie a užívateľského modelu zastupujú použiteľnú kombináciu pre daný algoritmus. Teda či existuje implementácia algoritmu, pre ktorú sú vlastnosti zastupovaných typov dostatočné na použitie,
- samotné spúšťanie vyhľadávacieho procesu. K tomuto účelu sa vyberie pre vstupné argumenty vhodná implementácia algoritmu,
- možnosť získania *executor*-u daného algoritmu. Executor sa vytvorí na základe vhodnej implementácie algoritmu.

Inštancia každého použiteľného vyhľadávacieho algoritmu musí byť obsiahnutá v kolekcii vyhľadávacích algoritmov v príslušnom manažérovi. Inak nebude aplikácia daný vyhľadávací algoritmus registrovať.

Implementácie vyhľadávacích algoritmov sú reprezentované triedami, ktoré implementujú rozhranie **`ISearchingAlgorithmImplementation`**. Toto rozhranie definuje podobné myšlienky funkcionalít tým z rozhrania `ISearchingAlgorithm`: testovanie typov vstupných mapových reprezentácií a užívateľských modelov,

spúšťanie vyhľadávacieho procesu a vytváranie svojích executor-ov. V tomto prípade však nie je táto funkcionalita implementovaná rozhraním a je potrebné aby ju implementácie algoritmov doplnili sami. Na to aby implementácia algoritmu mohla byť použitá, musí byť obsiahnutá v kolekcii implementácií odpovedajúceho vyhľadávacieho algoritmu.

Výsledkom vyhľadávania je inštancia triedy, ktorá implementuje rozhranie `IPath<out TVertexAttributes, out TEdgeAttributes>`. Toto rozhranie reprezentuje nájdenú cestu algoritmom pričom môže v sebe niesť atribúty typov `TVertexAttributes` a `TEdgeAttributes`. Neskôr v Model view vrstve je z tejto cesty vytvorený report, ktorý je následne vyššími vrstvami spracovaný a predvedený užívateľovi. Pre komunikáciu mimo Model vrstvy sa využíva jeho predchodca `IPath`. Toto rozhranie by nemalo byť nikdy priamo implementované.

Vyhľadávacie algoritmy taktiež môžu počas svojho behu podávať report-y o stave vyhľadávania prostredníctvom objektov tried, ktoré implementujú rozhranie `ISearchingState<out TVertexAttributes, out TEdgeAttributes>`. Algoritmus nechá z týchto stavov agregovať report a podá ho ku následnému spracovaniu a predvedeniu užívateľovi.

Obidve vyššie spomenuté rozhrania taktiež podporujú návrhový vzor *generic visitor pattern*.

Pokiaľ je po algoritme požadované vytvorenie jeho executor-u, algoritmus zavolá vhodnú svoju implementáciu nech executor vytvorí. Tá ho inicializuje pomocou dodanej mapovej reprezentácie, užívateľského modelu a pridá delegáta na svoju špecifickú metódu zabezpečujúcu beh algoritmu pre executor.

Všetky triedy reprezentujúce vyhľadávacie algoritmy a ich implementácie by mali byť implementované ako singleton triedy. Ich inštancie budú totiž vytvorené v aplikácii len jeden krát.

4.6 Výškové dáta

Oblasť pre správu a manipuláciu s výškovými dátami. Viac informácií o funkcii výškových dát v aplikácii je možné nájsť v podsekcii 1.1.4.

Hlavným komunikačným uzlom s touto oblasťou z vrstvy Model view je singleton trieda `ElevDataManager`. Tá ponúka kolekciu všetkých použiteľných zdrojov výškových dát v aplikácii. Popri tom doručuje metódy pre manipuláciu s výškovými dátami (sťahovanie a odstraňovanie) a taktiež metódy, ktoré testujú prítomnosť a vracajú prítomné výškové dáta odpovedajúce rozlohe konkrétnej mapy.

Výškové dáta sú v aplikácii reprezentované triedami, ktoré implementujú rozhranie `IElevData`. Toto jednoduché rozhranie definuje metódy, ktoré dokážu ku zadanej geografickej polohe vrátiť jej nadmorskú výšku. Inštancie týchto tried su na mieru vytvárané tak, aby dokázali dodať výškové dáta odpovedajúce polohe a rozlohe konkrétnej mapy.

Výškové dáta sú dodávané jednotlivými zdrojmi. Tie sú v aplikácii reprezentované triedami, ktoré implementujú rozhranie `IElevDataSource`. Zdroje výškových dát sú zložené z viacerých distribúcií, ktoré následne už dodávajú potrebnú funkcionalitu pre prácu s nimi spravovanými, výškovými dátami. Preto rozhranie

`IElevDataSource` definuje kolekciu, v ktorej by mali byť uložené všetky distribúcie daného zdroja na to, aby mohli byť v aplikácii použité. Popri tom definuje ďalších pár vlastností, ktoré sú využívané mimo vrstvy Model (napr. meno daného zdroja).

Jednotlivé distribúcie výškových dát sú v aplikácii reprezentované triedami, ktoré implementujú rozhranie **`IElevDataDistribution`**. Každá z týchto tried je zodpovedné za prácu s konkrétnou distribúciou výškových dát. Zabezpečujú sťahovanie, odstraňovanie a informovanie o ich prítomnosti. Je ponechané na zodpovednosti implementácií, akým spôsobom budú dáta ukladané, načítané do pamäte a spracovávané do inštancií implementácií rozhrania `IElevData`.

Rozhranie `IElevDataDistribution` by nemalo byť implementované priamo. Namiesto toho by mali byť implementované rozširujúce rozhrania **ICredentialsNotRequiringElevDataDistribution** a **ICredentialsRequiringElevDataDistribution** ktoré pridávajú samotnú metódu umožňujúcu sťahovanie výškových dát. Tieto metódy, resp. rozhrania, sa líšia v potrebe autorizácie pri získavaní výškových dát zo vzdialených zdrojov.

Manipulovanie s výškovými dátami prebieha po takzvaných *regiónoch*. Každá distribúcia si tvar a veľkosť svojich regiónov určuje sama. Následne tieto regióny sprostredkováva v kolekcii `AllTopRegions`, ktorá je definovaná rozhraním `IElevDataDistribution`.

Regióny sú reprezentované triedami, ktoré dedia od abstraktnej triedy **`Region`** a jej potomkov **`TopRegion`** a **`SubRegion`**. Región ako taký definuje svoje meno, svoj tvar za pomoci reťazca geografických súradníc a množinu svojich pod-regiónov. Taktiež definuje identifikátor svojej prítomnosti. Teda toho, či sú jemu odpovedajúce výškové dáta stiahnuté v počítači a pripravené na použitie alebo nie.

Tento indikátor by mal byť aktualizovaný vždy keď sa prítomnosť dát regiónu zmení. Dôležité však je, že táto informácia by mala byť zachovaná naprieč behmi aplikácie. Teda ak sú v jednom behu aplikácie stiahnuté výškové dáta pre konkrétny región, v následujúcom behu by mal región indikovať, že sú jemu zodpovedajúce dáta stále k dispozícii.

Regióny sú naďalej delené na *vrcholové regióny* a *pod-regióny*. Pod-regióny sú vždy viazané na nejaký vyšší región a reprezentujú nejakú jeho časť. Vrcholový región potom jednoducho nie je nikoho pod-regiónom. Spomínaná kolekcia `AllTopRegions` obsahuje práve vrcholové regióny definované danou distribúciou.

4.7 Grafika

Táto oblasť sa zaoberá problematikou vytvárania objektov, z ktorých sa skladajú grafické reprezentácie rôznych dátových štruktúr. Agregácia grafických objektov sa vykonáva špecifickým asynchrónnym postupom. Vytvorené grafické objekty sa plnia do tzv *kolektoru*. Tým pádom je možné vytvorené objekty paralelne spracovávať a zobrazovať ihneď po ich vytvorení.

Aplikácia obsahuje hneď dva hlavné uzly pre komunikáciu s touto oblasťou:

- `GraphicsManager` je hlavným uzlom pre komunikáciu prichádzajúcu z Model view vrstvy. Zabezpečuje agregáciu grafických objektov pre rôzne dátové štruktúry aplikácie ako sú napríklad mapy.

- `GraphicsSubManager` slúži k rovnakému účelu ako `GraphicsManager`, avšak pre komunikáciu priamo z vrstvy Model. Reprezentuje prívetivejší spôsob komunikácie so zachovaním typových informácií. Dátové štruktúry, pre ktoré zabezpečuje táto trieda agregáciu grafiky, sú napríklad nájdené cesty a stavy vyhľadávacích algoritmov.

Obidve tieto triedy sa riadia návrhovým vzorom singleton.

Grafické objekty sú v aplikácii reprezentované triedami, ktoré implementujú rozhranie `IGraphicObject`.Toto rozhranie neimplementuje takmer žiadnu funkcionalitu okre podpory návrhového vzoru *Generic visitor pattern*.

Aby bolo možné grafiku dodanej mapy, cesty či stavu extrahovať, musí pre ňu existovať trieda implementujúca rozhranie `IGraphicsAggregator`, teda presnejšie jedného z jeho špecializovaných potomkov. Títo *agregátori* následne dostanú objekt na spracovanie a kolektor, do ktorého sa majú naklásť vytvorené grafické objekty.

V prípade ciest a stavov vyhľadávacieho algoritmu dostane agregátor aj užívateľský model, ktorý môže využiť na výpočet niektorých hodnôt z vrcholových a hranových atribútov uložených v dodaných cestách/stavoch. Je však potrebné zdôrazniť, že užívateľský model možno nebude schopný tieto služby doručiť. V takom prípad sa musí agregátor zaobísť bez nich. Je možnosť, aby tieto vlastnosti vynucoval napríklad vyhľadávací algoritmus, ktorý pozná potreby pre extrahovanie grafiky ním používaného typu cesty či stavu vyhľadávania.

Bolo by namieste, keby každý užívateľský model dokázal z atribútov vyťažiť aspoň pozície vrcholov mapovej reprezentácie, aby bolo možné nakresliť aspoň základnú reprezentáciu nájdenej cesty či stavu.

Nakoniec sú v tejto oblasti definované dve rozhrania ktoré slúžia pre reprezentáciu grafického zdroja. `IGraphicsSource` definuje jedinú vlastnosť a to zdrojovú kolekciu grafických objektov. Táto zdrojová kolekcia je typu `SourceList` patriaceho do framework-u *Reactive UI*. Vo vyšších vrstvách MVVM(MV) architektúry je následne možné tento zdrojový zoznam sledovať a reagovať na jeho aktualizácie.

Špecializácia predošlého rozhrania `IGroundGraphicsSource` dodáva ešte nutnosť, aby daný grafický zdroj definoval aj svoju rozlohu. Triedy implementujúce toto rozhranie sú často využívané ako akési základné grafiky, ktorých rozlohe sa ostatné grafické zdroje prispôsobujú.

Implementácie týchto rozhraní sú väčšinou vytvárané mimo vrstvy Model. Sú na mieru vytvorené pre konkrétnu aplikačnú logiku.

4.8 Reportovanie

Oblasti spravujúcej grafiku je architektúrou veľmi podobná oblasť vytvárajúca report-y na základe rôznych typov dátových štruktúr. V aktuálnej podobe aplikácie sú týmito štruktúrami cesty, nájdené vyhľadávacími algoritmami a stavy vyhľadávacích algoritmov. Táto oblasť veľmi často využíva služby triedy `GraphicsSubManager` pre získavanie grafiky, ktorá je pridaná do obsahu report-ov.

Oblasť obsahuje opäť dva hlavné komunikačné uzly:

- `ReportManager` je hlavným uzlom pre komunikáciu prichádzajúcu z Model view vrstvy. Aktuálne zabezpečuje vytváranie report-ov pre nájdené cesty vyhľadávacím algoritmom.

- `ReportSubManager` slúži k rovnakému účelu ako `ReportManager`, avšak pre komunikáciu priamo z vrstvy Model. Reprezentuje prívetivejší spôsob komunikácie so zachovaním typovej informácie. Aktuálne zabezpečuje vytváranie report-ov pre nájdené cesty a stavy vyhľadávaní.

Obidve tieto triedy sa riadia návrhovým vzorom singleton.

Pre cesty a stavy vyhľadávania sú reporty v aplikácii reprezentované triedami, ktoré implementujú rozhrania **`IPathReport`** a **`ISearchingReport`**. Tieto rozhrania nedefinujú takmer žiadnu funkcionalitu, okrem podpory návrhového vzoru *generic visitor pattern*.

Podobne ako v oblasti spravujúcej grafiku, na to, aby pre konkrétny typ dátovej štruktúry mohol byť report vytvorený, musí preňho existovať vhodná trieda implementujúca rozhranie **`IReportAggregator`**, resp. jedného z jeho špecializovaných potomkov. Títo *agregátori* následne dostanú dátovú štruktúru na spracovanie a v niektorých prípadoch aj užívateľský model, ktorý môžu využiť na výpočet hodnôt z vrcholových a hranových atribútov uložených v dodaných dátach. Podobne však ako u grafických agregátorov, nie je zaručené, že užívateľský model bude schopný požadované služby doručiť.

4.9 Parametre

Posledná, jednoduchšia ale o to dôležitejšia oblasť je využívaná na spravovanie a ukladanie všemožných parametrov aplikácie. Hlavným uzlom pre komunikáciu s touto oblasťou je singleton trieda `ParamsManager`. V tejto triede je možné uložiť od každého typu parametrov práve jednu inštanciu. Táto inštancia môže byť počas behu algoritmu rôzne upravovaná. Inštancie parametrov sú uložené v slovníku pod kľúčom reprezentujúcom ich vlastný typ.

Keď je zavolaná metóda `SaveAllParams`, tak sa trieda `ParamsManager` pokúsi všetky uložené parametre v slovníku serializovať do súborov pre možnosť ich použitia v budúcich behoch aplikácie.

Pri následnom behu aplikácie sú parametre deserializované zo súborov tzv. „lenivým“ spôsobom. Keď je požiadané o parametre typu, ktorý v slovníku nie je zastúpený, trieda sa pokúsi najprv zistiť, či preňho neexistuje odpovedajúca serializácia. Ak áno, deserializuje sa inštancia odpovedajúcich parametrov, vloží sa do slovníku a vráti užívateľovi. Pokiaľ nie, poznačí sa do slovníku neexistencia parametru daného typu a navráti sa hodnota `null`.

Serializácia a deserializácia parametrov do súborov, v ktorých prežívajú beh aplikácie, prebieha za pomoci singleton triedy **`DataSerializer`**. Táto trieda serializuje objekty do súborov pomocou systémovej triedy ***`JsonSerializer`***. Súbory sú pomenované podľa typu daného serializovaného objektu. To znamená, že v jednu chvíli pre každý dátový typ dokáže táto trieda serializovať jednu jedinú inštanciu. Pri deserializácii, na základe vstupného generického typového parametru nájde súbor s odpovedajúcim menom a pokúsi sa ho deserializovať do inštancie daného typu.

Závěr

Cielom tejto bakalárskej práce bolo vytvoriť aplikáciu, v ktorej bude možné vyhľadávať cesty v otvorenom teréne na základe dodaného mapového súboru. Vyhľadávanie ciest malo byť konkrétne implementované pre mapy z prostredia športového odvetvia *orientačný beh*.

Práca mala zahrňovať návrh architektúry aplikácie a jej samotnú implementáciu, špecifickú implementáciu na spracovávanie mapových súborov z oblasti orientačného behu, vytvorenie vyhľadávacieho algoritmu, ktorý by dokázal vyhľadávať na vytvorenej mapovej reprezentácii a možnosť užívateľa zahrnúť svoje preferencie do procesu vyhľadávania.

Tieto úlohy boli čiastočne splnené. Bol vytvorený kvalitný návrh aplikácie na základe MVVM(MV) návrhového vzoru. Na jeho základe bol následne implementovaný mechanizmus zabezpečujúci logiku za vyhľadávaním ciest v mapách a taktiež mechanizmus hlavného okna, v ktorom je možné spravovať nastavenia určené pre celú aplikáciu. Následne bolo vytvorených 9 oblastí slúžiacich ku získavaniu a spracovávaniu dátových štruktúr. Celý program je navrhnutý takým spôsobom, aby bol čo najľahšie rozšíriteľný, či už v oblasti aplikačnej logiky alebo typov dátových štruktúr, s ktorými dokáže aplikácia pracovať.

Na druhej strane však nezvýšil čas na vytvorenie konkrétnej implementácie spracovávania máp z oblasti orientačného behu, vytvorenie vyhľadávacieho algoritmu a ani mechanizmu ktorým by užívateľ mohol zahrnúť svoje preferencie do procesu vyhľadávania. Konečným výsledkom je teda kvalitne vytvorená kostra ktorej chýba funkčné vnútro.

Literatura

1. SZOS. *Čo je to orientačný beh?* [online]. [cit. 2024-06-29]. Dostupné z: https://www.orienteering.sk/page/co-je-to-orientacny-beh.
2. ADEGEO; IHSANSFD; ALEXBUCKGIT; V-TRISSHORES; DCTHEGEEK. Desktop Guide (WPF .NET). 2023. Dostupné také z: https://learn.microsoft.com/en-us/dotnet/desktop/wpf/overview/?view=netdesktop-8.0.
3. DAVIDBRITCH; MHRASTEGARI; JONPRYOR; BANOVVV; JCONREY. What is .NET MAUI? 2024. Dostupné také z: https://learn.microsoft.com/en-us/dotnet/maui/what-is-maui?view=net-maui-8.0.
4. JAMES, M. Avalonia UI and MAUI - Something for everyone. 2024. Dostupné také z: https://avaloniaui.net/blog/avalonia-ui-and-maui-something-for-everyone.
5. *Welcome* [online]. [cit. 2024-06-30]. Dostupné z: https://docs.avaloniaui.net/docs/welcome.
6. *The MVVM pattern* [online]. [cit. 2024-06-30]. Dostupné z: https://docs.avaloniaui.net/docs/concepts/the-mvvm-pattern/.
7. *Reactive Programming* [online]. [cit. 2024-06-30]. Dostupné z: https://www.reactiveui.net/docs/reactive-programming/index.html.

A Přílohy

A.1 První příloha