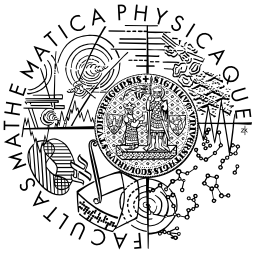


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

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

Poděkování.

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 základe 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. Tím pádem nezbyl čas na vytvoření vyhledávacích alogirtmů ani mechanizmů na spracováni map na kterých by jse cesty hledaly. Z tohto důvodu vznikla jenom 'skořápka' aplikace, která toho v týhle chvíli sama o sobě nic nedelá. Objektový návrh je za to velmi 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: objektový návrh, 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 'eggshell' of application was created which currently does not do nothing usefull. 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: object desing, MVVM, path finding in open terrain

Obsah

Úvod

Následuje několik ukázkových kapitol, které doporučují, jak by se měla závěrecná práce sázet. Primárně popisují použití TEXové šablony, ale obecné rady poslouží dobře i uživatelům jiných systémů.

1 Obecné informácie o aplikácii

1.1 Motivácia

Veľmi často v živote narážame na situácie, kedy sa potrebujeme dostať z bodu A do bodu B čo najrýchlejším spôsobom. V mnohých prípadoch mám 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 aplikácie ktoré majú podrobne zmapovanú cestnú sieť a vedia nám na základe cestných parametrov určiť najrýchlejšiu alebo najúspornejšiu trasu.

Avšak môžu nastať v živote aj situácie, kedy cestná sieť je priveľmi riedka, až takmer neprítomná. Väčšinou takéto situácie nastávajú vo voľnej prírode, v odľahlých častiach od civilizácie. V takých prípadoch nám konvenčné navigácie veľmi dobre neposlúžia. Ak máme kvalitnú, detailnú 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 software, 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. Vo všetkých je potrebné sa z času na čas dostať na miesto ‚uprostred ničoho' aby mohli byť vykonané 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 civilizácie, či už po alebo mimo cesty.

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.

1.2 Aspekty hľadania najrýchlejšej trasy (v OB)

Mapy pre orientačný beh bývajú veľmi detailné a bežec má preto dostatok informácie na to aby si mohol vybrať 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 polygonálne (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 vyhľadávať. Mapovou reprezentáciou by mal byť konštrukt, ktorý dobre vystihne topografiu mapy a umožní hľadanie najideálnejšej trasy. V našej aplikácii budú týmito konštruktami *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 znamená najrýchlejšia.

V následujúcich podsekciách spomeniem koncepty, ktoré budú v procese hľadania najrýchlejších ciest vystupovať. Medzi jednotlivými konceptami sú tvorené rozne závislosti. Grafické zhrnutie týchto závislostí je možné nahliadnuť na konci sekcie v Obrázku 1.1.

1.2.1 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 v prívetivejšej forme.

Touto formou bude práve koncept *Mapy*. 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 zahodno zmieniť že koncept mapy je odlišný od konceptu *mapovej reprezentácie*. Mapa narozdiel 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 velkosti mapového súboru.

1.2.2 Mapové reprezentácie

Mapové reprezentácie sú jednou z dôležitých zložiek procesu hľadania ciest. Sú to jednotky, na ktorých sa samotné vyhľadávanie uskutočňuje. Mapové objekty su oproti *mapám* zložitejšie konštrukty ktoré už v sebe zahrňujú plno závislostí a prepojení. Ich generovanie môže zabrať oveľa viac času ako generovanie mapových objektov.

Ako už bolo spomenuté 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 funguje vnútorne. Popisuje jeho vlastnosti, mechanizmy a spôsoby akými generuje výsledný *graf*, v ktorom sa následne hľadá cesta. Objekt ktorý je abstrahovaný z mapy je v prvom rade mapová reprezentácia a až v druhom graf.

  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 mapového spracovania.

  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 koresponduje s vyššie zmienenou mapovou reprezentáciou 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.

1.2.3 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 prispôsobené.

Užívateľské modely vďaka svojej informovanosti o užívateľských preferenciách nadobúdajú zodpovednosť za výpočty hodnôt ktoré sú závislé práve na daných preferenciách. Stávajú sa teda dôležitou zložkou procesu hľadania ciest, kde sa vyhľadávacie algoritmy stávajú závislé na nimi vykonávaných výpočtoch.

1.2.4 Výškové dáta

Jedným z dôležitých a neopomenuteľných faktorov voľby najrýchlejšieho postupu je prevýšenie, ktoré je potrebné pri jeho prevedení zdolať. Mnoho typov máp obsahuje 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 pre reprezentáciu nadmorskej výšky. Vypracovanie reliéfneho obrazu za pomoci vrstevníc samotných je veľmi ťažká úloha. V niektorých prípadoch (mapy pre OB napríklad) dokonca jednotlivé vrstevnice ani nie sú v mapovom súbore reprezentované jedným líniovým objektom. Kvôli dobrej čitateľnosti máp sú často vrstevnice 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.

1.2.5 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 samotné 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.

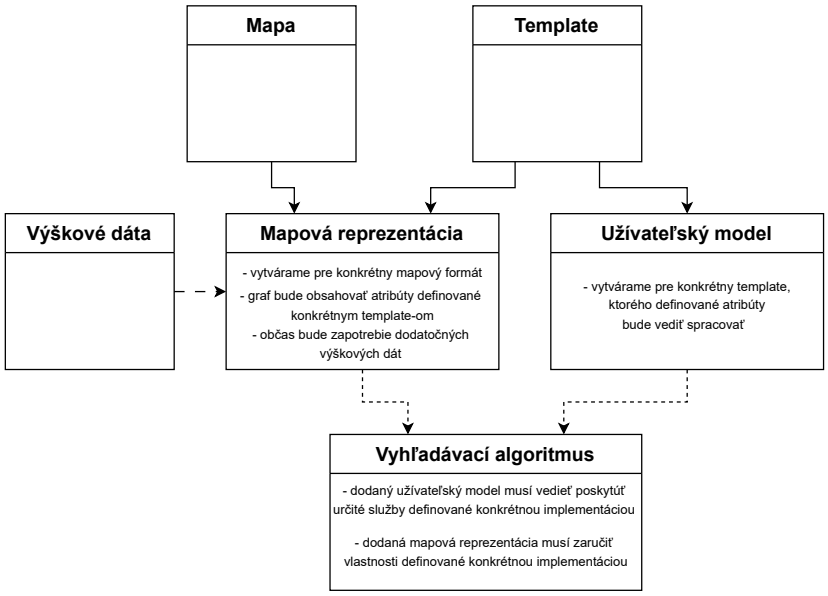
1.2.6 Vyhľadávacie algoritmy

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 hrany grafu, dodať ešte aj výpočet heuristiky využívanej v tomto algoritme.



Obrázek 1.1 Diagram závislostí jednotlivých konceptov

1.3 Zvolené prostriedky

1.3.1 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 buď typicky vysoko-úrovňový jazyk Python alebo na druhú stranu nízko úrovňový jazyk C++. Nakoľko v aplikácii bude prebiehať mnoho výpočtov, python by nebol vhodnou voľbou pre nedostatočnú rýchlosť z neho vytvoreného strojového kódu. Na druhú stranu C++ by bol veľmi dobrým kandidátom z hladiska výpočtovej sily. V tomto prípade však narážame opäť na nie veľmi dobrú znalosť tohto jazyka a na nie úplne pohodlnú prácu s ním. Pri takomto väčšom projekte sme považovali za potrebné istotu, že nás použitý jazyk podrží (ak s ním nie sme zžitý na 100%).

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

1.3.2 Užívateľské rozhranie

Pre implementáciu užívateľského rozhrania som sa rozhodol pre *Avalonia UI* framework.

V C# existuje viacero možných framework-ov 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, 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. 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 pochopiť, 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].

1.3.3 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 dostačujúcu 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.

1.3.4 Reaktívne programovanie

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

Tá samotná 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. Vrstvy model view a model sa väčšinou zaobídu bez jeho využitia, nakoľko reaktívna komunikácia prebieha medzi vrstvami view a view model.

2 Architektúra ako celok

O architektúre aplikácie sa dá premýšlať ako o *mriežke*. Je rozdelená na horizontálne (MVVM návrhový vzor) a vertikálne (*session*-y + hlavné okno) vrstvy. V nasledujúcich 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 1.3.3, v aplikácii je využívaný návrhový vzor MVVM s drobnou obmenou. Táto obmena sa týka rozdelenia originálnej vrstvy view model na dve časti: 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 nasledujúcich 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. Viewy sú viazané na zodpovedajúce view modely za pomoci reaktívneho programovania. Spracovávajú akcie užívateľa a iniciujú reakcie zvyšných vrstiev architektúry prostredníctvom naviazaného view modelu. Následne zabezpečujú grafické znázornenie ním dodaných výsledkov reakcie.

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

- **Window** - reprezentuje špecifické okno aplikácie, top-level kontajner, ktorý drží v sebe nejaký obsah. Samo o sebe veľmi nedefinuje vzhľad aplikácie. Slúži 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 obsahom okna. Naviazaný view model taktiež obsahuje vlastnosti ktoré priamo súvisia s vlastnosťami daného okna.

- **View** - sú to hlavné zložky, ktoré nesú grafiku toho, čo sa aktuálne zobrazuje v konkrétnom okne. Reprezentujú jeden obraz ktorý je užívateľovi vykresľovaný v konkrétnom okne. Každý view je naviazaný na špecifický view model. Viaže sa na jeho vlastnosti a vykresluje dáta ktoré tieto vlastnosti obsahujú.

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

- **DataTemplate** - definuje grafickú reprezentáciu dát dodávaných view modelom naviazanému view-u. 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 Modelov 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 modelu, 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 a to C# a XAML. Pomocu XAML definujeme všetky polohy a tvary grafických objektov a bindujeme 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 a disponuje vedomím toho, aké akcie, v akom poradí sa majú vykonať pre zabezpečenie patričnej reakcie na daný impulz. K tomuto účelu využíva služby nižších vrstiev prostredníctvom volania na vrstve Model view. Tá na základe svojej vnútornej logiky vráti odpoveď s požadovanými údajmi. View model následne spracuje dodané dáta a predostrie ich view-u aby ich mohla ukázať užívateľovi.

View model zároveň, 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 modelmi za účelom dodania ich doplňujúcich služieb a aplikačnej logiky do jeho vlastného procesu.

Podobne ako vo View vrstve sa view modely delia natri 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 2.2. Výnimkou je práve *Main window view model*, ktorý je naviazaný na hlavné okno aplikácie a zabezpečuje preňho aplikačnú logiku. Klasické session view modely sú taktiež naviazané zvyčajne na jedno okno z view vrstvy 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 modelu informuje session view model naviazané okno, ktoré potom v sebe zobrazuje odpovedajúci view aktívneho view modelu. Informovanie o aktívnom view modelu je aj hlavnou pracovnou náplňou session view modelu.

K ďalším jeho povinnostiam patrí spracovávanie užívateľom vyvolaných akcií, ktoré sa týkajú samotného viazaného 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. Každý view model vo väčšine prípadov spadá pod réžiu konkrétneho session-u (alebo hlavného okna), pre ktorý implementuje určitú časť jeho mechanizmu. Klasicky sú view modely naviazané na príslušné view-y z View vrstvy. S tými následne reaktívne komunikujú a reagujú na ich podnety. View modely 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 modelov by mala byť zahrnutá v zodpovedajúcom session view modele (poprípade Main window view modele). Ten potom zabezpečuje správu toho, ktorý view model je v danej chvíli aktívny. Výnimkou sú view modely, ktoré sú výhradne používane pre interakcie z iných view modelov, ktorým týmto spôsobom doručujú svoje služby.

  Tieto view modely sú väčšinou vytvorené na mieste interakcie a po jej dokončení zanikajú. Pri ich vytvorení im sú zvyčajne predané nejaké vstupné parametre a keď je ich práca dokončená, tak 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 modelov 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 vykreslenie. Data view modely môžu dodané informácie drobne upraviť takým spôsobom, aby ich bolo jednoduchšie vo View vrstve zobraziť.

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

  Niektoré dátové view modely nielen že obsahujú informácie príslušných dát ale obsahujú aj dáta samotné. Takéto dátové view modely označujeme pomocou slova *wrapping*. (tvoria akýsi obal okolo dátových inštancií). Táto funkcionalita je dôležitá hlavne v prípadoch, kedy dátový view model slúži taktiež pre spätnú komunikáciu s Model view vrstvou. V takých prípadoch musí byť možné identifikovať, ktorú dátovú inštanciu „obaluje“. Wrapping dátové view modely 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 konštruktov 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 View modelu 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 modelom.
- konverzia dát, získaných od Model vrstvy, do odpovedajúcich Data view modelov 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 2.2).

  Ich hlavnou úlohou je vytvoriť a distribuovať model view-y odpovedajúce danému session-u. Pri ich inicializácii vytvorí medzi nimi väzby, ktoré sú následne počas behu aplikácie využívané na spomínanú vnútro-session-ovú komunikáciu. Taktiež zabezpečujú spracovávanie požiadaviek pre odpovedajúce session view modely. Tieto požiadavky sa typicky týkajú akcií, ktoré súvisia so session-om ako takým (nie s nejakou jeho časťou).

  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, odpovedajúcim view modelom.

- **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 modelom požadovaných služieb. Nie je to však pravidlo, view model môže obsahovať odkazy na viacero model view-ov, ktorých služby následne využíva alebo sú predané vytvoreným, v interakciách využívaným, view modelom.

  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. 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 informačné alebo výpočtové. Predstavujú interface-y ponúkajúce prívetivejší spôsob práce s vnútornými mechanizmami modelov.

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 obsluhovania je Model jediná vrstva, ktorej konštrukty niesu viazané na žiadnu vertikálnu vrstvu (session/hlavné okno). Z tohto návrhu Model vrstvy vyplýva ešte jedna dôležitá vlastnosť modelov a to, že musia byť schopné svoje služby dodávať paralelne pre viacero session-ov.

Vrstva modelov je jednoducho rozšíriteľná o nových manažérov. Vďaka *singleton* štruktúre sú dosiahnuteľný v podstate z akéhokoľvek miesta v programe a teda nemusia by5 nikde zahrnutí. Manažéri by zasa nemali mať problém prijímať nové, primerane vytvorené implementácie mechanizmov z ich oblastí. Napríklad by nemalo byť zložité dodať nový vyhľadávací algoritmus odpovedajúcemu manažérovi, ktorý ho následne bude ponúkať zvyšku aplikácie.

Špecifickým znakom komunikácie medzi Model a Model view vrstvami je, 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 generík 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 modelov do model view-ov.

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

V modelu existujú popri manažéroch ešte aj tzv. *sub-manažéri*. Tieto entity sú ale určené pre využitie priamo z modelov. Podporujú generickú komunikáciu, na ktorej báze modely fungujú, bez straty typovej informácie a teda sú príjemnejšie pre modelovú komunikáciu než klasickí manažéri.

2.2 Session-y + *hlavné okno*

2.2.1 Session-y

Aplikácia, či už z vizuálneho, logického či 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.

Následne môžu session-y využívať dodatočné objekty z týchto vrstiev, ktoré patria 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 objektu.

Poprípade je možné využívať špecifické model viewy a view modely prostredníctvom interakcií. V takom prípade by ale dané objekty mali byť na daný účel prispôsobené (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 fázy jeho procesu. Tieto časti sú väčšinou tvorené špecifickými objektmi naprieč prvými tromi vrstvami horizontálnej štruktúry.

Session-ové inštancie by medzi sebou nemali navzájom komunikovať ani zdielať svoje dáta. Mohlo by to viesť ku problémom s paralelizáciou služieb vykonávaných Model vrstvou.

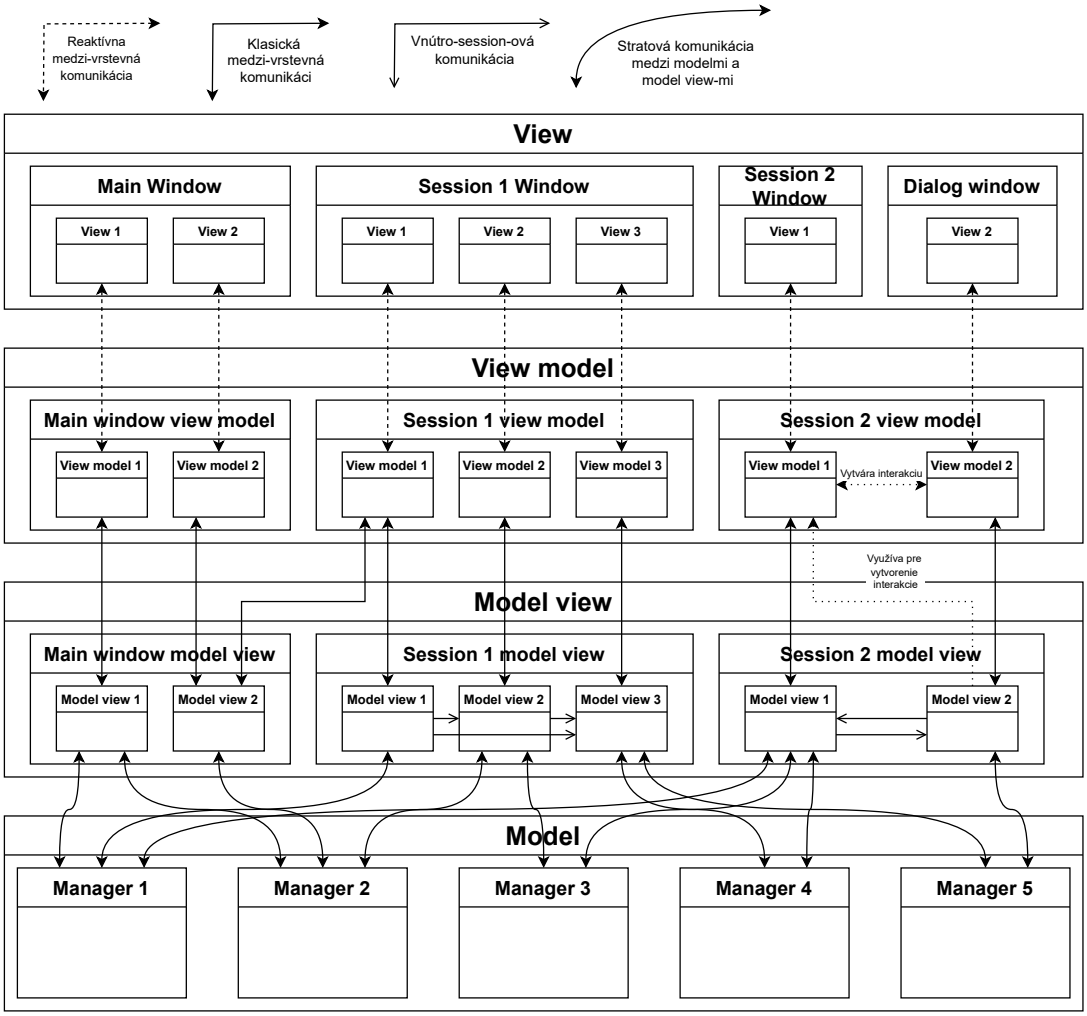
Session-y ako také reprezentujú cestu ku všeobecnej rozšíriteľnosti aplikácie o nové mechanizmy. Ak by vznikla potreba, aby aplikácia obsahovala nejaký nový mechanizmus, stačí preňho vytvoriť odpovedajúci session a upraviť hlavné okno tak, aby ho vedelo ponúknuť užívateľovi ako jednu z možností.

2.2.2 Hlavné okno

Session-y sú vytvárané a spravované *hlavným oknom*. Je to jediná perzistentná „vertikálna“ vrstva aplikácie. Beh aplikácie začína s 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 nejaké 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 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 otvorených okien, spravuje session-y pri ukončovaní aplikácie, poskytuje dodávateľa hlavných parametrov, inicializuje pri vytváraní session-u jeho view modely a model viewy.



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*). V pláne bolo taktiež vytvoriť session určený na vytváranie užívateľských modelov ale z časových dôvodov nakoniec nebol zahrnutý. Pre obecných 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 sa pozrieme na aktuálne implementovanú architektúru ako aj funkcionalitu hlavného okna.

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

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

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. Na popud 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 nastavení 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. Ak by táto potreba v budúcnosti vznikla, nemal by byť problém model view pre hlavné menu doimplementovať.

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

3.1.2 Hlavné nastavenia

Sú druhou časťou hlavného okna. Zabezpečujú možnosť pre užívateľa nastaviť parametre, ktoré sú následne aplikované na celú aplikáciu.

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 z daného zdroja).

Lokalizácia je implementovaná na základe lokalizačných .resx zdrojových súborov. Avalonia je schopná takéto zdroje využívať 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 v Model vrstve. Pre toto rozšírenie by sa však museli upraviť modeloví manažéri, aby tieto konfigurácie dokázali prijímať.

3.1.3 Konfigurácia výškových dát

Konfigurácia výškových dát ma 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 môže byť požadovaná iba pri niektorých jeho dátových distribúciach. V takom prípade mechanizmus zabezpečuje možnosť nechať užívateľa 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, neprítomne a mazané. Viacero regiónov môže byť sťahovaných naraz, či už z jednej distribúcie výškových dát alebo z rôznych. 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. Môže za to špecificky navrhnutý model view, ktorý drží informácie o tom, v akom štádiu sťahovania sú jednotlivé regióny. Jeho jediná inštancia je následne využívaná vo viacerých interakciách súčasne. Tým pádom sa do nich dostanú všetky potrebné informácie na to, aby v bolo možné správne korigovať manipuláciu s dátami.

Proces používania konfigurácie je nasledovný:

- Pri inicializácii interakcie je novo vytvorenému view modelu(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 mazať výškové dáta aktuálne vybranej distribúcie výškových dát. 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.

3.1.4 Zaujímavé implementačné postupy

V tejto podsekcii vymenujeme pár zaujímavých postupov využitých pri implementácii *hlavného okna*.

- **Zatváranie hlavného okna** - zatváranie hlavného okna je špecifické tým, že ukončuje beh celej aplikácie. Preto je v niektorých prípadoch potrebné, aby bolo možné sa opýtať užívateľa, či si je istý svojou požiadavkou na ukončenie aplikácie. Ideálnym spôsobom na zistenie užívateľovho názoru je použitie dialógového okna. Ač sa to môže zdať ako triviálna úloha, s Avalonia framework-om to zas tak jednoduché nebolo.

  Následujúci kód ukazuje nefunkčný príklad `MainWindow_OnClosing` metódy z triedy `MainWindow` určenej na zachytávanie a spracovanie udalosti zatvárania hlavného okna:

```
private async void MainWindow_OnClosing
    (object? sender, WindowClosingEventArgs e)
{
    bool close =
        await ViewModel!.OnClosingCommand.Execute();
    if (!close)
    {
        e.Cancel = true;
    }
}
```

  Na prvý pohľad by sa mohlo zdať že je všetko v poriadku. Pokiaľ spustený `ViewModel.OnClosingCommand` vráti indikáciu toho, že sa okno nesmie zavrieť, nastaví sa vlastnosť `Cancel` na true a tým sa zabráni zatvoreniu hlavného okna.

  Problém je však v tom, že spustenie daného príkazu na view modelu prebieha asynchrónne z dôvodu možnosti otvorenia dialógového okna na komunikáciu s užívateľom. Výsledok príkazu je potrebné očakávať za pomoci kľúčového slova `await`, nakoľko jeho použitie zabezpečí, že UI thread zostane aktívne a reagujúce.

To však na druhú stranu sa stáva problémom, nakoľko UI spracuje argument `e` prv než naša metóda dokáže správne dosadiť jeho vlastnosť `Cancel`. Teda okno sa zavrie skorej, než je užívateľovi daná možnosť to zvrátiť.

Preto bol použitý následujúci návrh metódy, ktorý zaručuje správny spôsob zatvárania hlavného okna:

```
private bool _alreadyAsked = false;
private async void MainWindow_OnClosing
    (object? sender, WindowClosingEventArgs e)
{
    if (_alreadyAsked) return;
    e.Cancel = true;
    bool close =
        await ViewModel!.OnClosingCommand.Execute();
    if (close)
    {
        _alreadyAsked = true;
        Close();
    }
}
```

V tomto prípade sme predošlý problém vyriešili malým trikom. Vždy keď zaznamenáme udalosť zatvárania okna iniciovanú užívateľom, nastavíme automaticky príznak `Cancel` argumentu `e` na true. Tým zabránime predčasnému zatvoreniu okna.

Následne podobne ako v predošlom prípade asynchronne zavoláme spustenie `ViewModle.OnClosingCommand` a počkáme na výsledný indikátor. Ak indikuje pokračovanie v zatváraní aplikácie, príde na radu náš trik.

Nastavíme hodnotu, k tomuto špecifickému účelu vytvoreného, privátneho poľa `_alreadyAsked` na true. Toto pole indikuje, že sa hlavné okno už raz užívateľa pýtalo na jeho názor na zatvorenie aplikácie a že jeho odpoveď bola pozitívna.

Po nastavení poľa je opäť zavolaná metóda Close() na hlavnom okne. Na základe tohto volania sa opäť hlavné okno pokúsi zavrieť. Tým pádom sa znova zavolá metóda `MainWindow_OnClosing`. Tentokrát sa však jej beh zastaví hneď na začiatku na dotaze, či pole `_alreadyAsked` indikuje už zistený užívateľov súhlas so zavretím okna. Tým pádom sa vlastnosť `Cancel` nestihne nastaviť na true a teda hlavné okno sa následne zavrie.

Tento princíp je možné využiť taktiež v ostatných oknách aplikácie, ak by mali potrebu rovnakého mechanizmu ich zatvárania.

- **Súbežné využívanie inštancie `ElevDataModelView-u`** - za možnosťou súbežného využívania model view-u konfigurácie výškových dát je malý trik. Na začiatok je potrebné si uvedomiť, ktoré dáta je potrebné držať synchrónne vo všetkých používaných konfiguráciách výškových dát a ktoré na druhú stranu majú byť pre každú konfiguráciu jedinečné:

  – Synchrónne je potrebné držať informácie o stave prítomnosti jednotlivých regiónov všetkých distribúcií. Keby sa v týchto dátach objavila akákoľvek nesúmernosť, mohlo by to mať fatálne dôsledky na manipuláciu s výškovými dátami.

– Na druhú stranu každá konfigurácia by si mala sama určovať, ktorá distribúcia výškových dát je aktuálne konfigurovaná. Predsa len to je jedným zo zámerov procesu konfigurácie. Nechať užívateľa vybrať ním žiadanú distribúciu na použitie či upravenie.

Teda potrebujeme, aby inštancia `ElevDataModelView` v sebe držala informáciu o prítomnosti jednotlivých regiónov pre všetky distribúcie. Pre konkrétny región je táto informácia uložená v data view modele, do ktorého je príslušný región zabalený (viac informácií ku data view modelom nájdete v podsekcii 2.1.2). Samotný región si nesie informáciu len o tom, či sú preňho dáta stiahnuté alebo nie. Neinformuje ale o tom, čí sú dáta aktuálne sťahované alebo odstraňované.

Typicky model view-y vytvárajú vždy nový data view model pre každý údaj získaný z Model vrstvy v momente posúvanie jeho informácie do vrstvy View model. Tu však prichádza na rad malý trik, kedy `ElevDataModelView` vytvorí data view modely pre všetky existujúce regióny počas svojej inicializácie a uloží ich do slovníka `TopRegionsOfAllDistributions`. Následne je tento slovník ponúkaný všetkým inštanciám triedy `ElevConfigViewModel` a teda všetky tieto inštancie pracujú s jednými a tými istými objektmi regiónových data view modelov.

Tým pádom všetky inštancie triedy `ElevConfigViewModel` zdieľajú informácie o stave prítomnosti jednotlivých regiónov a teda nemôže nastať nekonzistencia v akciách manipulujúcich s výškovými dátami. (samozrejme za predpokladu, že užívateľ nie je schopný stlačiť dve tlačidlá v rôznych konfiguračných oknách v ten istý moment. Pri klasickom používaní aplikácie by tento scenár nikdy nemal nastať).

Na druhú stranu každá inštancia triedy `ElevConfigViewModel` si sama drží informáciu o aktuálne konfigurovanej distribúcii výškových dát a teda v každej z týchto inštancií môžem v rovnakom čase konfigurovať inú distribúciu.

3.2 Session pre vyhľadávanie ciest v mapách

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

- výber vstupných parametrov tak aby ich kombinácia bola validná
- vytvorenie grafickej reprezentácie mapy
- vytvorenie mapovej reprezentácie, v ktorej sa bude hladat 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 vyššie uvedená časť.
- spúšťanie samotného vyhľadávania 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 aplikácia tvorená.

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, ktorý bude použitý na samotné hľadanie ciest v mapovej reprezentácii

Vyberanie parametrov sa musí riadiť istými pravidlami. To z dôvodu závislostí jednotlivých typov objektov popísaných v podsekciách sekcie 1.2.

- ukladanie parametrov - plan na pridani konfiguracie vyskovych dat do vytvarania mapovej reprezentacie

4 Súčasná podoba vrstvy Model

Závěr

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.

Seznam obrázků

Seznam použitých zkratek

- OB - orientačný beh
- GUI - grafické užívateľské rozhranie (graphical user interface)
- XAML - Extension application markup language
- MVVM - Model-View-ViewModel

A Přílohy

A.1 První příloha