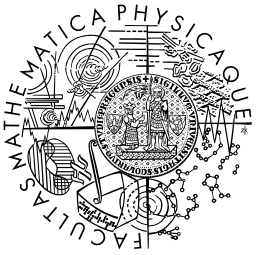


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Jakub Sýkora

GTTG – aplikace pro práci s grafikonem vlakové dopravy

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce:	Mgr. Pavel Ježek, Ph.D.
Studijní program:	Informatika
Studijní obor:	Softwarové a datové inženýrství

Praha 2019

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) 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

Chtěl bych poděkovat vedoucímu Mgr. Pavlu Ježkovi, Ph.D. za jeho čas, rady a pomoc při vypracování této práce. Zároveň bych chtěl poděkovat i mé rodině a všem, kteří mě při studiu podpořili nebo nějak pomohli.

Název práce: GTTG – aplikace pro práci s grafikonem vlakové dopravy

Autor: Jakub Sýkora

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Součástí aplikací pracujících s grafikonem vlakové dopravy je obvykle vizualizace provozu na železniční trati, takzvaný nákresný jízdní řád. Cílem práce je vytvořit knihovnu ulehčující vývojářům tvorbu takovýchto aplikací poskytnutím grafické komponenty, která vykresluje jejich vlastní vizualizaci nákresného jízdního řádu.

Komponenta, určená pro aplikace na platformě .NET, využívá ke kreslení 2D grafickou knihovnu SkiaSharp a je integrovatelná do více frameworků uživatelských rozhraní. Knihovna umožňuje interaktivně pracovat s vykreslovaným nákresným jízdním řádem, například přibližováním zobrazovaného obsahu nebo klikáním na vykreslované prvky pomocí myši. Aby se dále ulehčilo vytváření aplikací, knihovna nabízí lehce rozšiřitelnou základní vizualizaci nákresného jízdního řádu.

Jako referenční příklad využití knihovny jsme vytvořili v GUI frameworku WPF aplikaci pro prohlížení nákresných jízdních řádů organizace Správy železniční dopravní cesty spravující provoz na železničních tratích České republiky.

Klíčová slova: Nákresné jízdní řády Grafikon vlakové dopravy SkiaSharp .NET Knihovna

Title: GTTG – application for manipulation with train timetable diagrams

Author: Jakub Sýkora

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Applications for organization of rail transport often contain visualization of traffic situations, referred to as train timetable diagrams. The goal of this thesis is to create a library that simplifies the development of such applications by providing graphical component which allows developers to implement their own custom train timetable diagrams.

The component, developed for .NET platform, utilizes 2D graphics library SkiaSharp for drawing and can be integrated into various GUI frameworks. The library features interactive rendered content of train timetable diagrams – for example, zooming in on specific areas of the content or clicking on visualized elements. The library also offers an easily extendable implementation of the basic train timetable diagram.

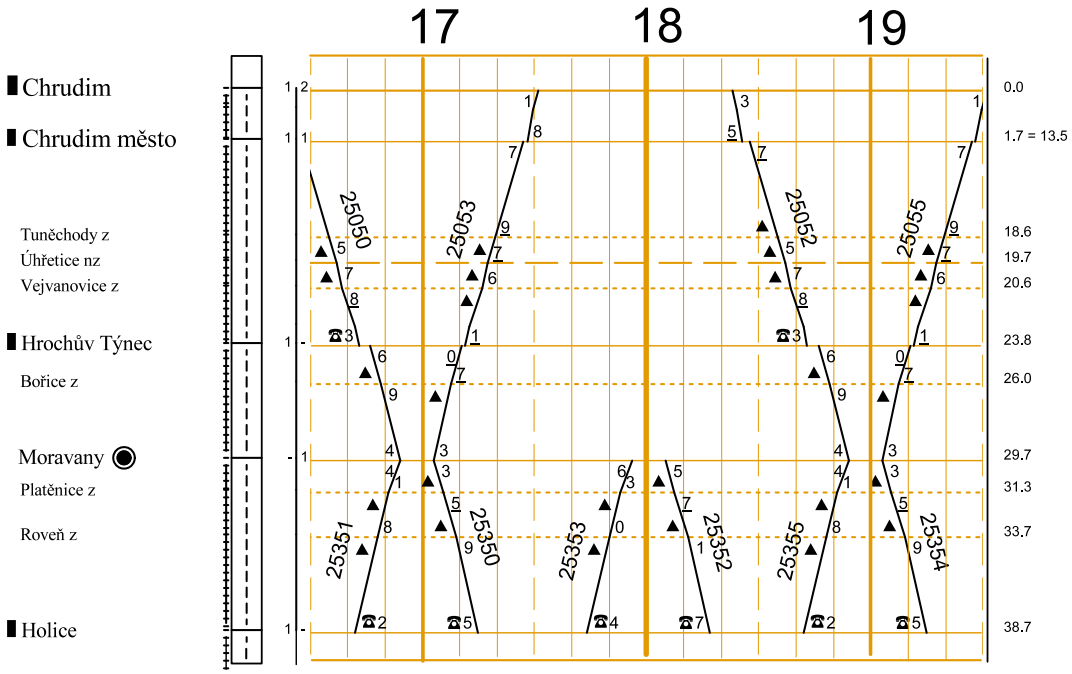
As an example of our library utilization, we also developed a WPF application intended for viewing train timetable diagrams of Czech railways.

Keywords: Train timetable diagrams Train graph SkiaSharp .NET Library

Obsah

1. Úvod

Česká republika patří mezi země s nejhustší sítí železnic na světě. Každým rokem se pro všechny tratě v této síti vytváří plán jízdy vlaků, zohledňující velké množství často protichůdných požadavků. Cestující se s částmi tohoto plánu, jízdami vlaků určených pro osobní dopravu, seznamují skrze tištěné knižní jízdní řády nebo uživatelsky přívětivé webové služby. Před běžnými cestujícími tak zůstává skryté množství pomůcek určených pro služební účely. Jednou z takových pomůcek je nákresný jízdní řád, jehož příklad můžeme vidět na obrázku 1.1.



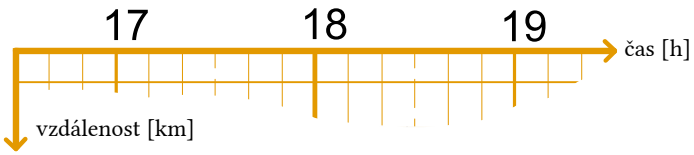
Obrázek 1.1: Příklad nákresného jízdního řádu

Všichni zaměstnanci zúčastněni na provozu železniční dopravy potřebují mít přehled o plánovaném i aktuálním provozu v traťových úsecích. Jelikož pro tuto potřebu reprezentovat data textovou formou není vhodné, využívá se znázornění průběhu jízd vlaků ve formátu, který se v železniční dopravě označuje jako *nákresný jízdní řád*. Existují aplikace, které s nákresným jízdním řádem pracují. Aplikace se však často při jeho zobrazování potýkají s problémy. Dále existují nástroje, které ke svému fungování nákresný jízdní řád nepotřebují, ale práce s nimi by se mohla zlepšit, pokud budou nákresný jízdní řád zobrazovat. Chtěli bychom proto vytvořit nástroj, který bude nákresné jízdní řády zobrazovat a umožní s nimi co nejlépe pracovat. Hlavním cílem práce bude vytvořit takový nástroj jako grafickou komponentu integrovatelnou do různých aplikací.

Než ale blíže určíme, jaké konkrétní požadavky by měla tato grafická komponenta splňovat, podrobně si v této kapitole nákresný jízdní řád představíme a zvážíme jeho existující uplatnění i možná využití. Jeho vlastnosti, které si budeme popisovat, jsou často specifické pro české prostředí.

1.1 Nákresné jízdní řády

Základem nákresného jízdního řádu je síť vodorovných a svislých čar umístěná v grafu, který je znázorněn na obrázku 1.2. Svislou osou grafu je kilometrická poloha na trati. Vodorovná osa značí čas, kde čísla na obrázku představují celé hodiny.

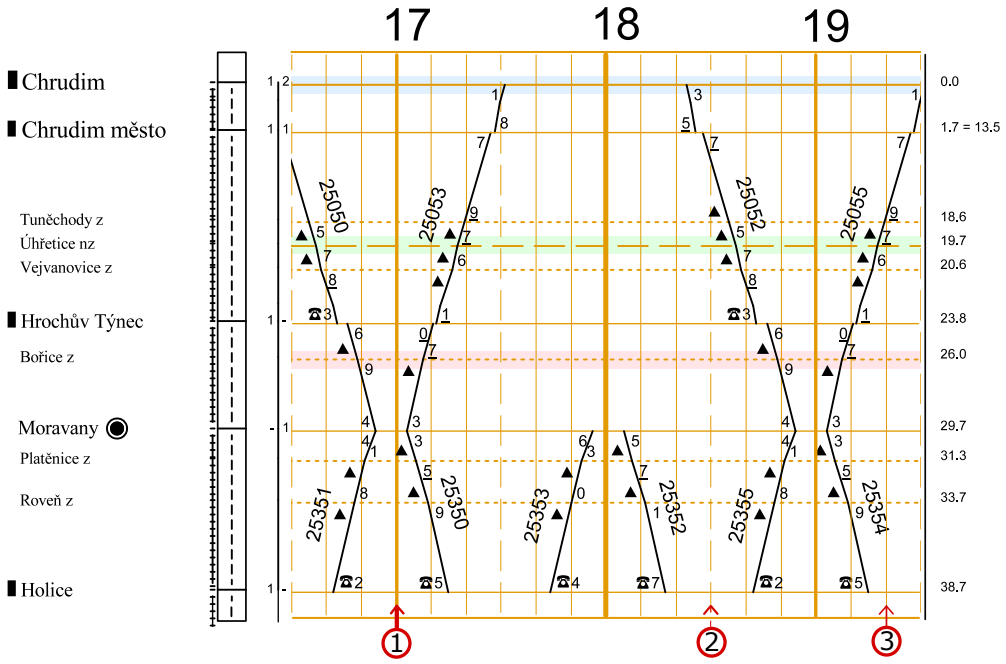


Obrázek 1.2: Osy nákresného jízdního řádu

Význam vodorovných a svislých čar si vysvětlíme s pomocí obrázku 1.3, který obsahuje nákresný jízdní řád doplněný anotacemi ke zmíněným čarám.

Na trati se nachází významná místa (například stanice, zastávky nebo výhybny), které označujeme jako *dopravní body*. Vodorovné čáry odpovídají dopravním bodům rozmístěných v grafu podle jejich kilometrické polohy vzhledem k trati. Na obrázku 1.3 pro přehlednost modře podbarvená plná čára reprezentuje stanici Chrudim. Přerušovaná čára podbarvená zelenou barvou patří nákladišti a zastávce Úhřetice. Červeně podbarvená tečkovaná čára značí zastávku Bořetice. Vzor (nebo i tloušťka) čáry tak určuje typ dopravního bodu.

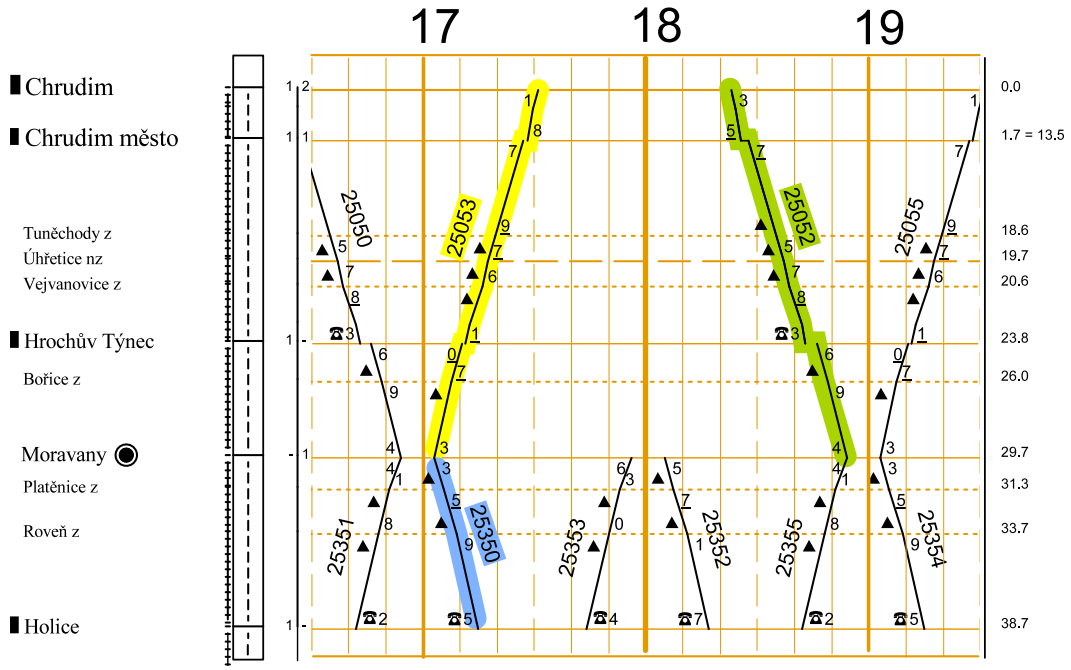
Stejným způsobem se rozlišují svislé čáry značící časové údaje v intervalech po deseti minutách. Plná čára na obrázku 1.3 označena ① je určena pro hodiny, ② pro půlhodiny a tenká plná čára označená ③ je určena pro jiné časové údaje.



Obrázek 1.3: Nákresný jízdní řád s vyznačenými typy vodorovných a svislých čar

Průběh jízdy vlaků v nakresném jízdním řádu

Popsaná síť vodorovných a svislých čar je nutným základem pro vyobrazení nejdůležitějšího obsahu nákresného jízdního řádu, kterým jsou průběhy jízd vlaků. Pomocí obrázku 1.4 obsahující zvýrazněné průběhy jízd některých vlaků si popíšeme princip jejich vyobrazování.



Obrázek 1.4: Nákresný jízdní řád s vyznačenými průběhy jízd vlaků 25350, 25052 a 25053

Na obrázku 1.4 se nachází vlak označený číslem 25053 se žlutě podbarvenou šikmou čarou reprezentující průběh jeho jízdy. Všimněme si, že tato čára tvoří průsečíky s vodorovnými čarami reprezentující dopravní body. Takový průsečík pak představuje příjezd, odjezd nebo průjezd vlaku dopravním bodem v čase, který odpovídá pozici průsečíku na časové ose. Průsečík šikmé čáry, který se vyskytne na časové ose jako první, patří k dopravnímu bodu, ze kterého vlak vyjíždí. Poslední průsečík naopak odpovídá dopravnímu bodu, kde jízda vlaku končí. V případě vlaku 25053 vlak vyjíždí přibližně v sedmnáct hodin ze stanice Moravany a jede směrem na Chrudim, do které dojede kolem půl šesté. Tímto popsaným principem můžeme sledovat celý průjezd vlaku tratí v závislosti na čase a jeho poloze.

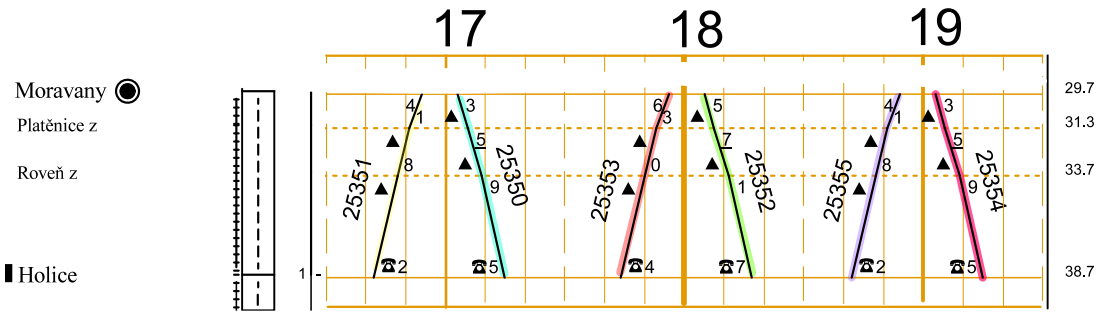
Podobně je možné z obrázku 1.4 určit průběh jízdy i u ostatních vlaků. Zeleně podbarvená lomená čára představuje průběh jízdy vlaku 25052, který jede v opačném směru do Moravan. Modře podbarvený vlak 25350 odjíždí z Moravan ve stejný čas jako vlak 25053, ale v opačném směru na Holice. Svislé čáry těchto dvou vlaků, které jsou vizuálně spojeny v Moravanech, od sebe můžeme odlišit pochopením faktu, že kdybychom je vnímaly jako celek zobrazující průběh jízdy jednoho vlaku, záznam by vzhledem k časové ose nedával smysl, jelikož by se vracel do minulosti.

Nezbytnou součástí vyobrazení informací o jízdě vlaků v nákresném jízdním řádu je vhodně umístěné číslo vlaku v blízkosti šikmé čáry, která odpovídá průběhu jeho jízdy. Podle pravidel provozu na českých železničních tratích má každý vlak přiděleno unikátní číslo. Číslo je v nákresném jízdním řádu umístěno způsobem, z kterého je zřejmé, k jakému vlaku patří. Číslo se umisťuje v blízkosti šikmé čáry vlaku tak, aby směr čtení čísla odpovídal směru jízdy. Šikmá čára je pak pomyslným řádkem, na kterém je číslo napsáno.

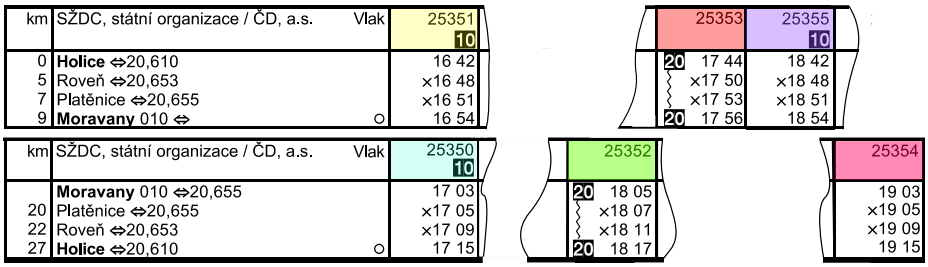
Výhody zobrazování průběhu jízd vlaků nákresným jízdním řádem

Abychom pochopili, v jakých situacích je možné nákresné jízdní řády používat, popíšeme si výhody, které přináší oproti jiným formám jízdního řádu. Z nákresného jízdního řádu je lehké zjistit, jak jsou úseky mezi dopravními body obsazeny jízdami vlaků. Drážní zaměstnanec tak má přehled o dopravní situaci na celé trati.

Vedle nákresných jízdních řádů existují jízdní řády v textové formě, které jsou užitečné například pro cestující, ale nemůžou nahradit roli nákresného jízdního řádu. Jedním z textových formátů zobrazující informace o jízdě vlaků je knižní jízdní řád, jehož výřez je uvedený na obrázku 1.6. Můžeme vidět, že informace v něm obsažené jsou vztaženy k jen konkrétnímu vlaku a těžko se mezi sloupečky, obsahující plány jízd vlaků, dají hledat hlubší souvislosti. Pro porovnání jsou vlaky z výřezu knižního jízdního řádu vyobrazeny na nákresném jízdním řádu na obrázku 1.5.



Obrázek 1.5: Nákresný jízdní řád s podbarvenými průběhy jízd vlaků z knižního jízdního řádu na obrázku 1.6

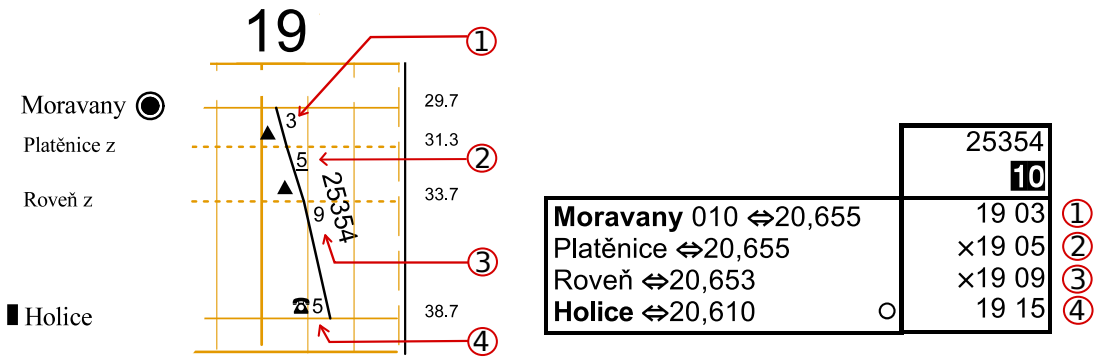


Obrázek 1.6: Výřez z knižního jízdního řádu se zvýrazněnými čísly vlaků odpovídající podbarveným průběhům jízd těchto vlaků na obrázku 1.5

V další části si popíšeme, jak z nákresného jízdního řádu můžeme vyčíst přesné časy příjezdů a odjezdů obsažených ve sloupcích knižních jízdních řádů.

Informace k jízdě vlaků v nákresném jízdním řádu

Ještě jsme si neuvedli, jak v nákresných jízdních řádech určit přesný čas příjezdů, odjezdů a průjezdů vlaku dopravními body. Tyto informace se nachází v ostrých úhlech, které dopravní body s trasami vlaků svírají, a nazývají se *kóty*. V kótách se pouze uvádí jednotky minut, například pro čas 19:15 se uvede 5. Na obrázku 1.7 můžeme vidět, jak jsou tímto způsobem ve výřezu nákresného jízdního řádu zaznamenány časy odjezdů (a příjezdu do Holic) z knižního jízdního řádu.



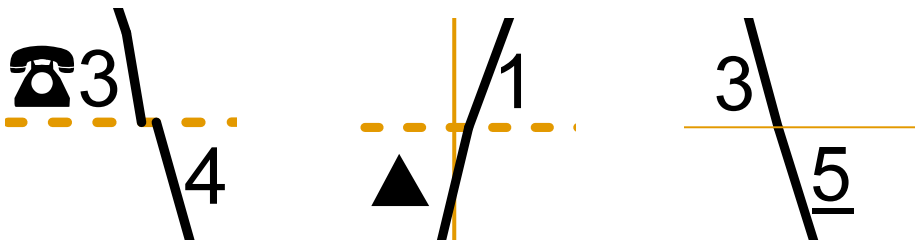
Obrázek 1.7: Označené kóty v nákresném jízdním řádu odpovídající časům v knižním jízdním řádu

Ostré úhly neslouží pouze k umisťování kót, ale je možné do nich přidat i další informace, které by se musely jinak složitě hledat jinde. Navíc se někdy zobrazení kót mění. Příklady zmiňovaných informací a změn se nachází na obrázku 1.8:

(a) Ikona telefonu značí ohlašovací povinnost strojvedoucího při příjezdu do dopravního bodu.

(b) Pokud má vlak v dopravním bodu dobu pobytu kratší než půl minuty a nejedná se o průjezd, kóta příjezdu se nahradí trojúhelníkem.

(c) Podtržená kóta značí o půlminuty více. Na obrázku vlak odjíždí z dopravního bodu mezi pěti minutami a 30 sekundami až šesti minutami, vhledem k používanému zaokrouhlení času pro kóty.

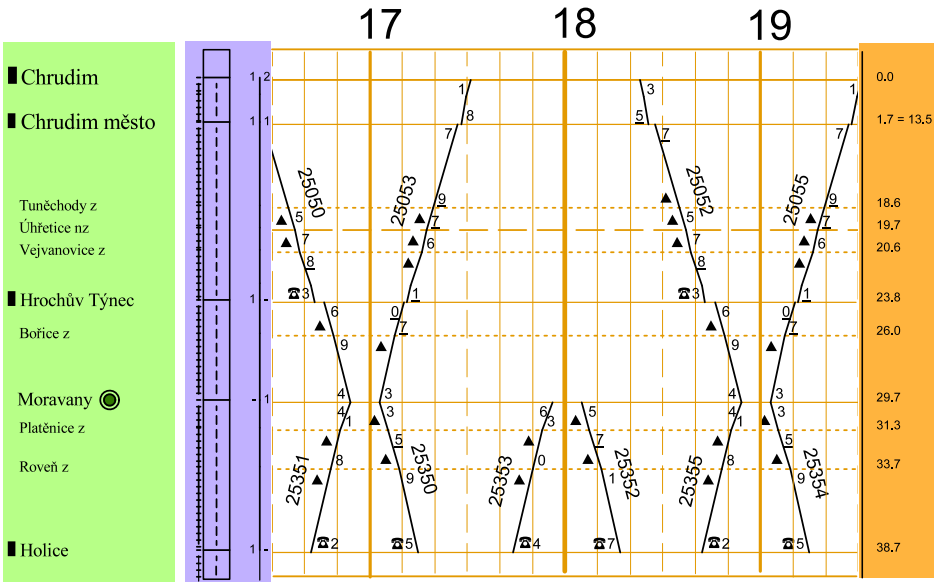


(a) Ohlašovací povinnost nařízena (b) Pobyt kratší než půl minuty (c) Odjezd o půl minuty později

Obrázek 1.8: Další informace umístitelné do ostrých úhlů

Další informace zobrazované v nákresných jízdních řádech

Doposud jsme si popisovali, jak se zobrazují informace související s jízdou vlaku. Nákresný jízdní řád se ale nemusí omezovat pouze na tyto informace. Na obrázku 1.9 můžeme vidět několik sloupců. Zeleně podbarvený sloupec obsahuje názvy dopravních bodů, umístěných podle kilometrické vzdálenosti na trati. V blízkosti názvů se pak nachází značení, přidávající další informace o dopravním bodu. Ve fialově podbarveném sloupci se nachází informace o zabezpečení traťového úseku. Oranžově podbarvený sloupec zobrazuje kilometrické vzdálenosti dopravních bodů na trati.

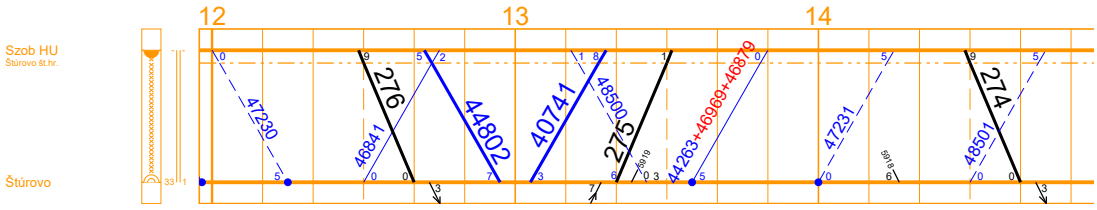


Obrázek 1.9: Nákresný jízdní řád s podbarvenými sloupci nesoucí další informace

Tímto jsme dokončili popis všech informací v nákresném jízdním řádu, poprvé uvedeném na obrázku 1.1.

Další typy nákresných jízdních řádů

Až do této chvíle jsme si pro lehčí pochopení ukazovali pouze jeden typ nákresného jízdního řádu. Existuje ale více typů nákresných jízdních řádů. Nákresný jízdní řád na obrázku 1.10 se používá pro organizaci železniční dopravy na Slovensku a jeho vizualizace se řídí doposud uvedenými pravidly, pouze se liší v drobnostech, kterými jsou použité barvy nebo vzor čar.



Obrázek 1.10: Další typ nákresného jízdního řádu

Než si ale detailně uvedeme jiné typy nákresných jízdních řádů, které budou obvykle součástí různých aplikací, zasadíme je do kontextu grafikonu vlakové dopravy, s kterým tyto aplikace pracují.

1.2 Grafikon vlakové dopravy

V železniční dopravě se pracuje s pojmem grafikon vlakové dopravy. Ten je v přeneseném významu jiným označením pro zobrazení jízdy vlaků v nákresném jízdním řádu. Proto se někdy nákresné jízdní řády označují jako listy grafikonu vlakové dopravy. Původně ale tento pojem představuje označení pro soubor předpisů a pomůcek určených k plánování vlakové dopravy. Na základě mnoha často protichůdných požadavků se pomocí předpisů a pravidel grafikonu vlakové dopravy sestaví plán, sloužící jako předloha a dokumentace, na kterou je se potřeba při organizaci dopravy a řešení konfliktů vznikajících při provozu odkazovat. Takto sestavený plán je obsažen v pomůckách grafikonu vlakové dopravy, mezi které patří právě listy nákresného jízdního řádu nebo jízdní řády pro cestující. Provoz v české železniční síti je pod správou státní organizace Správa železniční dopravní cesty, která pro každý rok sestavuje nový grafikon vlakové dopravy. Dokumentem popisující tento proces je Směrnice SŽDC č. 69 pro tvorbu jízdního řádu a pomůcek GVD[1]. Naše práce se problematice sestavování grafikonu vlakové dopravy věnovat nebude, více se o ní může čtenář dozvědět v knize Železniční doprava, Gašparík a Kolář [1].

1.3 Práce s grafikony vlakové dopravy

V této části se seznámíme s dalšími typy nákresných jízdních řádů. Pokud budeme mluvit o práci s grafikonem vlakové dopravy, budeme tímto spojením označovat činnost, která je založena na používání pravidel a pomůcek grafikonu vlakové dopravy – asi nejznámější činností je řízení provozu na železnici. Abychom věděli, jak může být námi vytvářený nástroj používán, představíme si jeho možná využití při práci s grafikonem vlakové dopravy.

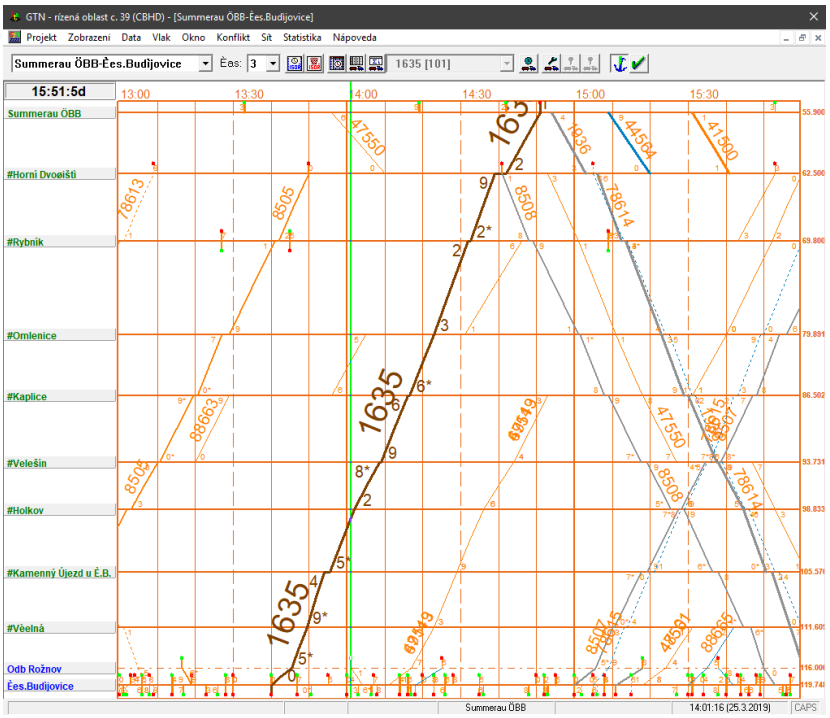
Aplikace pro vlakovou dopravu

Organizace vlakové dopravy se v české železniční síti až do vybudování infrastruktury výpočetní techniky odehrávala pouze na papíře. V současnosti existují aplikace pracující s grafikonem vlakové dopravy nasazené v reálném provozu, které mají pozitivní dopad na organizaci i zabezpečení provozu na trati. Některé takové aplikace si v této části popíšeme. Nové aplikace, používající námi vytvářenou grafickou komponentu zobrazující nákresný jízdní řád, se pak většinou budou snažit chování nyní uvedených aplikací napodobit.

[1] grafikon vlakové dopravy

Aplikace GTN

Aplikace GTN[2] společnosti AŽD Praha zobrazuje uskutečněnou a výhledovou (budoucí) dopravu. Zachycuje tak současnou situaci na trati. Okno aplikace s komponentou zobrazující nákresný jízdní řád se nachází na obrázku 1.11. V aplikaci je možné plánovat výhledovou dopravu úpravou průběhu jízd vlaků přímo v komponentě. Upravovaný vlak je navíc od ostatních vizuálně rozlišitelný hnědou barvou a zvětšenými kótami. Komponenta může zobrazit současnou dopravu v různých časových intervalech. Aplikace tak nezobrazuje nákresný jízdní řád, jehož obsah by byl neměnný, ale rozšiřuje ho o prvky, které z komponenty zobrazující nákresný jízdní řád vytváří interaktivní nástroj pro práci s grafikonem vlakové dopravy. Pokud by si chtěl čtenář práci s aplikací vyzkoušet, existuje volně přístupná demoverze [2] aplikace.



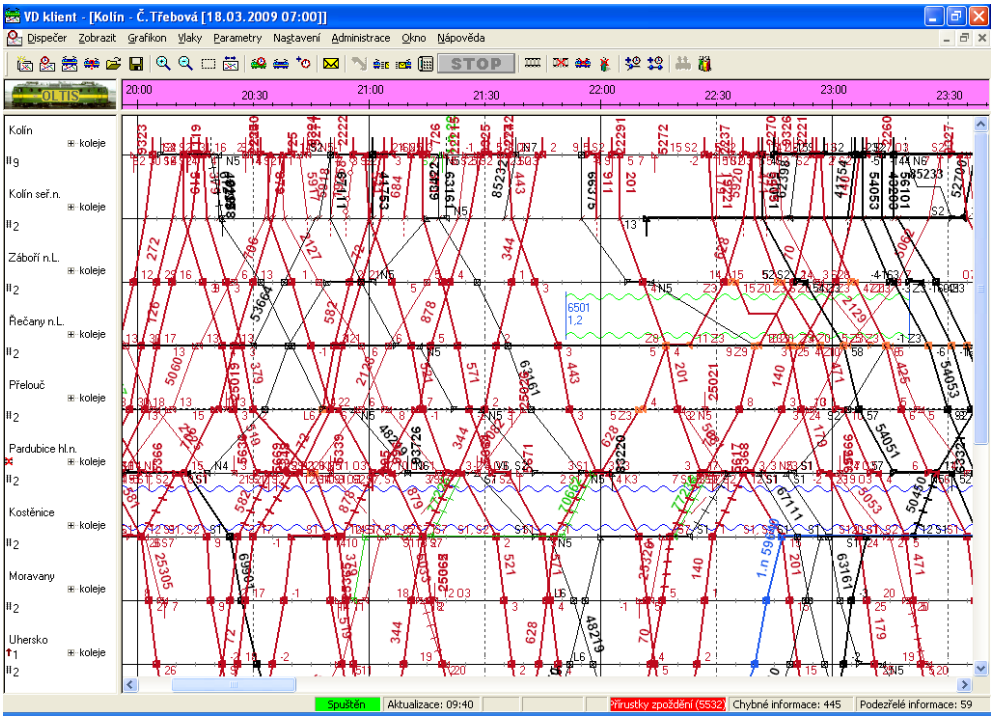
Obrázek 1.11: Okno aplikace GTN

Aplikace v rámci IS ISOR CDS

Aplikace pro práci s grafikonem vlakové dopravy nabízena v rámci informačního systému ISOR CDS [3] od OLTIS Group je zaměřena stejně jako aplikace GTN na prohlížení současného provozu. Na obrázku 1.12 můžeme vidět, jak aplikace vizualizuje nákresný jízdní řád. Všimněme si, že narozdíl od doposud popisovaného chování nezobrazuje dopravní body na vertikální ose podle jejich kilometrické polohy, ale rozmisťuje je rovnoměrně pro jednodušší zobrazení nákresného jízdního řádu. Taková změna v zobrazení není nijak závadná, ačkoliv porušuje námi popsané zásady pro zobrazení nákresného jízdního řádu. V dalších částech práce budeme narážet na podobné odchylky, nikdy ale zásadně neporuší popsané rozpoložení uvedené na začátku kapitoly. Na pravé hraně grafické komponenty

[2]Graficko-technologická nadstavba

zobrazující nákresný jízdní řád nabízí aplikace posuvník pro změnu zobrazené části traťového úseku. Stejným způsobem umožňuje na horizontále výběr zobrazovaného časového intervalu. Obě dvě aplikace tedy nabízí podobné možnosti při zobrazování nákresného jízdního řádu, volí jen jiný způsob ovládání.



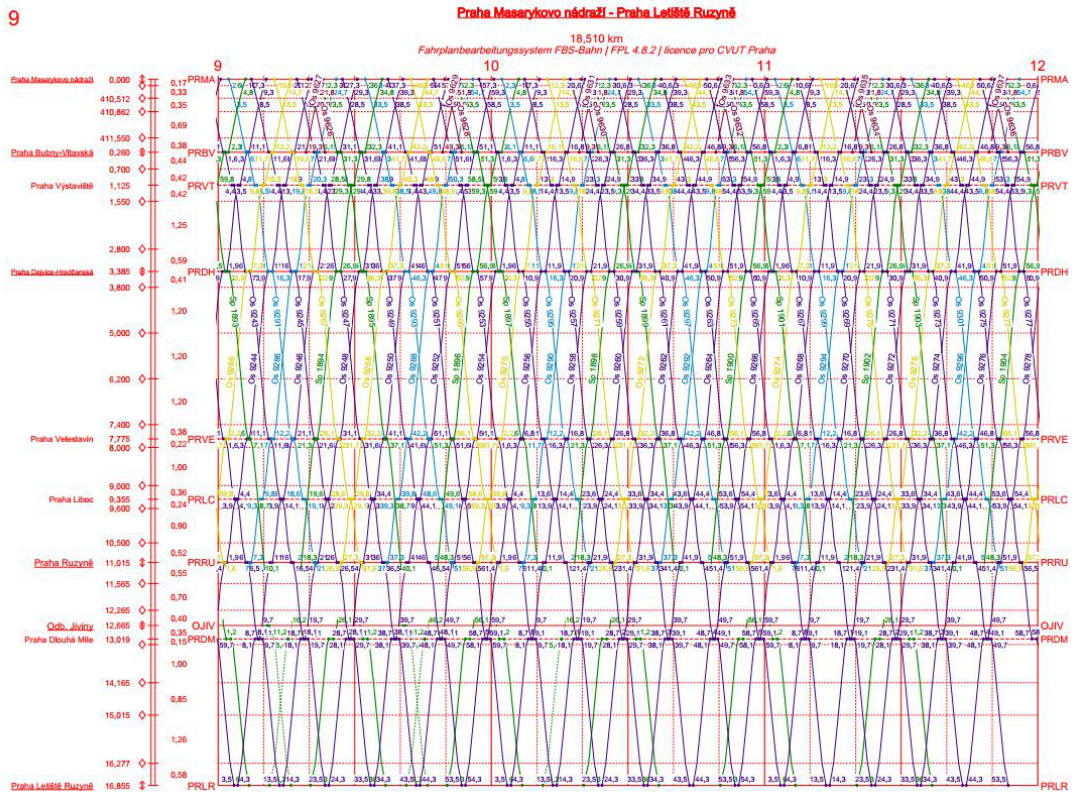
Obrázek 1.12: Aplikace Grafikon od OLTIS Group, převzato ze stránky společnosti [3]

Grafikon vlakové dopravy mimo skutečný provoz

V následující části zjistíme, že námi vytvářená grafická komponenta zobrazující nákresný jízdní řád může být součástí aplikací, které nejsou určeny pro nasazení v reálném provozu. Grafikon vlakové dopravy se zobrazeným nákresným jízdním řádem se nabízí použít i v jiných scénářích, zejména pak při aktivitách, které se snaží provoz na železnici simulovat.

Vytváření studií pro vlakovou dopravu

Nejblíže problémům vlakové dopravy řešených ve skutečné situaci jsou studie a koncepty věnující se provozu na železnici, které k prezentování svých výsledků využívají koncepčně sestavený grafikon vlakové dopravy a nákresné jízdní řády. Tyto studie jsou většinou zpracovávány v rámci různých bakalářských a diplomových prací studentů dopravních fakult. Existují komerční nástroje, které jsou k těmto účelům určené a na základě zadaných dat můžou autorům pomoci se sestavením grafikonu vlakové dopravy a vytvořením jeho pomůcek. Typickým řešením je pro fakultu koupit licenci takového nástroje. Jedním z těchto nástrojů je FBS-Bahn [4]. Tímto nástrojem byl vytvořen nákresný jízdní řád na obrázku 1.13.

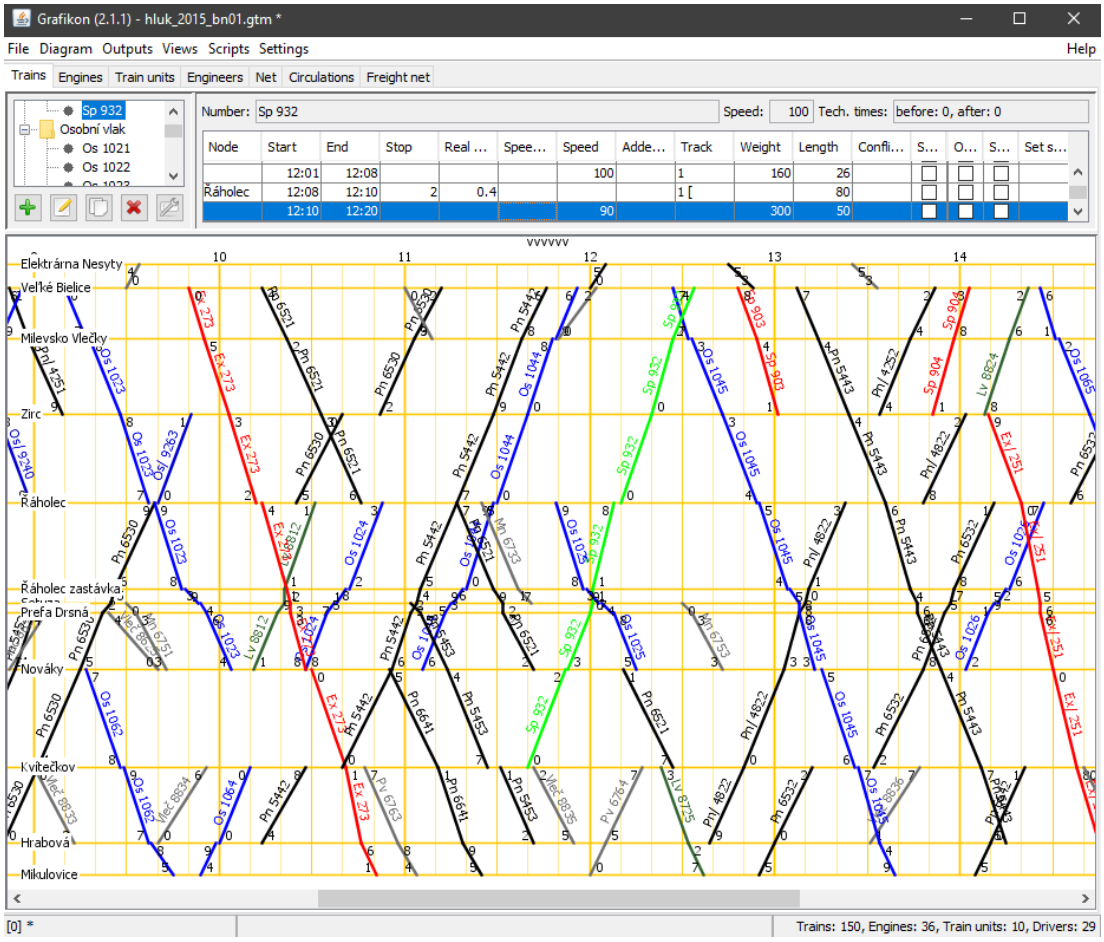


Obrázek 1.13: Nákresný jízdní řád generovaný nástrojem FBS-Bahn pro diplomovou práci Koncepce obsluhy letiště Ruzyně kolejovou dopravou, M. Drábek [5]

Aplikace v modelové železnici

Modelová železnice je co nejvěrnějším ztvárněním skutečné železnice ve zmenšeném měřítku. Za účelem napodobení skutečného provozu se nabízí jízdu vlaků na modelovém kolejišti organizovat podle grafikonu vlakové dopravy. Pokud se řízení vlaků na modelovém kolejišti věnuje více modelářů, v rámci přípravy jsou jízdy vlaků rozplánovány a zakresleny do nákresného jízdního řádu, který může být načrtnutý na papír.

Modeláři, kteří umí programovat, můžou vytvářet různé aplikace k organizaci provozu na modelové železnici. Jednou z nich je aplikace Grafikon [6], zobrazená na obrázku 1.14. Aplikace umožňuje sestavit grafikon vlakové dopravy pro modelová kolejiště. Nákresný jízdní řád, který vykresluje, ale slouží pouze k plánování a není možné ho při řízení provozu na modelovém kolejišti měnit, třeba pro porovnání současného provozu s původním plánem. Pro modeláře existují řešení, která umožňují řídit provoz na modelové železnici za pomoci počítače. Výrobci nástrojů pro řízení modelové železnice nabízí rozhraní, vůči kterým je možné vytvářet software sloužící k sestavení vlakové cesty, kdy modelář může z počítače měnit třeba nastavení výhybek. Takový software ale zcela nezachycuje proces organizace vlakové dopravy odpovídající skutečné situaci. Vytvářenou grafickou komponentu zobrazující nákresný jízdní řád by pak bylo možné zahrnout do nových nebo stávajících softwarových řešení, kterým by tato komponenta umožnila graficky zaznamenávat jízdy vlaků a porovnávat je se sestaveným plánem.



Obrázek 1.14: Hlavní okno aplikace Grafikon určené pro modelové železnice

Aplikace ve vlakových simulátorech

Námi vytvářený nástroj může být integrován do počítačových simulátorů, které se věnují řízení vlaku z pozice strojvůdce. Na začátku tisíciletí vzniklo mnoho takových simulátorů a her (Trainz [7], MSTS [8]) vydávaných firmami, bez možnosti přístupu k jejich zdrojovým kódům. V posledních letech vznikly open-source projekty snažící se na základě těchto her vytvořit lepší simulátory s větší konfigurovatelností, jelikož většina původních her už není dále vyvíjena a podporována. Mezi tyto open-source simulátory patří OpenRails [9] a OpenBVE [10]. Oba dva tyto simulátory jsou napsané v jazyce C#, existují několik let a jsou stále udržovány.

Simulátory, snažící se napodobit skutečný provoz na trati předpřipraveným scénářem jízd vlaků v časovém intervalu několika hodin, by bylo užitečné doplnit pomůckami grafikonu vlakové dopravy. Hráči by byla dostupná (třeba v okně vedle okna simulátoru) aplikace s grafikonem vlakové dopravy, což v současnosti ani jeden ze zmíněných open-source projektů nenabízí. Zajímavá by pak byla možnost hrát hru po síti s dalšími lidmi (což už tyto simulátory umožňují) a sledovat průběh hraného scénáře v nákresném jízdním řádu v porovnání s původním plánem.

1.4 Vývoj aplikací pro práci s grafikonem vlakové dopravy

Zaměřme se na proces vývoje aplikací, které budou pracovat s grafikonem vlakové dopravy. Všimněme si, že pro všechny uvedené příklady aplikací z podkapitoly 1.3 je nezbytné zobrazovat nákresný jízdní řád. Bez komponenty aplikace, která je zodpovědná za jeho vykreslení, by se data z grafikonu vlakové dopravy stala těžko spojitelnými množinami informací. Kdybychom se rozhodli vytvořit aplikaci pro práci s grafikonem vlakové dopravy, stáli bychom tak vždy před rozhodnutím, jak tuto komponentu navrhnout. Pokud by existovala knihovna, která poskytne nástroje pro zobrazování nákresného jízdního řádu a práci s jeho obsahem, vývoj aplikací pracujících s grafikonem vlakové dopravy by se tak ulehčil. Proto hlavní část této práce bude věnována vytváření takové knihovny. Námi vytvářenou knihovnu jsme se rozhodli pojmenovat GTTG (Generic Train Traffic Graph[3]).

Knihovna pro platformu .NET

Knihovnu, která bude představovat grafickou komponentu umožnující práci s nákresným jízdním řádem, jsme se rozhodli naimplementovat v jazyce C#, který je součástí platformy .NET. Odůvodníme si, proč toto rozhodnutí není závadné a seznámíme se s knihovnou SkiaSharp [11], kterou budeme v naší knihovně používat ke kreslení v grafické komponentě.

Jazyk C# a .NET Standard

Jelikož vytvářená knihovna bude reálně využívána hlavně modeláři, vývojáři nástrojů pro modelové železnice nebo programátory počítačových simulátorů, musí být snadno udržovatelná a použitelná k vytváření jejich vlastních implementací nákresného jízdního řádu. Protože knihovna bude většinou součástí nějaké aplikace s uživatelským rozhraním, je vhodné, aby uživatelům knihovny bylo dostupné množství GUI frameworků, do kterých bude možné knihovnu jako grafickou komponentu zahrnout.

Protože jazyk C# je jeden z nejpoužívanějších objektově orientovaných jazyků a nabízí mnoho GUI frameworků pro tvorbu desktopových aplikací, jeho výběr není v rozporu s tímto požadavkem a můžeme ho použít. Silnou motivací, proč knihovnu implementovat právě v tomto jazyce, je fakt, že oba dva zmíněné open-source simulátory jsou v jazyce C# napsané a knihovnu tak bude možné v těchto stávajících projektech využít.

Naší knihovnu bychom zároveň chtěli vytvořit tak, aby byla v rámci platformy .NET co nejvíce použitelná. Každý spustitelný program, který v jazyce C# napíšeme, může být spuštěn pouze vůči některému z běhových prostředí. Původním běhovým prostředím platformy .NET je .NET Framework, který je

[3]Název knihovny jsme chtěli co nejvíce přiblížit anglickému názvu pro nákresný jízdní řád, který se lidmi orientující se ve vlakové dopravě většinou označuje jako train graph. Pro případ, že se s názvem knihovny setká člověk bez představy o tom, co nákresný jízdní řád je, jsme chtěli vystihnout to, že se nejedná o graf zobrazující třeba železniční dvojkolí, ale graf zaznamenávající provoz na trati. Proto jsme přidali do názvu slovo traffic. Slovo generic na začátku názvu vystihuje záměr knihovny podporovat práci s co největším množstvím nákresných jízdních řádů.

určen pro operační systém Windows. Vedle tohoto běhového prostředí existuje Mono, které bylo vytvořeno jako open-source náhražka .NET Frameworku pro operační systémy Linux a MacOS. V posledních letech vzniklo běhové prostředí .NET Core, jehož vývoj je řízen společností Microsoft a je možné ho použít na operačních systémech Windows, Linux a MacOS. Je v plánu nahradit .NET Framework a Mono právě tímto novým běhovým prostředím. Aby bylo možné vytvářet knihovny, které budou použitelné všemi běhovými prostředími, vzniklo rozhraní .NET Standard. Knihovny, které vůči .NET Standard implementujeme, můžou stále používat skoro stejnou část používaných tříd a implementací, které jsou nabízeny například v rámci běhového prostředí .NET Framework. Aby .NET Standard docílilo přenositelnosti mezi více běhovými prostředími, určuje rozhraní, které běhová prostředí musí implementovat, aby kód knihovny .NET Standard mohl být na konkrétním běhovém prostředí vykonáván. Jelikož vytvářená grafická komponenta bude často umístěna v aplikaci s GUI frameworkem, který je spustitelný vůči jednomu běhovému prostředí, bude naše knihovna přenositelná v rámci rozhraní .NET Standard.

SkiaSharp

Naše knihovna potřebuje nějaký nástroj, který bude sloužit k vykreslování nákresného jízdního řádu. Práce s tímto nástrojem by měla být lehká, jelikož ho budeme chtít nabídnout uživatelům naší knihovny pro implementaci jejich vlastních typů nákresných jízdních řádů.

Pro jazyk C# existuje knihovna SkiaSharp, jako wrapper[4] nad vysoce výkonnou 2D grafickou knihovnou Skia napsanou v jazyce C++. Skia nabízí pro uživatele lehké kreslení různých komplexních typů čar a textu, které nákresné jízdní řády obsahují. Práce s knihovnou SkiaSharp je lehká, díky její dostupné dokumentaci, zcela nutné pro případ, kdy ji chceme také nabídnout uživatelům naší knihovny.

Knihovna SkiaSharp je navíc přenositelná v rámci rozhraní .NET Standard, čili je naší knihovnou použitelná a splňuje i všechny naše požadavky na grafický nástroj, který bude naše knihovna používat pro vykreslování nákresného jízdního řádu.

Implementace vlastní aplikace

Abychom mohli předvést využití naší knihovny, rozhodli jsme se vytvořit vzorovou aplikaci pro práci s grafikonem vlakové dopravy. Aplikace uživatelům umožní prohlížet nákresné jízdní řády vydávané Správou železniční dopravní cesty. Ověříme tak schopnost knihovny podporovat zobrazení specifické implementace nákresného jízdního řádu. Abychom předvedli, jak knihovna podporuje editaci nákresného jízdního řádu, přidáme mód vytvářející náhodnou dopravní situaci na trati, v rámci které bude možné upravovat interaktivně v nákresném jízdním řádu jízdy jednotlivých vlaků.

[4]Označení pro knihovnu pracující s cizí funkcionalitou, kterou vhodně upravuje nebo zpřístupňuje uživateli wrapperu, většinou s žádnými změnami v sémantice cizí funkcionality, které chce uživatel wrapperu využívat. SkiaSharp jako wrapper překládá volání funkcí z jazyka C# do kódu knihovny Skia implementované v jazyce C++.

1.5 Cíle práce

G1 Vytvořit knihovnu pro platformu .NET, představující grafickou komponentu, umožňující práci s nákresnými jízdními řády

- **a)** Podporovat zobrazování různých typů nákresných jízdních řádů
- **b)** Podporovat integraci komponenty do aplikací pracujících s grafikonem vlakové dopravy
- **c)** Umožnit replikovat chování existujících aplikací pracujících s grafikonem vlakové dopravy
- **d)** Zajistit přenositelnost knihovny na úrovni .NET Standard

G2 Použít tuto knihovnu pro implementaci aplikace, která bude sloužit pro práci s grafikonem vlakové dopravy

- **a)** Aplikace bude interaktivně zobrazovat listy nákresného jízdního řádu vydávané Správou železniční dopravní cesty.
- **b)** Aplikace bude pro ilustrační účely knihovny nabízet mód simulující provoz na trati, v jehož rámci bude možné upravovat výhledovou dopravu.
- **c)** Aplikace bude navržena tak, aby část pracující s modelem dat, představující logiku aplikace, byla zapojitelná do více GUI frameworků platformy .NET.

G3 Ověřit možnost využití 2D grafické knihovny SkiaSharp pro zobrazování komplexních dat obsažených v nákresných jízdních řádech

2. Analýza požadavků pro práci s knihovnou GTTG

Z kapitoly 1 jsme získali základní představu o tom, jaký účel bude knihovna GTTG splňovat a určili jsme si cíle, kterými se při implementaci knihovny budeme řídit. V této kapitole si tyto cíle více upřesníme a vytvoříme detailní souhrn požadavků, představujících kritéria pro správný výběr implementace. Požadavky je možné rozdělit do dvou hlavních kategorií:

- Požadavky na chování grafické komponenty jako okna, v kterém je zobrazován nákresný jízdní řád
- Požadavky na vykreslování nákresného jízdního řádu a práci s jeho obsahem

2.1 Požadavky na chování grafické komponenty

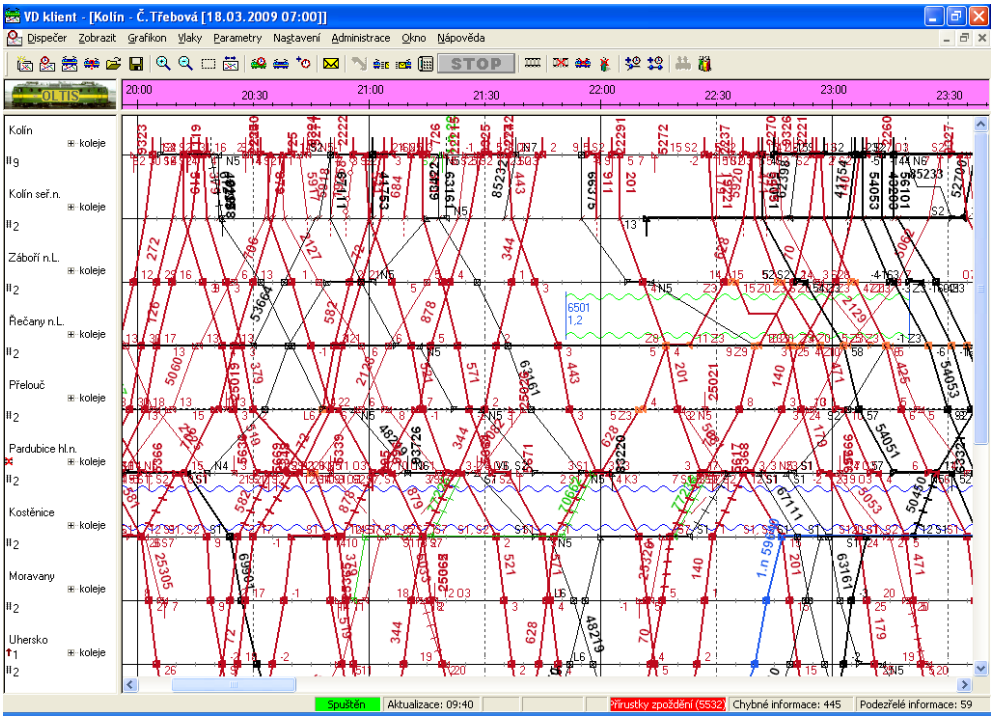
V této části vytvoříme požadavky určující chování grafické komponenty, která představuje okno s vykreslovaným obsahem. Určíme, jak se komponenta chová, například v rámci integrace s uživatelským rozhraním. Určením požadavků pro práci s touto komponentou vytvoříme prostředí, které umožní práci s nákresným jízdním řádem. Při určování těchto požadavků jsme zejména brali ohled na náš cíl zajistit replikovatelnost chování původních aplikací. Narozdíl od původních aplikací by ale knihovna měla nabízet i nové chování, které by zlepšilo interakci uživatelů s komponentou.

Interakce uživatelů s komponentou

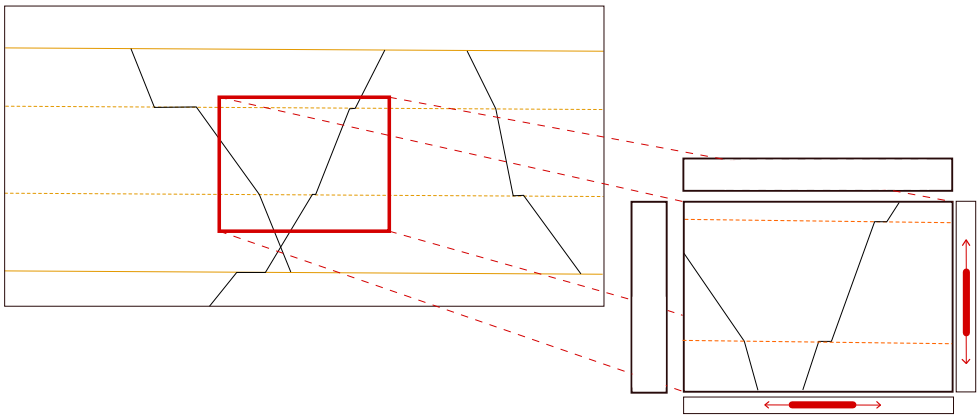
Jelikož vytváříme knihovnu pro více aplikací, měli bychom podporovat co největší množinu operací, kterými lze komponentu modifikovat. Aplikace pracující s knihovnou si z této množiny operací vybere pouze ty, které chce podporovat. V této části určíme požadavky, které umožní interakcemi uživatele s komponentou měnit zobrazovaný obsah komponenty. Podívejme se, jak existující aplikace zmíněné v podkapitole 1.3 umožňují uživatelům pracovat s komponentou. Na obrázku 2.1 se nachází aplikace od OLTIS Group. Na pravé hraně komponenty nabízí posuvník pro změnu zobrazované části tratového úseku. Stejným způsobem komponenta umožňuje na horizontále výběr zobrazovaného časového intervalu. Uživateli je tak zobrazen jen nějaký výřez z celého obsahu nákresného jízdního řádu, který vytváří hranice, v rámci kterých je možné zobrazení měnit posuvníky. Grafické znázornění hranic a zobrazované části nákresného jízdního řádu v komponentě se nachází na obrázku 2.2. Jelikož se stejným způsobem chovají i ostatní existující aplikace a toto chování chceme zachovat i u nových aplikací pracujících s knihovnou, definujeme následující požadavky.

R1 Komponentou zobrazovaný obsah se nachází v ohraničení, které je určeno zobrazitelným obsahem nákresného jízdního řádu

R2 Komponentou zobrazovaný obsah je možné měnit posunem po vertikální a horizontální ose v rámci ohraničení definovaném **R1**



Obrázek 2.1: Aplikace od OLTIS Group s posuvníky, převzato ze stránky společnosti [3]

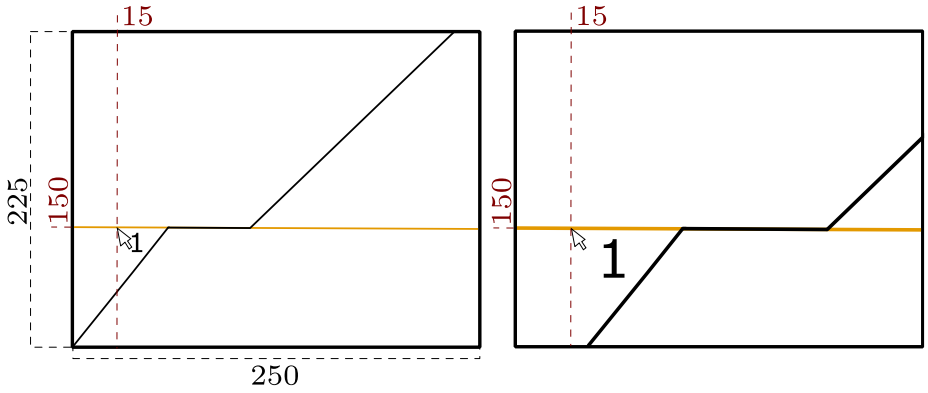


Obrázek 2.2: Komponentou zobrazovaný obsah a ohraničení určené zobrazitelným obsahem nákresného jízdního řádu

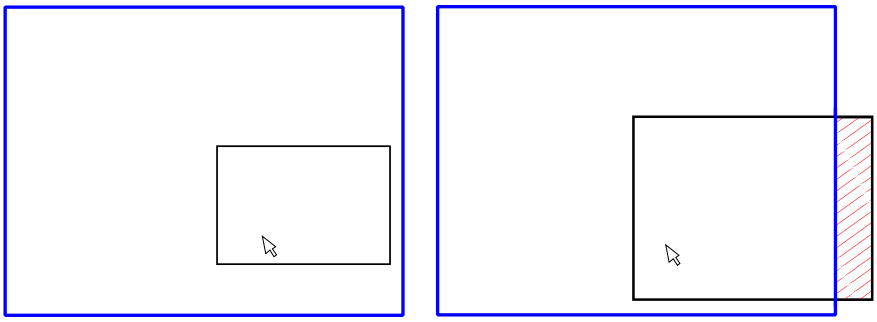
Existující aplikace umožňují měnit zobrazený obsah pouze posunem. Interakci uživatelů s grafickou komponentou bychom chtěli zlepšit přidáním možnosti zobrazovaný obsah přiblížit a oddálit, jak je ukázáno na obrázku 2.3. Pohled se bude oddalovat nebo přibližovat vůči zobrazovanému bodu, jehož umístění na plátně bude stále stejné – v případě, že uživatel bude zobrazení přibližovat myší, bude tento bod odpovídat kurzoru myši. V případě, že operace oddálení způsobí, že by se komponenta nacházela mimo ohraničení zabrazitelného obsahu a porušila by požadavek **R1**, je potřeba vzniklou situaci deterministicky opravit. Případ, kdy k takové situaci dochází, je uveden na obrázku 2.4.

R3 Komponenta podporuje operace přiblížení a oddálení pohledu. Při těchto operacích je určen v zobrazení bod, který po operaci v komponentě nezmění svou polohu

R4 V případě, že by se oddálení komponenty dostalo mimo ohraničení definované v **R1**, použije knihovna deterministický postup, který operaci oddálení vhodně opraví



Obrázek 2.3: Přiblížení pohledu na bod v komponentě, kde se nachází kurzor myši. Tento bod zůstává po přiblížení v komponentě na stejném místě.



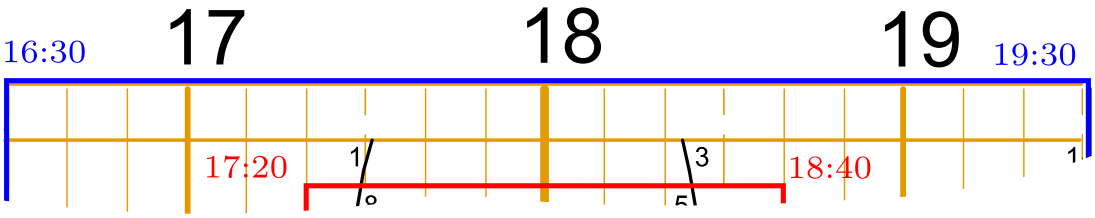
Obrázek 2.4: Situace, kdy se komponenta po oddálení nachází mimo ohraničení definované **R1**.

Mapování časových intervalů na horizontální osu

Víme, že horizontální osa grafu, v němž je nákresný jízdní řád umístěn, značí čas. V nákresném jízdním řádu pak zobrazujeme provoz z nějakého časového intervalu určeného právě časy na horizontální ose. Pokud zobrazujeme výřez z nákresného jízdního řádu, časový interval tohoto výřezu je podinterval zmíněného časového intervalu. Znázornění těchto intervalů se nachází na obrázku 2.5. Jelikož se zmíněnými intervaly budeme často pracovat, zavedeme jejich označení:

R5 Komponenta mapuje časový interval, který se nazývá *časový interval zobrazení*, na body horizontální osy komponenty

R6 Komponenta mapuje časový interval, který se nazývá *časový interval ohraničení*, na body horizontální osy v rámci ohraničení definovaného **R1**

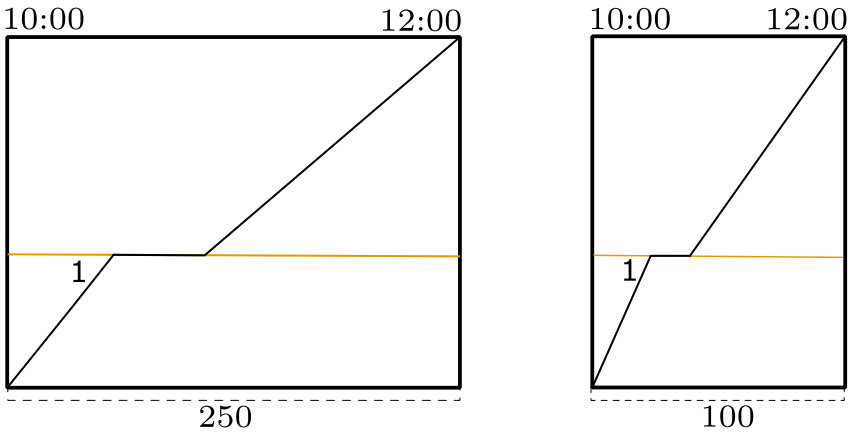


Obrázek 2.5: Časový interval zobrazení (červeně) a ohraničení (modře).

Změna velikosti komponenty a hranic obsahu

Jelikož bude komponenta obvykle součástí uživatelského rozhraní, může být její velikost v rámci procesu rozmístění uživatelského rozhraní změněna. Na obrázku 2.6 je zobrazena změna šířky komponenty. Pokud porovnáme původní a nové zobrazení po změně, vidíme, že se časový interval zobrazení nezměnil. Existující aplikace se při změně šířky komponenty chovají stejným způsobem, bude to tedy chování komponenty, které od knihovny při změně šířky požadujeme.

R7 Při změně velikosti komponenty zůstává časový interval zobrazení definovaný **R5** nezměněn

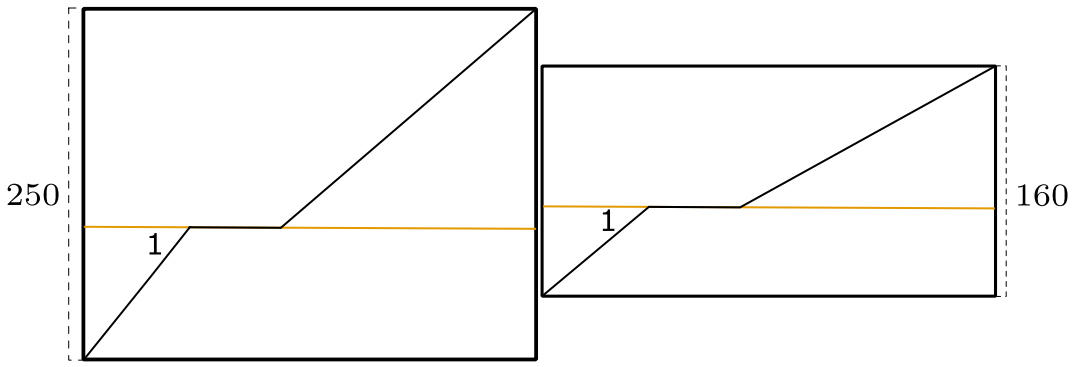


Obrázek 2.6: Nezměněný časový interval při změně šířky komponenty

Při změně výšky komponenty existuje více způsobů, jak zobrazení upravit. Jednou z možností je použít stejné chování, které se používá při změně šířky komponenty. Zobrazovaný obsah by tak zůstal stejný – v nejjednodušším scénaři budou dopravní body v komponentě rozmístěny podle poměru jejich kilometrické vzdálenosti. Chování je znázorněno na obrázku 2.7.

R8 Při změně výšky komponenty může zůstat zobrazovaný obsah komponenty nezměněn a jeho rozmístění je vzhledem k nové výšce proporčně upraveno

Problémem předchozího požadavku je chování v případě, kdy je výška komponenty příliš malá. Zobrazované informace v komponentě jsou pak nepřehledné.

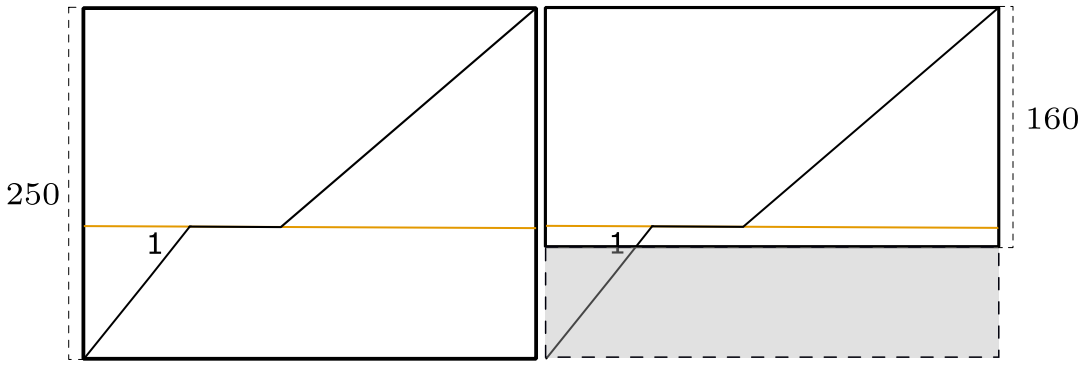


Obrázek 2.7: Proporční rozmístění stejného obsahu při změně výšky komponenty

Proto chceme vedle tohoto řešení nabídnout jiné, které zachová původní rozmístění. V případě zmenšení výšky se ze zobrazovaného obsahu zobrazí jeho výřez a nově skrytou část původního obsahu je možné zobrazit posunutím. Při zvětšení výšky se k původnímu obsahu přidá doposud skrytá zobrazitelná část navazující na původní obsah. Zvětšení výšky je možné tímto způsobem aplikovat, pokud je komponenta stále umístěna v hranicích zobrazitelného obsahu. Chování je znázorněno na obrázku 2.8.

R9 Je možné zmenšit výšku komponenty tak, aby byla zobrazena jen část původního výřezu

R10 Je možné zvětšit výšku komponenty tak, aby byla zobrazena původní část s nově odkrytou částí obsahu. Modifikace je možná, pokud zvětšení komponenty nepřekročí ohraničení definované **R1**



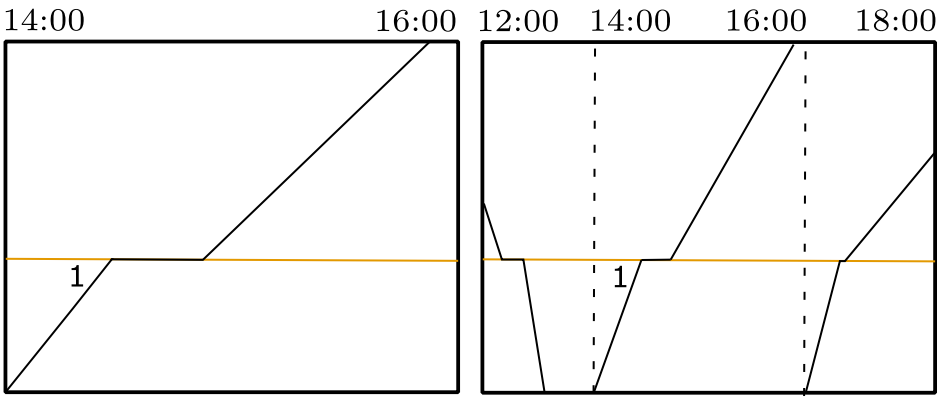
Obrázek 2.8: Změna zobrazeného obsahu při změně výšky komponenty

Modifikace časových intervalů

V této části popíšeme už zavedené modifikace ve formě nastavování časových intervalů. Je užitečné, aby zobrazovaný obsah, odpovídající nějakému časovému intervalu, bylo možné podle určení tohoto intervalu nastavit. Uvedeme si příklad, kdy se tento způsob modifikace uplatňuje. Aplikace často nabízí možnost zobrazit obsah v přednastavených úsecích několika hodin (například 2h, 4h, 6h). Pokud

například komponenta zobrazuje interval o délce dvě hodiny (14:00 - 16:00), je možné změnit obsah komponenty tak, aby nyní zobrazoval interval šesti hodin (12:00 - 18:00). Tato modifikace je zachycena na obrázku 2.9. Pokud toto chování zobecníme, požadujeme, aby mohl být nastaven jakýkoliv podinterval časového intervalu ohraničení.

R11 Zobrazení může být změněno tak, aby obsah komponenty odpovídal zobrazení nově nastavovaného časového intervalu zobrazení, který musí být součástí časového intervalu ohraničení



Obrázek 2.9: Změna časového intervalu zobrazení

Existují případy, kdy se v komponentě zobrazuje obsah odpovídající aktuálnímu času v rámci nějakého časového intervalu (například okno 12:00 - 18:00 s aktuálním časem 15:00). Periodicky se pak například po hodině celé toto časové okno o hodinu posune. Chtěli bychom proto umožnit i změnu časového intervalu ohraničení.

R12 Zobrazitelný obsah je možné změnit nastavením nového časového intervalu ohraničení. V rámci změny časového intervalu ohraničení je nutné poskytnout podinterval, který bude nastaven jako časový interval zobrazení

Navíc se knihovna musí jednotně chovat ve změně vertikální polohy komponenty v rámci ohraničení při změně časového intervalu. Jako nejvhodnější se ukazuje vertikální umístění komponenty neměnit, jelikož můžeme změnou časového intervalu vytvořit posun, zobrazující část původních informací z výřezu na stejné vertikální úrovni.

R13 Změny zobrazení určené časovými intervaly nemění vertikální umístění komponenty v ohraničení definovaném **R1**

Podporované jednotky určující velikost komponenty

Doposud jsme neurčili, v jakých jednotkách se měří velikosti komponenty a jejího ohraničení. Jelikož komponenta zobrazuje nějaké plátno, které je určeno jednotkami pixelů, požadujeme, aby s jejími velikostmi bylo možné pracovat

v různých jednotkách pixelů – například jednotkách pixelů nezávislých na zařízení a jednotkách pixelů zařízení. Pokud by se vývojář rozhodl používat jednotky pixelů zařízení, knihovna by mu měla poskytnout možnost zjistit aktuální DPI[1] hodnotu zařízení, na kterém je aplikace spuštěna, aby mohl případně zajistit třeba stejnou velikost textu mezi různými zařízeními.

R14 Velikost komponenty může být udávána v různých jednotkách pixelů:

- Jednotky pixelů nezávislých na zařízení
- Jednotky pixelů zařízení

R15 V případě práce s jednotkami pixelů zařízení bude mít vývojář možnost zjistit hodnotu DPI zařízení

Stav pro určení zobrazovaného obsahu

Předchozími požadavky jsme vytvořili základ pro prostředí, v kterém je možné pracovat se zobrazením nákresného jízdního řádu. Toto prostředí je popsáno informacemi, které tvoří jeho stav.

R16 Aby bylo možné určit, jaký obsah se má v komponentě zobrazit, zahrne komponenta do svého stavu:

- Velikosti komponenty a ohraničení
- Umístění komponenty v ohraničení definovaném **R1**
- Faktor škálování, který určuje, jak moc je zobrazení v komponentě přiblížené a odpovídá násobkům neupravené velikosti
- Časový interval zobrazení a časový interval ohraničení

2.2 Vykreslení obsahu nákresných jízdních řádů

Tato část určuje požadavky, které souvisí s vykreslováním obsahu komponenty, jejíž chování jsme určili požadavky předchozí části. Dále uvedeme požadavky, které vývojáři ulehčí přípravu nákresného jízdního řádu k jeho vykreslení. Tyto požadavky nejsou tak přesné jako v předchozí části, ale spíše určují směr, jakým se implementace knihovny musí vydat, aby mohla být dostatečně rozšiřitelná a využitelná implementacemi více typů nákresných jízdních řádů.

Plátno pro kreslení

Knihovna by měla vývojářům ulehčit práci při vykreslování nákresného jízdního řádu. Změnou stavu komponenty se neustále mění obsah, který by měl být zobrazen. Při každé změně je potřeba na komponentě zobrazit nově odpovídající část z celého nákresného jízdního řádu. Pokud bychom po knihovně požadovali co nejméně, stačilo by, aby vývojáři byla dostupná informace o stavu komponenty určena v **R16** a plátno, jehož obsah je zobrazen v komponentě. Vývojář by pak

[1] dots per inch

musel spočítat, jaký obsah se má zobrazit a tento obsah vykreslit. Takový přístup je ale pro vývojáře knihovny velmi náročný. Knihovna by mu měla nabídnout plátno, na které by se zobrazil celý obsah nákresného jízdního řádu a knihovna by pak sama přepočítala, jaký výřez z plátna se má v komponentě zobrazit.

Knihovna by ale měla nabízet i možnost pracovat pouze s částí plátna, která je zobrazena v komponentě. Tato změna je užitečná v případě, pokud chceme v komponentě zobrazit informace, které nejsou součástí nákresného jízdního řádu, ale mají informativní charakter v rámci aplikace, jako je současné datum nebo notifikace o změně související s nákresným jízdním řádem. Během kreslení tak může uživatel pohled na plátno měnit.

R17 Uživateli je přístupné plátno, které odpovídá obsahu ohraničení definovaném **R1**

R18 Uživateli je přístupný pohled na plátno definované v **R17**. Obsah tohoto pohledu je vždy vykreslen v komponentě

Převod časových údajů na horizontální osu

Jelikož vývojář pracuje s nákresným jízdním řádem, kde horizontální osa představuje časový interval, chceme, aby knihovna nabízela nástroje umožňující převod mezi horizontální polohou bodu na plátně a časovými údaji. Pokud vývojář bude kreslit průběh jízdy vlaku, nejspíše využije tento nástroj pro přepočítání časového údaje příjezdu, odjezdu nebo průjezdu do horizontální polohy na plátně, z které povede čáru značící část průběhu jízdy vlaku.

R19 Knihovna umožní mezi sebou převádět horizontální polohu bodu na plátně a časový údaj z časového intervalu, který odpovídá horizontální ose plátna

Kreslení po vrstvách

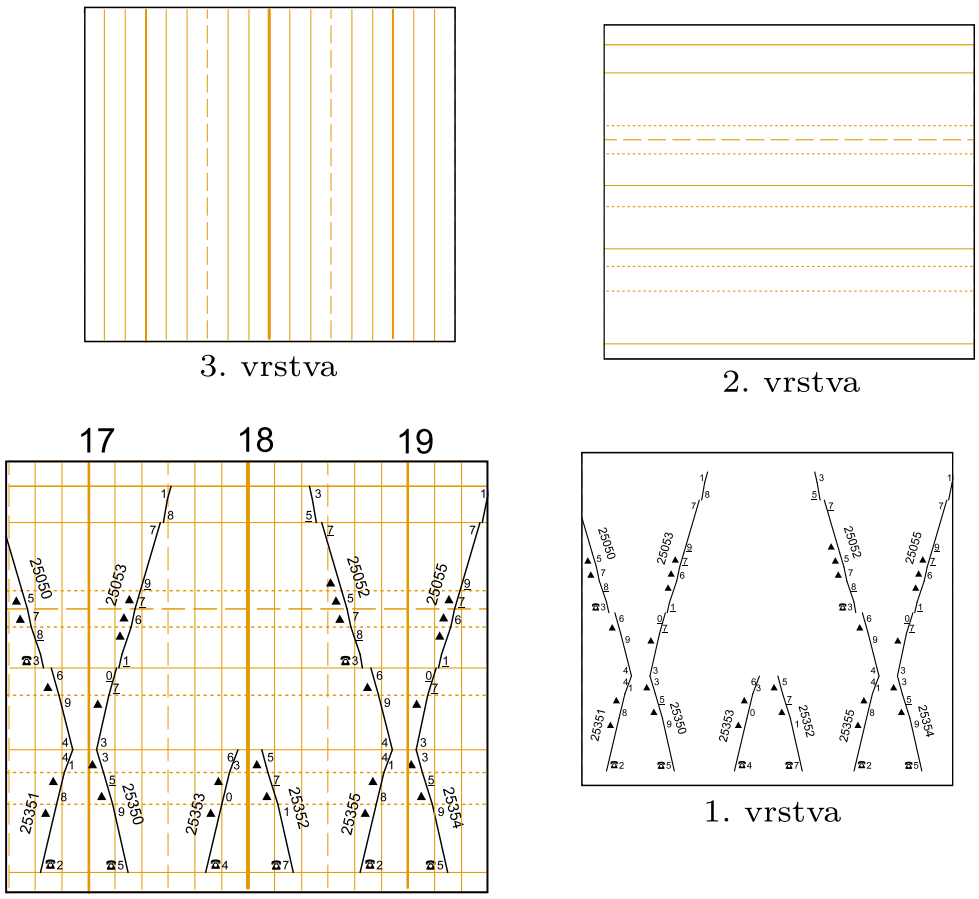
Na obrázku 2.10 jsme nákresný jízdní řád rozdělili do několika vrstev (síť vodorovných a svislých čar, šikmé čáry reprezentující průběh jízdy vlaků s kótami). Vrstvy se na sebe vykreslují v nějakém pořadí. K lepšímu strukturování obsahu komponenty umožníme vývojáři obsah do těchto vrstev rozdělit.

R20 Obsah komponenty je vykreslován ve vrstvách vytvořených vývojářem

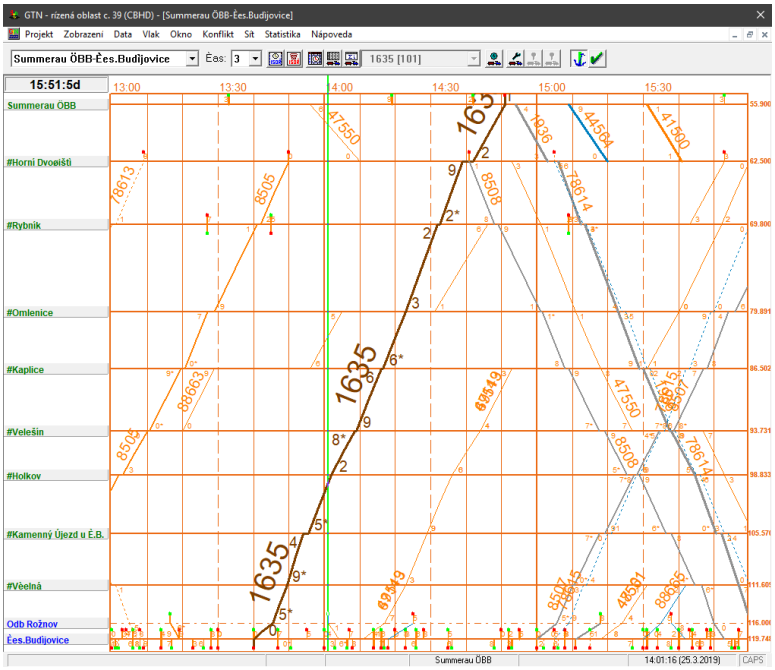
Jelikož vykreslení vrstev podle nějakého pořadí je proces, který je pro různé implementace stejný, bude knihovna poskytovat nástroj pro správu vrstev. Nástroji se předají vrstvy v pořadí definovaném vývojářem, podle kterého jsou vrstvy vykreslovány. Uvažme případ, kdy aplikace bude zvýrazňovat průběh jízdy nějakého vlaku. Pro přehlednost se šikmá čára a kóty zvýrazněného vlaku přenesou do popředí, případně se vybarví odlišně jinou barvou. Popsanou funkcionalitu nabízí na obrázku 2.11 aplikace GTN. Potřebovali bychom informace o vlaku přenést do nové vrstvy, vykreslené v popředí. Některé aplikace by navíc chtěly průběh jízdy ostatních vlaků, umístěných v nějaké vrstvě, při výběru vlaku nezobrazit.

R21 Vývojáři je dostupný nástroj, který vykresluje vrstvy **R20** podle pořadí určeného vývojářem

R22 V rámci nástroje z **R21** je možné vrstvy ke kreslení přidávat i odebírat



Obrázek 2.10: Možné rozdělení obsahu nákresného jízdního řádu na vrstvy



Obrázek 2.11: Okno aplikace GTN s vybraným vlakem vyneseným do popředí

Prvky nákresného jízdního řádu

Během změn zobrazení nákresného jízdního řádu je nutné jeho obsah pokaždé znovu vhodně rozmístit. Tento proces nazveme jako *cyklus rozmístění*[2]. Pokud například zvětšíme výšku komponenty, zobrazovaný obsah se proporcionálně přerozmístí. Jelikož se změnou výšky změnil i úhel sklonu šikmých čar, musí se znovu rozmístit prvky jako čísla vlaků nebo informace zobrazované v ostrých úhlech.

Některé implementace nákresných jízdních řádů můžou navíc obsahovat složitější konfigurace, které rozmístění komplikují. Dopravní bod například může být místo jedné horizontální čáry reprezentován souborem horizontálních čar, které odpovídají jeho kolejím. Zároveň je možné v cyklu rozmístění vyhradit úseky kolem dopravních bodů, do kterých se mají umístit informace zobrazené v ostrých úhlech. Všechny tyto prvky, které reprezentují nějakou část nákresného jízdního řádu a je nutné je rozmístit, nazveme jako *zobrazitelné prvky*. Základem pro rozmístění všech těchto prvků bude určení jejich velikosti, která může být nastavena různými způsoby, uvedených v **R24**.

Pro jednoduchost implementace různých typů nákresných jízdních řádů tak chceme vývojářům umožnit co nejvíce obsahu, který je nutné rozmístit, systematicky popsat pomocí zobrazitelných prvků. S obsahem nákresného jízdního řádu by pak bylo možné pracovat jako se strukturou stromu, jako na obrázku 2.12.

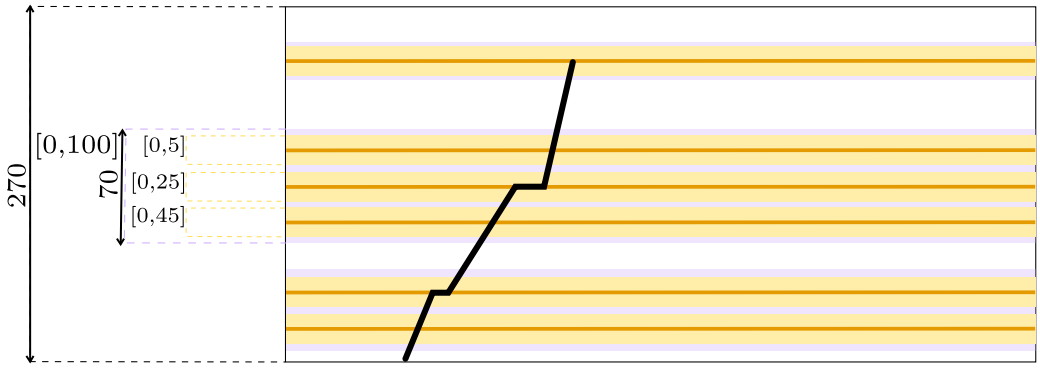
R23 Obsah nákresného jízdního řádu je možné systematicky popsat strukturou prvků, které označíme jako *zobrazitelné prvky*. Pomocí prvků je možné upravit zobrazení nákresného jízdního řádu v závislosti na nastavení komponenty.

R24 Implementace zobrazitelných prvků umožní konfigurovat jejich velikosti různými způsoby:

(a) Prvku může být přiřazena pevná velikost

(b) Prvku je přiřazena jím požadovaná velikost

(c) Prvku přiřazená velikost je závislá na nastavení komponenty



Obrázek 2.12: Rozdělení obsahu nákresného jízdního řádu do stromové struktury prvků zobrazení

[2]layout cycle

Strategie rozmisťování zobrazitelných prvků

Pro některé skupiny zobrazitelných prvků (informace v ostrých úhlech nebo čísla vlaků) chceme, aby bylo možné vybrat specifické rozmístění, které nazveme *strategií*. Výběr strategie ovlivňuje, kam a jak moc přehledně se zobrazitelné prvky těchto skupin umístí. Každý typ nákresného jízdního řádu totiž umisťuje tyto zobrazitelné prvky jiným způsobem. Aplikace, s kterými jsme se seznámili, preferují v implementaci rozmístění jednoduchost, jelikož většina prostředků a času strávených jejich vývojem byla určena na implementaci aplikační logiky a složitější návrh grafické komponenty nebyl při návrhu aplikace zvažován. Knihovna by měla umožnit přehledné rozmisťování zobrazitelných prvků, jelikož se jedná o vlastnost, která je důležitá pro uživatele aplikací při práci s nákresným jízdním řádem. S pomocí obrázku 2.13 si pro ilustraci uvedeme několik přístupů, které se můžou aplikovat při rozmisťování čísel vlaků.

- Strategie (a) je nejjednodušší na implementaci. Její jednoduchost spočívá v tom, že ve většině případů zobrazuje rozmístěné prvky dostatečně přehledně. Náhodně se vybere místo úseku mezi dopravními body, kam se má číslo vlaku umístit. Může ale dojít k problému, kdy je úsek krátký a číslo vlaku se bude překrývat s kótou.

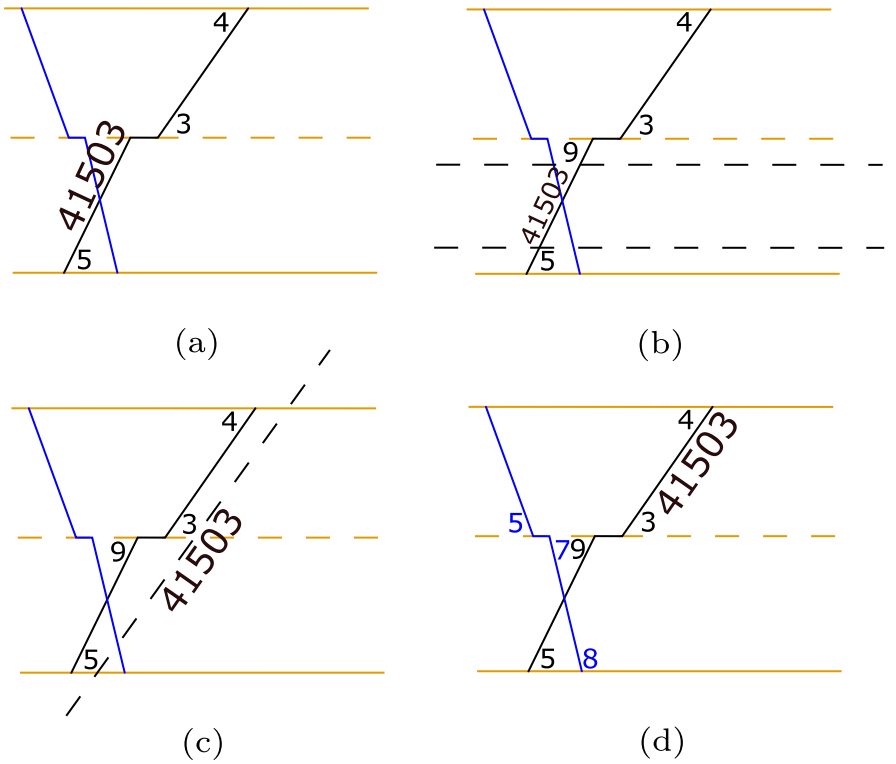
- Strategie (b) je dalším jednoduchým způsobem, jak prvky rozmístit přehledně. Strategie se snaží předejít překrývání čísla vlaku s kótami. Číslo vlaku umístí na střed do nejdelšího dostupného úseku mezi dvěma sousedními dopravními body. Pokud se číslo do úseku nevejde, strategie ho dostatečně zmenší.

- Strategie (c) umisťuje číslo na rovnou aproximaci svislé čáry mezi více dopravními body, tak, aby se nepřekrýval s kótami.

- Strategie (d) je použitelná v případech, pokud vykreslujeme nákresný jízdní řád, který se nemění, jelikož je její implementace náročná na výpočet. Strategie podle složitějšího algoritmu může určit, jak se prvky do nákresného jízdního řádu vhodně rozmístí tak, aby se nepřekrývaly.

Naše knihovna by neměla mít pevně určené strategie, které se musí používat. Vývojář si tak může zvolit strategii, která mu přijde nejvíce vhodná. Není v našich silách implementovat všechny možné strategie, ale můžeme knihovnu navrhnout tak, že zásadně ulehčí vytváření dalších strategií.

Mělo by být možné v rámci jedné strategie zobrazitelné prvky nahradit jinými, beze změny její implementace. Pro strategii totiž obecně není podstatné, s jakými prvky pracuje. Všimněme si, že s prvky pracuje jako s obdélníky s pevnou velikostí. Tento proces je zachycen na obrázku 2.14. Strategie nejdříve obdélník změří a za pomoci operací škálování a rotace obdélník vhodně umístí. Možnost oddělit strategii od konkrétních zobrazitelných prvků pak umožňuje pro více typů nákresných jízdních řádů použít již existující strategie na jejich specifické zobrazitelné prvky, jako je uvedeno na obrázku 2.15.



Obrázek 2.13: Aplikovatelné strategie při rozmístění čísel vlaků

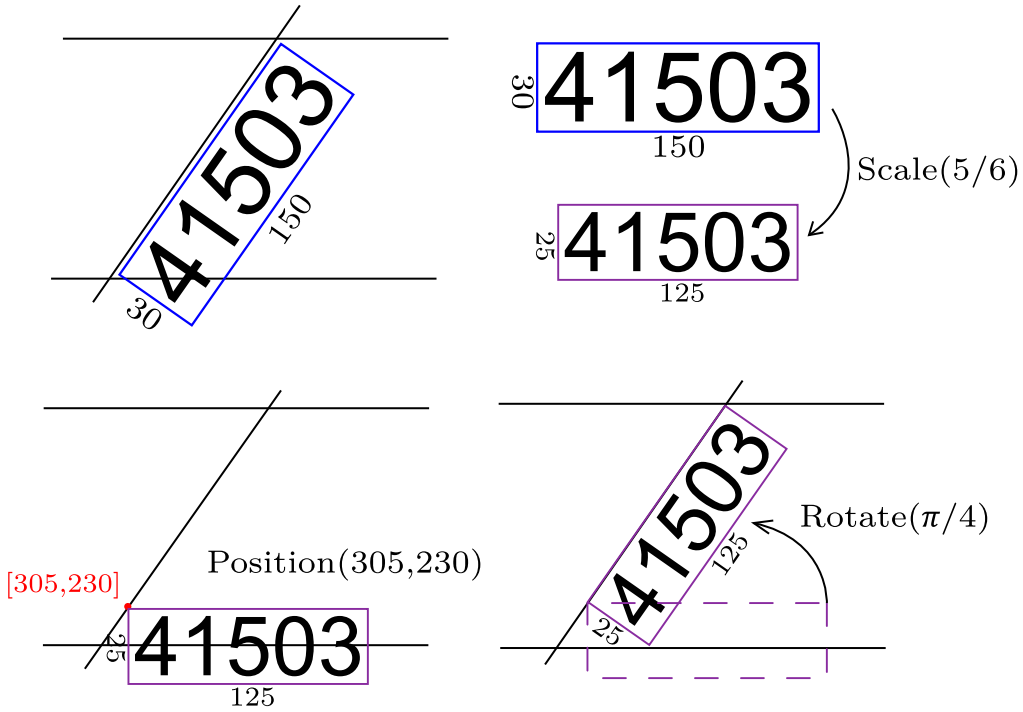
R25 Knihovna poskytne nástroje k implementaci strategií, které rozmisťují zobrazitelné prvky do specifických míst jako ostré úhly nebo oblasti podél svislých čar

R26 Strategie pracují se zobrazitelnými prvky jako s obdélníky, jejichž zobrazení a umístění je možné měnit:

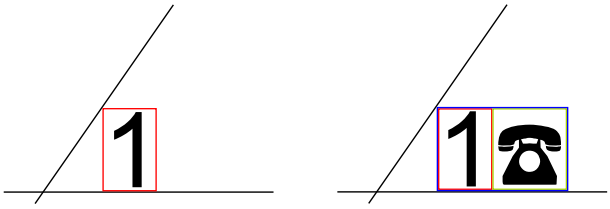
(a) Zobrazení prvku je možné strategií upravit transformací rotace

(b) Zobrazení prvku je možné strategií upravit transformací škálování

(c) Prvek je možné umístit na místo určené strategií



Obrázek 2.14: Příklad aplikování strategie na zobrazitelný prvek



Obrázek 2.15: Možnost použít stejnou strategii na jiné zobrazitelné prvky

Interakce se zobrazitelnými prvky a hit-testy

Jelikož by měla práce s nákresným jízdním řádem nabízet co největší možnou úroveň interakce, chceme, aby bylo možné při práci s komponentou pracovat interaktivně i se zobrazitelnými prvky. Pokud by uživatel aplikace například klikl na kótu nebo prvek zobrazující složitější informace, nějaká jiná komponenta uživatelského rozhraní by na tuto událost mohla zareagovat a zobrazit dodatečné informace doplňující vizualizaci v nákresném jízdním řádu. Proces, kdy zjišťujeme, zda se nějaký bod nachází uvnitř rozmístěného geometrického útvaru, se nazývá *hit-testování*.

R27 Rozmístěné zobrazitelné prvky je možné otestovat, zda leží na konkrétním bodu plátna

R28 Na základě funkcionality **R27** bude možné získat všechny zobrazitelné prvky nacházející se na specifikovaném bodu plátna podle pořadí vykreslení

Kreslení zobrazitelných prvků

Jedním ze složitých procesů, který bychom chtěli vývojářům ulehčit, je vykreslování zobrazitelných prvků. Pokud bychom vývojářům, kteří pracují na vykreslování zobrazitelných prvků, poskytli pouze plátno pro celý obsah nákresného jízdního řádu s nástroji knihovny SkiaSharp, museli by sami do kreslení prvku zahrnout vzniklé transformace a umístění prvku. Chtěli bychom, aby vývojáři mohli prvky vykreslovat způsobem, kdy by měl každý zobrazitelný prvek přidělené vlastní plátno na kreslení a vykreslení by tak probíhalo nezávisle na zobrazení v nákresném jízdním řádu.

R29 Pro vykreslení obsahu zobrazitelného prvku je vytvořeno plátno, které odpovídá velikosti prvku. Vykreslení prvku tak probíhá nezávisle na celém zobrazení nákresného jízdního řádu.

Základní model obsahu nákresných jízdních řádů

Jelikož nákresné jízdní řády mají mnoho společných vlastností, v rámci splnění cíle **G1 a**) bychom chtěli vytvořit implementaci základního modelu nákresných jízdních řádů. Jejím rozšířením bude možné vytvořit konkrétní typ nákresného jízdního řádu. V rámci základní implementace poskytneme vývojářům implementaci několika strategií.

3. Analýza implementace knihovny GTTG

Text této kapitoly podrobně analyzuje možné přístupy k implementaci částí knihovny GTTG, aby byly splněny cíle **G1 a**) - **d**) uvedené na konci kapitoly 1 a požadavky **R1** - **R29** uvedené v kapitole 2.

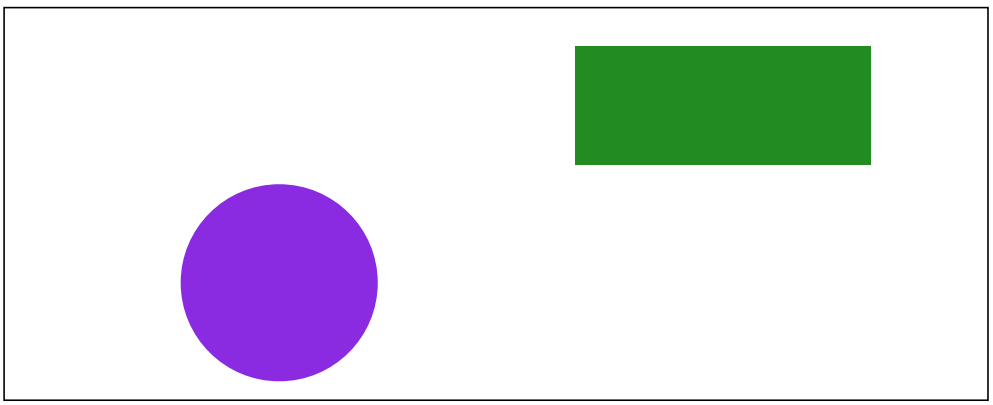
3.1 Implementace komponenty a prostředí pro kreslení

Jedním z cílů pro vývoj knihovny je zajistit její integrovatelnost do aplikací (cíl **G1b**)). Toho dosáhneme pomocí správné implementace komponenty. Jelikož knihovna SkiaSharp bude při určování implementace komponenty a integrace knihovny do aplikací hrát zásadní roli, nejdříve se před samotným rozborem implementace seznámíme s jejími základními principy fungování.

3.1.1 Úvod do práce s knihovnou SkiaSharp

Základem pro kreslení v knihovně SkiaSharp je třída `SKCanvas`, která představuje plátno pro kreslení. Mezi instanční metody této třídy patří kreslící příkazy (například `DrawText()`, `DrawCircle()`), které se na základě dodaných parametrů nanáší na plátno. Jelikož je knihovna SkiaSharp 2D, pořadí nanášení příkazů na plátno je určeno pořadím volání metod. Kreslení na plátno, vzniklé jako výsledek aplikování několika kreslících příkazů na následujícím fragmentu kódu, je uvedeno na obrázku 3.1.

```
1: var paint = new SKPaint {Color = SKColors.Green, Style = SKPaintStyle.Fill};
2: canvas.DrawRect(SKRect.Create(x: 145, y: 10, width: 75, height: 30), paint);
3: paint.Color = SKColors.BlueViolet;
4: canvas.DrawCircle(new SKPoint(x: 70, y: 70), radius: 25, paint: paint);
```

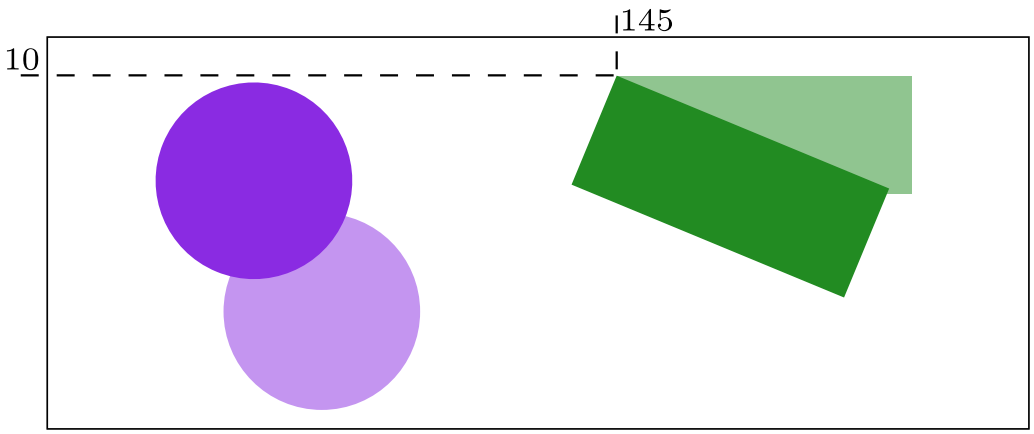


Obrázek 3.1: Kreslení na plátno odpovídající kreslícím příkazům v předchozím fragmentu kódu

Transformace kreslících příkazů

Plátno `SKCanvas` je možné konfigurovat přenastavením matice označované jako `transform matrix` a výřezu plátna, který se nazývá `clip`. Část kreslícího příkazu, který se nachází mimo `clip`, se nevykreslí. Nastavení `transform matrix` ovlivňuje výsledek kreslících příkazů. Na obrázku 3.2 se na plátno aplikovaly stejné příkazy jako při kreslení na obrázku 3.1, pouze se před jejich aplikací nastavila `transform matrix` na následujícím fragmentu kódu. Obsah výkresu je nyní transformován rotací o $\pi/8$ po směru hodinových ručiček se středem v levém horním rohu zeleného obdélníku.

```
1: var pivotX = 145; var pivotY = 10;
2: SKMatrix.Rotate(ref matrix, (float) Math.PI / 8, pivotX, pivotY);
3: canvas.SetMatrix(matrix);
```



Obrázek 3.2: Výkres 3.1 se změněnou `transform matrix`. Původní výkres je pro porovnání uveden ve stínu

Konfigurace `transform matrix` se vykonává voláním `SetMatrix()`, `clip` se aplikuje vzhledem k současné `transform matrix` například pomocí `ClipRect()`. `Transform matrix` i `clip` je možné efektivně měnit a vracet do původních stavů, jelikož konfigurace jsou plátnem ukládány na jeho zásobník modifikací. Pro každou sérii kreslících příkazů je tak možné použít jinou transformaci.

Sestrojení plátna

Pokaždé, když chceme pracovat s knihovnou SkiaSharp, musíme pro kreslení získat instanci plátna `SKCanvas`. Plátno může být sestrojeno nad existující bitmapou nebo může být vytvořeno jako instanční proměnná třídy `SKSurface`, která je zodpovědná za práci s nástrojem poskytující pixely, na které `SKCanvas` kreslí. Je tak možné rozhodnout, jestli jsou pixely alokovány v paměti nebo GPU. Dále je možné, aby SkiaSharp vykreslovala obsah plátna do PDF nebo jako vektorovou grafiku v SVG. Jednou z vlastností `SKCanvas` i `SKSurface` je, že po vytvoření již nemůžou měnit svou velikost.

Podpora v rámci uživatelských rozhraní

Knihovna SkiaSharp nabízí pro některé GUI frameworky implementaci prvku uživatelského rozhraní, jehož plocha se používá jako plátno `SKCanvas`. To přináší vývojářům zásadní ulehčení jejich práce, jelikož prvek sám sestrojí `SKSurface` a při změně velikosti prvku vytvoří nový s upravenou velikostí. Kdykoliv, když je vizualizace prvku invalidována, je obnoven kreslící cyklus, v kterém vývojář pracuje s plochou prvku uživatelského rozhraní jako s `SKCanvas`. Vývojáři díky těmto implementacím nemusí pracovat s kreslícím *backendem*, který by museli zapojit do `SKSurface` za poskytnutí mnoha parametrů popisující jeho nastavení.

3.1.2 Integrace knihovny do aplikací

Na základě zmíněných vlastností knihovny SkiaSharp si nyní popíšeme, jak naše knihovna bude integrována do aplikací. V rámci této integrace je nutné určit, jaké nástroje musí vývojář knihovně dodat.

Reprezentace jednotek pixelů

V rámci implementace komponenty můžeme splnit požadavky **R14** a **R15** související s existencí plátna v různých jednotkách pixelů. Uvedli jsme si, že SkiaSharp nabízí implementaci UI prvků určených pro kreslení. U těchto prvků je možné nastavit jednotky pixelů plátna `SKCanvas`. V prvku `SKElement` GUI frameworku WPF se jednotky pixelů nastavují vlastností `IgnorePixelScaling` uvedené na následujícím fragmentu kódu.

```
1:  public class SKElement : FrameworkElement {
2:
3:  /// <summary>
4:  /// Gets or sets a~value indicating whether the drawing canvas
5:  /// should be resized on high resolution displays.
6:  /// </summary>
7:  /// <remarks>
8:  /// By default, when false, the canvas is resized to 1 canvas
9:  /// pixel per display pixel. When true, the canvas is resized to device
10: /// independent pixels, and then stretched to fill the view. Although
11: /// performance is improved and all objects are the same size on different
12: /// display densities, blurring and pixelation may occur.
13: /// </remarks>
14: public bool IgnorePixelScaling {
```

Komentář vysvětluje, že pokud pixely plátna `SKCanvas` odpovídají pixelům nezávislých na zařízení, je dodána `SKSurface` bitmapa v jednotkách pixelů s DPI 96, které odpovídá ve WPF jednotkám pixelů nezávislých na zařízení. Bitmapa pak pro každý takový pixel může v závislosti na zařízení použít několik fyzických pixelů.

Pokud bychom řešili tento problém obecně, plátno by bylo vytvořeno ve velikosti odpovídající ploše prvku v jednotkách pixelů zařízení a více pixelů zařízení by mohlo odpovídat jednomu pixelu nastavením škálování v `transform matrix` plátna. Pokud bychom velikost komponenty určovali pomocí velikosti plátna `SKCanvas`, nastala by tímto způsobem nekonzistence v reprezentaci velikostí – v případě nastavení `IgnorePixelScaling` je plátno udáváno v konečné

velikosti pixelů nezávislých na zařízení a v případě nyní zmiňovaného způsobu by odpovídalo velikosti pixelů zařízení, ačkoliv v obou případech chceme v komponentě pracovat s jednotkami pixelů nezávislých na zařízení. I přes tyto nevýhody bychom chtěli takovou konfiguraci umožnit.

Rozhodli jsme se proto, že velikost komponenty bude explicitně uváděna jiným způsobem a s velikostí `SKCanvas` knihovna pracovat nebude. Vývojář, který bude chtít použít zvětšování pomocí `transform matrix`, sám matici nastaví a knihovna mu tuto konfiguraci nezmění. Protože existují různé způsoby, jak můžou být pro `SKSurface` alokovány pixely, nechceme v knihovně vytvářet nové plátno v bitmapě a budeme po vývojáři požadovat, aby dodal vlastní `SKSurface` s `SKCanvas`. Jiná implementace by byla neefektivní vzhledem k možnostem knihovny SkiaSharp. Zároveň požadujeme, aby vývojář provedl všechna doposud popisovaná nastavení sám mimo knihovnu. Pokud se komponentě předávají body z plochy prvku, která pracuje s jinými jednotkami pixelů, vývojář musí převod jednotek sám implementovat. Jelikož už vývojář v této konfiguraci pracuje s hodnotou DPI, která je uvedena v požadavku **R15**, hodnotu DPI komponentě nastaví sám. Všechny tyto konfigurace se tak odehrávají na stejné úrovni a knihovna bude konzistentně pracovat se stejnými jednotkami velikosti.

3.1.3 Reprezentace časových intervalů

Musíme najít způsob, jak reprezentovat časové intervaly definované **R5** a **R6**. Zároveň **R19** požadujeme, aby bylo možné časové údaje používané v reprezentaci intervalů převádět do bodů plátna. Převod by měl být co nejméně komplikovaný. Doposud jsme si uváděli situace, kdy časový interval nepřesáhne několik hodin a proto by se nabízelo použít k reprezentaci časových údajů struktury, které reprezentují pouze čas, ale neobsahují už informaci o datu. Každá aplikace ale pracuje s jinou reprezentací časových údajů v jejím modelu dat. Existují aplikace, které sledují současný provoz a v časových údajích zahrnují i datum.

K reprezentaci časových údajů v knihovně jsme se proto rozhodli použít strukturu `DateTime`. Převod jiných struktur pracujících s časovými údaji do hodnot `DateTime` není komplikovaný a převod v rámci **R19** je přímočarý – rozdíl konce a začátku intervalu vytvoří strukturu `TimeSpan` obsahující hodnotu `TotalMilliseconds`, s jejíž násobky jde dále při převodu pracovat. Při použití data i času nemůže docházet k nejednoznačnostem při porovnávání a převodu časových intervalů.

3.1.4 Modifikace komponenty

Modifikace komponenty jako translace nebo škálování nesmí překročit ohraničení definované **R1**. Jelikož se chybné modifikace mohou při interakci uživatele s komponentou vyskytovat běžně, nechceme, aby vyhazovaly výjimky. Zároveň ale chceme vývojáře informovat o úspěšnosti modifikace, jelikož na jejím provedení může být například závislé vykreslení obsahu a nechceme zbytečně vykreslovat modifikací nezměněný obsah. Rozhodli jsme se, že každé volání modifikace bude mít jako návratovou hodnotu výčtový typ, který bude poskytovat informace o úspěšnosti modifikace. Na následujícím fragmentu kódu je příklad takového výčtového typu pro modifikaci **R3** škálující zobrazení:

```
1: public enum ScaleTransformationResult {
2:     ViewModifiedWithTransformedOrigin,
3:     ViewModifiedWithSameOrigin,
4:     ViewUnmodified
5: }
6:
7: public ScaleTransformationResult TryScale(SKPoint origin, float delta) {
8:     /*...*/
9: }
```

Pro případ, kdy nastane stav **R4** a pohled by se vyskytl mimo ohraničení **R1**, je vrácen `ViewModifiedWithTransformedOrigin`. Rozhodli jsme se, že se pohled v rámci volání `TryScale()` deterministicky upraví tak, aby byl posunut o délku, kterou v ohraničení překračuje. V případě, že faktor škálování je menší než jedna, je vrácen `ViewUnmodified`.

Práce s komponentou ve vícevláknovém prostředí

Může docházet k případům, kdy vývojář bude ke knihovně přistupovat z více vláken. Například je v rámci aplikace spuštěn periodický časovač, který běží v dalším vlákně a opakovaně upravuje nějakou část komponenty, aby odpovídala modelu dat v aplikaci. Pokud by se taková změna stavu komponenty z jiného vlákna vyskytla během vykreslování nebo jiných změn v komponentě, může se vyskytnout chybový stav označovaný jako *race condition*. Chceme určit postup, jak systematicky těmto problémům předcházet. Jelikož knihovna bude obvykle součástí uživatelských rozhraní, která řeší podobné problémy pro jejich prvky, nabízí se jejich řešení rozšířit i na naší knihovnu. Každé uživatelské rozhraní definuje svá pravidla pro práci ve vícevláknovém prostředí označovaná jako *threading model*. Většinou je pravidlem povolovat u každého prvku změnu jeho stavu pouze z vlákna, které se nazývá *UI vlákno*. Kdykoliv uživatel změní stav prvku z jiného vlákna, je vyhozena výjimka. Práci z jiného vlákna modifikující stav prvku je však možné zařadit do UI vlákna přes strukturu obecně nazývanou *dispatcher*. Takto je řešena synchronizace na úrovni prvků uživatelského rozhraní. Například v GUI frameworku WPF je každý UI prvek potomkem abstraktní třídy `DispatcherObject`, která nabízí nástroje pro ověření přístupu přes metody uvedené na následujícím fragmentu kódu. Při nastavování vlastností prvků uživatelského rozhraní se pak kontroluje přístup v rámci volání `SetValue()` a metody přístup kontrolují přes `VerifyAccess()`.

```
1: public abstract class DispatcherObject {
2:
3:     public void VerifyAccess() {
4:       this._dispatcher?.VerifyAccess();
5:     }
6:
7:     public void SetValue(DependencyProperty dp, object value) {
8:       this.VerifyAccess();
```

Pokud bychom se rozhodli řešit tento problém na úrovni knihovny, museli bychom najít řešení, které bude jednoduché na implementaci a zajistí konzistentní chování při práci s celým obsahem knihovny. Nemůžeme převzít žádné

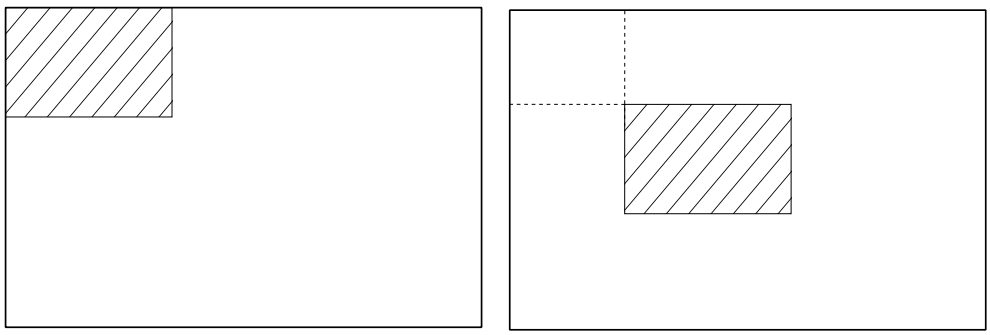
konkrétní řešení, které existuje v rámci nějakého GUI frameworku, jelikož jedním z cílů je zajistit přenositelnost knihovny na úrovni .NET Standard (cíl **G1d**)). Mohli bychom implementovat vlastní řešení, které by se chovalo podobně jako zmíněný `DispatcherObject` a dispatcher, ale zajistit jeho správnost i udržitelnost by bylo náročné. Nalezením řešení na úrovni knihovny bychom také způsobili zpomalení aplikací, které budou ke kódu knihovny přistupovat z jednoho vlákna. Z těchto důvodů jsme se rozhodli, aby vývojář řešil problémy synchronizace na úrovni uživatelského rozhraní s nástroji, které GUI frameworky nabízí. Následující fragment kódu z vlákna běžícího vedle UI vlákna v aplikaci ve WPF zařadí do UI vlákna přes `Dispatcher.Invoke()` vykonání lambda funkce, která přidá nový vlak do modelu vykreslovaného komponentou.

```
1: Train newTrain = /*...*/;
2:
3: Dispatcher.Invoke(() => {
4:     Model.Trains.Add(newTrain);
5: });
```

3.1.5 Reprezentace plátna v knihovně

Podle požadavku **R17** chceme vývojářům poskytnout plátno nákresného jízdního řádu, které má být nastaveno tak, aby v komponentě byl zobrazen jeho výřez v závislosti na jejím stavu. Toto plátno nazveme `ContentCanvas`. Implementaci jeho nastavení v komponentě se nyní budeme věnovat.

Pro vykreslení správného výřezu plátna `ContentCanvas` můžeme nastavit `transform matrix` na plátně `SKCanvas`, které podle 3.1.2 knihovně dodává vývojář k vykreslení. V počátečním stavu zobrazeném na obrázku 3.3a je `transform matrix` identita. Modifikace komponenty popsaná **R2** odpovídá operaci translace, **R3** odpovídá škálování. Upravená `transform matrix` po modifikaci posunem je zobrazena na obrázku 3.3b. Bod [300,300] na `ContentCanvas` se nyní mapuje na [0,0] a vykreslí se na levém horním rohu komponenty.



$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

(a) Identita

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -300 & -300 & 1 \end{pmatrix}$$

(b) Translace

Nastavení `transform matrix` plátnu `SKCanvas` pomocí `SetMatrix()` se v knihovně Skia v tomto případě optimalizuje – její implementace je schopna zjistit, že nastavovaná matice odpovídá matici posunu a při následujícím aplikování kreslících příkazů neproběhne obecná transformace násobením s nastavenou maticí, ale pouze se k bodům kreslícího příkazu přičte po osách délka posunu. Protože jiná implementace by byla neefektivní vzhledem k možnostem knihovny Skia, rozhodli jsme se výřez nastavit tímto způsobem.

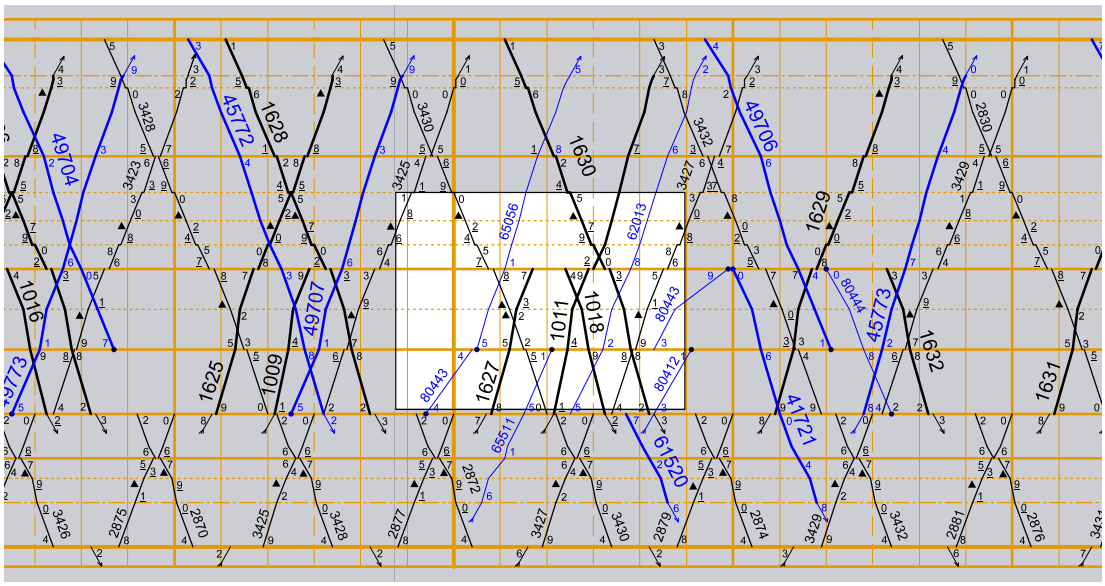
Vykreslování obsahu mimo zobrazovanou oblast v komponentě

Ve dvou bodech si uveďme, jaké výhody pro vývojáře knihovny přináší používání plátna `ContentCanvas`:

1. Pro vývojáře se oddělil proces zobrazení obsahu komponenty od vykreslení obsahu nákresného jízdního řádu. Díky používání `ContentCanvas` vývojář nemusí počítat, jaký jeho výřez se v komponentě zobrazí a může vykreslit celý obsah nezávisle na tomto zobrazení.

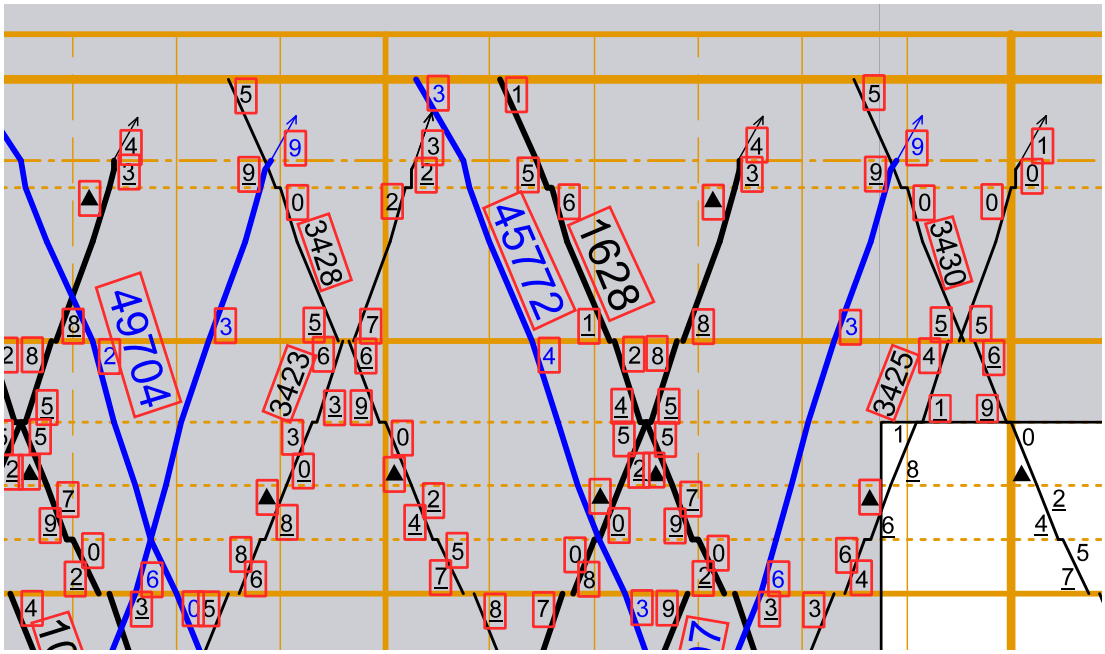
2. Výpočet, v němž se rozmístí obsah zobrazený v komponentě a který jsme nazvali *cyklem rozmístění*, probíhá pro celý obsah plátna `ContentCanvas`. Cyklus rozmístění musí probíhat v případě změny velikosti komponenty. Modifikace měnící zobrazovaný výřez v komponentě s cyklem rozmístění nesouvisí.

Používání plátna `ContentCanvas` může vedle těchto výhod představovat výkonnostní problém. Na obrázku 3.4 se nachází šedě podbarvená část jeho obsahu, která se v komponentě nezobrazí a přesto se vykresluje. Ušetřený čas za nehledání zobrazovaného obsahu tak může být převážen vykonáním kreslících příkazů, které se nezobrazí.



Obrázek 3.4: Šedě podbarvený obsah `ContentCanvas` nacházející se mimo zobrazení komponenty

Hledání zobrazovaného obsahu budeme místo vývojáře řešit na úrovni knihovny efektivní metodou, která je založena na následujícím pozorování. Podívejme se, na jaký obsah `ContentCanvas` se používá nejvíce kreslících příkazů. Souřadnicová síť jako několik čar nepředstavuje pro kreslení zásadní problém, stejně tak i lomené čáry znázorňující průběh jízdy vlaků. Ke každému vlaku ale existuje několik zobrazitelných prvků, které často obsahují pro kreslení složitý obsah, jakým je například text. Na obrázku 3.5 jsme v části obsahu, která se v komponentě nezobrazí, ohraničili zobrazitelné prvky červenými obdélníky.



Obrázek 3.5: Zobrazitelné prvky označené červenými obdélníky se nacházejí mimo zobrazení komponenty

Toto pozorování nyní spojíme s prací se zobrazitelnými prvky v rámci cyklu rozmístění. Strategie, které jsme si představili v 2.2, v cyklu rozmístění pracují se zobrazitelnými prvky jako s obdélníky, které někam umístí a transformacemi rotací a škálování je upraví. Tyto obdélníky odpovídají červeným obdélníkům na obrázku 3.5. Jelikož probíhá cyklus rozmístění na celém obsahu, jsou všechny zobrazitelné prvky před vykreslením již rozmístěny a obdélníky je tak možné určit. Zobrazitelný prvek vykreslíme pouze v případě, pokud se část jeho obdélníku nachází v komponentě.

Nalezli jsme řešení, které je založeno na existující implementaci knihovny a vzniká jako vedlejší výsledek rozmístění zobrazitelných prvků v cyklu rozmístění. Kreslení obsahu tak zabere méně času než v případě, kdy se vykreslí celý obsah nebo když by vývojář používal své vlastní řešení pro hledání zobrazeného obsahu v komponentě.

3.2 Práce s obsahem nákresného jízdního řádu

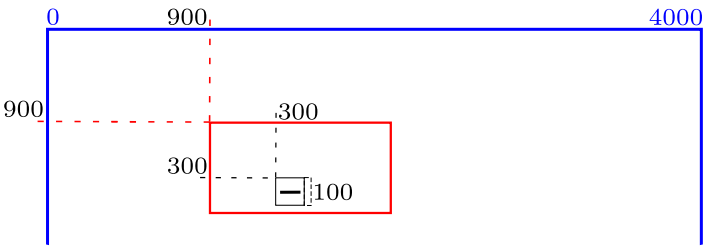
V této části určíme, jak implementovat knihovnu tak, aby umožnila tvorbu různých typů nákresných jízdních řádů (cíl **G1a**)) a byly splněny požadavky kapitoly 2, které souvisí s prací s obsahem nákresného jízdního řádu.

3.2.1 Implementace zobrazitelných prvků

Zobrazitelným prvkům chceme v cyklu rozmístění podle **R24** přidělovat různými způsoby jejich velikost. S velikostí a umístěním prvku pracují požadavky *hit-testování* **R27**, vykreslování **R29** a umístění do strategií **R26**. Před zvážením možné implementace cyklu rozmístění prvků určíme, jak na ní bude závislá implementace těchto požadavků.

Vykreslení zobrazitelných prvků

Pro vykreslování zobrazitelných prvků chceme vývojářům podle **R29** předávat vlastní plátno, které bude odpovídat velikost prvku. V podkapitole 3.1.2 jsme se rozhodli, že knihovna bude pracovat pouze s jedním plátnem – `SKCanvas` poskytnutým vývojářem. Plátno pro zobrazitelný prvek tak bude představovat vytvoření pohledu na toto plátno, podobně jako `ContentCanvas`, jehož pohled jsme nastavili pomocí `transform matrix`. Na základě výhod uvedených v 3.1.5 jsme se také rozhodli pro vykreslení plátna zobrazitelných prvků nastavit `transform matrix`. Hodnoty této matice budou závislé na umístění prvku na `ContentCanvas` a `transform matrix` plátna `ContentCanvas`. Umístění prvku budeme reprezentovat maticí, kterou nazveme `placement matrix` a abychom získali `transform matrix` prvku, vynásobíme ji s `transform matrix` plátna `ContentCanvas`. Obrázek 3.6 obsahuje zobrazitelný prvek, pro nějž tuto matici vypočítáme.



Obrázek 3.6: Příklad zobrazitelného prvku v komponentě a plátně `ContentCanvas`

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ -900 & -900 & 1 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 1200 & 1200 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 300 & 300 & 1 \end{pmatrix}$$

Obrázek 3.7: Výpočet `transform matrix` zobrazitelného prvku pomocí modré `transform matrix` plátna `ContentCanvas` a červené `placement matrix`

Podle R26 chceme navíc zobrazení modifikovat rotací a škálováním v rámci strategií. Tyto transformace pak budou zaznamenány v `placement matrix`. K násobení bychom mohli použít obecné maticové násobení implementované knihovnou SkiaSharp, které se ale provolává do C++ kódu knihovny Skia. Jelikož ale pracujeme pouze s maticovými operacemi rotace, posunu a škálování, můžeme násobit pouze některé prvky matic. Na následujícím fragmentu kódu je výsledná operace vytvářející `transform matrix` prvku, která zabere pouze šest násobení.

```
1:  var canvasScaleX = canvasTransformMatrix[0];
2:  var canvasScaleY = canvasTransformMatrix[4];
3:  var canvasTransX = canvasTransformMatrix[2];
4:  var canvasTransY = canvasTransformMatrix[5];
5:
6:  /* Other indices set to 0 */
7:  var elementTransformMatrix = new SKMatrix {
8:      /*A[3,3]*/ Persp2 = 1,
9:      /*A[1,1]*/ ScaleX = placementMatrix.ScaleX * canvasScaleX,
10:     /*A[1,2]*/ SkewY = placementMatrix.SkewY * canvasScaleY,
11:     /*A[2,1]*/ SkewX = placementMatrix.SkewX * canvasScaleX,
12:     /*A[2,2]*/ ScaleY = placementMatrix.ScaleY * canvasScaleY,
13:     /*A[3,1]*/ TransX = placementMatrix.TransX * canvasScaleX
14:                         + canvasTransX,
15:     /*A[3,2]*/ TransY = placementMatrix.TransY * canvasScaleY
16:                         + canvasTransY
17: };
```

Cyklus rozmístění zobrazitelného prvku bude muset pro výpočet matice `placement matrix` prvku nastavit jeho umístění na `ContentCanvas` a jeho hodnoty škálování a rotace při transformaci strategií.

Vliv strategií na cyklus rozmístění

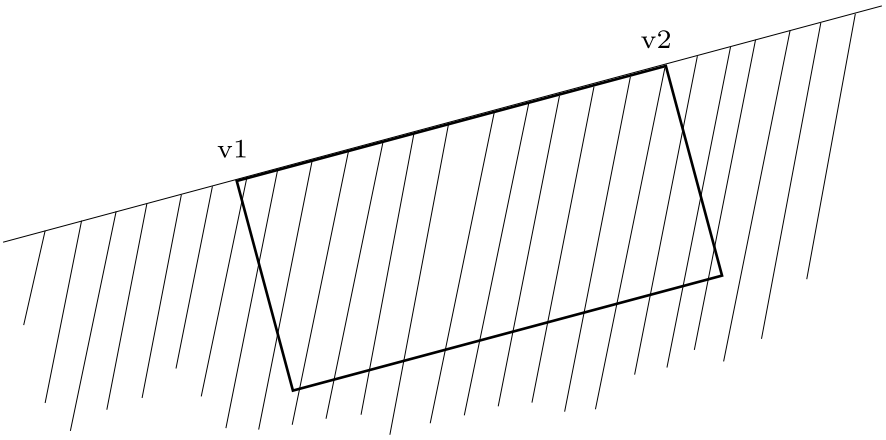
Podle požadavku R24 chceme zobrazitelné prvky strategiemi transformovat operacemi škálování a rotace. Transformace může mít dopad pouze na vykreslení prvku nebo může změnit i jeho velikost. Operace škálování by například mohla prvku proporcionálně přidělit jinou velikost a prvek by na změnu reagoval. Měnila by se tak i velikost plátna pro prvek a zároveň s ní i umístění jeho kreslících příkazů. Taková implementace neodpovídá původnímu záměru prvky vykreslovat pokud možno nezávisle na zobrazení. Proto jsme se rozhodli transformace implementovat tak, aby strategie prvky pouze vizuálně transformovaly pomocí `placement matrix`. Během transformací se ale mění i umístění prvků, pokud má prvek potomky. Cyklus rozmístění pak bude muset pro celý strom prvků při rotaci přepočítat umístění prvků a jejich hodnoty pro výpočet `placement matrix`.

Umístění prvků na plátně

Podle R27 chceme dokázat zjistit, zda se nějaký bod plátna `ContentCanvas` nachází v zobrazitelném prvku umístěném na tomto plátně. Jelikož můžou být prvky transformovány rotací a škálováním, určení, zda se bod v prvku nachází, může představovat složitý problém. Prvek umístěný v plátně můžeme reprezentovat jako obdélník. K jeho určení potřebujeme znát umístění, rotaci a škálování prvku. Podle předchozí části dokážeme vytvořit `placement matrix`, která v sobě

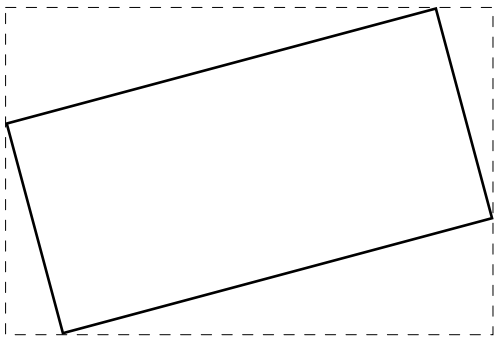
nese tyto transformace. Pomocí této matice reprezentové `SKMatrix` dokážeme metodou `MapPoint()` levý horní vrchol obdélníku [0,0] nebo dolní pravý vrchol odpovídající velikosti prvku přetransformovat na body v plátně `ContentCanvas`.

Tyto body by pak tvořily přímky popisující obdélník na plátně. Každá přímka určuje polorovinu, v které se nachází obsah obdélníku, jako na obrázku 3.8.



Obrázek 3.8: Polorovina určená hranou obdélníku

Pokud se prvek nachází ve všech čtyřech polorovinách, je součástí obdélníku. Aby se pokaždé složitě neporovnávalo umístění přes poloroviny, můžeme vytvořit obdélník, který nazveme `bounding rectangle` a bude ohraničovat vrcholy původního obdélníku, jako na obrázku 3.9, kde je zobrazen `bounding rectangle` prvku transformovaného rotací.



Obrázek 3.9: Čárkovaný `bounding rectangle` zobrazitelného prvku

Rozhodli jsme se, že hit-testing bude na prvcích transformovaných rotací implementován tak, že se bod nejdříve porovná s `bounding rectangle` a v případě úspěchu se provede přesné porovnání pomocí polorovin. V případě prvků netransformovaných rotací stačí k přesnému porovnání použít pouze `bounding rectangle`. `Bounding rectangle` by bylo možné použít i k určení, zda se prvek nachází v komponentě (3.1.5) – `bounding rectangle` by odpovídal červeným obdélníkům na plátně na obrázku 3.5. V případě transformace rotací zabírá `bounding rectangle` o něco více prostoru než je skutečná velikost prvku, ale

jeho porovnání je rychlejší než přesné porovnání pomocí polorovin. Odhad, který používáme, v nejhorších případech způsobí vykreslení velmi malého počtu prvků navíc.

Implementace zobrazitelných prvků

Implementaci zobrazitelných prvků bychom mohli reprezentovat rozhraním s informacemi, které jsme si v předchozích částech této podkapitoly uvedli. Toto rozhraní označíme jako `IViewElement`. Výhodou rozhraní by bylo, že vývojář může implementovat vlastní cyklus rozmístění z **R24** podle konkrétního prvku metodami mimo rozhraní, odpovídající potřebám jeho implementace. Pokud bychom například měli zobrazitelný prvek, který by představoval dopravní bod s několika kolejemi, v metodě umisťující prvek na plátno se nastaví, jestli má horizontální čáry jako koleje seskupit kvůli nedostatku prostoru na plátně do jedné čáry nebo je má vykreslit pro větší přehlednost vedle sebe.

Komplikací je, že pro implementaci rozhraní vývojář musí vypočítat přesné umístění prvků na plátně `ContentCanvas`. V případě práce se stromem prvků popisující obsah nákresného jízdního řádu podle **R23** by vývojář spíše preferoval možnost potomky prvku umístit v rámci jeho souřadnic. Převod do souřadnic plátna `ContentCanvas` by se tak prováděl jen kvůli knihovně.

Další komplikací je aplikování strategií na prvky implementující toto rozhraní. Pokud bychom strom prvků přemístili nebo ho transformovali škálováním a rotací, musí se změnit hodnoty pro výpočet `placement matrix` všech jeho prvků. Tento problém by musely řešit strategie úpravou hodnot rozhraní nebo by `IViewElement` mohlo obsahovat metody jako `Scale()` nebo `Rotate()`, které by strategie volaly a úpravu stromu prvků by implementoval vývojář. V předchozí části věnující se vlivu aplikování strategií na cyklus rozmístění jsme si ale uvedli, že by tyto operace měly implementaci prvku vývojářem ovlivnit co nejméně.

Informace rozhraní `IViewElement` se převádí na různé struktury jako `placement matrix` nebo `bounding rectangle`. Všechny tyto převody by se odehrávaly během vykreslování, které ale nechceme zbytečně zpomalovat. Bylo by vhodnější, kdyby se už tyto struktury předpočítaly během cyklu rozmístění. Mohli bychom vytvořit základní implementaci zobrazitelných prvků, která implementuje nějaký obecný cyklus rozmístění, předpočítá tyto hodnoty a umožní prvky umístit i v souřadnicích plátna rodičovského prvku.

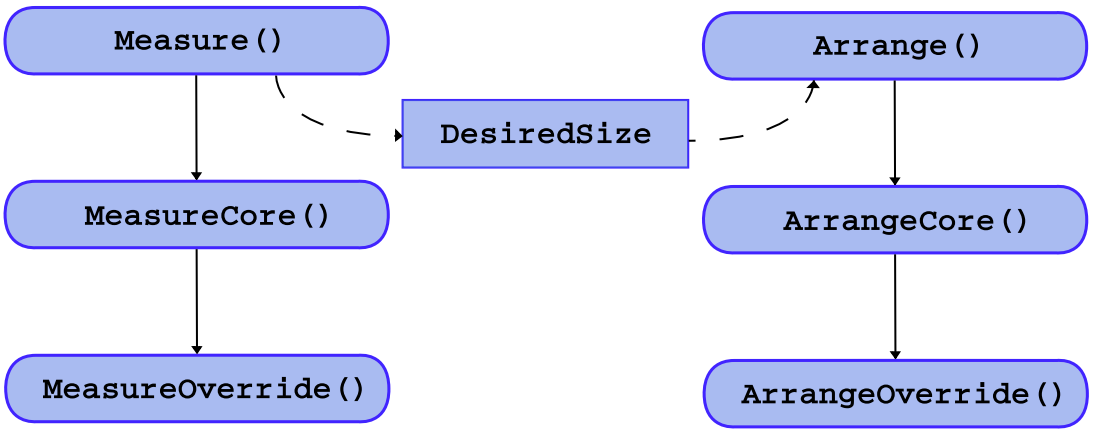
Jelikož jsme určili, že by bylo vhodné, aby cyklus rozmístění produkoval další informace nebo struktury používané knihovnou, rozhodli jsme se, že jejich správné vytvoření zajistíme v obecné implementaci, kterou dále budeme hledat. Vývojář bude v potomcích základní implementace pouze přidávat konkrétní vlastnosti prvku – vykreslení, určení požadované velikosti a rozmístění obsahu na základě přiřazení konečné velikosti. Vývojáři tak obdrží hotovou implementaci, která bude vytvářet části důležité pro funkcionalitu knihovny.

Obecná implementace cyklu rozmístění zobrazitelných prvků

Obecný cyklus rozmístění prvků musí být dostatečně konfigurovatelný pro různé scénáře. Podle **R24** například chceme změřit velikost prvku a na základě této velikosti mu přiřadit jeho konečnou velikost. Konečná velikost prvku bude určovat velikost plátna pro vykreslení, ale i hit-testovanou oblast nebo `bounding`

`rectangle`. S konečnou velikostí také budou pracovat strategie. Pokud by prvek tuto velikost přesáhl, nástroje by s prvkem nemohly správně pracovat. Protože vytváříme nějakou obecnou implementaci pro knihovnu, chceme, aby špatné nastavení cyklu rozmístění bylo nějak deterministicky vyřešeno. Jelikož s konečnou velikostí prvku pracuje mnoho částí knihovny, rozhodli jsme se, že pokud velikost prvku přesáhne přidělenou velikost, nebude rozmístěn. Problém s rozmístěním se tak projeví hned a ne při práci knihovny s prvkem. Nechceme vyhovazovat výjimky, protože vytváříme obecnou implementaci a některé aplikace pouze příliš velký prvek v některých konfiguracích komponenty nezobrazí. Vývojáři umožníme ale implementaci rozšířit a chybové stavy řešit i jiným způsobem.

Podobný problém, na který jsme narazili, řeší i GUI frameworky. Ty vytváří základní implementaci cyklu rozmístění pro jejich prvky uživatelského rozhraní. Frameworky pak pomocí hierarchie tříd cyklus rozmístění dále rozšiřují konfigurací jeho základní implementace. Toto řešení je vhodné i pro naší knihovnu. Nemuseli bychom tak vymýšlet nové řešení, ale mohli bychom nějaké převzít a upravit ho. Rozhodli jsme se proto použít nějakou existující implementaci a přizpůsobit ji navíc požadavkům naší knihovny. Implementace se mezi různými frameworky zásadně neliší a proto jsme se rozhodli vybrat k implementaci cyklus rozmístění frameworku WPF, který se skládá z *measure* a *arrange* průchodů znázorněných na obrázku 3.10:



Obrázek 3.10: Schéma cyklu rozmístění pro prvek GUI frameworku WPF

Prvkům je možné nastavit vlastnosti jako `Margin`, `MinWidth`, `MaxWidth`, `MinHeight`, `MaxHeight`. V measure průchodu začínající metodou `Measure()` nejdříve zjistíme velikost prvku. Velikost je v rámci hierarchie volání získána nejdříve z metody `MeasureOverride()`, kterou přetěžuje vývojář pro konkrétní implementaci prvku. Tato velikost je předána `MeasureCore()`, která v základní implementaci ověřuje správnost vrácené hodnoty například vůči nastavení `MaxWidth`, `MinWidth` a případně ji upravuje. Vývojář může metodu přetížit za podobným účelem. Ověřená hodnota je předána `Measure()`, která ji přiřadí do vlastnosti `DesiredSize`. V arrange průchodu je metodě `Arrange()` předána konečná velikost prvku. Metody `ArrangeOverride()` a `ArrangeCore()` se chovají podobným způsobem jako jejich ekvivalenty v measure průchodu. Cyklus rozmístění WPF vytváří mezi prvky explicitní vztahy, aby například invalidování potomka mohlo notifikovat o změně rodiče, který by mohl změnu velikosti potomka vyřešit v jemu přiřazené velikosti, případně by se notifikace šířila do jeho

rodiče. Jelikož k těmto situacím docházet při práci se zobrazitelnými prvky nebude, pro jednoduchost jsme se rozhodli explicitní vazby mezi prvky nevytvářet. Na potomky se volá pouze z rodičovských prvků `Arrange()` a `Measure()`.

Výpočet struktur používaných knihovnou v cyklu rozmístění

Určili jsme, že v základní implementaci cyklu rozmístění spočítáme hodnoty jako `placement matrix` nebo `bounding rectangle`. Navíc podle **R26** chceme, aby zobrazení prvků bylo možné upravovat rotací a škálováním a prvek přerozmístit. Všechny tyto úpravy je možné zahrnout do `placement matrix`. Při vykonávání těchto modifikací prvku chceme, aby byly aplikovány na všechny jeho potomky.

Jelikož se výpočet těchto hodnot bude aplikovat i pomocí metod, kterými strategie prvek transformují a je celkem náročné ho implementovat, chtěli bychom najít nějakou jednu metodu, v které by se výpočet provedl. Pokud se podíváme, jak transformace souvisí s cyklem rozmístění, strategie nejdříve musí zjistit velikost prvku, přiřadit mu konečnou velikost a pak ho případně transformovat rotací nebo škálováním. Metoda `Arrange()`, která by prvku přiřadila konečnou velikost, by mohla získat v rámci jejího volání `placement matrix`. Konfigurace `placement matrix` by pak mohly být následující:

1. Aplikováním `placement matrix` rodičovského prvku můžeme prvek `Arrange()` umístit v souřadnicích rodičovského prvku
2. Aplikováním `placement matrix` jako identity můžeme prvek `Arrange()` umístit v souřadnicích `ContentCanvas`
3. Aplikováním rotace nebo škálování na `placement matrix` můžeme vytvářet transformace rotace a škálování strategiemi

Měli bychom tedy nějakou interní metodu `ArrangeInternal()`, která by přijímala různé `placement matrix`. Vývojáři by volali na zobrazitelném prvku `Arrange()`, `Scale()`, `Rotate()` a `Reposition()`. Tyto metody upraví `placement matrix` a předají jí `ArrangeInternal()` metodě. Pokud se zavolá v `ArrangeOverride()` na potomka `Arrange()`, předá se mu upravená `placement matrix` a stejný výpočet probíhá znovu v potomku. Pomocí `placement matrix` se pak vypočítá `bounding rectangle` i ostatní hodnoty související s transformací. Ačkoliv cyklus rozmístění tak probíhá pro každou tuto operaci znovu, rozhodli jsme se tento přístup použít, protože neduplikuje kód a aplikování operace znovu nepředstavuje velké výkonnostní omezení, jelikož transformace strategiemi neprobíhají na větších stromech prvků a vyskytují se v cyklu rozmístění, ke kterému při běhu aplikace nedochází často.

3.2.2 Kreslení po vrstvách

Podle podkapitoly 2.2 chceme umožnit obsah vykreslovat ve vrstvách. Ve zmíněné podkapitole jsme uvedli příklad, kdy chceme do nové vrstvy v popředí vynést vybraný vlak z vrstvy vykreslující všechny vlaky. Zásadní problém, kterému se budeme snažit předejít, je vykreslení stejného obsahu ve více vrstvách opakovaně – vybraný vlak by se vykreslil v nové i původní vrstvě. Všechny vlaky se nachází v nějakém seznamu a v původní vrstvě se vykreslí následovně:

```
1: foreach(var train in Trains) {
2:     drawingCanvas.Draw(train);
3: }
```

Mohli bychom vybraný vlak umisťovaný do nové vrstvy z `Trains` odebrat. Tímto ale zasahujeme do modelu aplikace, při operaci, která svým významem model nemění – vybraný vlak se pouze přenáší do popředí. Další možností by bylo vytvořit nový seznam, který obsahuje prvky nacházejí se ve vrstvě nebo každému zobrazovanému vlaku přidat informaci, zda je ho možné ve vrstvě vykreslit. Zásadním problémem obou těchto řešení je fakt, že zasahují do implementace obsahu v původní vrstvě:

```
1: foreach(var train in Trains) {
2:     if (currentLayer.Contains(train)) {
3:         drawingCanvas.Draw(train);
4:     }
5: }
```

Problém můžeme řešit na úrovni knihovny tak, aby tato implementace nebyla změněna. Prvky jako vlak by implementovaly rozhraní `IVisual`, které bude nést informaci o přidělené vrstvě prvku. `DrawingCanvas` bude vědět, jakou vrstvu vykresluje. Pokud budeme chtít `IVisual` na plátno vykreslit, nejdříve se provede porovnání vrstev a pokud jsou stejné, prvek se vykreslí. Jediná nevýhoda je, že se vrstva musí přenastavit všem prvkům stromu reprezentující vybraný vlak, aby se po porovnání vykreslily.

Jelikož jsme v požadavku **R23** určili, že obsah nákresného jízdního řádu bude systematicky popsatelný různými prvky, rozhodli jsme se, že prvky budou implementovat rozhraní jako `IVisual` a `DrawingCanvas` ověří jejich vykreslení. `IVisual` bude implementováno v nějaké základní třídě všech vykreslitelných prvků, která je předkem `ViewElement`. V knihovně tak vytváříme mechanismus, který vývojář může používat a nevytvářet vlastní řešení. Zmíněnou nevýhodu v další části odstraníme a ještě řešení vylepšíme.

Konfigurace vrstev

V dosavadní představě si v rozhraní `IVisual` uchováváme pouze vrstvu, v které se prvek nachází. Když vývojář změní prvku vrstvu, musí si pamatovat, v jaké předchozí vrstvě se prvek nacházel. Proces přemisťování prvků mezi vrstvami probíhá tak, že vývojář vrstvu změní a pak ji zase nahradí původní. V příkladu prvků vlaku vyneseného do popředí se jeho prvky nejdříve při inicializaci obsahu přiřadí do vrstvy všech vlaků a při výběru vlaku se přenesou do vrstvy popředí. Pak se případně vrací do vrstvy původní. Proces přiřazování a odebírání vrstev tak odpovídá práci se zásobníkem. Aby si vývojář nemusel pamatovat, v jaké původní vrstvě se prvek nacházel, tento zásobník každému z prvků přidáme.

Nevýhodou současného řešení je potřeba nastavovat vrstvu všem prvkům, které bychom chtěli vykreslit. Tuto nevýhodu bychom mohli na základě následujícího pozorování odstranit. Předpokládejme, že v počátečním stavu aplikace problém, kdy by se jeden prvek vykreslil ve více vrstvách zároveň, nenastává. Chtěli bychom nalézt řešení, jak nastavování vrstev v počátečním stavu přeskočit. Vytvoříme vrstvu, kterou nazveme jako `default layer`. Pokud ji má prvek nastavenou, kontrola vrstvy se před vykreslením nevykoná. Jelikož pracujeme se zásobníkem, bude tato vrstva umístěna na spodku zásobníku, jako počáteční stav. Pokud použijeme příklad s vybraným vlakem, při jeho přesunu do nové vrstvy bychom už vrstvu změnili všem jeho prvkům.

3.2.3 Implementace hit-testingu

V části 3.2.1 jsme určili, jak probíhá hit-testování na každém zobrazitelném prvku. Nyní si vysvětlíme, jak podle požadavku **R28** implementujeme průchod zobrazitelných prvků obsahu nákresného jízdního řádu a získáme z něj ty, které v hit-testování uspěly. Tyto prvky bude chtít vývojář získat podle pořadí jejich vykreslení, s tím, že většinou bude pracovat s prvkem, který je vykreslován v popředí a překrývá tak ostatní prvky, které v hit-testu také uspěly.

Průchod prvků by mohl implementovat sám vývojář. Postupně by procházel všechny prvky, tvořící dohromady stromovou strukturu. V některých prvcích stromu by ale mohl narazit na problém. V příkladu uvedeném v 3.2.2 máme zobrazitelný prvek vykreslující všechny vlaky v seznamu `Trains`. Vývojář by tento seznam prošel a každý prvek vlaku ověřil v hit-testu. Podle 3.2.2 se ale vlaky můžou nacházet v různých vrstvách, čili by sekvenční průchod seznamu případně neposkytl správné pořadí vykreslení a při průchodu prvků ve více vrstvách by se konkrétní prvek mohl vyskytnout v úspěšně otestovaných prvcích vícekrát.

Průchod stromu prvků jsme se proto rozhodli implementovat v knihovně. Otestování celého obsahu odpovídá průchodu stromem, kdy otestujeme obsah rozdělený do vrstev, vykreslovaných podle jejich pořadí. Každá vrstva bude obsahovat prvek, který bude kořenem jejího obsahu. K rozhraní `IVisual` z 3.2.2 bychom přidali metodu `HitTest()`, hit-testující vizuální prvek vůči nějakému bodu a metodou poskytující potomky k hit-testování `ProvideChildrenToHitTest()`. Z této metody se pak vyberou jen ty prvky, které se nachází ve stejné vrstvě jako rodičovský prvek. Musíme ještě určit pořadí, v kterém bude knihovna poskytovat úspěšně otestované prvky. Metodu `ProvideChildrenToHitTest()` bychom chtěli nechat vývojáře implementovat tak, aby prvky poskytovala v pořadí vykreslení:

```
1:  public void OnDraw(DrawingCanvas drawingCanvas) {
2:      foreach(var train in Trains) {
3:          drawingCanvas.Draw(train);
4:      }
5:  }
6:
7:  public IEnumerable<IVisual> ProvideChildrenToHitTest() {
8:      foreach(var train in Trains) {
9:          yield return;
10:     }
11: }
```

To ale znamená, že první poskytnutý prvek je ten, který se vykreslí na spodku. Vývojář by mohl projít seznam v opačném pořadí, to ale představuje komplikaci v implementaci rozhraní. Úspěšně otestované prvky bychom při průchodu stromem mohli uchovávat v nějakém seznamu, odpovídající pořadí vykreslení. Jelikož úspěšně testovaných prvků je malé množství, rozhodli jsme se, že vývojář v rozhraní poskytne prvky podle pořadí vykreslení. Průchod tedy bude implementován tak, že se postupně navštíví všechny vrstvy s jejich obsahem podle pořadí vykreslení. Pokud chceme prvek, který překrývá ostatní, najdeme ho na konci tohoto seznamu. Proto bychom si místo seznamu mohli pamatovat jen jeden prvek, který se vykreslí jako poslední. Bylo by možné místo pevného seznamu použít jinou implementaci průchodu, kterou si nyní představíme.

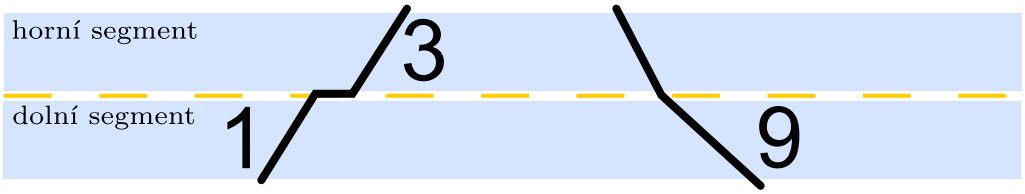
Průchod stromu prvků pro hit-testování řeší i uživatelská rozhraní. Například GUI framework WPF implementuje obecný průchod stromem, kdy se při navštívení prvku zavolá vývojářem dodaná metoda delegáta `HitTestFilterCallback`, v které může vývojář podle prvku rozhodnout, jak má dále průchod stromem pokračovat. Průchodu stromem se dále předává metoda delegáta `HitTestResultCallback`, která se volá, pokud byl prvek vůči hit-testingu úspěšný. Také určí, jak při průchodu stromem pokračovat. V základním nastavení je tato implementace lehká na konfiguraci – v `HitTestResultCallback` by se úspěšně testované prvky přidávaly do seznamu, který jsme chtěli v původní implementaci použít. Vývojář může průchod stromem urychlit, jelikož dodání implementace delegáta `HitTestFilterCallback` umožňuje přeskočit všechny prvky v nějakém podstromu, u kterých víme, že se s nimi pracovat nebude. Jelikož implementace odpovídá našim potřebám a můžeme převzít funkční kód doplněný o potencionálně využitelná rozšíření, rozhodli jsme se ji použít.

3.2.4 Implementace strategií

Pro splnění požadavku **R23** musíme najít nástroje, které umožní systematicky popsat proces implementace strategií. Rozhodli jsme se, že tento proces rozdělíme do částí, kde každá část má jednu zodpovědnost[1]. Tyto částí si nyní popíšeme a zdůvodníme jejich používání.

Segmenty strategií

Strategiím je potřeba vyhradit nějaký vodorovný pruh, do kterého můžou umístit zobrazitelné prvky. Tyto pruhy označíme jako *segmenty*. Umístění i výšku segmentu vývojář určí v cyklu rozmístění před samotným aplikováním strategie, jelikož jsou segmenty součástí zobrazitelných prvků popisujících nákresný jízdní řád, jako třeba místa pro umisťování kót kolem horizontálních čar dopravních bodů. Nechceme, aby vývojář vytvářel ve svém modelu vlastní struktury, které by segmenty určovaly. Nemůžeme pak jejich konkrétní implementaci zapojit do nástrojů pracujících se strategiemi a navíc představují další část pro vývojáře, kterou musí implementovat. Ke strategii pro rozmisťování kót do ostrých úhlů bychom potřebovali dva segmenty kolem dopravních bodů, jako na obrázku 3.11. Segmenty v příkladu je možné rozlišit – podle dopravního bodu a dolního nebo horního umístění. Proto pro segment zavedeme identifikaci označovanou jako `SegmentType`, která umožní segmenty kategorizovat. Segmenty patřící pod jeden typ bude spravovat registr `SegmentRegistry`, který bude přijímat registrace segmentů a poskytovat segmenty podle `SegmentType`, čehož můžou využít další nástroje pracující se strategiemi.



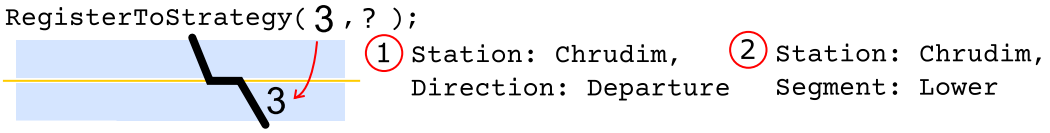
Obrázek 3.11: Horní a dolní segment dopravního bodu pro umístění kót

Zařazení zobrazitelných prvků do strategií

Když chce vývojář přidat do strategie zobrazitelný prvek, musí se zjistit, do jakého `SegmentType` prvek patří. Pokud by vývojář přidával prvek přímo do segmentu, musel by sám určit `SegmentType`. To ale může představovat problém. Uvažme případ, kdy chceme přidat odjezdovou kótu z nějakého dopravního bodu, jako na obrázku 3.12. `SegmentType` je závislý na směru jízdy vlaku, čili vývojář musí sám složitě podle výpočtu zjistit, kam prvek podle směru patří. Vývojář by mohl při přidání prvku uvést jiný typ, přirozenější pro specifikaci umístění, označovaný jako `PlacementType`, který se na `SegmentType` převede. Vývojář by při přidání prvku uvedl informaci v `PlacementType` ① se směrem (příjezd nebo odjezd). Bude pak existovat rozhraní `ITypeConverter`, které převede tuto informaci

[1]Single responsibility principle [12]

na `SegmentType` ② (lower, upper). Rozhodli jsme se tyto dva typy s rozhraním pro převod zavést.



Obrázek 3.12: Přidání kóty odjezdu do strategie

Spravování zobrazitelných prvků v strategiích

Vývojář bude přidávat prvky do strategie pomocí nástroje `StrategyManager` uvedením `PlacementType`. `StrategyManager` zajišťuje přiřazení prvků do správných segmentů a strategií. Vývojář tak nemusí přímo pracovat s jejich konkrétními implementacemi. Existují situace, kdy je nutné invalidovat všechny přidané zobrazitelné prvky – například, když se odstraňuje vlak nebo se jeho trasa zcela mění. Pro každý logický celek prvků, který má omezenou dobu existence (*scope*), bude vytvořen nový `StrategyManager`, zajišťující odstranění prvků ze struktur strategií nebo segmentů – zabraňujeme tak *memory leakům*, kdy by tyto struktury držely jedinou referenci na prvek jinak již neexistující části obsahu, jelikož je vývojář zapomněl odebrat.

Dockery zobrazitelných prvků

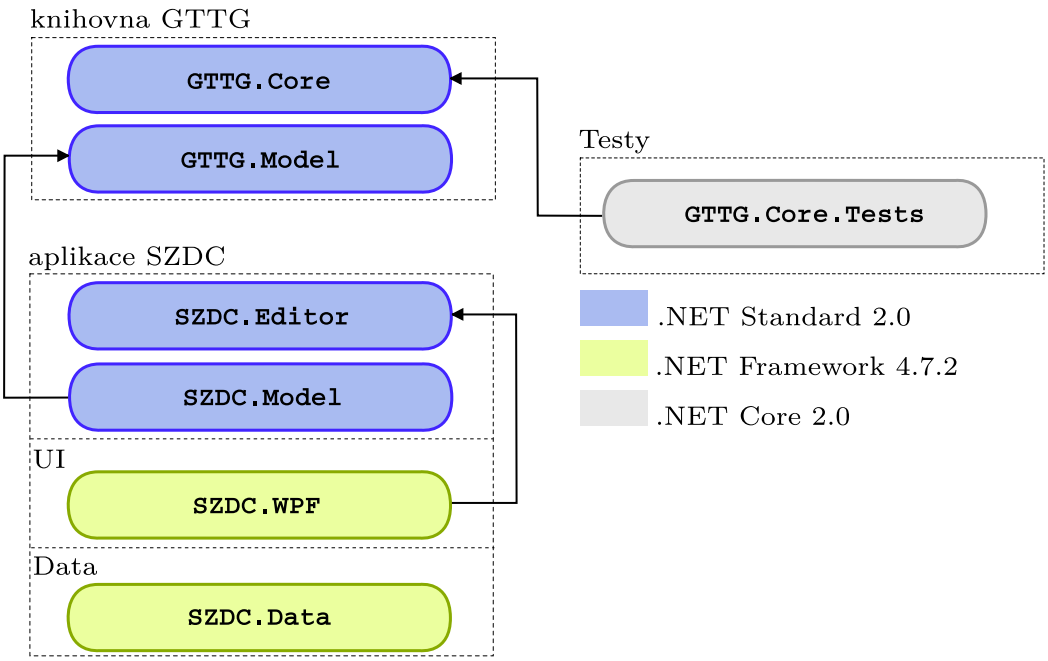
Docker implementuje vhodné rozmístění zobrazitelných prvků. V cyklu rozmístění obdrží od `StrategyManager` všechny přidané prvky a umístí je do oblastí segmentů. Implementace dockeru může vypadat tak, že nejdříve prvek změří, přidělí mu požadovanou velikost přes `Arrange()` a pak na něj podle `SegmentType` a dalších parametrů aplikuje správnou metodu, která ho podle **R26** transformuje a umístí. Docker je takto možné v aplikaci vyměnit a reimplementovat, jelikož přímo není propojený s nástroji pro strategii.

4. Vývojová dokumentace knihovny GTTG

Ve Visual Studio 2017 *solution* se nachází celý obsah práce, kde knihovna GTTG je jednou z jeho částí. Představíme si, jak je celý solution strukturovaný a jaké jsou jeho další části. Solution jsme rozdělili do tří složek:

`build`	Společná nastavení verzí balíčků používána v projektech. Na soubory v `build` typu **.props* obsahující verze balíčků se odkazují projekty v *.csproj* souborech.
`src`	Zdrojový kód v solution
`test`	Unit testy pro zdrojový kód v `src`

Solution je rozděleno do sedmi projektů uvedených na obrázku 4.1, představující dvě části – knihovnu GTTG, jejímuž popisu jsme se doposud věnovali, a aplikaci SZDC, implementující práci s grafikonem vlakovy využitím knihovny podle cíle **G2**. Popisu aplikace se budeme věnovat v kapitole 6.



Obrázek 4.1: Struktura projektů v solution

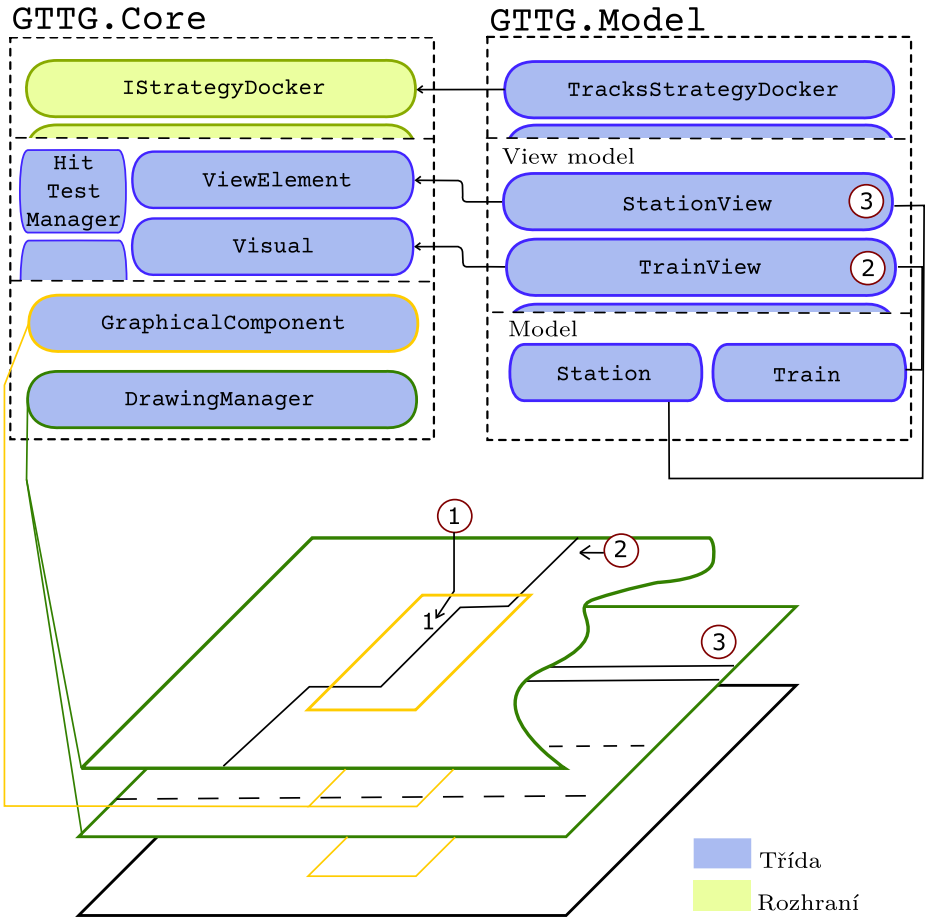
Podle cíle **G2d**) jsme pro co největší přenositelnost knihovny GTTG všechny její projekty implementovali vůči rozhraní .NET Standard 2.0. Jelikož podle cíle **G2c**) chceme logiku aplikace SZDC zapojit do více uživatelských rozhraní, je také její editor `SZDC.Editor` a model dat `SZDC.Model` implementován vůči rozhraní .NET Standard. Jako GUI framework aplikace `SZDC` jsme zvolili WPF, který je v naší aplikaci spustitelný na běhovém prostředí .NET Framework 4.7.2. Nově je možné i WPF spustit na běhovém prostředí .NET Core 3. Výběru GUI frameworku a běhového prostředí pro zvolené WPF se věnujeme v kapitole 6.1.

Struktura knihovny GTTG

Knihovna je rozdělena do dvou spolu souvisejících částí, jejichž schéma rozdělení se nachází na obrázku 4.2.

`GTTG.Core` obsahuje nástroje umožňující uživatelskou interakci s knihovnou a její integraci do aplikací. Základem knihovny je *engine* `GraphicalComponent` a nástroj `DrawingManager` rozdělující kreslení do vrstev. Dále jsou v projektu implementovány prvky umožňující systematicky popsat implementaci nákresných jízdních řádů, jako třeba `ViewElement` a `Visual`. Projekt obsahuje nástroje pro práci s těmito prvky, jako třeba `HitTestManager` a nabízí rozhraní jako `IStrategyDocker`, jehož implementací vytváří vývojář strategie umisťující prvky jako kóty do nákresného jízdního řádu.

`GTTG.Model` používá `GTTG.Core` k vytvoření základního modelu nákresných jízdních řádů. Třídy jako `TrainView` jsou potomky základní třídy zobrazitelných prvků `ViewElement` a představují vizualizaci datového modelu tříd jako `Train` nebo `Station`. Projekt tak rozděluje třídy na *view model* a *model*. Dále jsou v projektu implementována rozhraní knihovny `GTTG.Core` pro práci se zobrazitelnými prvky. Například `TracksStrategyDocker` jako implementace rozhraní `IStrategyDocker` z `GTTG.Core` rozmisťuje kóty ① do ostrých úhlů průběhu jízdy vlaku `TrainView`.



Obrázek 4.2: Části knihovny GTTG

4.1 GTTG.Core

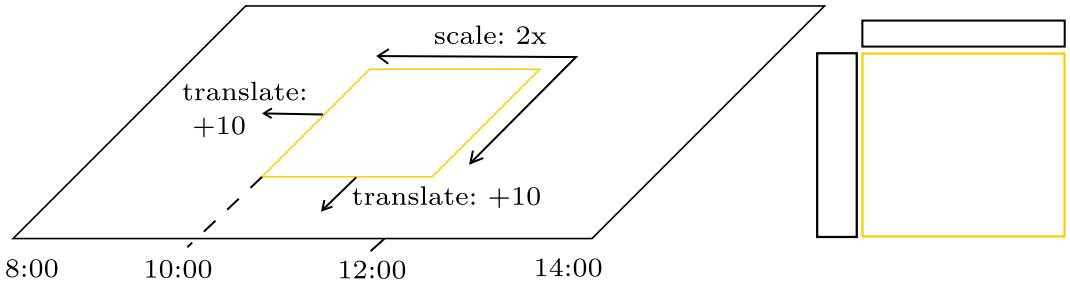
`GTTG.Core` *assembly* je rozdělena do několika oblastí jmenných prostorů:

`Base`	Hierarchie tříd implementující zobrazitelné prvky
`Component`	Třídy pro práci s grafickou komponentou
`Drawing`	Nástroje pro kreslení
`Extensions`	Knihovní a SkiaSharp *extension* metody
`HitTest`	Nástroje pro hit-testování obsahu
`Strategies`	Nástroje pro práci se strategiemi
`Time`	Třídy pro reprezentaci časových intervalů
`Utils`	Matematické funkce pro implementaci strategií

V následujících částech si tyto jmenné prostory popíšeme.

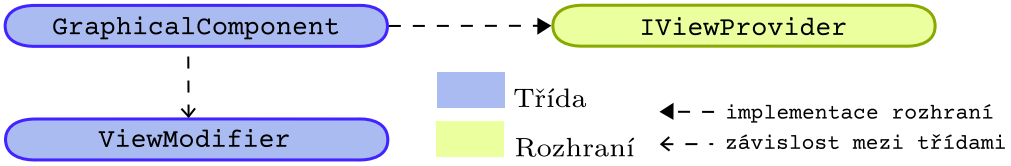
4.1.1 Grafická komponenta

Základem knihovny GTTG je třída `GraphicalComponent` představující *engine*, který umožňuje vytvářet pohled na nákresný jízdní řád podle interakcí uživatele s aplikací nebo jiných nastavení, jako na obrázku 4.3. Poskytuje modifikace jako translace a škálování pohledu v rámci ohraničení nákresného jízdního řádu. Na základě nastavení pohledu pak vzniká stav komponenty, s kterým pracují ostatní části knihovny. Například vykreslení nákresného jízdního řádu je nastaveno maticí `ContentMatrix`, kterou podle 3.1.5 třída modifikacemi upravuje.



Obrázek 4.3: *Engine* `GraphicalComponent`

Obrázek 4.4 obsahuje schéma tříd jmenného prostoru `GTTG.Core.Component` související s implementací grafické komponenty, jehož součásti si nyní blíže představíme:



Obrázek 4.4: Schéma tříd a rozhraní implementujících grafickou komponentu

Stav komponenty R16 je představován vlastnostmi, které jsou mimo třídu pouze pro čtení. Abychom mohli stav v komponentě poskytnou dalším částem knihovny nebo implementacím bez přístupu k modifikacím komponenty, `GraphicalComponent` implementuje rozhraní `IViewProvider`. Rozhraní poskytuje i převody mezi body komponenty a časovými údaji, které jsou využívány nástroji knihovny (průběh jízdy vlaku se vykreslí převodem jeho časových údajů do souřadnic) nebo naopak aplikační logikou pracující s knihovnou, která chce souřadnice kurzoru myši v komponentě převést na časový údaj zobrazený v aplikaci.

Komponenta pracuje s třídou `ViewModifier` určenou pro vykonávání složitějších matematických modifikací stavu komponenty. Implementace modifikací v `GraphicalComponent` upraví parametry a předá je odpovídající modifikaci ve `ViewModifier`. V případě úspěšné modifikace je pak upraven celý stav komponenty. Každá modifikace vrací podle 3.1.4 výčtový typ nesoucí informace o jejím výsledku.

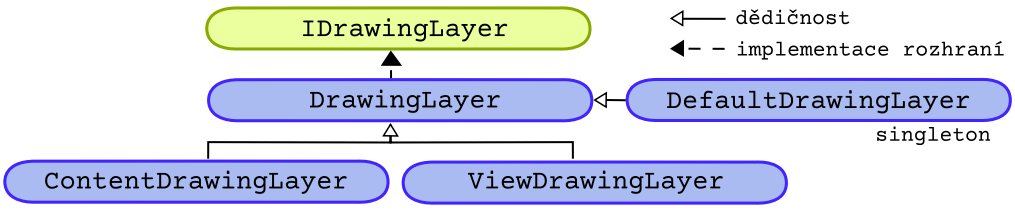
Práce s časovými údaji

Knihovna obsahuje strukturu `DateTimeInterval` představující časové intervaly `DateTime` hodnot. `GraphicalComponent` tuto strukturu používá k reprezentaci časových intervalů v komponentě popsaných v 2.1. Třídou `DateTimeContext`, která tyto intervaly slučuje a zajišťuje jejich validitu, se můžou komponentě v rámci jejího stavu přenastavit časové intervaly. Pohled na nákresný jízdní řád je pak možné modifikovat i pomocí časových intervalů jako v situacích zmíněných v 2.1.

4.1.2 Vykreslování

Obsah knihovny je vykreslován ve vrstvách, které jsou reprezentovány rozhraním `IDrawingLayer` s metodou `Draw()` k vykreslení vrstvy. Podle schématu 4.5 abstraktní třída `DrawingLayer` implementuje toto rozhraní a je předkem tří abstraktních typů vrstev:

`ContentDrawingLayer`	Typ vrstvy, které je k vykreslení poskytnuto plátno pro celý obsah nákresného jízdního řádu podle R17.
`ViewDrawingLayer`	Typ vrstvy, které je k vykreslení poskytnuto plátno komponenty podle R18.
`DefaultDrawingLayer`	*Singleton* reprezentující vrstvu, která je při kontrole vrstvy v 3.2.2 vždy validní.



Obrázek 4.5: Schéma tříd a rozhraní představující vrstvy

Správa kreslících vrstev

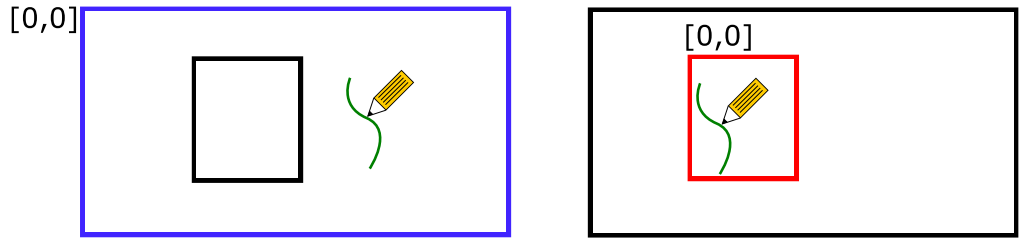
Třída `DrawingManager` vykresluje vrstvy podle pořadí definovaného vývojářem. Instance vrstev jsou přidávány s určením jejich pořadí pomocí metod `AddOnCurrentTop()`, `AddOnCurrentBottom()`, `Insert()`. Instanci vrstvy je možné odstranit pomocí `RemoveDrawingLayer()`. Jelikož existují vrstvy, které se neodstraňují a jejich pořadí je během běhu aplikace neměnné, konstruktoru `DrawingManager` se předává implementace rozhraní `IRegisteredLayersOrder`, které obsahuje pořadí typů (z `typeof()`) vrstev, které označíme jako *registrované*. Takovou vrstvou může být vrstva svislých čar reprezentující časové údaje nebo vrstva s dopravními body. Metoda `ReplaceRegisteredDrawingLayer()` umístí instanci registrované vrstvy podle jejího typu v pořadí určeným rozhraním.

Kreslící plátno

Knihovna na plátno `SKCanvas`, které ji poskytne vývojář, vytváří různé pohledy. Tyto pohledy budeme reprezentovat plátnem `DrawingCanvas`. Vhodným nastavením `transformation matrix` na `SKCanvas` může `DrawingCanvas` představovat plátno pro vykreslování zobrazitelných prvků z 3.2.1 i celého obsahu plátna `ContentCanvas` z 3.1.5. Plátno nabízí kreslící příkazy `SKCanvas` a příkazy pro vykreslení prvků knihovny. Plátno obsahuje vlastnosti o své velikosti `Width`, `Height`. Vlastnost `DrawingLayer` udává, v jaké vrstvě bylo plátno vytvořeno.

Proces vykreslování vrstev

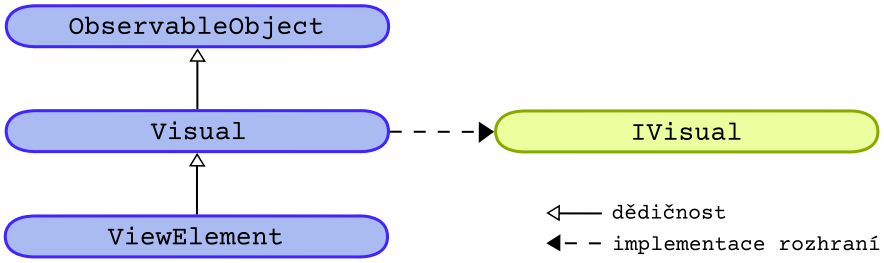
Entry pointem pro vykreslení obsahu knihovny je metoda `Draw(SKSurface surface)` ve třídě `DrawingManager`. Vrstvy se vykreslí v sestaveném pořadí tak, že se pro každou vrstvu zavolá metoda `CreateCanvas()` factory rozhraní `ICanvasFactory`, která poskytnutím `IDrawingLayer` a `SKCanvas` vytvoří pro vrstvu konfigurovaný `DrawingCanvas`. Vrstvám `ContentDrawingLayer` a `ViewDrawingLayer` poskytuje knihovna implementaci rozhraní `CanvasFactory`.



Obrázek 4.6: Modře ohraničený `ContentDrawingCanvas` a červeně ohraničený `ViewDrawingCanvas`

4.1.3 Zobrazitelné prvky

Jmenný prostor `GTTG.Core.Base` obsahuje hierarchii tříd uvedenou na obrázku 4.7, z níž vzniká třída `ViewElement` představující zobrazitelné prvky určené k systematickému popisu nákresného jízdního řádu podle 2.2.



Obrázek 4.7: Hierarchie tříd tvořící zobrazitelné prvky

ObservableObject

Na začátku hierarchie tříd se nachází abstraktní třída `ObservableObject` implementující rozhraní `INotifyPropertyChanged`, které slouží k notifikacím o změnách vlastností v třídě pomocí eventu `PropertyChanged`. Tuto základní třídu mohou využít vývojáři k vytvoření *data bindingu* z uživatelských rozhraní na aplikační logiku a její model. Pokud bychom měli například model vlaku `Train` obsahující plán jízdy `Schedule`, aplikace by `Schedule` navíc zobrazovala v jiné komponentě uživatelského rozhraní jako textová data. V případě, kdy by se `Schedule` změnila, notifikaci o změně obdrží jiná komponenta v aplikaci a zároveň i `TrainView`, které může upravit svou vizualizaci dat. Z této třídy dědí `GraphicalComponent`, která tak umožňuje vývojářům získávat notifikace o změně `DateTimeContext`.

Třída implementuje mechanismus pro bezpečné a lehké používání notifikací na *setterech* vlastností, který jsme převzali z projektu `Core2D` [13] z třídy `ObservableObject`. Následující fragment kódu ukazuje přiřazení nové hodnoty do vlastnosti s notifikováním pomocí `Update()` metody. Update by se jinak musel provádět pomocí *invoke* eventu `PropertyChanged` poskytnutím jména vlastnosti.

```
1: public class GraphicalComponent {
2:
3:     public DateTimeContext DateTimeContext  {
4:         get => _dateTimeContext;
5:         protected set => Update(ref _dateTimeContext, value);
6:     }
7:
```

Visual

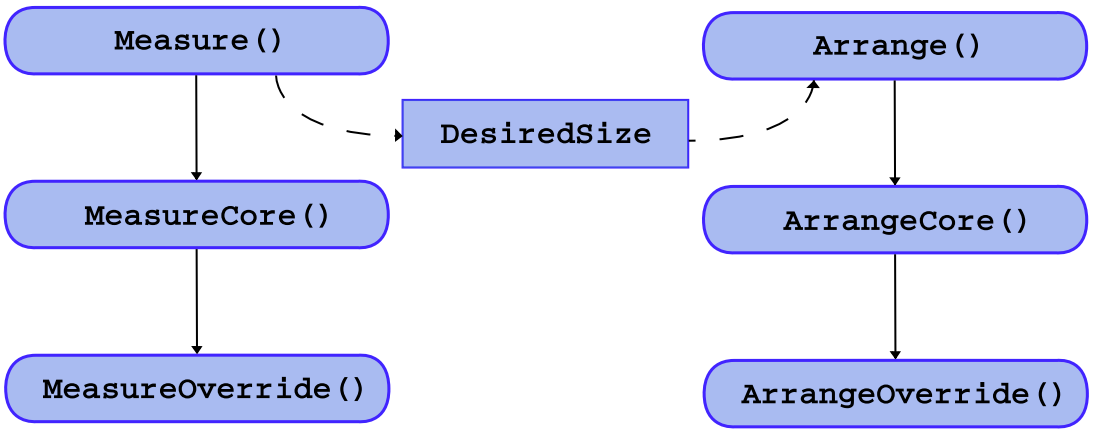
Abstraktní třída `Visual` představuje prvky, které popisují nákresný jízdní řád a je možné je vykreslit na `DrawingCanvas`, ale narozdíl od zobrazitelných prvků se nerozmisťují. Příkladem může být prvek `TrafficView`, který sdružuje a vykresluje na plátno `ContentDrawingCanvas` všechny vlaky `TrainView`. `Visual` obsahuje podle 3.2.2 zásobník vrstev `IDrawingLayer` s neměnným spodkem obsahující singleton `DefaultDrawingLayer`. Vlastnost `CurrentDrawingLayer` odpovídá vrchu zásobníku. Vykreslení probíhá tak, že se virtuální metodě `Draw()` předá `DrawingCanvas` a nejdříve se ověří možnost prvek vykreslit pomocí `IsInDrawingLayer()` porovnáním `CurrentDrawingLayer` a `DrawingLayer` plátna. Po ověření se zavolá virtuální `protected OnDraw()` k vykreslení implementace prvku.

Vyvojáři implementují `protected ProvideVisuals()`, která dodává prvky v pořadí vykreslení. Metoda je volána `ProvideVisualsInSameLayer()` poskytující pouze z těchto prvků ty, které se nachází ve stejné vrstvě jako prvek, na kterém se metoda volá. Tyto metody se používají při hit-testování a nastavování vrstev potomkům.

Pro případy, kdy existující prvek (například model jako potomek jiné třídy) nemůže již od `Visual` dědit a přesto chce kreslit na plátno, použije rozhraní `IVisual`, s kterým pak pracuje i kód knihovny jako s reprezentací `Visual`. Rozhraní se používá i pro *mockování* v unit testech.

ViewElement

Abstraktní třída `ViewElement` představuje základní implementaci zobrazitelných prvků z podkapitoly 3.2.1, které je možné umístit a určit jim velikost. Pro implementaci cyklu rozmístění `ViewElement` jsme použili cyklus rozmístění prvků uživatelského rozhraní GUI frameworku WPF. Implementace jeho metod na schématu 4.8 odpovídá zdrojovému kódu metod z WPF, který byl ale upraven, aby mohl vytvářet během cyklu rozmístění i struktury, s kterými pracuje knihovna.

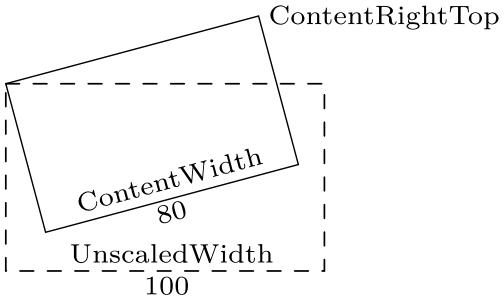


Obrázek 4.8: Schéma cyklu rozmístění pro prvek GUI frameworku WPF

Metody `Arrange()` a `Measure()` zajišťují správné používání virtuálních metod typu `Core`, které volají. Ty slouží k určení obecné implementace prvků, která se používá v rámci nějaké aplikace. V základní implementaci upravují vrácené hodnoty z `Override` metod. Ty jsou určeny ke změření a rozmístění konkrétního prvku. V rámci volání metody `Arrange()` se podle 3.2.1 vytváří struktury, s kterými knihovna pracuje:

`BoundingRectangle`	Obdélník k určení, zda se prvek nachází v komponentě podle 3.1.5.
`PlacementMatrix`	Matice `placement matrix` pro vytvoření `transform matrix` plátna `DrawingCanvas` z 3.2.1.
`Clip`	Oblast ořezu vykreslení z 3.1.1 pokrývající prvek, která se může aplikovat pomocí vlastnosti `HasClipEnabled`.

Přetíženou metodou `Arrange()` se prvek umístí na plátno `ContentCanvas` – přímo v jeho souřadnicích nebo v souřadnicích již umístěného prvku, který se uvede jako parametr `Arrange()` metody. To je užitečné v případě, kdy v `ArrangeOverride()` rodičovského prvku získáme konečnou velikost rodiče a musíme v rámci této velikosti rozmístit jeho potomky `Arrange()` metodou. Zároveň `ViewElement` implementuje metody `Reposition()`, `Scale()` a `Rotate()`, které jsou používány strategiemi. Transformované zobrazení prvku je popsáno vlastnostmi, které jsou zobrazeny na obrázku 4.9. Vlastnosti s prefixem `Content` odpovídají transformované velikosti a umístění. Vlastnost umístění jako `ContentRightTop` odpovídá umístění netransformovaného horního pravého vrcholu po aplikování transformace.



Obrázek 4.9: Popis vlastností `ViewElement`

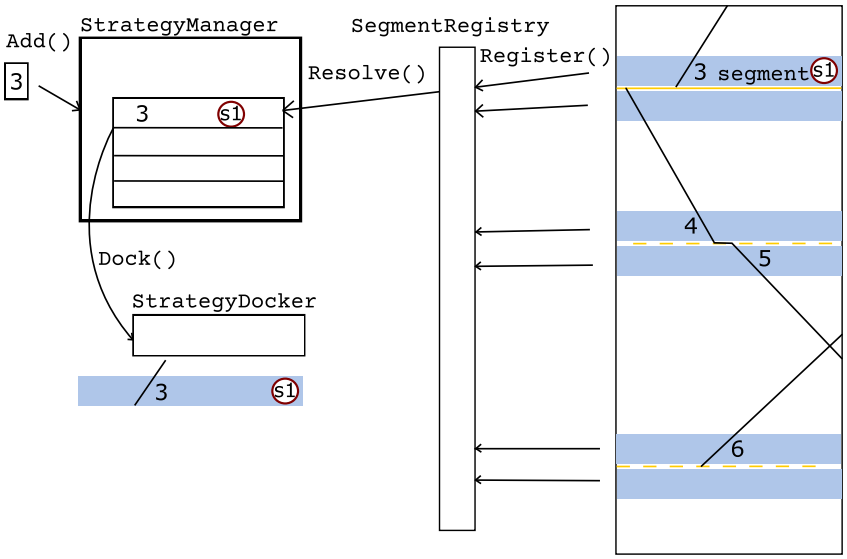
4.1.4 Průchod stromu prvků pro hit-testování

Průchod stromu prvků `Visual` hit-testováním popsaný v 3.2.3 je implementován třídou `HitTestManager` v přetížených metodách `HitTest()`. Průchod stromem je implementován podle implementace GUI frameworku WPF – vývojář poskytuje metody delegátů `HitTestResultCallback` a `HitTestFilterCallback` pro řízení průchodu stromem a získání úspěšně otestovaných prvků, více popsané v 3.2.3. Vývojáři jsou dostupná přetížení metody `HitTest()`, která se svým chováním liší. Prvky jsou pokaždé poskytovány v pořadí jejich vykreslení. První přetížení s parametry dvou zmíněných delegátů prochází celý obsah nákresného jízdního řádu po vrstvách, kdy z každého prvku `Visual` navštíví potomky pouze ve

stejné vrstvě pomocí `ProvideVisualsInSameLayer()`. `HitTestManager` obdrží pořadí vrstev v konstruktoru z `DrawingManager`. Druhému přetížení se dodává navíc prvek `Visual` a při průchodu se získávají potomci z `ProvideVisuals()`, poskytující prvky i v jiných vrstvách. Seznam prvků `Visual`, které uspěly v hit-testu, je pak možné seřadit pomocí *extension* metody `OrderByLayers()` podle pořadí vrstev. Druhé přetížení je určeno pro případy, kdy vývojář testuje obsah umístěný původně v jedné vrstvě a je možné, že jeho část je nyní součástí jiné vrstvy, například při testování seznamu vlaků, kdy vlaky můžou být rozděleny do více vrstev podle jejich zařazení a nechceme procházet celý vykreslovaný obsah. Předpokládá se, že při průchodu podstromu poskytnutého prvku se navštíví všechny prvky pouze jednou.

4.1.5 Nástroje pro strategie

V této části si představíme implementaci nástrojů pro práci se strategiemi, které jsme si popsali v 2.2. Na obrázku 4.10 se nachází závislosti mezi těmito nástroji. Instance třídy `Segment` představují pruhy, jejichž umístěním a velikostí vývojář určuje místa, do kterých mají strategie rozmisťovat prvky. Přidání prvků do strategie probíhá ve třídě `StrategyManager`, která podle informací dodaných při přidávání prvku daný prvek zařadí do vhodného segmentu. Během cyklu rozmístění pak `IStrategyDocker` představující implementaci strategie obdrží všechny přidané prvky a umístí je do segmentů podle pravidel strategie.



Obrázek 4.10: Schéma nástrojů pro práci se strategiemi

Segmenty

Třída `Segment` představuje podle 3.2.4 ohraničený horizontální pruh, kterému se určuje výška. Podle výšky segmentu se do něj strategiemi rozmisťují prvky. Pokud je segment pro prvek příliš krátký, může strategie podle 2.2 prvek zmenšit

škálováním. Výška segmentu se určuje v rámci `ArrangeOverride()` nějakého zobrazitelného prvku. Například prvek `StrategyStationView` obsahuje segmenty, které se umisťují kolem horizontálních čar představující koleje k rozmístění údajů do ostrých úhlů. Segmentům se přiřadí ohraničení a velikost pomocí přetížených metod `SetBounds()`. Pokud vývojář chce určit výšku segmentu na základě prvků, které jsou do něj přidány, použije potomka `MeasurableSegment`, který výšku měří hledáním maxima voláním eventu, do nějž se přidají měření výšek jednotlivých prvků v segmentu.

Instance segmentů je možné registrovat v třídě `SegmentRegistry`, do které se segmenty přidávají pod typovým argumentem `TSegmentType`, zvoleným při vytváření `SegmentRegistry`. Pomocí `TSegmentType`, který může být například strukturou obsahující přesné informace o umístění segmentu, pak můžou ostatní části knihovny získávat ze `SegmentRegistry` konkrétní registrované instance. Dále `SegmentRegistry` umožňuje pomocí *fluent syntax* registrovat jednu instanci segmentu pod více `TSegmentType` argumenty, jako na následujícím fragmentu kódu:

```
1: var segmentType1 = /*...*/;
2: var segmentType2 = /*...*/;
3: var segment = /*...*/;
4: segmentRegistry.Register(segment).As(segmentType1).As(segmentType2);
```

Rozhraní `ISegment` slouží pro přístup ke čtení vlastností segmentu mimo prvek, který ho rozmisťuje.

StrategyManager

Vývojář přidává prvky do segmentů k aplikování strategií pomocí třídy `StrategyManager`, která je určena 4 typovými argumenty:

`TPlacementType`	Typ, kterým vývojář určí, kam se prvek umístí
`TElement`	Typ přidávaného prvku, který je potomkem `IVisual`
`TSegmentType`	Typ používaný ke kategorizaci segmentů
`TSegment`	Typ segmentu

S `TPlacementType` pracuje `StrategyManager` jako s klíčem, pod nímž nemůže být přidáno více prvků. Pokud bychom chtěli podle 3.2.4 přidat odjezdovou kótu do segmentu koleje, `PlacementType` bude obsahovat výčtový typ s hodnotami `Departure` / `Arrival`, udávající umístění v kontextu průběhu jízdy vlaku. `StrategyManager` převede pomocí implementace převodů `ITypeConverter` (dodané v konstruktoru) tento typ na `TSegmentType`, který je strukturou s výčtovým typem s identifikátorem stanice a hodnotami `Upper` / `Lower`, udávající umístění segmentu nad nebo pod horizontální čarou stanice. Celý průběh přidání prvku je pak následující:

1. Klíč `TPlacementType` se pomocí `ITypeConverter` převede na `TSegmentType`
2. Pomocí `TSegmentType` se získá ze `SegmentRegistry` instance segmentu

3. Do `StrategyManager` se uloží přidávaný prvek, typ umístění i typ segmentu a segment, který se prvku přiřadil. Ty jsou pak přístupné implementacím `IStrategyDocker`.

`MeasureableStrategyManager` jako rozšíření této třídy v konstrukturu přijímá rozhraní `IElementMeasureProvider` sloužící ke změření výšky přidávaného prvku do segmentu `MeasureSegment`. Pokud vývojář nepotřebuje pracovat s instancemi segmentů a `SegmentRegistry`, může využít `BasicStrategyManager`, která je předkem `StrategyManager` a pouze pracuje s `TPlacementType` a `TSegmentType`.

IStrategyDocker

Dockery strategií jako implementace rozhraní `IStrategyDocker` rozmisťují prvky do segmentů aplikováním různých strategií, uvedených v 2.2. Rozhraní `IStrategyDocker` obsahuje pouze metodu `Dock()`, která je volána v cyklu rozmístění. Metoda rozmístí prvky, které získá v konstruktoru, nejčastěji třídu `StrategyManager`. Jelikož docker prvky rozmisťuje, může také určovat jejich výšku v segmentu – proto může implementovat `IElementMeasureProvider`. `StrategyManager`, jehož prvky docker umisťuje, poskytuje ke každému prvku jeho segment a další ukládané informace, na jejichž základě je docker schopný určit přesné umístění prvku pomocí vyměřování matematickými funkcemi z jmenného prostoru `GTTG.Core.PlacementUtils`.

4.2 Pokrytí požadavků unit testy

Unit testy jsou implementovány v projektu `GTTG.Core.Tests` vůči běhovému prostředí `.NET Core` 2.0. K implementaci testů jsme použili unit test framework xUnit [14] a *mock* knihovnu Moq [15]. V následující tabulce se nachází oblasti požadavků, které jsme v testech pokryli. Každý test má pomocí atributu přidělené číslo požadavku v projektu testů, skládající se ze zkratky oblasti požadavků a unikátního čísla v této oblasti (neodpovídající očíslování požadavků v textu), jako na následujícím fragmentu kódu:

```
1: [Trait("Req.no", "COMP16")]
2: public void TranslateScaleOutOfBounds() { /*...*/ }
```

Rozdělení do oblastí požadavků

Oblasti požadavků	**Rozsah požadavků**	**Tag pro unit test**
Modifikace komponenty	**R1** - **R4**	**COMP**
Práce s čas. intervaly	**R5** - **R7**, **R11**-**R13**	**DTI**
Zobrazitelné prvky	**R23** - **R24**	**VE**
Vrstvy kreslení	**R20** - **R22**	**DL**
Hit-testování	**R27** - **R28**	**HT**

V další tabulce jsou uvedena přímá pokrytí požadavků konkrétními testy.

Přímé pokrytí

R1	**COMP**12-16	**R2**	**COMP**7-10
R3	**COMP**6,11	**R4**	**COMP**16
R5	**DTI**10-12	**R6**	**DTI**14
R7	**COMP**16	**R11**	**DTI**10
R22	**DL**1-6	**R21**	**DL**7,8
R24	**VE**11-20	**R26**	**VE**1-10
R28	**HT**1-15		

4.3 GTTG.Model

`GTTG.Model` *assembly* obsahuje implementaci základního modelu obsahu nákresného jízdního řádu, kterou můžou aplikace používající knihovnu dále rozšiřovat, podle 2.2. Implementace je v projektu rozdělena do dvou jmenných prostorů:

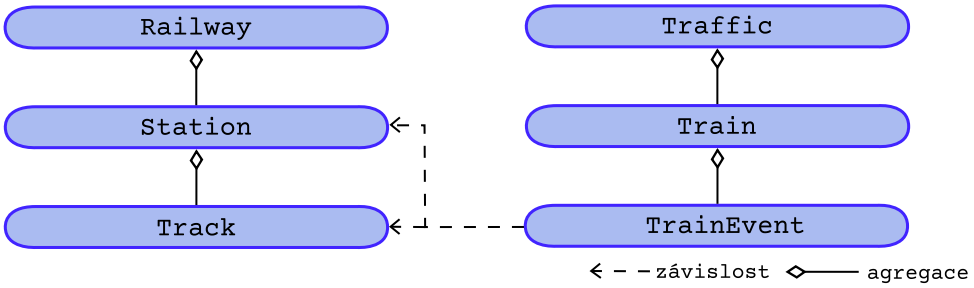
`Model`	Třídy obsahující datovou reprezentaci modelu
`ViewModel`	Třídy vytvářející vizuální reprezentaci modelu v knihovně

Model se v rámci obou těchto reprezentací rozděluje do následujících částí:

`Infrastructure`	Model traťového úseku s dopravními body
`Traffic`	Model vlaků v traťovém úseku

4.3.1 Model

Knihovna vytváří nejmenší možnou základní implementaci modelu dat popisující obsah nákresných jízdních řádů. Všechny třídy modelu jsou potomky třídy `ObservableObject` z 4.1.3, pomocí které je možné vytvářet na model data binding. Na obrázku 4.11 se nachází schéma tříd modelu. Nejdříve si popíšeme třídy v levém sloupci vytvářející model infrastruktury traťového úseku.



Obrázek 4.11: Schéma tříd vytvářejících model

Model infrastruktury

Třída `Railway` představuje traťový úsek obsahující kolekci tříd `Station` odpovídající dopravním bodům. Každá `Station` obsahuje koleje reprezentované třídou `Track`.

Model provozu na trati

Třída `Traffic` seskupuje všechny vlaky. Vlaky jsou reprezentovány třídou `Train` a obsahují plán jízdy, který se skládá z immutable událostí `TrainEvent`. Události jsou popsány `DateTime`, `Station` a `Track` – odpovídají tak nějakému dopravnímu bodu na trati v čase. Navíc obsahují vlastnost výčtového typu `TrainEventType`, který udává informaci, zda je událost příjezdem, odjezdem nebo průjezdem.

4.3.2 View modely

View model i model představuje stromovou strukturu tříd, které jsou na sobě závislé. Na následujícím fragmentu kódu se nachází view model pro `Station` obsahující view modely kolejí `Track`. U modelu jako je `Station` máme k dispozici pouze základní třídy, které je možné *downcasty* přetypovat na konkrétní implementaci. Instance modelu se v základních třídách view modelu vyskytují jako vlastnosti. Aby nebylo nutné provádět downcasty mezi třídami view modelu, v kterých se musí často pracovat s konkrétní implementací, jsou třídě dodány typové argumenty určující konkrétní implementaci. Na fragmentu kódu je dodán `StationView` typový argument `TTrackView` jako konkrétní implementace `TrackView`. V konkrétní implementaci `StationView` je možné přistoupit k poli konkrétní implementace `TrackView` bez downcastů.

```
1:  public class StationView<TTrackView> : InfrastructureViewElement
2:     where TTrackView : TrackView {
3:
4:      public Station Station { get; }
5:      public ImmutableArray<TTrackView> TrackViews { get; }
```

Knihovna nekontroluje jakoukoliv logickou správnost (*sanity*) dat dodaných modelem. V případě existujících dat, které mají špatné hodnoty, komponenta vykresluje obsah špatným způsobem.

Factory metody pro tvorbu view modelu

Aby bylo možné vytvořit view model přímo v konstruktorech, vytvořili jsme rozhraní s *factory* metodami, které vytváří z instancí modelu view model. Na následujícím fragmentu kódu se nachází rozhraní pro vytvoření konkrétní třídy `TTrackView` jako potomka `TrackView` z modelu `Track`. View model `StationView` pak vytváří pomocí factory metod seznam konkrétních `TTrackView`. Pomocí rozhraní je tak možné nad rámec základní implementace vytvořit konkrétní třídy, kterým je možné dodat další parametry.

```
1:  public interface ITrackViewFactory<out TTrackView>
2:     where TTrackView : TrackView {
3:
4:      TTrackView CreateTrackView(Track track);
5:  }
6:
7:  public class StationView<TTrackView> : InfrastructureViewElement
8:     where TTrackView : TrackView {
9:
10:     public StationView(Station station,
11:                        ITrackViewFactory<TTrackView> trackViewFactory) {
12:
13:         Station = station;
14:         TrackViews = ImmutableArray
15:         .CreateRange(station.Tracks.Select(trackViewFactory.CreateTrackView));
16:     }
17: }
```

Základní implementace poskytuje dvě verze view modelu. Pro jednoduchost zavedeme [*] jako zkratku za název entity modelu, například „Train“:

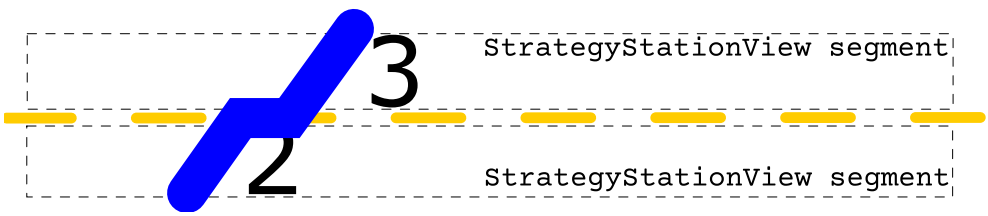
`[*]View`	Verze bez podpory strategií
`Strategy[*]View`	Třídy obsahující prefix *Strategy* implementují práci se strategiemi

View model infrastruktury bez podpory strategií

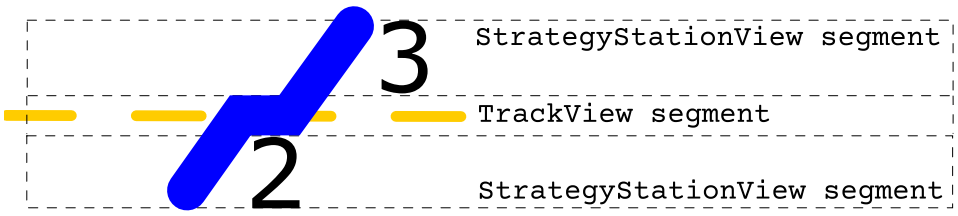
Základní view model infrastruktury pracuje se základním modelem, který jsme si představili. Neobsahuje tedy údaje kilometrické polohy dopravních bodů na trati. Prvky ve view modelu se proto rozmisťují na vertikální ose rovnoměrně. `RailwayView` obsahuje kolekci `StationView`, která obsahuje horizontální čáry kolejí `TrackView`. Pro všechny části view modelu infrastruktury platí, že v případě, kdy výška v `DesiredSize` přesahuje konečnou velikost přidělenou v metodě `Arrange()`, je obsah proporčně zmenšen podobně jako v R8.

TrackView a práce s čarami

Společným prvkem implementace view modelu se strategiemi i bez strategií je `TrackView`, který představuje oblast pro vykreslení horizontální čáry koleje `Track`. Jeho implementace zajišťuje správné umístění šikmých čar představující průběh jízdy vlaků i horizontální čáry dopravního bodu (koleje) do horizontálního pruhu, který je `TrackView` v cyklu rozmístění vyhrazen. Na obrázku 4.12 se nachází problém, který implementace odstraňuje. Když má vlak v nějakém dopravním bodu delší dobu pobytu, jeho čára má být vykreslena na prostředek čáry v `TrackView`. Jelikož čáry můžou mít obecně různou tloušťku, mohl by nastat problém, kdy se budou čáry překrývat se segmenty, které `StrategyStationView` kolem `TrackView` rozmisťuje. Problém se řeší v `TrackView` pomocí segmentů, jako na obrázku 4.13. V rámci jednoho nákresného jízdního řádu se vytvoří `SegmentRegistry` pro segmenty, které budou představovat horizontální čáry. Při vytváření `TrackView` se vytvoří instance segmentu a přidá se pod typem `LineType` obsahující instanci `Track` do `SegmentRegistry` segmentů `MeasureableSegment`. Ostatní view modely, jako třeba `TrainView`, pak přidají svou čáru tomuto segmentu ke změření. Samotný segment se změří implementací `MeasureOverride()` třídy `TrackView`. Při jeho `ArrangeOverride()` se nastaví konečná výška segmentu a v dalších částech cyklu rozmístění se upraví i výška čar ostatních view modelů, které čáry do segmentu registrovaly. Navíc je lehké pomocí vlastnosti `SegmentContentMiddle` ze segmentu přes `SegmentRegistry` získávat z jiného view modelu umístění horizontálních čar kolejí na plátně. To se hodí například `TrainView` pro vykreslení průběhu jízdy vlaku.



Obrázek 4.12: Čáry průběhu jízdy vlaků překrývající se s kótami

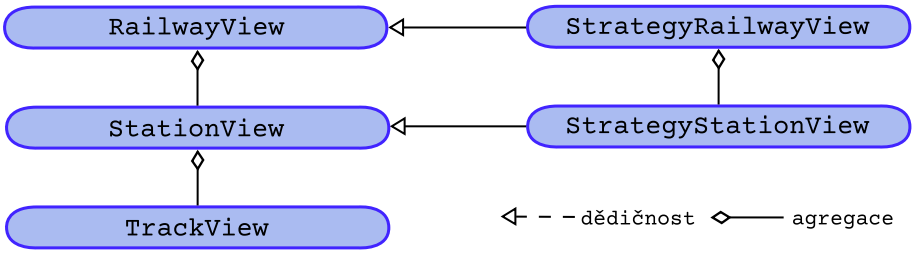


Obrázek 4.13: Čáry průběhu jízdy vlaků umístěné do segmentu čar

Třída `LinePaint` představuje čáry, které používá například `TrackView` nebo `TrainView` k reprezentaci čar, které, jak jsme si uvedli, můžou měnit svou velikost. `LinePaint` je *wrapperem* nad `SKPaint`, které v knihovně SkiaSharp udává vlastnosti vykreslení čar, textu nebo různých objektů. Vlastnost `StrokeWidth` třídy `SKPaint` představuje tloušťku čáry. Třída `LinePaint` si pamatuje z konstruktoru přidělenou požadovanou tloušťku čáry `DesiredStrokeWidth`, kterou poskytuje při měření `Measure()`. Pomocí `Arrange()` se pak nastaví jiná tloušťka čáry.

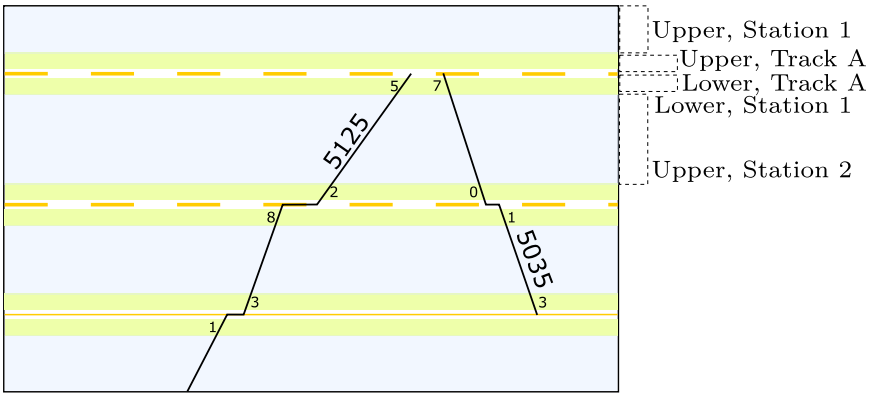
Rozšiřující implementace view modelu infrastruktury se strategiemi

Rozšiřující implementace view modelů podporující strategie s prefixem `Strategy` v názvu je potomkem základní implementace view modelů. Tento vztah mezi třídami se nachází na obrázku 4.14. View model infrastruktury se strategiemi rozšiřuje základní implementaci tak, že k třídám `RailwayView` a `StationView` přidává segmenty v třídách `StrategyRailwayView` a `StrategyStationView`.



Obrázek 4.14: Závislosti mezi třídami základního view modelu a rozšiřujícího view modelu se strategiemi

Na obrázku 4.15 se nachází `MeasureableSegment` segmenty, které zmíněné rozšířené view modely obsahují, kategorizované strukturou `SegmentType<T>`. `SegmentType<T>` nese informaci o vizuálním umístění segmentu ve view modelu (`Lower`, `Upper`) a instanci modelu `T`, kterou je v těchto view modelech `Track` nebo `Station`. `StrategyRailwayView` obsahují a určují výšku `SegmentType<Station>` segmentů, které jsou používány k umístění čísel vlaků. `StrategyStationView` obsahují a určují výšku `SegmentType<Track>` segmentů, do kterých se umísťují údaje v ostrých úhlech související s průběhem jízdy vlaků, jako jsou kóty.



Obrázek 4.15: Zvýrazněné segmenty na nákresném jízdním řádu

View model vlaků

Základní view model vlaků `Train` je implementován třídou `TrainView`, která vykresluje šikmou čáru představující průběh jízdy vlaku. Třída pracuje s typovým argumentem `TTrain` odpovídající konkrétní implementaci modelu `Train`. Jelikož se `TrainView` nikam na plátně nerozmisťuje, dědí od `Visual`. Pracuje s třídou `TrainPath` jako s implementací `ITrainPath`, představující šikmou čáru průběhu jízdy vlaku. `TrainView` obsahuje několik virtuálních metod, jejichž význam si vysvětlíme. Metoda `UpdateTrainViewContent()` se volá v případě, když došlo ke změně modelu. Metoda `Arrange()` se volá v rámci cyklu rozmístění, kdy se například v `TrainPath` musí přepočítat umístění šikmé čáry. Metoda `OnDraw()` slouží k vykreslení `TrainPath` a informací o vlaku. Nyní si představíme, jak tyto metody modifikují třídu `TrainPath`.

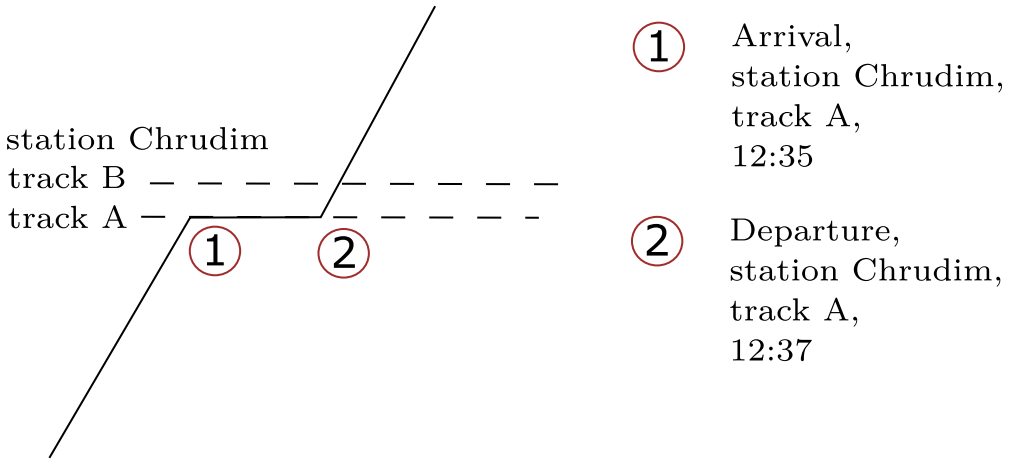
Vykreslení průběhu jízdy vlaku

Třída `TrainPath` jako součást `TrainView` spravuje vykreslování šikmé čáry představující průběh jízdy vlaku. Metoda `Update()` přijímá kolekci `TrainEvent`, které odpovídají jednotlivým událostem jako odjezd nebo příjezd do stanice. Tyto události pak představují body na čáře průběhu jízdy vlaku, jako na obrázku 4.16.

Tloušťka čáry je proměnná a určuje se při cyklu rozmístění. V části věnující se `TrackView` jsme si popsali, že `TrackView` přidává segmenty do `SegmentRegistry`, které jsou určeny k vyměření tlouštěk čar procházející kolejí. `TrainPath`, představující takovou čáru, získá v konstruktoru `SegmentRegistry` a čáru `LinePaint`, která se bude vykreslovat a její tloušťka se bude měnit. V `Update()` metodě přidá tuto čáru do segmentů kolejí `Track` ze všech instancí `TrainEvent`.

Metoda `Update()` se volá v případě, kdy došlo ke změně průběhu jízdy vlaku. Je možné, že se změní i koleje `Track`, kterými vlak projíždí. Proto se v `Update()` metodě před provedením zmíněné procedury registrací odstraní původní registrace čar ze všech segmentů.

Metoda `Arrange()` se volá v rámci cyklu rozmístění, kdy se musí přerozmístit šikmá čára představující průběh jízdy vlaku. Pomocí rozhraní `IViewProvider` z 4.1.1 se převedou `DateTime` údaje z událostí `TrainEvent` na horizontální souřadnice plátna. Abychom určili vertikální polohu kolejí, získáme ze `SegmentRegistry` instance segmentů podle instance `Track` z každé události. Segmenty obsahují



Obrázek 4.16: `TrainEvent` jako body tvořící čáru průběhu jízdy vlaku

vlastnost `SegmentContentMiddle`, která určuje umístění středu horizontální čáry koleje na plátně. Z těchto dvou údajů je sestaven bod `SKPoint` na čáře, reprezentovanou `SKPath`. V metodě `Draw()` dojde k vykreslení této čáry použitím `SKPaint` z `LinePaint`.

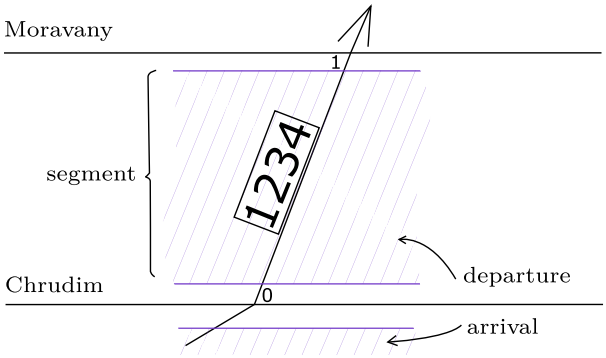
Rozšířený view model vlaků o strategie

Třída `StrategyTrainView` rozšiřuje `TrainView` o nástroje, které umožní přidávat k šikmé čáře vlaku zobrazitelné prvky jako čísla vlaků a kóty v ostrých úhlech. Nástroje jsou obsaženy ve *facade* rozhraní `IStrategy`, jehož konkrétní implementace je vývojáři přístupná jako typový argument `TStrategy`. Rozhraní obsahuje obecné metody `Dock()` a `Update()` pro práci se strategiemi v rámci instance `TrainView`. Metoda `Dock()`, která rozmístí prvky strategiemi, je volána v rámci přetížené `Arrange()` na `TrainView`. Rozhraní `IStrategy` implementuje `IVisual`, čili je vykrelováno v `Draw()` metodě a prvky registrované ve strategiích je možné hit-testovat.

Práce se strategiemi pro rozšířený view model infrastruktury

Rozhraní `IStrategy` je v `GTTG.Model` implementováno třídou `Strategy`, která je provázána se segmenty view modelu infrastruktury `StrategyRailwayView` a `StrategyStationView`, do kterých umisťuje přidávané prvky. Pro každý z těchto view modelů s jejich segmenty existuje unikátní strategie. První strategie související se segmenty v `SegmentRailwayView` kategorizovaných podle `SegmentType<Station>` slouží k umisťování čísel vlaků na prostředek čáry mezi dvěma dopravními body, jako na obrázku 4.17.

Prvky se do této strategie přidávají přes instanci `StrategyManager` nazvanou `StationStrategyManager` uvedením struktury `TrainEventPlacement`, která prvek umístí ve specifikovaném dopravním bodu `Station` do horního nebo dolního segmentu, který odpovídá příjezdu nebo odjezdu podle výčtového typu v třídě `TrainEvent`, která se do struktury přidává spolu s výčtovým typem `AnglePlacement` umísťující prvek do ostrého nebo tupého úhlu. Z `TrainEvent`

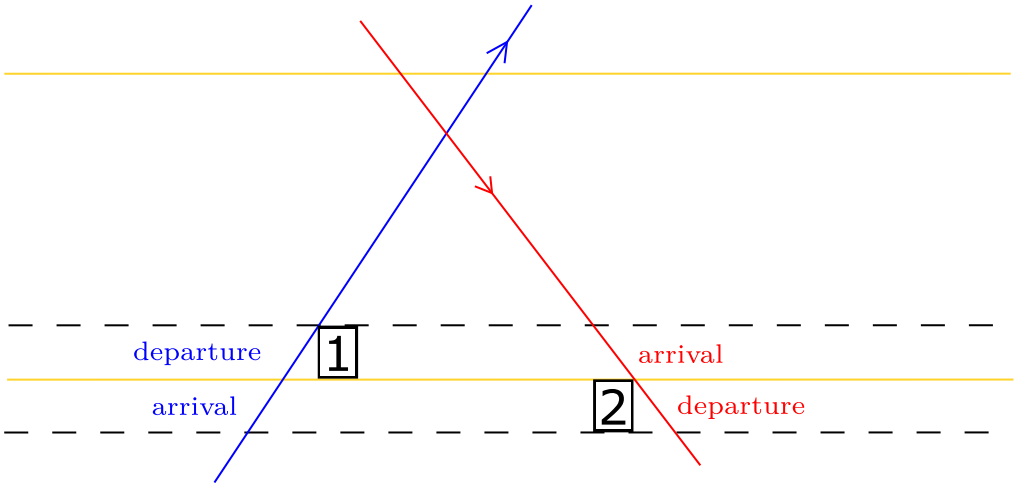


Obrázek 4.17: Aplikování strategie rozmisťující číslo vlaků mezi dopravní body

se pak určí i instance `Station`, tvořící `SegmentType<Station>`. Na následujícím fragmentu kódu přidáváme prvek představující číslo vlaku do tupého úhlu z události `TrainEvent`, jejíž typ `TrainEventType` je odjezd – `Departure`. Stejné umístění jako `Departure` má hodnota průjezdu `Passage`. Výsledek aplikování zmíněné strategie na umístěný prvek odpovídá obrázku 4.17.

```
1: var trainNumber = /*.. (ViewElement) ..*/
2: var trainEvent = /* Station: Chrudim, Type: Departure ...*/
3: var anglePlacement = AnglePlacement.Obtuse;
4: var eventPlacement = new TrainEventPlacement(trainEvent, anglePlacement);
5: strategyManager.Add(eventPlacement, trainNumber);
```

Druhá strategie slouží k umisťování prvků do segmentů kategorizovaných `SegmentType<Track>` v `StrategyTrainView`. Prvky se přidávají do instance `StrategyManager` nazvané `TrackStrategyManager` uvedením stejné struktury `TrainEventPlacement` jako pro první strategii. Strategie umisťuje prvky do ostrých nebo tupých úhlů co nejblíže průsečíku šikmé čáry s horizontální čarou, jako na obrázku 4.18.



Obrázek 4.18: Aplikování strategie rozmisťující kóty kolem kolejí

V metodě `Dock()` se na přidané prvky z obou `StrategyManager` instancí aplikují strategie, které jsou implementací rozhraní `IStrategyDocker`. V další části si uvedeme, jak jsou rozhraní pro práci se strategiemi pro tento view model implementovány.

Implementace strategií pro rozšířený view model infrastruktury

Pro používané strategie v rámci rozšířeného view modelu existují i implementace rozhraní `ITypeConverter` a `IStrategyDocker`. Třída `TrainEventPlacementConverter` implementuje rozhraní převodů typu `TrainEventPlacement` do `SegmentType<Track>` nebo `SegmentType<Station>`. K převodům se používá třída `TrainPath`, čili pro každou instanci `StrategyTrainView` vzniká nový konvertor. Implementace k převodům používá rozmístění bodů `TrainPath`, které odpovídají událostem `TrainEvent`.

Z typu `TrainEventPlacement` získáme událost, kterou převedeme. Podle směru události `Departure` nebo `Arrival` se začne procházet seznam bodů v `TrainPath`. Nejdříve se najde index odpovídající události. Budeme chtít najít vektor k jinému bodu v určeném směru – pro `Arrival` hledáme body menších indexů, pro `Departure` vyšší. Podle vertikálního směru vektoru pak určíme umístění segmentu `Lower` nebo `Upper`, jako na obrázku 4.18.

Třídy `StationStrategyDocker` a `TrackStrategyDocker` jako implementace `IStrategyDocker` pak pomocí tohoto vektoru nachází část cesty představující šikmou čáru ohraničující oblast, kam je prvky možné umístit.

Licence knihovny a NuGet balíčky

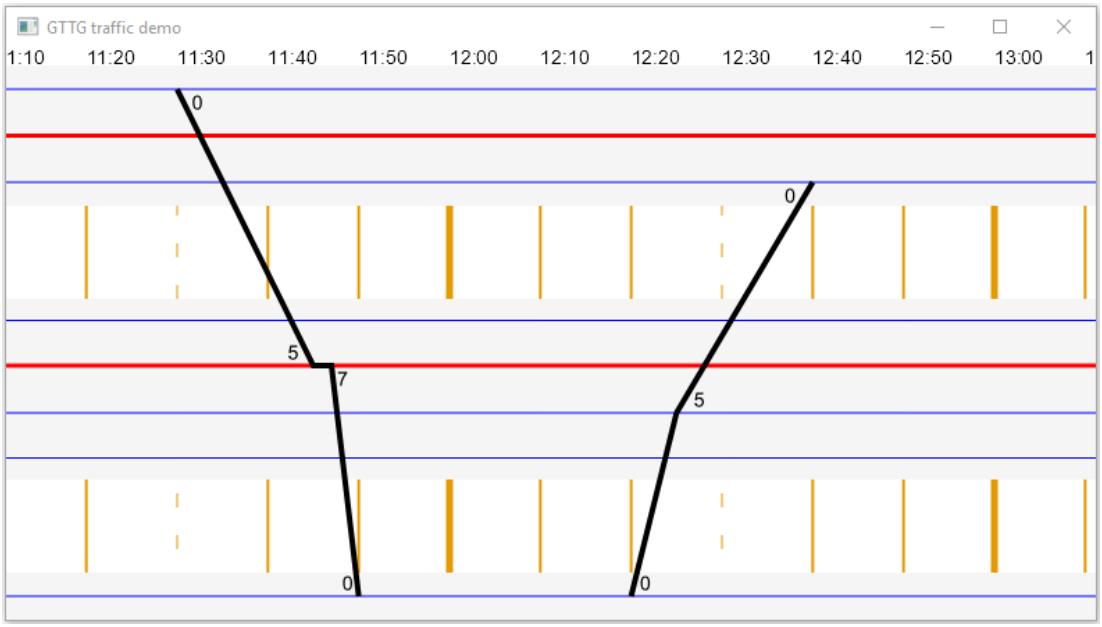
Celá knihovna GTTG je nabízena pod licencí MIT, více popsané v souboru `LICENSE.TXT` umístěném v kořenovém adresáři příloh práce. Oba dva projekty jsou dostupné jako NuGet balíčky `GTTG.Core` [16] a `GTTG.Model` [17].

5. Uživatelská dokumentace knihovny GTTG

Tato kapitola pomocí série tutoriálů popisuje, jak knihovnu GTTG začlenit do aplikací a vytvořit obsah nákresného jízdního řádu rozšířením jejího základního modelu. Budeme chtít vytvořit jednoduchou aplikaci pro práci s grafikonem vlakové dopravy pro modelová kolejiště, jejíž vývoj rozdělíme do tří částí, které si detailněji popíšeme:

1. Integrace grafické komponenty nákresného jízdního řádu do aplikace
2. Vytvoření vrstev a view modelu infrastruktury
3. Vytvoření view modelu vlaků a práce se strategiemi

Výsledná aplikace se nachází na obrázku 5.1 a je dostupná v repozitáři [18] nebo v příloze práce jako solution v `/tutorials/traffic`. V každém tutoriálu pak uvedeme odkaz na současný meziprodukt aplikace. Pro běh aplikace je potřeba mít instalovaný .NET Framework nejméně ve verzi 4.7.2 [19], který je součástí Windows 10 Fall Creators Update (version 1709). Pro práci se solution potřebujeme .NET Framework 4.7.2 Developer Pack [20]. Dále si v této kapitole uvedeme, jak implementovat strategie a pracovat s obsahem nákresného jízdního řádu.



Obrázek 5.1: Výsledná aplikace vytvořená tutoriály z této kapitoly

5.1 Tutoriál integrace knihovny do aplikací

Cílem tohoto tutoriálu je grafickou komponentu vytvářenou knihovnou integrovat do prvku uživatelského rozhraní. Konečný výsledek tutoriálu se nachází na obrázku 5.2 a je dostupný jako solution v `/tutorials/integration` v příloze práce nebo repozitáři [21].

Obrázek 5.2: Výsledek tutoriálu integrující knihovnu s aplikací

Jako GUI framework pro aplikaci zvolíme WPF. Ve Visual Studiu založíme projekt WPF aplikace. Podle 3.1.1 využijeme knihovnou `SkiaSharp` předpřipravený prvek uživatelského rozhraní `SKElement`, jehož plocha je pokrytá plátnem `SKCanvas`. Proto do projektu přidáme balíček `SkiaSharp.Views`, poskytující třídu `SKElement`. Zároveň instalujeme balíček `GTTG.Core` knihovny GTTG.

Vytvoříme nový *user control* pojmenovaný `GraphicalComponentUserControl` a přepíšeme jeho předka z `UserControl` na `SKElement`. Náš user control nyní obsahuje pouze konstruktor:

```
1: public partial class GraphicalComponentUserControl : SKElement {
2:
3:         public GraphicalComponentUserControl() {
4:             InitializeComponent();
5:         }
6: }
```

Tuto třídu použijeme k implementaci jednoduché aplikační logiky a proto ji nyní doplníme o grafickou komponentu `GraphicalComponent`, která bude vytvářet *engine* pro práci s nákresným jízdním řádem, a umožníme uživateli pracovat s komponentou pomocí vstupu z myši.

Inicializace grafické komponenty

Před vytvořením komponenty se musíme rozhodnout, v jakých jednotkách pixelů bude plátno `SKCanvas` pracovat (více v 3.1.2). Budeme používat jednotky pixelů zařízení a proto vlastnost `IgnorePixelScaling` třídy `SKElement` zůstane podle 3.1.2 nastavená na původní hodnotě `false`. Jelikož vlastnosti prvku uživatelského rozhraní jako `ActualWidth` jsou udávány v jednotkách nezávislých na

zařízení, může se tato délka lišit od velikosti plátna `CanvasSize`, v tomto případě uvedené v jednotkách pixelů zařízení.

V konstruktoru našeho uživatelského prvku vytvoříme privátní proměnnou třídy `GraphicalComponent` inicializovanou bazparametrickým konstruktorem. Protože budeme chtít komponentě při inicializaci dodat velikost prvku `SKElement` a v konstruktoru uživatelského prvku ještě určená není, přidáme v něm event handler `OnLoaded()` k eventu `Loaded`, který se volá, když v cyklu rozmístění uživatelského rozhraní má již `SKElement` přidělenou velikost. V metodě `OnLoaded()` nejdříve komponentě nastavíme časové intervaly, které zobrazuje. Pro všechna data, s kterými bude aplikace pracovat, vytvoříme třídu `TrainTimetableData`. V této třídě vytvoříme `DateTimeContext`, udávající časový interval pokrývající komponentu v `ViewDateTimeInterval` a její zobrazitelnou oblast nákresného jízdního řádu intervalem `ContentDateTimeInterval` délky čtyř hodin:

```
1:  public static class TrainTimetableData {
2:
3:      public static DateTime Start { get; } = DateTime.Today.AddHours(10);
4:
5:      public static DateTimeInterval ViewDateTimeInterval { get; }
6:       = new DateTimeInterval(Start.AddHours(1), Start.AddHours(3));
7:
8:      public static DateTimeInterval ContentDateTimeInterval { get; }
9:       = new DateTimeInterval(Start, Start.AddHours(4));
10:
11:     public static DateTimeContext DateTimeContext =
12:      new DateTimeContext(ContentDateTimeInterval, ViewDateTimeInterval);
```

`DateTimeContext` nastavíme komponentě v metodě `OnLoaded()`:

```
1:  private void OnLoaded() {
2:      _graphicalComponent
3:      .TryChangeDateTimeContext(TrainTimetableData.DateTimeContext);
```

Dále zajistíme nastavování velikosti komponenty, aby odpovídala velikosti `SKElement`. V konstruktoru uživatelského prvku proto přidáme k eventu `SizeChanged` handler `OnSizeChanged()`, v kterém přidělíme grafické komponentě velikost `CanvasSize` v jednotkách pixelů zařízení:

```
1:  private void OnSizeChanged() {
2:      _graphicalComponent?.TryResizeView(CanvasSize.Width, CanvasSize.Height);
3:  }
```

V rámci inicializace komponenty v `OnLoaded()` tento handler zavoláme k nastavení nové velikosti, při dokončování inicializace:

```
1:  private void OnLoaded() {
2:      /*...*/
3:      OnSizeChanged();
4:  }
```

Aby bylo možné zobrazení měnit, v další části propojíme modifikace komponenty s uživatelským vstupem z myši.

Modifikace komponenty

Komponentu budeme chtít modifikovat její metodou pro translaci `TryTranslate()` a škálování `TryScale()` pomocí handlerů eventů vstupu z myši. Škálování bude odpovídat skrolování a translace buda implementována držením levého tlačítka myši při jejím posunu. Jak jsme si uvedli, komponenta pracuje s jinými jednotkami pixelů než WPF, které souřadnice kurzoru myši uvádí v jednotkách nezávislých na zařízení. Proto bychom museli aplikovat převod do jednotek pixelů zařízení u každého handleru, jako na následujícím kódu:

```
1:  /* in method of SKElement inherited class */
2:  var uiHorizontalPosition = 100;
3:  var dpiFactor = CanvasSize.Width / ActualWidth; // e.g.: (2000 / 1000)
4:  var componentHorizontalPosition = uiHorizontalPosition * dpiFactor;
```

Aby nebylo potřeba pro každý handler vstupu z myši přepočítávat souřadnice, vytvořili jsme třídu `WpfMouseInputSource`, kterou najdeme v tutoriálu `/tutorials/integration/` v příloze práce. Přidáme ji do projektu spolu se strukturami `MouseInputArgs` a `MouseZoomArgs` a třídou `DragProcessor`, která bude sloužit pro implementaci posunu pohledu myší. Zároveň stáhneme balíček `System.Reactive`. V metodě `OnLoaded()` inicializujeme `WpfMouseInputSource` a `DragProcessor` a vytvoříme handlery na následující eventy, jako třeba handler `Scroll()`:

```
1:  private void OnLoaded() {
2:
3:    /*...*/
4:    var source = new WpfMouseInputSource(this);
5:    source.LeftUp.Subscribe(LeftUp);
6:    source.LeftDown.Subscribe(LeftDown);
7:    source.Move.Subscribe(Move);
8:    source.Scroll.Subscribe(Scroll);
9:    source.Leave.Subscribe(Leave);
10:
11:   _dragProcessor = new DragProcessor();
12:
13:   OnSizeChanged();
14: }
15:
16: public void Scroll(MouseZoomArgs args) {
17:
18:     var point = new SKPoint(args.X, args.Y);
19:     var result = _graphicalComponent.TryScale(point, args.Delta);
20:     if (result != ScaleTransformationResult.ViewUnmodified) {
21:         InvalidateVisual();
22:     }
23: }
```

Kreslení, které probíhá v `OnPaintSurface()`, je voláno `SKElement`, když proběhla invalidace zobrazení, například přes metodu `InvalidateVisual()`. Handler `Scroll()` volá tuto metodu v případě, kdy je modifikace úspěšná. Posun zobrazované části v komponentě implementujeme posunem při držení levého tlačítka myši. V handleru `LeftDown()` nejdříve inicializujeme posun:

```
1:  public void LeftDown(MouseInputArgs args) {
2:      _dragProcessor.TryInitializeDrag(args);
3:  }
```

Třída `DragProcessor` si nyní uloží pozici kurzoru myši. V následujících posunech myší bude handler `Move()` předávat `DragProcessor` pozici kurzoru myši, z které se spočítá délka posunu, předávaná grafické komponentě:

```
1:  public void Move(MouseInputArgs args) {
2:
3:    if (!_dragProcessor.IsEnabled) {
4:      return;
5:    }
6:
7:    var translation = _dragProcessor.GetTranslation(args);
8:    var result = _graphicalComponent.TryTranslate(translation);
9:
10:   if (result == TranslationTransformationResult.ViewModified) {
11:     InvalidateVisual();
12:   }
13: }
```

Operace posunu je ukončena při opuštění uživatelského prvku kurzorem myši nebo uvolněním levého tlačítka:

```
1:  public void LeftUp(MouseInputArgs args) {
2:      _dragProcessor.TryExitDrag(args);
3:  }
4:
5:  public void Leave(MouseInputArgs args) {
6:      _dragProcessor.TryExitDrag(args);
7:  }
```

S translací pomocí vstupu z myši vzniká jeden problém, který je vhodné řešit. V případě, že výška obsahu odpovídá výšce komponenty a není tedy možné výřez obsahu v komponentě posouvat po vertikální ose, dochází k trhavému pohybu, pokud má vektor pohybu myši nenulovou vertikální souřadnici. Modifikace translace se takovým vektorem totiž správně komponentou neaplikují. Situace nastává, pokud vývojář chce pohled posunout po horizontále. Problém vyřešíme tím, že vertikální souřadnici v těchto případech nastavíme 0:

```
1:  public void Move(MouseInputArgs args) {
2:      /*...*/
3:      var translation = _dragProcessor.GetTranslation(args);
4:
5:      var viewProvider = _graphicalComponent;
6:      if (viewProvider.ContentMatrix.ScaleY.Equals(1.0f) &&
7:          viewProvider.ContentHeight.Equals(viewProvider.ViewHeight)) {
8:          translation.Y = 0;
9:      }
10:
11:     var result = _graphicalComponent.TryTranslate(translation);
12:     /*...*/
13: }
```

Kreslení v komponentě

Pokaždé, když se invaliduje prvek, proběhne vykreslení v `OnPaintSurface()`, z jehož argumentů získáme `SKCanvas`. Aby vykreslení proběhlo i v případě, kdy je obsah grafické komponenty inicializován, přidáme na konec metody `OnLoaded()` invalidaci vizualizace:

```
1:  private void OnLoaded() {
2:      /*...*/
3:      InvalidateVisual();
4:  }
```

Vykreslíme do komponenty horizontální pruh časových údajů reprezentujících `DateTimeContext`. Nastavíme plátnu matici `ContentMatrix` grafické komponenty, která zobrazí v komponentě správný výřez obsahu. Všechny časové údaje, které musíme vykreslit, se nachází v intervalu `ContentDateTimeInterval` grafické komponenty. Přesné časové údaje můžeme získat metodou intervalu `GetDateTimesByPeriod()`. K vykreslení textu vytvoříme v uživatelském prvku instanční proměnnou `_textPaint` třídy `SKPaint`:

```
1:  private readonly SKPaint _timePaint =
2:      new SKPaint { Color = SKColors.Black,
3:                    Style = SKPaintStyle.Fill,
4:                    IsAntialias = true, TextSize = 14 };
```

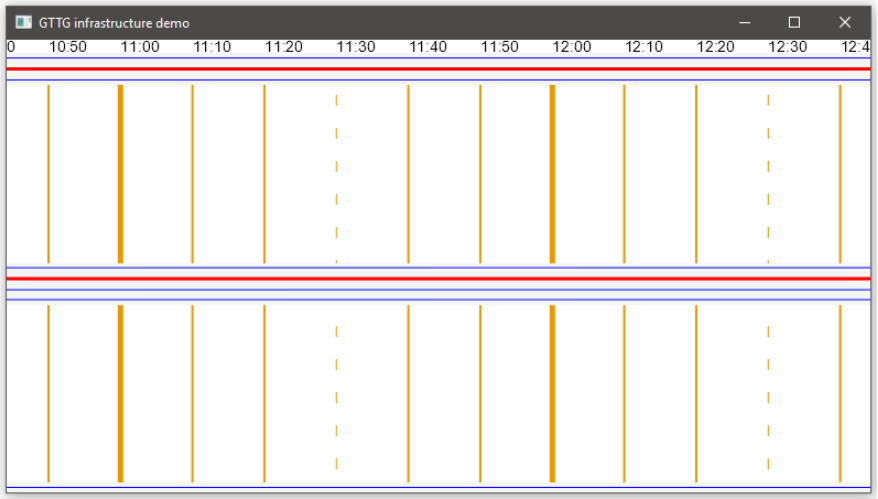
Převody časových údajů do bodů plátna `GetContentHorizontalPosition()` vykreslíme tyto údaje na horizontální osu umístěnou na vrchu komponenty:

```
1:  protected override void OnPaintSurface(SKPaintSurfaceEventArgs e) {
2:
3:      if (_graphicalComponent == null) return;
4:      e.Surface.Canvas.Clear();
5:      e.Surface.Canvas.SetMatrix(_graphicalComponent.ContentMatrix);
6:
7:      var start = _graphicalComponent.ContentDateTimeInterval.Start;
8:      var dateTimes = _graphicalComponent.ContentDateTimeInterval
9:                      .GetDateTimesByPeriod(start, TimeSpan.FromMinutes(10));
10:
11:     foreach (var dateTime in dateTimes) {
12:
13:         var x = _graphicalComponent.GetContentHorizontalPosition(dateTime);
14:         var timeString = dateTime.ToString("HH:mm");
15:         e.Surface.Canvas.DrawText(timeString, new SKPoint(x, 30), _timePaint);
16:     }
17: }
```

Takto jsme zapojili komponentu do WPF aplikace. Při spuštění by se měl zobrazit horizontální pruh časových údajů jako na obrázku 5.1. Pomocí vstupu z myši můžeme translací a škálováním konfigurovat nastavení pohledu v komponentě. V další části vytvoříme a vykreslíme základní model infrastruktury trati delegováním kreslení z metody `OnPaintSurface()` do jednotlivých vrstev.

5.2 Tutoriál práce s vrstvami a vytvoření modelu infrastruktury

V této části k existující aplikaci pracující s komponentou přidáme vykreslování souřadnicové sítě svislých a vodorovných čar a rozšíříme základní implementaci modelu, uvedenou v podkapitole 4.3. Konečný výsledek tutoriálu se nachází na obrázku 5.3 a je dostupný jako solution v `/tutorials/infrastructure` v příloze práce nebo repozitáři [22]. Začneme otevřením předchozího tutoriálu ve Visual Studiu, kde do projektu přidáme balíček `GTTG.Model` obsahující základní implementaci modelu, s kterým budeme pracovat.



Obrázek 5.3: Výsledek tutoriálu vytvářející model infrastruktury

Rozdělení obsahu do vrstev

Jelikož budeme vykreslovat do komponenty několik spolu nesouvisejících částí obsahu, rozdělíme ho do vrstev, které se na sebe budou vykreslovat. Vytvoříme tři vrstvy, které jsou potomkem `ContentDrawingLayer`:

1. Spodní vrstva svislých čar vizualizující časové údaje - `TimeLinesLayer`
2. Prostřední vrstva vizualizující stanice s kolejemi - `InfrastructureLayer`
3. Vrchní vrstva vykreslující původní časovou osu - `TimeAxisLayer`

Vrstvy jako potomci `ContentDrawingLayer` umožní vykreslovat celý zobrazitelný obsah nákresného jízdního řádu. Doposud jsme kreslili v metodě `OnPaintSurface()` v uživatelském prvku `GraphicalComponentUserControl`. Nyní v této metodě budeme vykreslovat instanci `DrawingManager`, kterou vytvoříme v konstruktoru `GraphicalComponentUserControl`. Jako první argument jí předáme `CanvasFactory` z knihovny `GTTG.Core`, která bude vytvářet pro vykreslení vrstev typu `ContentDrawingLayer` plátno pro celý nákresný jízdní řád. Jako druhý argument předáme konstruktoru třídu `DrawingLayersOrder`, kterou vytvoříme jako implementaci rozhraní `IRegisteredLayersOrder` a bude obsahovat definici pořadí vykreslení našich vrstev:

```
public class DrawingLayersOrder : IRegisteredLayersOrder {

    ImmutableList<Type> IRegisteredLayersOrder.DrawingLayerTypeList { get; }
    = new List<Type> {
          typeof(TimeLinesLayer),
          typeof(InfrastructureLayer),
          typeof(TimeAxisLayer)
    }.ToImmutableList();
}
```

Nyní implementujeme všechny tři naše vrstvy, kdy každé vrstvě budeme předávat několik parametrů.

Kreslení ve vrstvě

V této části implementujeme vykreslování vrstvy `TimeLinesLayer` se svislými čarami představující časové údaje. Pro vrstvu vytvoříme konstruktor, v kterém získáme rozhraní `IViewProvider` pro převod časových údajů na horizontální souřadnice. Pro vrstvu vytvoříme instanční proměnnou `SKPaint`, v které budeme udávat vlastnosti tří typů vykreslovaných svislých čar a instanční proměnnou `SKPath` představující svislou čáru:

```
public class TimeLinesLayer : ContentDrawingLayer {

  private readonly IViewProvider _viewProvider;

  private readonly SKPaint _hourLinePaint = new SKPaint {
    Style = SKPaintStyle.Stroke,
    Color = new SKColor(228, 154, 1),
    IsAntialias = true
  };

  private readonly SKPath _verticalHourLine = new SKPath();

  public TimeLinesLayer(IViewProvider viewProvider) {
    _viewProvider = viewProvider;
  }
}
```

Zároveň implementujeme metody třídy `ContentDrawingLayer`. V metodě `OnDraw()` vykreslíme svislé čáry, které odlišíme tloušťkou podle časového údaje, který reprezentují. Například svislé čary hodin vykreslíme takto:

```
var timeContext = _viewProvider.DateTimeContext.ContentDateTimeInterval;

  foreach (var hour in timeContext
      .GetDateTimesByPeriod(timeContext.Start, TimeSpan.FromHours(1))) {

    _verticalHourLine.Reset();
    var canvasX = _viewProvider.GetContentHorizontalPosition(hour);

    _verticalHourLine.MoveTo(new SKPoint(canvasX, 0));
    _verticalHourLine.LineTo(new SKPoint(canvasX, drawingCanvas.Size.Height));
```

```
11:
12:     _hourLinePaint.StrokeWidth = 5 / _viewProvider.Scale;
13:     sdrawingCanvas.Canvas.DrawPath(_verticalHourLine, _hourLinePaint);
14: }
```

Pracujeme stále s jednou instancí `SKPath`, kterou pokaždé resetujeme pomocí `Reset()`. Je důležité, jak budou instance `SKPath` a `SKPaint` vytvářeny, jelikož jsou alokovány jako *unmanaged* kód v knihovně Skia, v C++. Není pak správné vytvářet nové instance během kreslícího cyklu, kterých může být v krátkém časovém intervalu velké množství. Instance se proto vytvoří v konstruktoru vrstvy a případně se upravují. Vytváříme konfiguraci `SKPaint`, v které při jakémkoli nastavení přiblížení pohledu zůstává tloušťka čáry stále stejná, použitím vlastnosti `Scale` rozhraní `IViewProvider`, kdy vhodně nastavíme vlastnost `StrokeWidth`:

```
1: _hourLinePaint.StrokeWidth = 5 / _viewProvider.Scale;
```

Ostatní čáry jiných časových údajů vykreslíme stejným způsobem, pouze metodě `GetTimesByPeriod()` dodáme například `TimeSpan.FromMinutes(10)` a nastavíme jinou tloušťku vykreslované čáry. Na začátku foreach cyklu přeskočíme čáry vykreslované v jiných částech:

```
1: if (minute.Minute == 00 || minute.Minute == 30) {
2:    continue;
3: }
```

Jelikož vrstva nemá žádný další obsah, který se má vykreslovat, metoda `ProvideVisuals()` vrací prázdnou kolekci:

```
1: public override IEnumerable<IVisual> ProvideVisuals() {
2:     yield break;
3: }
```

V další vrstvě `InfrastructureLayer` vykreslíme model infrastruktury – dopravní body a jejich koleje. Nejdříve v další části vytvoříme reprezentaci dat tohoto modelu a poté ho popíšeme třídami, které ho budou vizualizovat.

Vytvoření modelu

K vytvoření modelu použijeme základní implementaci dodanou knihovnou v projektu `GTTG.Model`. Celý model infrastruktury zahrnutý v instanci `Railway` vytvoříme v třídě `TrainTimetableData`. Model si nyní popíšeme. Každá stanice může mít několik kolejí. Koleje budou dvou typů – nákladní a osobní. Vytvoříme třídu `TutorialTrack` jako potomka základního modelu kolejí `Track`. V konstruktoru mu dodáme hodnotu vytvořeného výčtového typu `TrackType` s hodnotami `Cargo` a `Passenger`, udávající, zda je kolej určena pro nákladní nebo osobní dopravu:

```
1:  public class TutorialTrack : Track {
2:
3:      public TrackType TrackType { get; }
4:
5:      public TutorialTrack(TrackType trackType) {
6:          TrackType = trackType;
7:      }
8:  }
```

Na následujícím fragmentu kódu vytvoříme v třídě `TrainTimetableData` instanci modelu:

```
1:  public static Railway Railway { get; } =
2:      new Railway(
3:          new List<Station> {
4:              new Station(
5:                  new List<TutorialTrack> {
6:                      new TutorialTrack(TrackType.Cargo),
7:                      new TutorialTrack(TrackType.Passenger),
8:                      new TutorialTrack(TrackType.Cargo),
9:               /*...*/
```

Při vykreslení od sebe hodnoty `TrackType` vizuálně odlišíme a koleje ve stanicích vykreslíme vedle sebe. Na základě těchto požadavků vytvoříme *view model*, který bude rozšiřovat základní implementaci view modelu z `GTTG.Model`.

Nejdříve vytvoříme view model pro koleje `TutorialTrackView` jako potomka `TrackView`. Všechny prvky view modelu jsou potomky třídy `ViewElement` popsané v 3.2.1, čili můžeme jejich rozmístění a vykreslení konfigurovat. Základní implementace prvku `TrackView` pouze obsahuje čáru představující kolej a `StationView` tyto prvky skládá vedle sebe. Abychom mohli koleje ve stanici od sebe rozlišit, umístíme kolem nich mezery. V `TutorialTrackView` proto přetížíme metody `ArrangeOverride()` a `MeasureOverride()`, kde budeme pracovat s výškou mezery v instanční proměnné `Space`. Při příliš malé výšce `finalSize` v metodě `ArrangeOverride()` se proporcionálně mezery i čára zmenší:

```
1:  public class TutorialTrackView : TrackView {
2:    public readonly int Space = 4;
3:
4:    protected override SKSize MeasureOverride(SKSize availableSize) {
5:      var size = base.MeasureOverride(availableSize);
6:      return new SKSize(size.Width, size.Height + 2 * Space);
7:    }
8:
9:    protected override SKSize ArrangeOverride(SKSize finalSize) {
10:
11:     var scale = finalSize.Height / DesiredSize.Height;
12:     TrackLineSegment
13:     .SetBounds(this, Space * scale, finalSize.Height - Space * scale);
14:     if (LinePaint.DesiredStrokeWidth > finalSize.Height - 2 * Space) {
15:         LinePaint.Arrange(LinePaint.DesiredStrokeWidth * scale);
16:     }
17:     return new SKSize(float.MaxValue, finalSize.Height);
18:   }
19: }
```

Pro vytvoření instance `TutorialTrackView` vytvoříme třídu `TutorialTrackViewFactory` jako implementaci rozhraní `ITrackViewFactory`. V metodě `CreateTrackView()` předáme konstruktoru `TutorialTrackView` podle instance `TutorialTrack` barvu `LinePaint` jiné barvy – `Cargo` získává modrou barvu, `Passenger` červenou. Konstruktoru předáme i vytvořenou instanci segmentu `MeasureableSegment`, s kterou zatím nebudeme pracovat.

Implementace rozhraní `ITrackViewFactory` se bude předávat konstruktoru vytvořeného view modelu `TutorialStationView` jako potomka `StationView`. V jeho konstruktoru nastavíme `HasClipEnabled` na `true`, aby vybarvení proběhlo pouze v obsahu stanice a ne v obsahu celého plátna. Zároveň přetížíme metodu vykreslení:

```
protected override void OnDraw(DrawingCanvas drawingCanvas) {
    drawingCanvas.Canvas.DrawColor(SKColors.Gray);
    base.OnDraw(drawingCanvas);
}
```

Vytvoříme `TutorialStationViewFactory` jako implementaci rozhraní `IStationViewFactory`. Konstruktoru dodáme `TutorialTrackViewFactory`, která je předávána v metodě `CreateStationView()` vytvořené instanci `TutorialStationView`. V metodě `OnLoaded()` našeho uživatelského prvku pak inicializujeme celý view model infrastruktury a získáme instanci `RailwayView`:

```
private void OnLoaded() {
  /*...*/
  var trackViewFactory = new TutorialTrackViewFactory();
  var stationViewFactory = new TutorialStationViewFactory(trackViewFactory);
  _railwayView = new RailwayView(Railway, stationViewFactory);
  /*...*/
}
```

Nyní vytvoříme vrstvu infrastruktury `InfrastructureLayer`, která bude v konstruktoru přijímat `RailwayView`. V metodě `OnDraw()` vykreslíme instanci `RailwayView` a vrátíme jí v metodě `ProvideVisuals()`.

Vrstvu `TimeAxisLayer` implementujeme stejně jako původní kreslení horizontálního pruhu časových údajů v metodě `OnPaintSurface()`. Vrstva bude obsahovat vlastnost `Height`, udávající výšku pruhu, závislou na výšce textu. Tu je obtížnější zjistit – použijeme odhad `FontMetrics.CapHeight` na `SKPaint`, odpovídající výšce textu velkých písmen. Jelikož nepokrývá plně velikosti čísel, které zasahují i pod *baseline*, od které se `CapHeight` měří a která slouží jako řádek pro vykreslení písma, vytvoříme výšku odpovídající desetině `CapHeight`, která se ke `CapHeight` přičte, jako mezera pod baseline. Takto v konstruktoru vrstvy, kterému dodáme `IViewProvider`, určíme výšku `Height`:

```
public TimeAxisLayer(IViewProvider viewProvider) {

    _viewProvider = viewProvider;
    var measuredHeight = Math.Abs(TimePaint.FontMetrics.CapHeight);
    Padding = measuredHeight / 10;
    measuredHeight += 2 * Padding;
    Height = measuredHeight;
}
```

Nyní v metodě `OnLoaded()` všechny tři vrstvy vytvoříme a registrujeme pomocí `ReplaceRegisteredDrawingLayer()` do `DrawingManager`:

```
1: _timeAxisLayer = new TimeAxisLayer(_graphicalComponent);
2: _drawingManager
3:   .ReplaceRegisteredDrawingLayer(new InfrastructureLayer(_railwayView));
4: _drawingManager.
5:   ReplaceRegisteredDrawingLayer(new TimeLinesLayer(_graphicalComponent));
6: _drawingManager
7:   .ReplaceRegisteredDrawingLayer(_timeAxisLayer);
```

V metodě `OnPaintSurface()` vykreslíme jenom `DrawingManager`:

```
1: protected override void OnPaintSurface(SKPaintSurfaceEventArgs e) {
2:     _drawingManager?.Draw(e.Surface);
3: }
```

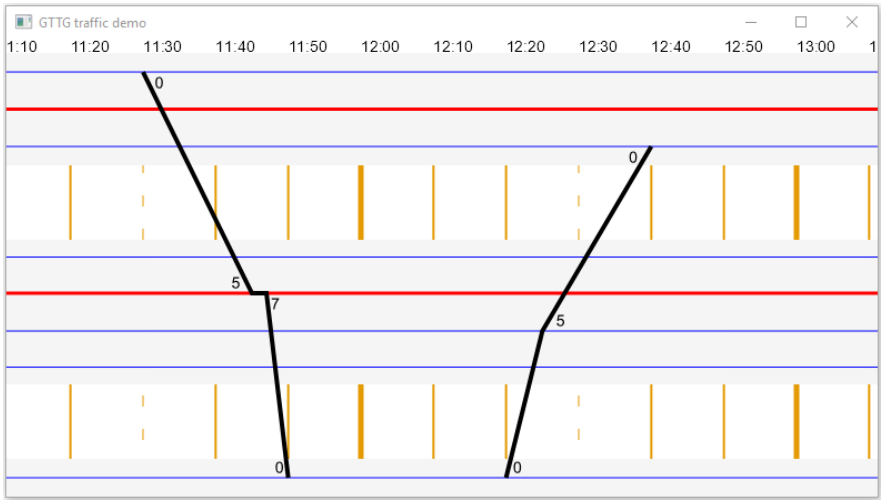
Integrace view modelu do aplikace

Jelikož je náš view model závislý na výšce zobrazení v komponentě, budeme volat `Arrange()` při každé změně velikosti komponenty. Proto přidáme do handleru `OnSizeChanged()` v `GraphicalComponentUserControl` i rozmístění view modelu. Protože na vrchu komponenty vykreslujeme pruh s časovými údaji, nechceme, aby se překrýval s obsahem `RailwayView`. Proto `RailwayView` posuneme o výšku pruhu a přidělíme mu i menší výšku. Všem view modelům infrastruktury předáváme délku `float.MaxValue`, jelikož v cyklu rozmístění pracujeme pouze s výškami.

```
1: _graphicalComponent?.TryResizeView(CanvasSize.Width, CanvasSize.Height);
2: _railwayView?.Measure(LayoutConstants.InfiniteSize);
3: /* Move railway view to avoid content being hidden by _timeAxisLayer atop */
4: var railwayY = CanvasSize.Height - _timeAxisLayer.Height;
5: var railwayPos = new SKPoint(0, _timeAxisLayer.Height);
6: _railwayView?.Arrange(railwayPos, new SKSize(float.MaxValue, railwayPos));
```

5.3 Tutoriál vykreslení průběhu jízdy vlaků a používání strategií

V předchozí části jsme vytvořili aplikaci, která vykresluje dopravní body s kolejemi. S vytvořeným modelem infrastruktury budeme dále pracovat. Na jeho základě vytvoříme view model vlaků a zobrazíme průběh jízdy vlaků doplněný informacemi umístěnými v ostrých úhlech jako kóty (1.1). Konečný výsledek tutoriálu se nachází na obrázku 5.4 a je dostupný jako solution v `/tutorials/traffic` v příloze práce nebo repozitáři [18].



Obrázek 5.4: Výsledek tutoriálu vykreslující průběh jízdy vlaků

Úprava view modelu infrastruktury pro práci se strategiemi

Nejdříve poupravíme stávající základní view model na verzi, která podporuje používání strategií umisťujících kóty do ostrých úhlů vedle šikmých čar představujících průběh jízdy vlaku.

Úprava view modelu kolejí

Začneme nejdříve úpravou třídy `TutorialTrackViewFactory`. Budeme vykreslovat šikmé čáry představující průběh jízdy vlaků, které jsou podle 4.3.2 tvořeny průsečíky s horizontálními čarami kolejí. Segment `MeasureableSegment`, představující horizontální čáru, umístíme v metodě `CreateTrackView()` do `SegmentRegistry` přidaném v konstruktory této třídy:

```
1: public TutorialTrackView CreateTrackView(Track track) {
2:
3:     var lineType = LineType.Of(track);
4:     var segment = new MeasureableSegment();
5:     _lineSegments.Register(segment).As(lineType);
6:     return new TutorialTrackView(track, CreateTrackLine(track), segment);
7: }
```

Ze `SegmentRegistry` pak view model vlaku získá horizontální čáry kolejí jako segmenty s jejich vertikální polohou. `SegmentRegistry` vytvoříme v třídě `GraphicalComponentUserControl` v metodě `OnLoaded()`, v které vytvořený `SegmentRegistry` předáme konstruktoru `TutorialTrackViewFactory`:

```
1:  private void OnLoaded() {
2:   /*...*/
3:   var segments = new SegmentRegistry<LineType, MeasureableSegment>();
4:   var trackViewFactory = new TutorialTrackViewFactory(segments);
5:   /*...*/
6:  }
```

V `TutorialTrackView` jsme základní implementaci `TrackView` doplnili o pruhy kolem horizontálních čar kolejí, které tyto čáry od sebe vizuálně oddělují. Přetížení metod `ArrangeOverride()` a `MeasureOverride()` pracujících s těmito pruhy můžeme odstranit, jelikož segmenty dalších view modelů, které si nyní uvedeme, původní pruhy nahradí.

Úprava view modelu stanice

Pruhy, které jsme z view modelu kolejí odstranili, nyní přidáme jako segmenty strategií, které budou rozmisťovat kóty. `TutorialStationView` bude potomkem `StrategyStationView`, která pro nás tyto segmenty vytvoří a registruje je do instance `SegmentRegistry`, kterou vytvoříme a předáme konstruktoru `TutorialStationViewFactory` podobně jako v předchozí části. V konstruktoru `TutorialStationView` přidáme eventu `HeightMeasureHelpers` vytvořených segmentů handler k měření výšky segmentu, který přednastaví minimální výšku segmentu a vytvoří tak mezery mezi horizontálními čarami:

```
1:  public TutorialStationView(/*...*/)
2:      : base(station, segmentRegistry, trackViewFactory) {
3:
4:      foreach (var track in TrackViews.Select(t => t.Track)) {
5:
6:          Segments
7:              .Resolve(new SegmentType<Track>(track, SegmentPlacement.Lower))
8:              .HeightMeasureHelpers += MeasureSegmentHeight;
9:
10:         Segments
11:             .Resolve(new SegmentType<Track>(track, SegmentPlacement.Upper))
12:             .HeightMeasureHelpers += MeasureSegmentHeight;
13:     }
14: }
```

View model trati

Nyní při konstrukci modelu v `OnLoaded()` vytvoříme místo `RailwayView` instanci `StrategyRailwayView`, které vedle původních parametrů `RailwayView` také předáme nově vytvořenou instanci `SegmentRegistry` pro strategie umisťující čísla vlaků podél šikmé čáry průběhu jízdy vlaků. `StrategyRailwayView` pak se segmenty pracuje obdobně jako `StrategyStationView`. Celé vytvoření view modelu infrastruktury vypadá následovně:

```
1:  private void OnLoaded() {
2:
3:   var lineSegments =
4:       new SegmentRegistry<LineType, MeasureableSegment>();
5:   var tracksSegments =
6:       new SegmentRegistry<SegmentType<Track>, MeasureableSegment>();
7:   var stationSegments =
8:       new SegmentRegistry<SegmentType<Station>, MeasureableSegment>();
9:
10:  var trackViewFactory =
11:       new TutorialTrackViewFactory(lineSegments);
12:  var stationViewFactory =
13:      new TutorialStationViewFactory(trackViewFactory, tracksSegments);
14:  var trainViewFactory =
15:      new TutorialTrainViewFactory(_graphicalComponent,
16:                                   lineSegments,
17:                                   tracksSegments,
18:                                   stationSegments);
19:
20:  _railwayView =
21:      new StrategyRailwayView<TutorialStationView, TutorialTrackView>
22:     (TrainTimetableData.Railway, stationViewFactory, stationSegments);
23:  _trafficView =
24:      new TrafficView<TutorialTrainView, Train>
25:     (TrainTimetableData.Traffic, trainViewFactory);
```

Vytvoření view modelu vlaků a vykreslení průběhu jízdy vlaků

Na základě upraveného view modelu infrastruktury nyní vytvoříme view model vlaků. K vytvoření modelu provozu na trati použijeme základní implementaci `Traffic`, která obsahuje kolekci vlaků `Train`. Pro jednoduchost tutoriálu vytvoříme dva vlaky, každý jedoucí v jiném směru a umístíme je do statické třídy `TrainTimetableData` s modelem dat v aplikaci. Pořadí stanic určuje směr jízdy vlaku. Na následujícím fragmentu kódu tento seznam vlaků vytvoříme:

```
1:  public static Traffic<Train> Traffic { get; } =
2:    new Traffic<Train>(
3:      new List<Train> {
4:
5:         new Train(
6:          new List<TrainEvent> {
7:              new TrainEvent(Start.AddMinutes(90),
8:                             Railway.Stations[0],
9:                             Railway.Stations[0].Tracks[0],
10:                            TrainEventType.Departure),
11:             new TrainEvent(     /*...*/
```

Nyní vytvoříme implementaci factory rozhraní a view model vlaku. Jako view model vlaku vytvoříme `TutorialTrainView` jako potomka `StrategyTrainView`. V konstruktoru bude přijímat stejné parametry jako jeho předek a nebudeme ho zatím nijak modifikovat:

```
public class TutorialTrainView
    : StrategyTrainView<Strategy, Train> {

    public TutorialTrainView(Train train,
                             ITrainPath trainPath,
                             Strategy strategy)
            : base(train, trainPath, strategy) {
    }
}
```

Konstruktoru factory třídy `TutorialTrainViewFactory` dodáme všechny tři vytvořené `SegmentRegistry` a v metodě `CreateTrainView()` je použijeme k vytvoření `Strategy` (4.3.2) a `TrainPath` obsahující `TrackLine`, v které nastavíme tloušťku čáry představující průběh jízdy vlaku na 4 pixely a obarvíme ji černě. Vytvořené instance pak předáme konstruktoru `TutorialTrainView`. Celá implementace metody se nachází na následujícím fragmentu kódu:

```
public TutorialTrainView CreateTrainView(Train train) {

  var trackLine = new TrackLine(4, SKColors.Black);
  var trainPath =
    new TrainPath(_viewProvider, _linesSegmentRegistry, trackLine);
  var strategy =
    new Strategy(trainPath, _trackSegmentRegistry, _stationSegmentRegistry);
  return new TutorialTrainView(train, trainPath, strategy);
}
```

V metodě `OnLoaded()` inicializujeme view model `TrafficView` vykreslující a spravující všechny vlaky v třídě `Traffic`, kterou mu v konstruktoru dodáme spolu s `TutorialTrainViewFactory`:

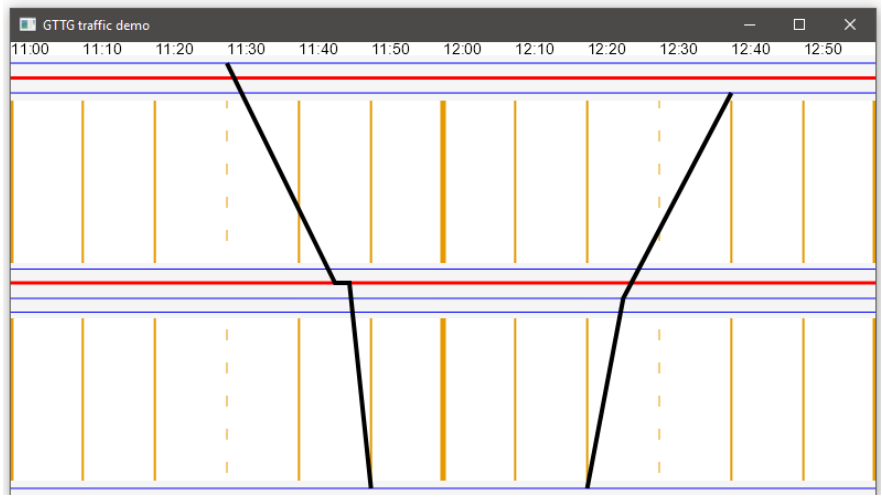
```
private void OnLoaded() {
  /*...*/
  var trainViewFactory =
  new TutorialTrainViewFactory(_graphicalComponent,
                               lineSegments,
                               tracksSegments,
                               stationSegments);
  _trafficView =
  new TrafficView<TutorialTrainView, Train>(TrainTimetableData.Traffic,
                                            trainViewFactory);
}
```

Zároveň vytvoříme novou vrstvu, v které `TrafficView` vykreslíme. Upravíme implementaci `IRegisteredLayersOrder`, kde vrstvu umístíme do popředí. Musíme upravit i `OnSizeChanged()` handler, aby se provedl cyklus rozmístění na `TrafficView`, ale až po rozmístění infrastruktury, jelikož na základě jejího rozmístění umisťuje své vlastní prvky:

```
private void OnSizeChanged() {

    /* ... */
    _trafficView?.Arrange();
}
```

Pokud nyní spustíme aplikaci, zobrazí se vykreslené šikmé čáry odpovídající průběhu jízdy vlaků, jako na obrázku 5.5. V další části doplníme nákresný jízdní řád o informace jako jsou kóty.



Obrázek 5.5: Výsledek tutoriálu vykreslující průběh jízdy vlaků bez kót

Přidání kót do nákresného jízdního řádu

V této části doplníme obsah nákresného jízdního řádu a view model vlaku o kóty, které vytvoříme jako zobrazitelné prvky `ViewElement` umístitelné do nákresného jízdního řádu strategiemi.

Vytvoření zobrazitelného prvku kót

Pro vykreslování kót vytvoříme třídu `TimeComponent`, která dědí od základního zobrazitelného prvku `ViewElement`. V kótách jsou podle 1.1 vykreslovány jednotky minut. Kóty implementujeme tak, že `TimeComponent` získá v konstruktoru `TrainEvent`, z nějž se zobrazí jednotky minut vlastnosti `DateTime`. Pro správnou implementaci vykreslení kóty v ostrém úhlu mezi šikmou čáru průběhu jízdy vlaku a horizontální čáru koleje musíme určit výšku a šířku `TimeComponent`, s kterou poté pracuje podle 2.2 strategie.

Šířku a výšku prvku určíme v `MeasureOverride()` změřením vykreslovaného textu. Vlastnosti textu určuje třída `SKPaint`, která se vykreslení textu předává. Nastavením vlastností `TextSize` a `TypeFace` na `SKPaint` měníme velikost textu. Délku textu zjistíme pomocí metody `MeasureText()` na `SKPaint`. Výšku textu určíme odhadem pomocí `CapHeight` uvedeném v předchozím tutoriálu u vykreslování vrstvy `TimeAxisLayer`. Text vykreslíme následujícím způsobem v metodě `OnDraw()`, kde `TextPathY` odpovídá výšce řádku, na kterém je text vykreslen:

```
1: protected override void OnDraw(DrawingCanvas drawingCanvas) {
2:
3:   TextPath = TimePaint.GetTextPath(text: TextString, x:0, y:TextPathY);
4:   drawingCanvas.Canvas.DrawPath(TextPath, TimePaint);
5: }
```

Nyní tento prvek představující kótu můžeme přidat do strategií.

Přidání kót do strategií

Kóty do strategií přidáme v implementaci view modelu vlaku `TutorialTrainView` v přetížené metodě `UpdateTrainViewContent()`:

```
1:  public override void UpdateTrainViewContent() {
2:
3:    base.UpdateTrainViewContent();
4:    foreach (var trainEvent in Train.Schedule) {
5:
6:      var movementEventType =
7:      new TrainEventPlacement(trainEvent, AnglePlacement.Acute);
8:
9:      Strategy.TrackStrategyManager
10:     .Add(movementEventType, new TimeComponent(trainEvent));
11:   }
12: }
```

Takto jsme určili, že ke každému `TrainEvent` přidáme `TimeComponent`, kterou strategie umístí správně do ostrého úhlu pomocí `AnglePlacement.Acute`.

Na následujícím fragmentu kódu se nachází série kroků nutná pro to, aby se správně vykreslil obsah s kótami už při inicializaci aplikace. Musíme upravit cyklus rozmístění v `OnLoaded()` ve třídě `GraphicalComponentUserControl`. Nejdříve provedeme rozmístění infrastruktury a vlaků bez prvků jako jsou kóty v `OnSizeChanged()`. Poté v `trafficView.Update()`, která volá `UpdateTrainViewContent()`, přidáme kóty, které již dokážeme na základě rozmístění infrastruktury správně umístit. Rozmístění nově přidaných kót v `Arrange()` se provede voláním `OnSizeChanged()`. Na konci zavoláme `InvalidateVisual()`, který provede vykreslení.

```
1:  private void OnLoaded() {
2:      /* ... */
3:
4:      OnSizeChanged();
5:      _trafficView.Update();
6:      OnSizeChanged();
7:      InvalidateVisual();
8:  }
```

Takto jsme získali aplikaci, která zobrazuje nákresný jízdní řád námi vytvořeného modelu. V dalších částech si uvedeme, jak jako vývojář můžeme implementovat nebo používat části knihovny GTTG.

Používání poskytnutých nástrojů knihovny

Tato podkapitola popisuje možnosti, jak knihovnu používat a vytvářet strategie, které je možné propojit s existujícími nástroji knihovny. Dále se budeme věnovat přidání dodatečných částí nákresného jízdního řádu, které nejsou přímo jeho obsahem. Nejdříve navážeme na cyklus rozmístění, kterým jsme zakončili poslední tutoriál předchozí kapitoly.

Cyklus rozmístění pro práci se základním modelem

Cyklus rozmístění musí vývojář použít v případě, kdy dochází ke změně velikosti plátna, na který je obsah vykreslován. V tomto případě probíhá cyklus rozmístění na celém obsahu. V implementaci základního modelu, kterou jsme doposud v tutoriálech používali a je dostupná v projektu `GTTG.Model`, musí být cyklus rozmístění implementován jako sekvence několika kroků:

1. Nejdříve se provede cyklus rozmístění na view modelu infrastruktury. V této části se rozmístí horizontální čáry představující dopravní body a segmenty, které se používají strategiemi. Pokud pracujeme se segmenty, které jsou typu `MeasureableSegment`, můžeme do nich podle 4.1.5 umístit handlery, které určí požadovanou velikost prvku v segmentu. Na základě této velikosti pak může prvek infrastruktury mít jinou výšku v `DesiredSize`. V té prvky infrastruktury nastavují jako požadovanou délku `float.MaxValue`.

2. V druhé části se rozmístí view modely vlaků. Podle vertikálního rozmístění prvků infrastruktury a segmentů se vykreslí šikmé čáry představující průběh jízdy vlaku. S již rozmístěnou infrastrukturou a těmito čarami je možné aplikovat různé strategie, které podle konečné velikosti a různých ohraničení prvky vizuálně transformují a rozmisťují.

Vedle cyklu rozmístění, který probíhá na celém obsahu nákresného jízdního řádu, dochází při běhu aplikace k úpravám, které ovlivňují rozmístění pouze nějaké podmnožiny obsahu. Příkladem je změna plánu jízdy vlaku. Vývojář nemusí provést cyklus rozmístění na celém nákresném jízdním řádu a pouze zavolá metodu `Arrange()` na `TrainView` (případně před ní metodu `Update()`, podle 4.3.2). Pokud nastane situace, kdy by například kóta jako jeden prvek chtěla změnit svou velikost, musí vývojář sám nalézt a znovu rozmístit nejmenší část, do které kóta patří. Předpokládá se, že k takovým změnám bude docházet zároveň na více prvcích, když se například vybírá vlak a všechny jeho kóty se mají upravit. V takovém případě stačí zavolat z aplikační logiky `Arrange()` na view modelu vlaku a obsah znovu vykreslit.

Vykreslování informací kolem nákresného jízdního řádu

V 1.1 jsme si uvedli, že se kolem samotného obsahu nákresného jízdního řádu, který jsme doposud vykreslovali, nachází i sloupce s dalšími informacemi – kilometrická poloha, informace o zabezpečení nebo názvy dopravních bodů. Pro tyto sloupce se vytvoří vlastní prvek uživatelského rozhraní, který bude umístěný vedle UI prvku komponenty s nákresným jízdním řádem a bude vykreslovat obsah na

své vlastní plátno `SKCanvas`. Aby jeho obsah odpovídal zrovna zobrazovanému výřezu celého nákresného jízdního řádu, získáme `IViewProvider` a provedeme úpravu matice `ContentCanvas`, kterou pak nastavíme plátnu prvku, jako na následujícím fragmentu kódu. Seznam stanic je pak možné kreslit na toto plátno podle rozmístění dopravních bodů ve view modelu.

```
1: public void Draw(SKSurface skSurface) {
2:
3:     var viewMatrix = _viewProvider.ContentMatrix;
4:     viewMatrix.TransX = 0;
5:     skSurface.Canvas.SetMatrix(viewMatrix);
```

Obdobně můžeme vykreslení horizontální časové osy přesunout do vlastního UI prvku, kterému také musíme přenastavit matici `ContentCanvas`:

```
1: public void Draw(SKSurface surface) {
2:
3:     var viewMatrix = _viewProvider.ContentMatrix;
4:     var timeSidebarMatrix = SKMatrix.MakeIdentity();
5:      timeSidebarMatrix.TransX = viewMatrix.TransX;
6:      surface.Canvas.SetMatrix(timeSidebarMatrix);
```

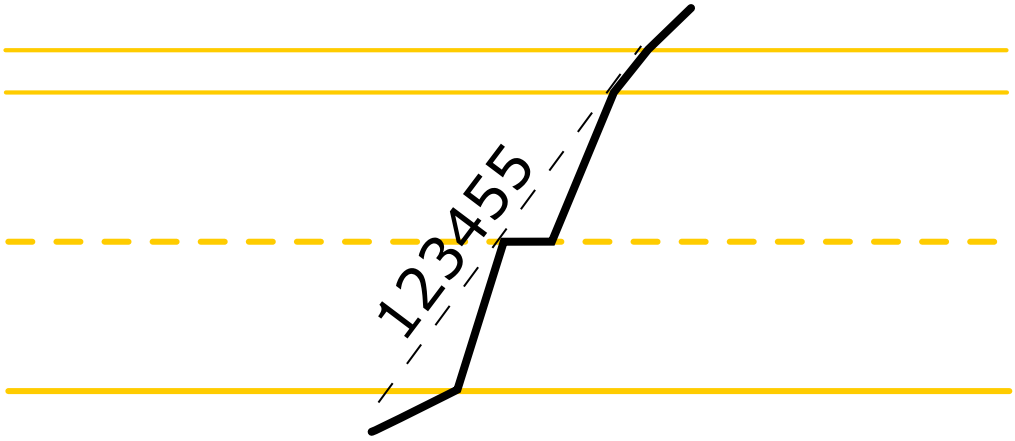
V obou těchto modifikacích matice tak pouze vynulujeme translaci ve směru, s kterým zobrazení nepracuje (například posun po horizontále pro sloupec stanic). Pokud nechceme, aby vykreslovaný text byl při modifikaci přiblížení zvětšený, použijeme postup z 5.2, kdy použitím vlastnosti `Scale` přenásobíme velikost textu.

Implementace vlastních strategií

Nyní si představíme, jak je možné implementovat jednu ze složitějších strategií vedle těch, které poskytuje `GTTG.Model`. Jednou ze strategií, o které jsme se v 2.2 zmínili, je strategie umisťující čísla vlaků na šikmou čáru aproximující průběh jízdy vlaku mezi více dopravními body, jako na obrázku 5.6. Tuto strategii je vhodné použít v případě, kdy trať obsahuje velké množství dopravních bodů, čili mezi nimi není možné najít dostatečně vysoký úsek pro umístění čísla vlaku. Ukažme si, jak je možné tuto strategii vytvořit využitím nástrojů knihovny GTTG.

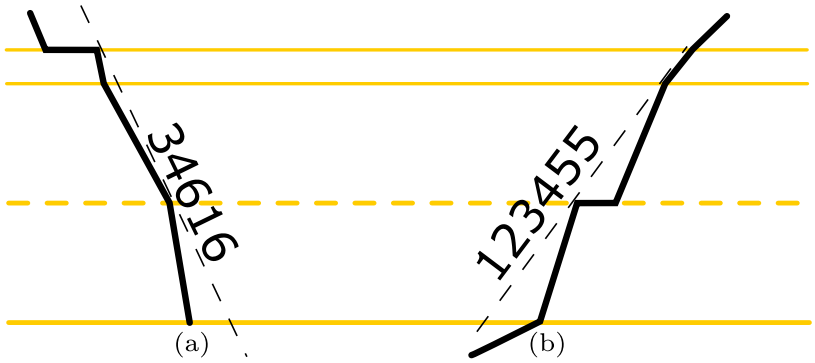
Předpokládejme, že vedle této strategie budeme vytvářet segmenty představující horizontální pruhy kolem dopravních bodů, do kterých nějaká další strategie bude umisťovat kóty. Implementace naší strategie se bude snažit předejít situaci, kdy by se číslo vlaku překrývalo s kótou z těchto segmentů.

Pro přidávání zobrazitelných prvků do strategie nám bude stačit `BasicStrategyManager` pracující s `TPlacementType` a `TSegmentType`. V naší implementaci totiž nemusíme přidávat vlastní segmenty pro umístění čísel vlaku do třídy `SegmentRegistry`, která je součástí `StrategyManager`. Bude nám pouze stačit, když uvedeme v `TPlacementType` dvě instance `Track`, mezi jejichž vizualizaci se umístí číslo vlaku. `TPlacementType` se pak převede pomocí bodů `TrainPath` na `TSegmentType`, který bude jako typ obsahovat dva segmenty pro umístění kót kolem instancí dodaných kolejí. V ohraničení segmentů se vytvoří aproximovaná čára průběhu jízdy vlaku.



Obrázek 5.6: Strategie umisťující čísla vlaků na aproximovanou čáru

Nalezení aproximované čáry i správného umístění čísla vlaku přenecháme implementaci `IStrategyDocker`, jejíž instanci vytvoříme pro každý vlak zvlášť. Konstruktoru dockeru předáme `TrainPath` obsahující body vytvářející průběh jízdy vlaku. Pomocí ní spolu se segmenty je docker schopný určit směr jízdy vlaku a zároveň tak i aproximovanou čáru s umístěním čísla vlaku. Nyní bychom pro každé ze dvou možných umístění, nacházejících se na obrázku 5.7, vytvořili metodu k umístění prvku. Přesnou implementaci metody pro situaci (a) si nyní detailně popíšeme.

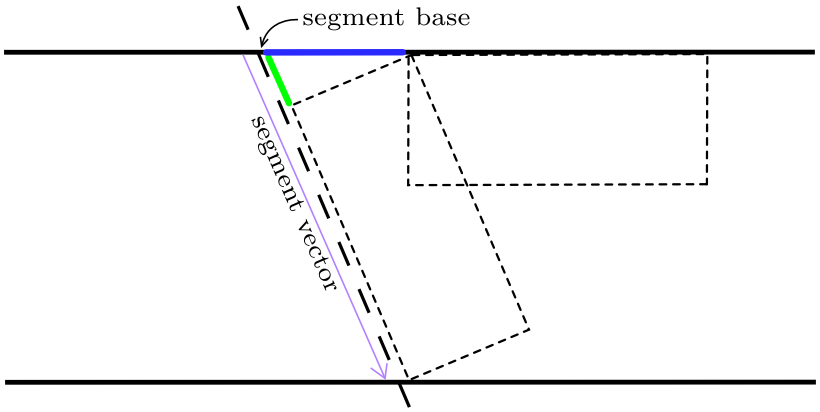


Obrázek 5.7: Různé možnosti vytvoření aproximované čáry s umístěním čísla vlaku

Strategii implementujeme tak, že se číslo vlaku umístí na prostředek aproximující čáry v ohraničeném úseku segmenty pro kóty. Budeme chtít dosáhnout stavu na obrázku 5.8, kdy horní levý vrchol prvku není umístěný na aproximující čáře, ale je přemístěný tak, aby se po rotaci nacházela jeho spodní hrana na aproximující čáře.

Transformace velikosti prvku

Budeme očekávat, že získáme dva body na aproximující čáře, které se budou zároveň nacházet na ohraničení segmenty pro kóty. Rozdílem těchto dvou bodů získáme vektor `segment vector` ve směru jízdy vlaku. Jeden z těchto dvou bodů, který se nachází jako první ve směru jízdy vlaku, nazveme `segment base`.



Obrázek 5.8: Konečná pozice prvku určená strategií

Nejdříve změříme `MeasureOverride()` velikost prvku `DesiredSize`, kterou mu přiřadíme v `Arrange()`. V případě, že délka prvku přesahuje délku `segment vector`, zapamatujeme si poměr rozdílu velikostí do proměnné `scale`. Dále by hrozilo, že prvek výškou bude i po vynásobení `scale` zasahovat mimo ohraničenou oblast. Změříme proto délku odvěsny, vyznačenou zeleně na obrázku 5.8, s novou výškou prvku přenásobenou `scale`. Pro měření a výpočty existují statické metody v třídě `PlacementUtils`. Délka odvěsny by se vypočítala metodou `ComputeLegLength()`. Pokud součet délky odvěsny a délky prvku bude přesahovat délku `segment vector`, jejich poměr vynásobíme s původní hodnotou `scale`. Na prvku zavoláme `Scale()` s nově vynásobenou hodnotou `scale`. Nyní pracujeme případně se zmenšeným prvkem, který se vejde do vyhrazeného místa.

Umístění a transformace rotací

Nyní provedeme samotné umístění – aplikujeme přesun prvku s rotací. Nejdříve vypočítáme přesun prvku pomocí délky zelené odvěsny a modré přepony z obrázku 5.8, pro škálovanou velikost prvku, která odpovídá vlastnostem prvku `ContentWidth` a `ContentHeight`. Jako výchozí bod pro umístění použijeme `segment base`. Na horizontální ose k němu přičteme součet délky modré přepony a polovinu tloušťky čáry `LinePaint`, aby se s ní číslo vlaku nepřekrývalo. Vertikální umístění bodu vypočítáme posunem od `segment base` po `segment vector` metodou `MoveInLine()` udáním délky posunu. Pokud je délka menší, do posunu započítáme polovinu volného prostoru, aby se prvek umístil na prostředek čáry. Takto jsme získali umístění, na které prvek přemístíme pomocí `Reposition()` metody. Protože se rotace aplikuje po směru hodinových ručiček, předáme `Rotate()` metodě ostrý úhel, který svírá `segment vector` a horizontální čára.

Přesouvání obsahu mezi vrstvami a jejich hit-testování

Nyní si na příkladu výběru vlaku k editaci ukážeme, jak implementovat přesun obsahu nákresného jízdního řádu do jiné vrstvy a jak s vrstvami pracovat. Předpokládejme, že máme view model `TrafficView`, který obsahuje seznam všech vlaků rozšířeného modelu se strategiemi, v rámci kterého se umisťují vedle šikmé čáry průběhu jízdy vlaku kóty. Pokud například uživatel klikne na kótu, pomocí

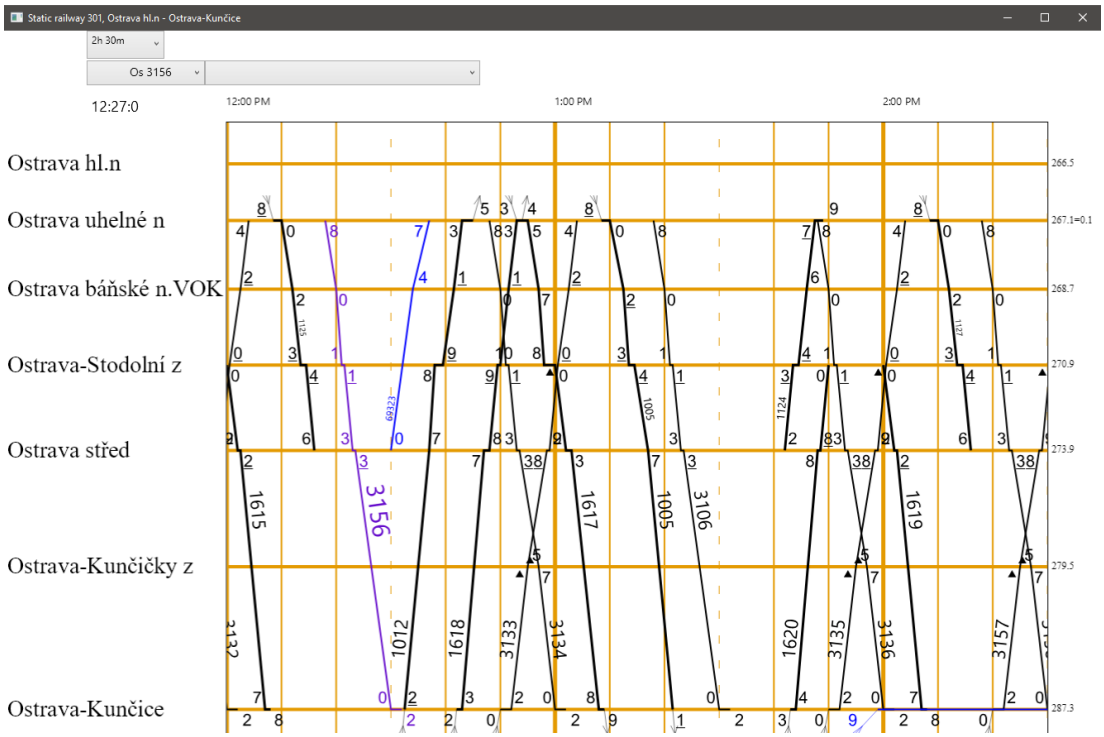
data bindingu zmíněného v 4.1.3 se zobrazí v nějakém uživatelském prvku doplňující textové informace, které jsou kótou vizualizovány. Pokud se však rozhodne uživatel vybrat vlak k editaci, přesune se jeho čára průběhu jízdy vlaku i kóty spojené s vlakem do popředí. Při editaci vlaku by pak bylo možné při kliknutí na kótu editovat informace spojené s kótou z jiného prvku uživatelského rozhraní.

Nejdříve implementujme přesunutí editovaného vlaku do vrstvy, která bude umístěna v popředí. Vrstvu umístíme pomocí `AddOnCurrentTop()` do instance `DrawingManager`. Ve vrstvě bude metoda `ProvideVisuals()` poskytovat editovaný view model vlaku. Tomu umístíme na vrch zásobníku pomocí `PushDrawingLayer()` tuto vrstvu. Rekurzivně se tato metoda ve view modelu vlaku zavolá na každý prvek z `ProvideVisualsInSameLayer()`. Celý strom těchto prvků je pak umístěn v nové vrstvě. Pro navrácení těchto prvků do původní vrstvy použijeme na view modelu vlaku opačně `PopDrawingLayer()`.

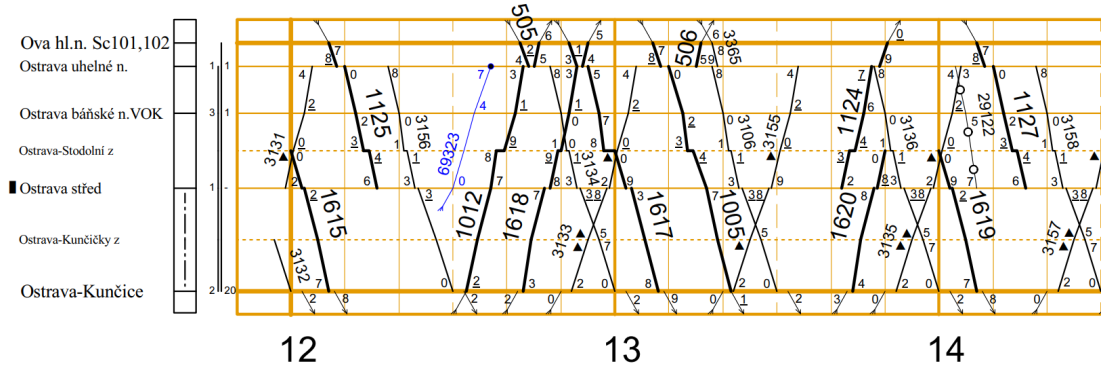
Dále si popíšeme, jak implementovat klikání na zobrazitelné prvky. Předpokládejme, že budeme chtít z úspěšně otestovaných prvků v hit-testu získat pouze instance třídy `TimeComponent` představující kóty. Textové informace kóty vykreslené na vrchu chceme ve vedlejším prvku uživatelského rozhraní zobrazovat a případně upravovat. Na instanci třídy `HitTestManager` se zavolá metoda `HitTest()` vůči vybranému bodu na prvek `TrafficView`, která projde podstrom prvků tohoto view modelu ve všech vrstvách. Získanou sekvenci úspěšně otestovaných prvků typu `TimeComponent` uložíme v seznamu v metodě delegáta `HitTestResultCallback`, který pak následně seřadíme podle vrstev v `OrderByLayers()` a vybereme poslední prvek z první vrstvy v pořadí. Ten se vykreslí jako poslední a jedná se o kótu nacházející se na vrchu.

6. Aplikace SZDC

Abychom předvedli využití knihovny GTTG, chtěli jsme vybrat existující implementaci nákresných jízdních řádů, na které bychom ukázali, že je naše knihovna dostatečně konfigurovatelná tak, aby umožnila vytvoření nákresného jízdního řádu podle skutečných a přesně definovaných požadavků. Proto jsme se rozhodli vytvořit aplikaci pro práci s nakresnými jízdními řády (cíl **G2**), které jsou vydávány Správou železniční a dopravní cesty (SŽDC) a jsou veřejně přístupné [23] spolu s dalšími pomůckami grafikonu vlakové dopravy nebo se nachází k nahlédnutí v příloze práce v `/szdc/njr/pdf`. Popíšeme si cestu, jak jsme získali data, která budou při zobrazení v aplikaci (obrázek 6.1) co nejvěrněji odpovídat vzoru zmíněných nákresných jízdních řádů (obrázek 6.2). Nejdříve si ale popíšeme architekturu aplikace a dva způsoby zobrazení, které aplikace nabízí.



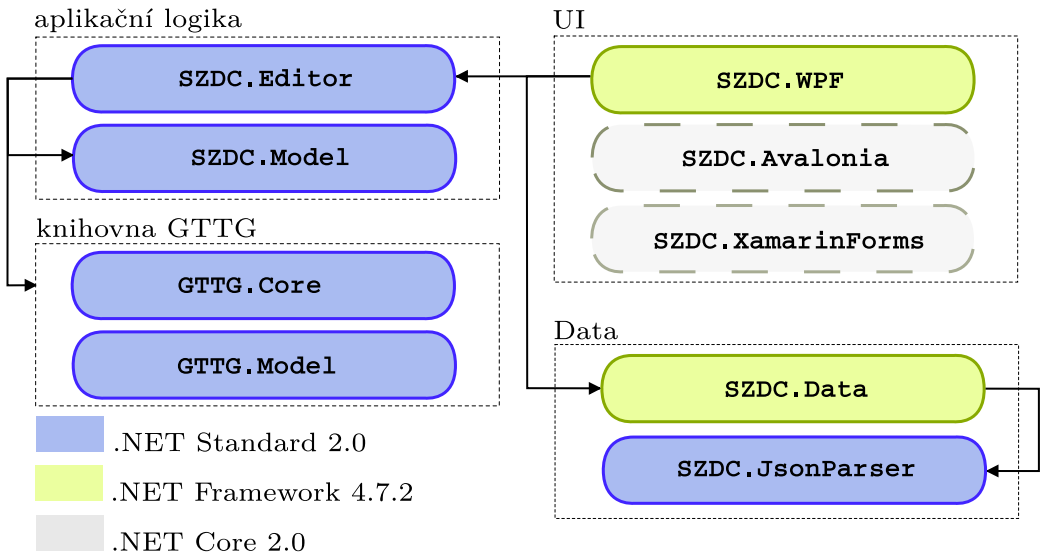
Obrázek 6.1: Zobrazení nákresného jízdního řádu v aplikaci



Obrázek 6.2: Nákresný jízdní řád vydávaný SŽDC

6.1 Architektura aplikace SZDC

Aplikaci jsme navrhli tak, aby byla znovupoužitelná v rámci více GUI frameworků platformy .NET (cíl **G2c**)). Možnost tohoto návrhu jsme si ověřili na existující a dostatečně složité aplikaci Core2D [13], která tento požadavek splňuje. Aplikaci SZDC jsme rozdělili na projekty, uvedené na obrázku 6.3, podobným způsobem jako ve zmíněné aplikaci. V rámci co největší přenositelnosti jsou projekty aplikační logiky a jejího modelu (`SZDC.Editor` a `SZDC.Model`) implementovány vůči rozhraní .NET Standard. Model aplikace je rozšířením modelu z `GTTG.Model`. Jako zdroj dat používá aplikace připojení k databázi, které je spravováno v projektu `SZDC.Data`, je ale možné pracovat s daty i z jiných zdrojů. V rámci práce s daty existují pomocné nástroje jako `SZDC.JsonParser`. Jako GUI framework aplikace jsme zvolili WPF, určené pro běhové prostředí .NET Framework 4.7.2. Vedle WPF by bylo možné aplikaci použít i v GUI frameworku Avalonia [24] nebo Xamarin.Forms. V následujících podkapitolách si všechny tyto projekty více popíšeme.



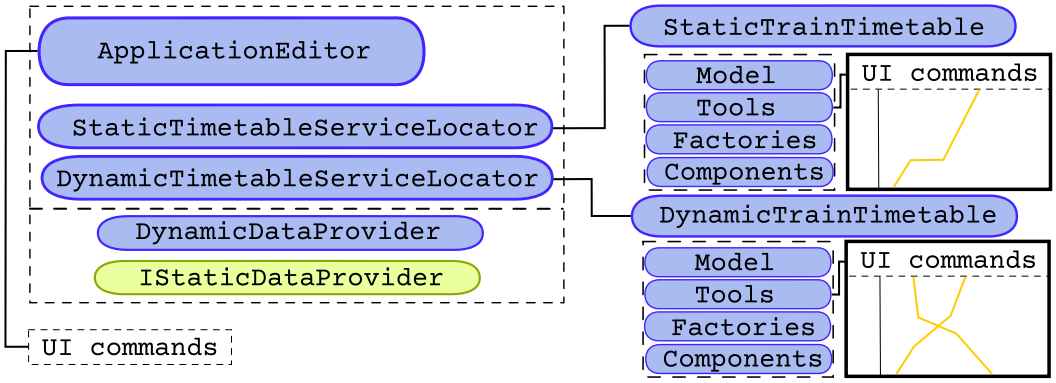
Obrázek 6.3: Rozdělení aplikace SZDC do projektů

SZDC.Editor

Aplikace umožní uživateli zobrazit více oken s nákresnými jízdními řády zároveň, každé umožňující práci s nákresným jízdním řádem v jednom ze dvou módů, které jsme podle **G2** chtěli vytvořit:

1. Mód k prohlížení nákresných jízdních řádů SŽDC, který nazveme jako *statický* (cíl **G2a**))
2. Mód simulující provoz na trati, který označíme jako *dynamický* – komponenta s nákresným jízdním řádem se periodicky obnovuje, aby odpovídala současnému časovému intervalu několika hodin. Uživatel může modifikovat průběh jízdy vlaku a tyto změny jsou viditelné mezi více okny v tomto módu (cíl **G2b**)).

Z pohledu uživatele si tyto módy více popíšeme v uživatelské dokumentaci 6.3. Pro rozdělení aplikační logiky, nacházející se na obrázku 6.4, je existence těchto dvou módů důležitá – každý zobrazuje podobný obsah, ale nabízí různou úroveň jeho modifikace.



Obrázek 6.4: Rozdělení aplikační logiky

Základem aplikační logiky je třída `ApplicationEditor`. Pokud například z uživatelského rozhraní klikneme na tlačítko k otevření nákresného jízdního řádu ve statickém módu, `ApplicationEditor` vytvoří, nastaví a vrátí instanci `StaticTrainTimetable`, která je základní třídou nákresného jízdního řádu ve statickém módu. Třída `StaticTrainTimetable` obsahuje tyto části:

1. Nástroje knihovny GTTG jako `GraphicalComponent` nebo `DrawingManager`
2. Model a view model nákresného jízdního řádu rozšiřující `GTTG.Model`
3. Implementace factory rozhraní view modelů z 4.3.2 v `FactoriesCollector`
4. Nástroje ve třídě `ToolsCollector` určené pro práci s nákresným jízdním řádem, obsahující například:
   (a) `CurrentDateTimeTool` převádějící pozici kurzoru myši na časový údaj
   (b) `TrainViewSelectionTool` zajišťující výběr zvýrazňovaného vlaku vyneseného do popředí
   (c) `ViewTimeModifierTool` umožňující nastavení časových intervalů komponenty
5. Komponenty uživatelského rozhraní ve třídě `ComponentsCollector`:
   (a) `TrainTimetableComponent` představuje komponentu nákresného jízdního řádu
   (b) `TimeSidebarComponent` představuje horizontální osu časových údajů nad nákresným jízdním řádem
   (c) `RailwayDistanceSidebarComponent` je sloupec kilometrických poloh dopravních bodů na trati
   (d) `StationNamesComponent` představuje sloupec názvů dopravních bodů

Inicializace `StaticTrainTimetable` a jeho částí probíhá přes *dependency injection* framework *Autofac* [25]. Nemusíme tak složitě inicializovat třídy používané v `StaticTrainTimetable` sekvencí konstruktorů, ale inicializace proběhne mimo náš kód. Ukažme si, jak inicializace probíhá. `StaticTrainTimetable` přímo využívá `GraphicalComponent` k modifikacím a existuje mnoho dalších částí třídy, například `HitTestManager` nebo `CurrentTimeTool`, které chtějí získat instanci `GraphicalComponent` jako implementované rozhraní `IViewProvider`.
`ApplicationEditor` obsahuje `StaticTimetableServiceLocator`, spravující registraci nástrojů, které se v `StaticTrainTimetable` inicializují. Přidáme do něj `GraphicalComponent`:

```
1:  public class CoreModule : Module {
2:
3:    protected override void Load(ContainerBuilder builder) {
4:
5:     builder
6:       .RegisterType<GraphicalComponent>()
7:       .AsSelf()
8:       .AsImplementedInterfaces()
9:       .InstancePerLifetimeScope();
10:
11:  /*...*/
```

`GraphicalComponent` se zaregistruje jako její typ a všechna rozhraní, která implementuje, tedy i `IViewProvider`. Instanci takto registrované třídy je pak možné po sestavení `Build()` získat:

```
1:  _lifetimeScope = builder.Build();
2:  var viewProvider = _lifetimeScope.Resolve<IViewProvider>()
```

Framework vybere bezparametrický konstruktor. Pokud by existoval pouze konstruktor `GraphicalComponent` s parametry, framework tranzitivně vyhledá registrované třídy doplnitelné do parametrů a inicializuje je. Pro každou instanci `StaticTrainTimetable` bychom chtěli jednu instanci `GraphicalComponent`. Abychom pomocí `Resolve()` získali stejnou instanci, nastavíme vytváření ve *scope* pomocí `InstancePerLifetimeScope()` a při vytváření editoru `StaticTrainTimetable` v `StaticTimetableServiceLocator` vytvoříme nový scope pomocí `GetScopedServiceLocator()`. Ve scope se pomocí `Resolve()` získá jedna a ta samá instance takto registrované třídy. Dále je možné nastavit, že se při každém `Resolve()` vytvoří nová instance nebo se bude poskytovat pouze jedna po celý běh aplikace.

Editor pro dynamický mód `DynamicTrainTimetable` je inicializován stejným způsobem. Od `StaticTrainTimetable` se liší v implementaci aplikační logiky a nástrojích, které používá. Editory obou těchto módů jsou potomkem `TrainTimetable`, která implementuje základní aplikační logiku pro oba dva módy. V potomcích se implementace doplňuje o další metody aplikační logiky a navíc je povolená jen určitá podmnožina nástrojů `ToolsCollector`. Nástroje je totiž možné povolit i zakázat, aby například nezískaly vstup z myši a nevytvářely modifikace.

Propojení aplikační logiky a UI

Základem propojení aplikační logiky a datového modelu v `GTTG.Editor` je *data binding* pomocí rozhraní `INotifyPropertyChanged`, zmíněného v 4.1.3. Uživatelská rozhraní získávají data binding na model dat pomocí nastavení *data context*. Třídu `StaticTrainTimetable` jako kořen stromu editoru, jehož součástí jsou různé nástroje v `ToolsCollector` nebo komponenty v `ComponentsCollector`, přiřadíme jako data context do okna nákresného jízdního řádu. Ostatní prvky uživatelského rozhraní v okně pak s částmi editoru přes data context pracují. Například prvek uživatelského rozhraní pracující s `TrainViewSelectionTool` nabízí textový seznam všech vlaků. Pomocí `INotifyPropertyChanged` vytváří data binding na seznam všech vlaků ve `TrafficView`. Při změně instance seznamu se pak v prvku uživatelského rozhraní zobrazí nový seznam. V případě, že uživatel vybere z tohoto seznamu vlak, přiřadí se vlastnosti `SelectedTrainView` nově vybraný vlak a aplikační logika pomocí `INotifyPropertyChanged` zaznamená tuto změnu a upraví nákresný jízdní řád.

Uživatelské rozhraní aplikace SZDC

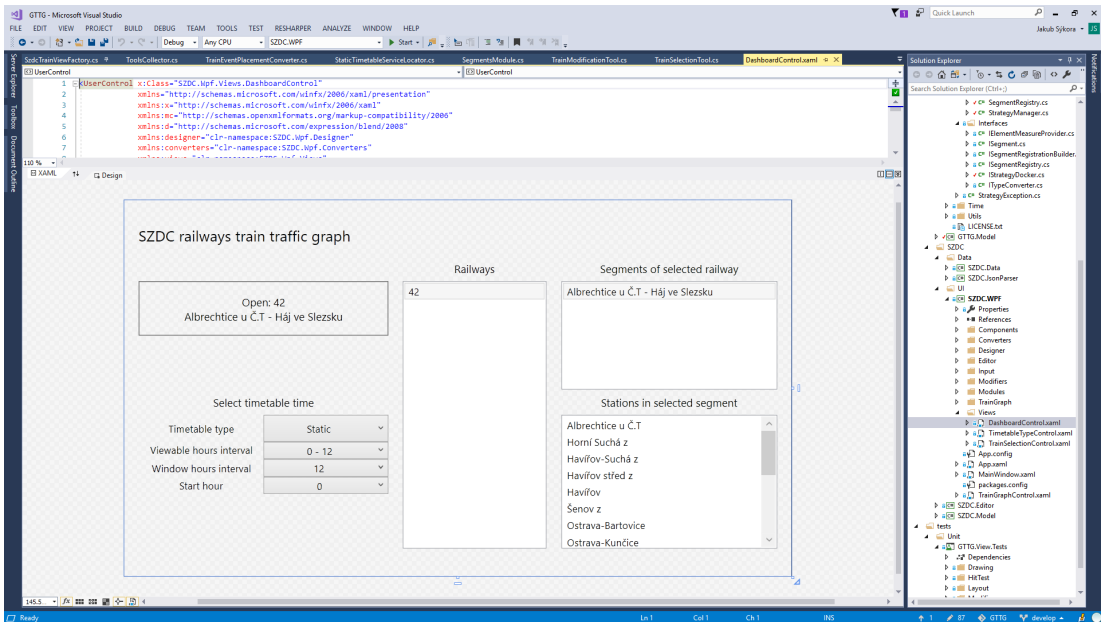
Jako GUI framework aplikace SZDC jsme vybrali WPF. Při vývoji aplikace pouze pro platformu Windows představuje nejvhodnější řešení, jako nástroj podporovaný přímo Microsoftem, poskytujícím k práci s WPF mnoho návodů. Vedle WPF by se nabízelo použít nový GUI framework Avalonia [24], který je narozdíl od WPF multiplatformní. V době, kdy jsme aplikaci vytvářeli, však nebyl ještě příliš zdokumentovaný a proto jsme se raději rozhodli použít ověřené WPF s větší podporou. Avalonia je nyní v beta verzi, ale používají ji již existují složitější aplikace jako Core2D [13] a další [26]. Vedle těchto dvou frameworků se v budoucnu nabízí použít pro vývoj aplikací Xamarin.Forms, primárně určený pro mobilní zařízení, který je nyní v *preview* verzi pro Linux, Windows i MacOS [27], vývoj UI by však pro každou platformu musel probíhat narozdíl od multiplatformní Avalonie v jiných projektech.

Výběr běhového prostředí pro WPF

V květnu roku 2018 byla ohlášena podpora běhového prostředí .NET Core 3 pro GUI framework WPF, stále však pouze v rámci platformy Windows – není plánováno z WPF vytvořit multiplatformní GUI framework. V době, kdy jsme začali s vývojem aplikace, bylo pro WPF dostupné pouze běhové prostředí .NET Framework. Pro různé prvky uživatelského rozhraní jsme vytvořili statické třídy s design daty pro *designer*, který zobrazuje náhled na vzhled prvků uživatelského rozhraní ve vývojových prostředích jako je Visual Studio nebo Rider [28], jako na obrázku 6.5.

V rámci posledních několika měsíců vývoje jsme už však měli možnost vyzkoušet .NET Core 3 a aplikaci jednoduše *přemigrovat* na toto běhové prostředí změnou jeho *.csproj* souboru. Zatím je však .NET Core 3 stále v preview verzi, kterou by si uživatel aplikace musel stáhnout a instalovat. Navíc designer pro Visual Studio 2017 nepodporuje .NET Core 3. Rozhodli jsme se, že první verze aplikace bude dostupná pro běhové prostředí .NET Framework a až bude nově

vydané Visual Studio 2019 více podporované s *release* verzí .NET Core 3, přemigrujeme další verzi aplikace na toto běhové prostředí.



Obrázek 6.5: Designer pro WPF ve Visual Studio 2017

6.2 Data aplikace SZDC a její model

Než jsme mohli začít vytvářet model a vykreslovat obsah nákresného jízdního řádu, museli jsme zjistit, podle jakých pravidel se nákresný jízdní řád Správy železniční a dopravní cesty vytváří a s jakými daty pracuje. Na základě získaných dat jsme poté vytvořili jejich zdroj, ke kterému se může aplikace připojit a používat je. Pravidla popisující tvorbu grafikonu vlakové dopravy se nachází v článku 13 dokumentu Směrnice SŽDC č. 69, dostupném jako `podklady-njr.pdf` v příloze práce v `/szdc/documents/`. Naším cílem bylo co nejvíce těchto pravidel vizualizovat. Potřebujeme ale data, na která je pravidla možné aplikovat. Data, která jsou zobrazována v nákresných jízdních řádech, jsou dostupná online jako PDF dokumenty představující knižní a sešitové jízdní řády [29]. Pro nahlédnutí se nachází v příloze práce v `/szdc/sjr/`. Data ve formátech pro strojové zpracování jako je třeba JSON bohužel nejsou veřejně dostupná. Proto jsme se rozhodli najít cestu, jak získat data z PDF souborů. Na základě zpracovaných dat jsme pak určili konkrétní podporovaná pravidla.

Celá jízda jednoho vlaku, který je označen unikátním číslem, je rozdělena do několika sešitových jízdních řádů určených pro konkrétní tratě. Každý obsahuje pouze nějakou část jeho celého plánu jízdy, jako na obrázku 6.6. Sloupce 5 a 7 odpovídají příjezdu a odjezdu do dopravního bodu. Dokument `tabulka-sjr.pdf` v příloze práce v `/szdc/documents/` tuto tabulku plánu jízdy vlaku v sešitovém jízdním řádu detailně popisuje.

Os 2951

Studénka - Bohumín os.n. - Mosty u Jablunkova

Elektrická jednotka ř. 471.

1	2	3	5	6	7	8
Studénka					**18 07**	**140/$^{141}_{98}$**
Jistebník		**5**	**18 12**	0^5	**12^5**	
Polanka nad Odrou z		**3**	× **15^5**	▲	**15^5**	
Výh Polanka n.Odrou		**1**			**16^5**	
Ostrava-Svinov		**2^5**	**19**	1	**20**	
Ostrava-Mar.Hory z		**3^5**	× **23^5**	▲	**23^5**	
Ostrava hl.n.		**3^5**	**27**	3	**30**	
Bohumín os.n.		**7**	**18 37**		*18 39*	

Obrázek 6.6: Část plánu jízdy vlaku 2951 v sešitovém jízdním řádu

Rozhodli jsme se, že PDF soubory sešitových jízdních řádů naparsujeme, převedeme do vhodného formátu a získáme z nich části plánu jízd vlaku, které pak sloučíme do jednoho celého plánu. Až budeme vytvářet nákresný jízdní řád, vlaky v něm získáme průnikem stanic se stanicemi v takto sloučených plánech.

Tabulková data ze sešitových jízdních řádů jsme se pokoušeli získat pomocí různých PDF nástrojů, ale ukázalo se, že tabulky jsou pro zpracování nějakým frameworkem pro práci s PDF soubory příliš komplexní. Vyzkoušeli jsme, jak tyto tabulky vypadají po převodu do tabulkového XLSX formátu. Data v tomto formátu se ukázala být již lépe zpracovatelná. Jelikož jsme měli zkušenost s parsováním XLSX souborů pomocí knihovny Apache POI [30] v Javě, rozhodli jsme se soubory zpracovat touto knihovnou. Výsledkem zpracování je vygenerovaný

JSON výstup, uvedený na následujícím fragmentu kódu. Čas se uvádí v půlminutách ve vlastnosti `"isAfter30Seconds"` odpovídající číslu pět v horním indexu za minutovým údajem ve sloupcích příjezdu (5) a odjezdu (7), které se seznamem stanic v prvním sloupci zpracováváme.

```
{ "trains": {
  "2951": {
    "trainType": "Os",
    "schedule": [
      {
        "departureTime": {
          "hours": 18,
          "minutes": 7,
          "isAfter30Seconds": false
        },
        "arrivalTime": null,
        "stationName": "Studénka"
      },
      { (+) },
      .
      .
      .
      { (+) },
      ]
   },
   "1345": { (+) }
}
}
```

Jiná data v dalších sloupcích sešitových jízdních řádů nezpracováváme, z důvodu jejich nespolehlivého převedení do XLSX souboru. Příklad převedeného sešitového jízdního řádu do souboru XLSX a vytvořené Java aplikace pro jeho parsování se nachází v příloze práce v `/szdc/parser/xlsx`. Pomocí tohoto postupu jsme z celkového počtu přibližně 20000 plánů nedokázali získat řádově desítky plánů jízd. Všechny takto úspěšně převedené plány jízd se nachází v příloze práce v `/szdc/njr/json`. Ve formátu JSON se vytváří i popis traťových úseků v rámci nákresného jízdního řádu, jako v souboru `/szdc/njr/json/railways/301.json`.

Pro parsování JSON dat do různých aplikací je vývojářům dostupná utilita `SZDC.JsonParser`, která převádí tato data formátu JSON do objektů C#. Parseru se předává objekt knihovny Newtonsoft [31], představující načtený JSON soubor. Data jsou parserem předána implementaci rozhraní `IParsedTrainReceiver`, kterou parseru dodá vývojář. Stejným způsobem jsou pak získatelná data popisující stanice.

Pravidla pro sestavení nákresného jízdního řádu

Nyní si na základě dostupných dat představíme pravidla, která podporujeme, podle článku 13 Směrnice SŽDC č. 69, ilustrovaná obrázky v 6.7 a nacházející se v příloze `/szdc/documents/podklady-njr.pdf`:

(a) (b) (c) (d)

Obrázek 6.7: Ilustrace pravidel tvorby nákresného jízdního řádu.

13.2.2 Trasy vlaků osobní dopravy (zkratky Os, Sp, R, Ex) se tisknou černě, trasy vlaků nákladní dopravy a služebních vlaků (zkratky Nex, Mn, Sl, Vl, Pn) modře.

13.2.2 Kóty, čísla vlaků a příp. značky před kótami se tisknou stejnou barvou jako trasa vlaků.

13.3.2 Trasy nákladních vlaků se ve výchozích a konečných stanicích označí zakreslením plného kolečka na vodorovné čáře stanice (c).

13.2.10 Projíždí-li vlak v odbočné stanici nebo na odbočce na jinou trať nebo projíždí-li stanici ohraničující zobrazený úsek, prodlouží se trasa krátkou šipkou k umožnění zapsání průjezdové kóty (d). Obdobně se vyznačí průjezd vlaku vstupujícího v odbočné stanici nebo ve stanici ohraničující zobrazený úsek.

13.2.12 U příjezdových kót v dopravnách a stanovištích se používají značky pro vyznačení:

1. *Pobytu kratšího než půl minuty. Kóta se nahradí trojúhelníkem jako v (a)*

Pokud má časový údaj o půl minuty více, kóta se podtrhává (b). Síla čar trasy vlaku je určena na obrázku 6.8 podle zkratek zmíněných v pravidle 13.2.2.

Obrázek 6.8: Tloušťky čar představující trasy vlaku

Práce s daty a SZDC.Data

Jako zdroj dat slouží aplikaci projekt `SZDC.Data`, který importuje data ve zmíněném JSON formátu a nahrává je do PostgreSQL [32] databáze pomocí EntityFrameworkCore [33]. Projekt implementuje rozhraní `IStaticDataProvider` v třídě `DbDataProvider`, pomocí kterého aplikace získává z `SZDC.Data` data. V rámci představení možností práce s knihovnou jsme ke každé stanici při parsování dat, které probíhá pomocí konzolové aplikace v tomto projektu, navíc vygenerovali její koleje.

Pokud chce vývojář dodat aplikaci vlastní databázi jako zdroj dat, v příloze práce v `/szdc/db` se nachází návod, jak propojit aplikaci s databází, inicializovat ji používaným schématem modelu a importovat do ní data ve formátu JSON.

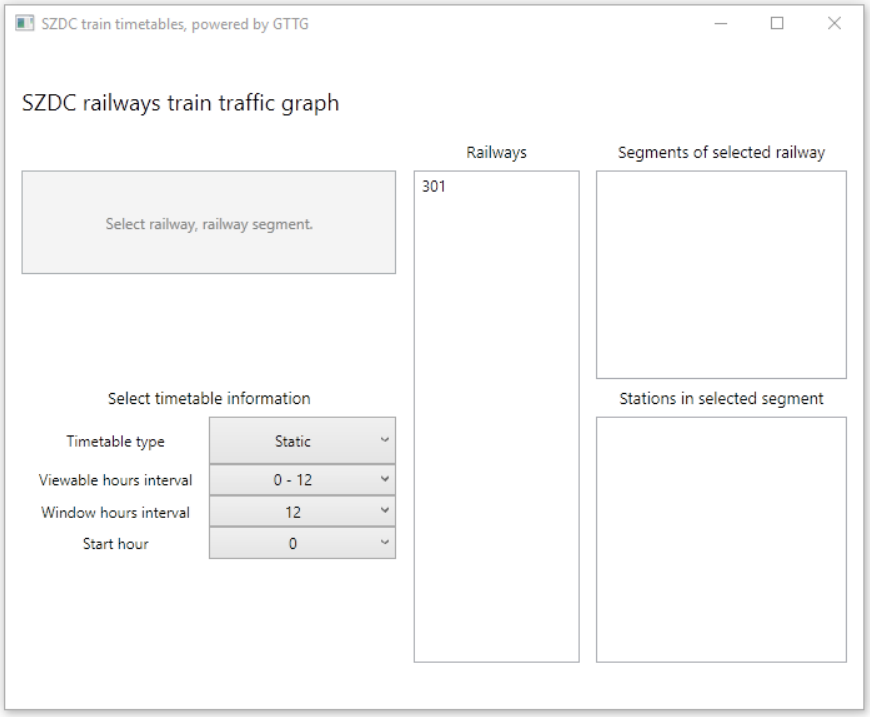
Model a view model aplikace

Pravidla, podle kterých chceme obsah zobrazovat, jsme zahrnuli do modelu vytvořeném v projektu `SZDC.Model`, který dále rozšiřuje základní model z knihovny GTTG. Implementovaný view model v tomto projektu pak s vlastnostmi modelu představující pravidla pracuje. Například tloušťka a vlastnosti čar se view modelům přiřazují v implementaci factory metod. Pokud se musí aplikovat pravidla 13.3.2 a 13.2.10 související s přidáváním prvků vizualizace trasy vlaku, používáme *decorator* pattern řetězením několika implementací rozhraní `ITrainPath`, každé dodávající vizualizaci nějakého pravidla.

Pravidla spojená s vizualizací kót jsou implementována různými zobrazitelnými prvky. V přetížené metodě `UpdateTrainViewContent()` na view modelu vlaku `SzdcTrainView` každé jeho události `SzdcTrainEvent` nastavíme *flagy* výčtového typu `TrainEventFlags` – například `LessThanHalfMinute` pro nahrazení číselné kóty trojúhelníkem podle 13.2.12. Dále se pak v této metodě podle kombinace flag každé události přidělí zobrazitelný prvek reprezentující kótu – například `TimeComponent` nebo `TriangleComponent`. Tyto prvky se pak v metodě `Arrange()` spolu s čísly vlaků rozmístí podle strategií základní implementace zmíněných v 4.3.2.

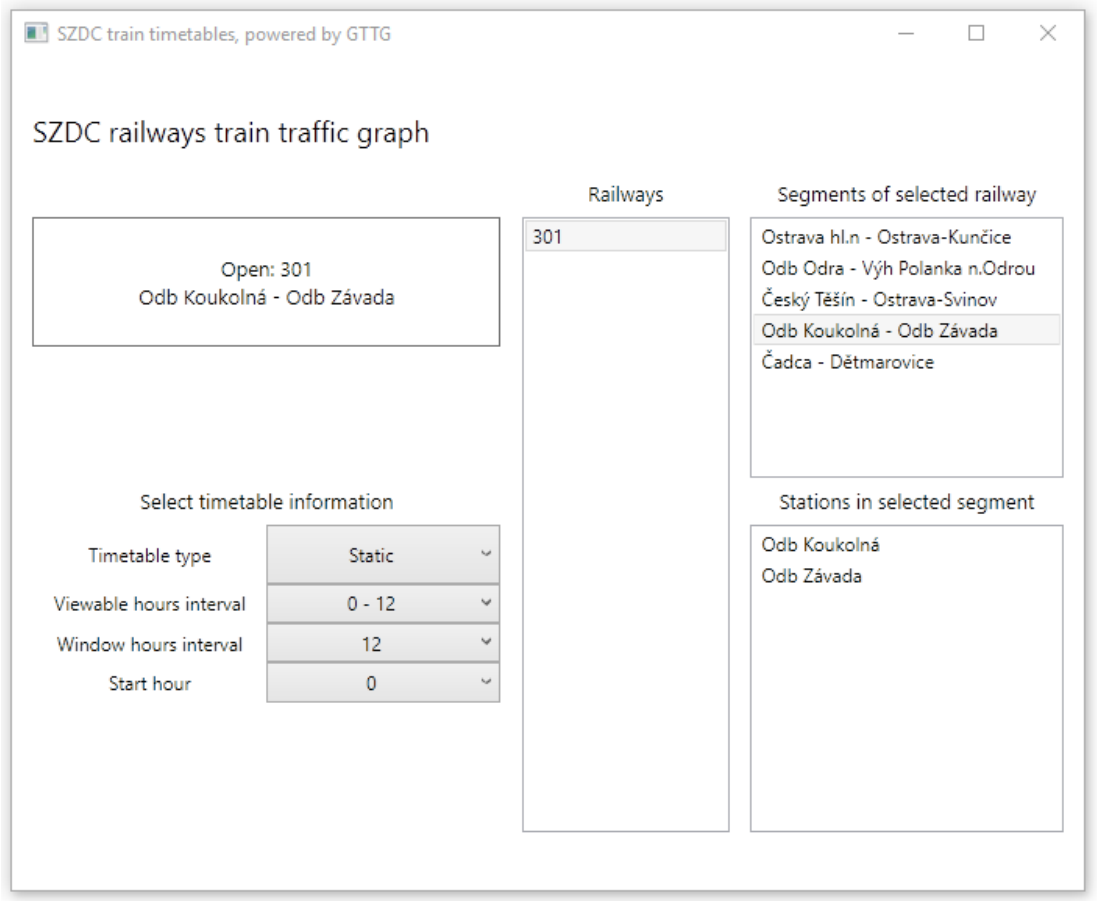
6.3 Uživatelská dokumentace

V této podkapitole si ukážeme možnosti práce s aplikací SZDC na dvou módech nabízených aplikací. Při spuštění aplikace se uživateli zobrazí okno na obrázku 6.9, pomocí kterého uživatel otevírá nová okna s nákresným jízdním řádem.



Obrázek 6.9: Okno pro otevření nákresného jízdního řádu v aplikaci SZDC

Ve sloupci `'Railways'` se nachází seznam očíslovaných tratí. Když na číslo v seznamu klikneme, ve sloupci `'Segments of selected railway'` se uživateli zobrazí úseky tratě (na obrázku 6.10), podle obsahu nákresného jízdního řádu zvolené trati. Například pro trať číslo 301 se uživateli nabízí zvolit pět úseků, nacházejících se v nákresném jízdním řádu v příloze práce v `/szdc/njr/pdf/L301_d.pdf`. Kliknutím na vybraný úsek se ve sloupci `'Stations in selected segment'` zobrazí jeho seznam dopravních bodů.

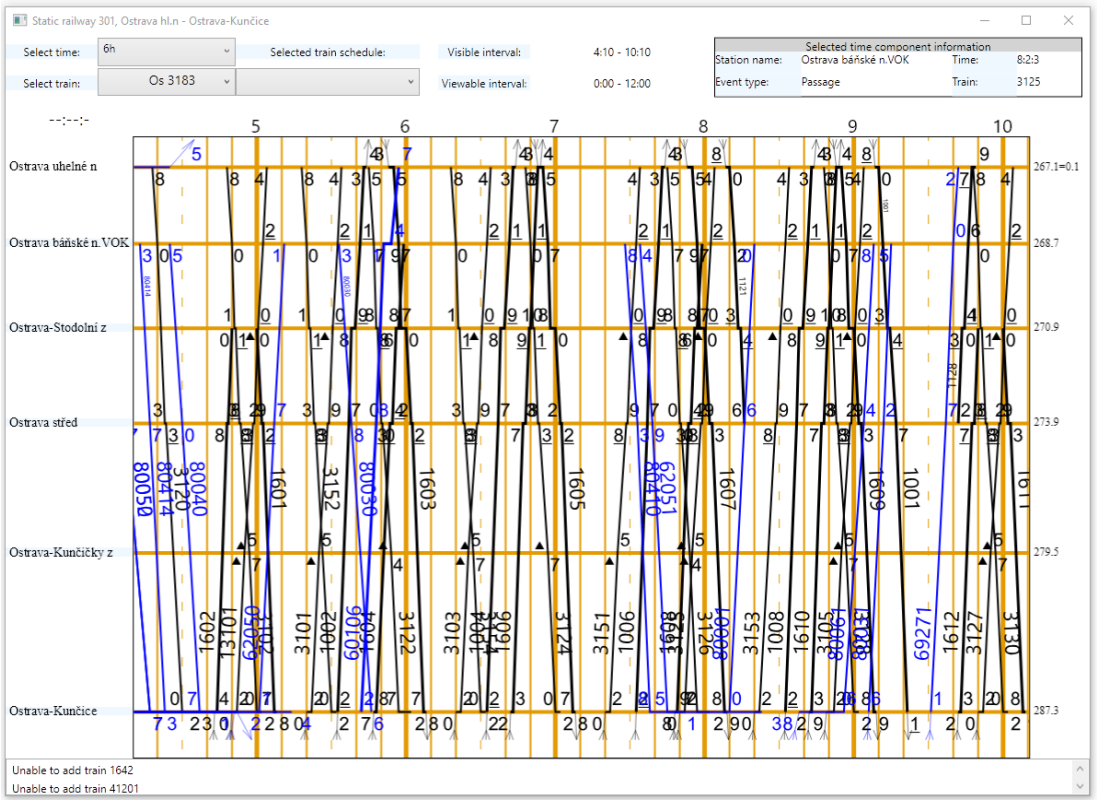


Obrázek 6.10: Otevření nákresného jízdního řádu v aplikaci SZDC

Tlačítko v levé horní části okna, určené k otevření okna nákresného jízdního řádu, se tímto výběrem nachází ve stavu, kdy text tlačítka obsahuje číslo vybrané trati a jejího úseku. Kliknutím na tlačítko se otevře nové okno s vybraným nákresným jízdním řádem. Pod tlačítkem se nachází oblast `'Select timetable information'`, v které konfigurujeme otevíraný nákresný jízdní řád. V kombinovaném seznamu `'Timetable type'` se vybírá jeden z módů zobrazení nákresného jízdního řádu `'Static'` a `'Realtime'`, které si představíme v dalších částech. Pro `'Static'` mód je možné vybrat určením času zobrazovaný obsah. Kombinovaný seznam `'Viewable hours interval'` udává časový rozsah, který je možné prohlížet. `'Window hours interval'` nastavuje počáteční délku zobrazovaného časového intervalu. `'Start hour'` udává hodinu, kterou bude začínat časový interval zobrazení v okně při otevření.

Statický mód zobrazení nákresného jízdního řádu

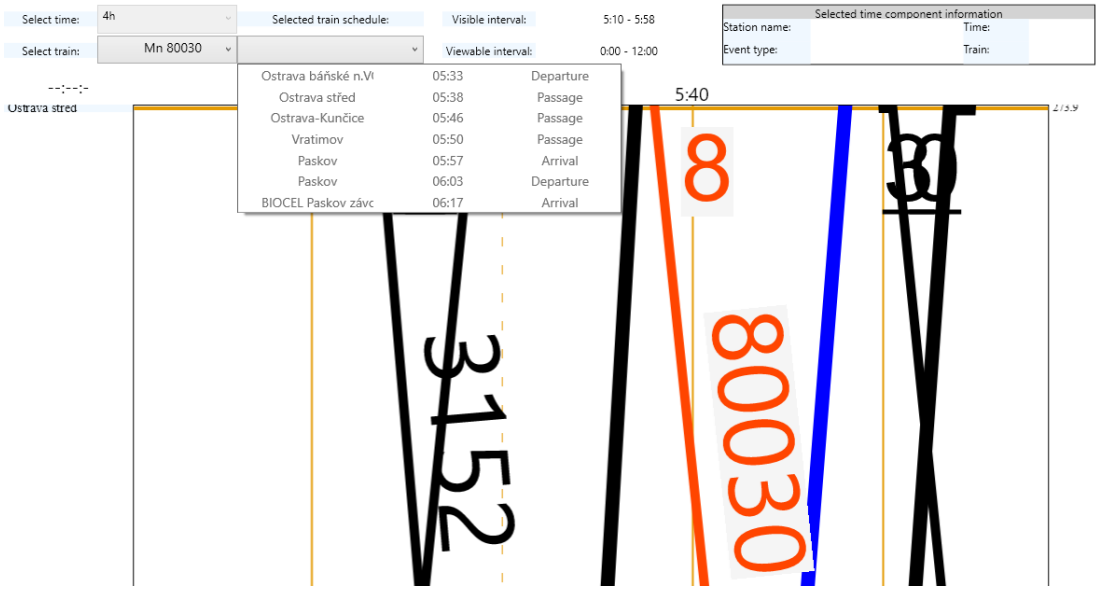
Nyní si popíšeme okno, které se otevře při nastavení hodnoty kombinovaného seznamu `'Timetable type'` na `'Static'`. Okno nákresného jízdního řádu ve statickém módu se nachází na obrázku 6.11 a umožňuje pouze prohlížení nákresného jízdního řádu. Skrolováním myši je možné pohled oddalovat a přibližovat. Držením levého tlačítka při posunu myši je možné měnit zobrazovaný obsah. Časový interval, který je možné zobrazit, je uveden v `'Viewable interval'`. Zobrazený časový interval je možné měnit pomocí `'Select time'`. Pohybem myši po obsahu nákresného jízdního řádu se v levém horním rohu ukazuje čas odpovídající pozici kurzoru.



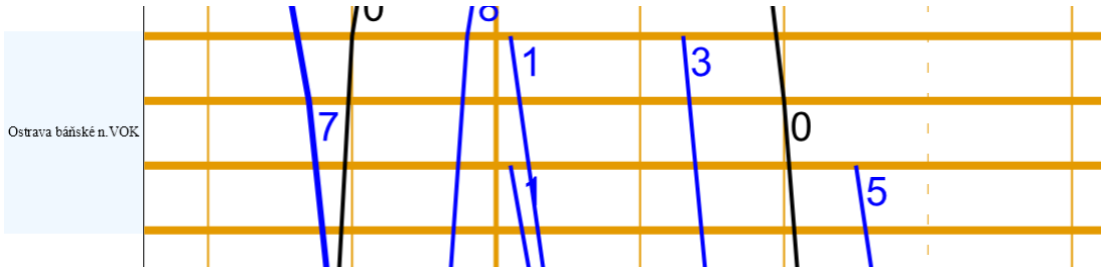
Obrázek 6.11: Nákresný jízdní řád ve statickém módu

Kliknutím na kombinovaný seznam označený `'Select train'` je možné vybrat vlak, který je pak v obsahu nákresného jízdního řádu oranžově zvýrazněn a přenesen do popředí, jako na obrázku 6.12. Navíc jsou jeho kóty a číslo vlaku pro větší přehlednost bíle podbarvené. Výběr vlaku je možné provést i kliknutím pravým tlačítkem na obsah nákresného jízdního řádu, kdy se vybere nejbližší vlak k pozici kurzoru myši. Otevřením seznamu pod `'Selected train schedule'` se otevře plán jízdy vlaku. Kliknutím levého tlačítka na kótu se zobrazí podrobnější informace v `'Selected time component information'` – typ události (příjezd, odjezd, průjezd), stanice, číslo vlaku a čas.

Pokud dopravní bod obsahuje více kolejí, je jeho jméno v levém sloupci modře podbarvené. Kliknutím na toto podbarvení je možné horizontální čáru dopravního bodu rozdělit do více kolejí, jako na obrázku 6.13. Toto rozdělení je pak znovu možné seskupit do jedné čáry kliknutím na podbarvení.



Obrázek 6.12: Zobrazený průběh jízdy vybraného vlaku

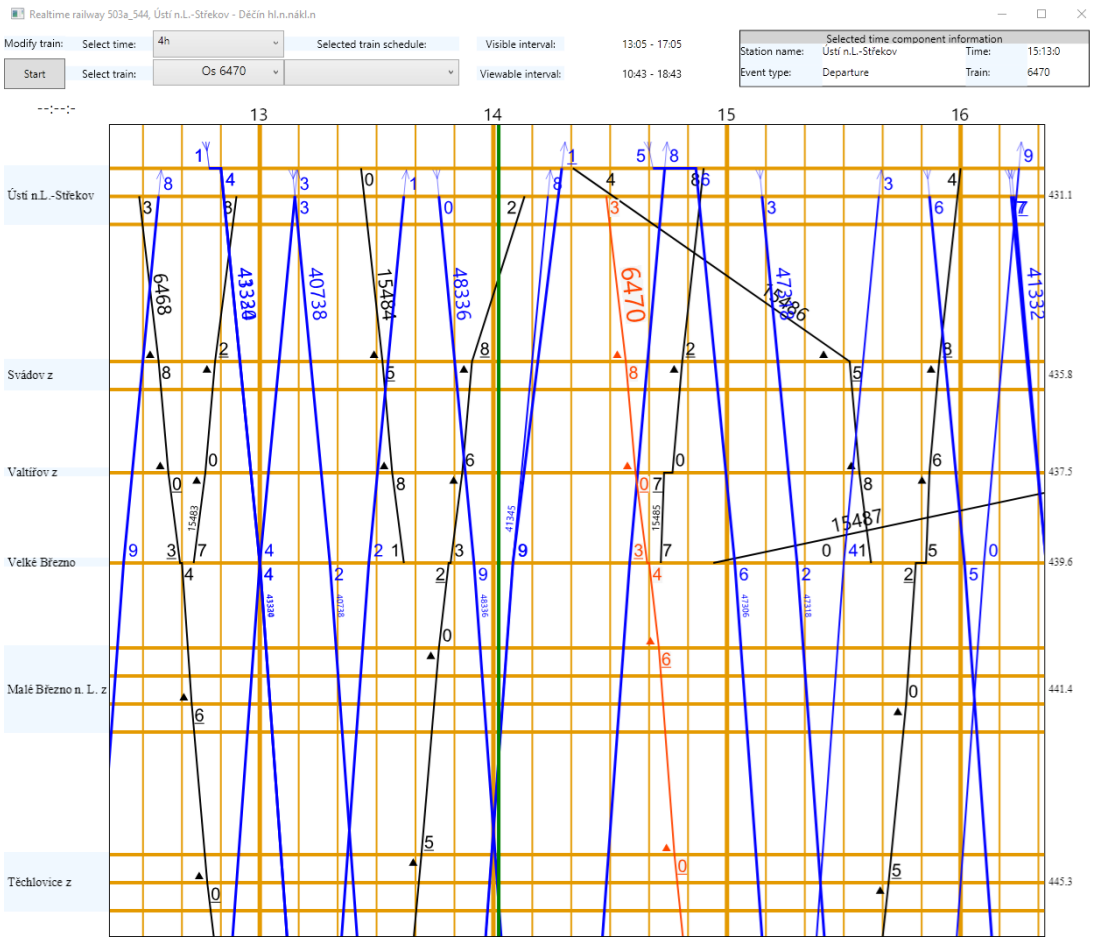


Obrázek 6.13: Rozdělení vodorovné čáry dopravního bodu do jednotlivých kolejí

Dynamický mód zobrazení nákresného jízdního řádu

Dynamický mód, jehož okno se nachází na obrázku 6.14, nabízí stejné nástroje jako statický mód a navíc umožňuje modifikaci plánovaného průběhu jízd vlaků. Aplikace sama v krátkých časových intervalech upravuje náhodně průběh jízdy některých vlaků. Při změnách se mění hodnoty časových kót i jejich zobrazení podle uváděných pravidel. Pokud je otevřeno více oken dynamického módu, které zobrazují stejný vlak, změny se zobrazí ve všech oknech. Zobrazuje se aktuální časový interval délky osmi hodin a průběžně se po hodině posouvá. Při posunu jsou přidávána i odebírána zobrazovaná data. Zelená svislá čára se obnovuje každou minutu a značí aktuální čas.

Po kliknutí na tlačítko pod textem `'Modify train'` v horní liště okna může uživatel sám interaktivně měnit plán jízd vlaků. Nejdříve musí uživatel vlak vybrat, aby se nacházel v `'Select train'`. Poté uživatel klikne na zmíněné tlačítko. Není možné již provádět pravým tlačítkem výběr jiných vlaků. Naopak klikáním na obsah nákresného jízdního řádu je nyní možné průběh jízdy vlaku upravovat. Na pozici kurzoru myši je přesunuta ta událost, která se k pozici nachází nejblíže. Úpravy je však možné pouze aplikovat na plánovaný provoz, který se nachází napravo od zelené svislé čáry značící aktuální čas. Modifikace se projeví v ostatních oknech a zároveň i v ostatních uživatelských prvcích okna, jako například v seznamu `'Selected train schedule'`.



Obrázek 6.14: Dynamický mód zobrazení v aplikaci SZDC

Závěr

V závěru této práce zhodnotíme splnění cílů vytyčených v podkapitole 1.5.

G1 *Vytvořit knihovnu pro platformu .NET, představující grafickou komponentu, umožňující práci s nákresnými jízdními řády.*

- **a)** *Podporovat zobrazování různých typů nákresných jízdních řádů* – Naše knihovna umožňuje strukturováním obsahu do vrstev a poskytnutím obecných struktur systematicky popsat jakýkoliv typ nákresného jízdního řádu. Nabízí konfigurovatelnou základní vizualizaci nákresného jízdního řádu, rozšířením použitelnou pro vizualizaci konkrétních typů nákresných jízdních řádů. Popsali a vyvinuli jsme nástroje, které práci s obsahem nákresného jízdního řádu ulehčují.
- **b)** *Podporovat integraci komponenty do aplikací pracujících s grafikonem vlakové dopravy* – Nabízené konfigurace komponenty a její obecné úpravy umožňují její integraci do různých aplikacích, lišících se způsobem ovládání komponenty. Uživatel aplikace může navíc interaktivně pracovat přímo s obsahem nákresných jízdních řádů.
- **c)** *Umožnit replikovat chování existujících aplikací pracujících s grafikonem vlakové dopravy* – Knihovnu jsme implementovali podle požadavků v kapitole 2 založených na chování existujících aplikací. Komponentu je navíc možné narozdíl od existujících aplikací interaktivně ovládat přiblížováním a oddalováním pohledu.
- **d)** *Zajistit přenositelnost knihovny na úrovni .NET Standard* – Knihovna je implementována vůči rozhraní .NET Standard 2.0. Je tak použitelná v různých aplikacích, které jsou vyvíjeny vůči konkrétním běhovým prostředí platformy .NET.

G2 *Použít tuto knihovnu pro implementaci aplikace, která bude sloužit pro práci s grafikonem vlakové dopravy.*

- **a)** *Aplikace bude interaktivně zobrazovat listy nákresného jízdního řádu vydávané Správou železniční dopravní cesty* – Získali jsme textovou reprezentaci dat vizualizovaných v nákresných jízdních řádech Správy železniční dopravní cesty. Na základě získaných dat jsme vytvořili model, který při vizualizaci odpovídá předloze. Ověřili jsme, že je možné knihovnu využít pro vytvoření konkrétního typu nákresného jízdního řádu. Ukázalo se, že implementace knihovny umožňuje plynule zobrazovat větší množství zobrazovaných tras vlaků.
- **b)** *Aplikace bude pro ilustrační účely knihovny nabízet mód simulující provoz na trati, v jehož rámci bude možné upravovat výhledovou dopravu* – Implementací tohoto módu jsme předvedli, že je možné obsah nákresného jízdního řádu interaktivně upravovat.
- **c)** *Aplikace bude navržena tak, aby část pracující s modelem dat, představující logiku aplikace, byla zapojitelná do více GUI frameworků platformy .NET* – Aplikační logika a model jsou implementovány vůči

rozhraní .NET Standard a využitím nástrojů používaných GUI frameworky platformy .NET je aplikace do těchto frameworků plně zapojitelná.

G3 *Ověřit možnost využití 2D grafické knihovny SkiaSharp pro zobrazování komplexních dat obsažených v nákresných jízdních řádech.*
– Použití knihovny SkiaSharp se ukázalo jako správné řešení. Propojením optimalizací knihovny SkiaSharp a GTTG jsme dokázali vytvořit nástroj, který zvládá plynule zobrazovat i velmi frekventovaný provoz na trati. Jednou z věcí, s kterou bylo problematické pracovat, je určení přesné velikosti textu. Tento problém je ale pro vývojáře lehce řešitelný díky návrhu knihovny, kdy změřené hodnoty není potřeba přepočítávat pro různé velikosti.

Možné pokračování

V budoucnu bychom chtěli knihovnu GTTG případně rozšiřovat:

- Knihovnu rozšířit na framework umožňující vývojářům specifikovat různé požadavky pro vytvoření view modelu. Framework by vytvořil požadovanou implementaci za vývojáře.
- Vytvořit nástroje ulehčující vývojářům práci s některými částmi knihovny SkiaSharp, například nástroj nahrazující měření textu pomocí vlastností `SKPaint.FontMetrics`, které jsou pro vývojáře složitější na pochopení.
- Rozšířit projekt `GTTG.Model` o další strategie a konfigurace view modelu

Aplikaci SZDC by bylo možné dále rozšířit a upravit:

- Při získání vhodného zdroje dat doplnit obsah nákresného jízdního řádu vizualizací dalších pravidel. Vytvořili jsme již další pravidla, která jsou součástí modelu a view model s nimi pracuje, ale současná implementace `IStaticDataProvider` je nahrazuje základními hodnotami, jelikož k nim nemá žádný relevantní zdroj dat:
  - Vykreslované dekorace šikmé čáry průběhu jízdy vlaku – bílá kolečka (vlak jedoucí podle potřeby) nebo černé čárky kolmo k čáře (vlak jede po nesprávné koleji), podle pravidel záhlaví nákresného jízdního řádu
  - Vizualizace horizontálních čár dopravních bodů podle jejich typů (pravidlo 13.1.2 směrnice č. 69 v `/szdc/documents/podklady-njr.pdf`)
  - Doplnění informací umístěných vedle kót v ostrých úhlech s vytvořením jejich vizualizace a získáním zdroje dat
- Vytvořit JSON soubory popisující další tratě podle existujících nákresných jízdních řádů
- Optimalizovat rychlost načítání dat do aplikace
- Při rozšiřování dynamického módu přemístit okna a správu obsahu aplikace do separátních vláken
- Přemigrovat v budoucnu WPF projekt na .NET Core 3

Seznam použité literatury

[1] J. Gašparík, J. a Kolář. *Železniční doprava*. První vydání. Grada, Praha, 2017.

[2] GTN demoverze. https://www.azd.cz/cs/media/ke-stazeni.

[3] Aplikace pracující s grafikonem vlakové dopravy v rámci IS ISOR CDS. https://www.oltis.cz/produkty/nakladni-doprava/isor-cds-centralni-dispecersky-system/.

[4] FBS-Bahn. http://www.en.irfp.de/the-timetable-construction-system-fbs.html.

[5] Martin Drábek. Koncepce obsluhy letiště ruzyně kolejovou dopravou. Master's thesis, Czech technical university in Prague, Faculty of Transportation Sciences, Plzeň, 2007.

[6] Aplikace Grafikon pro modelové železnice. https://github.com/jub77/grafikon.

[7] Trainz. https://www.trainzportal.com/.

[8] MSTS. https://cs.wikipedia.org/wiki/Microsoft_Train_Simulator.

[9] OpenRails. http://openrails.org/.

[10] OpenBVE. https://openbve-project.net/.

[11] SkiaSharp. https://github.com/mono/SkiaSharp.

[12] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[13] Core2D. https://github.com/wieslawsoltes/Core2D.

[14] xUnit. https://xunit.net/.

[15] moq. https://github.com/moq/moq4.

[16] GTTG.Core.NuGet. https://www.nuget.org/packages/GTTG.Core/.

[17] GTTG.Model.NuGet. https://www.nuget.org/packages/GTTG.Model/.

[18] GTTG.Traffic tutoriál. https://github.com/Sykoj/GTTG.Tutorials.Traffic.

[19] .NET Framework 4.7.2 runtime. https://dotnet.microsoft.com/download/dotnet-framework/net472.

[20] .NET Framework 4.7.2 developer pack. https://dotnet.microsoft.com/download/dotnet-framework/net472.

[21] GTTG.Integration tutoriál. https://github.com/Sykoj/GTTG.Tutorials.Integration.

[22] GTTG.Infrastructure tutoriál. https://github.com/Sykoj/GTTG.Tutorials.Infrastructure.

[23] GVD nákresné jízdní řády. http://gvd.cz/czx/data/njr.html.

[24] Avalonia. http://avaloniaui.net/.

[25] Autofac. https://autofac.org/.

[26] Aplikace v Avalonii. https://github.com/AvaloniaUI/Avalonia/wiki/Projects-that-are-using-Avalonia.

[27] Xamarin.Forms na jiných platformách. https://docs.microsoft.com/en-us/xamarin/xamarin-forms/platform/other/.

[28] Rider. https://www.jetbrains.com/rider/.

[29] GVD sešitové jízdní řády. http://gvd.cz/cz/data/sjr-os.html.

[30] Apache POI. https://poi.apache.org/.

[31] Newtonsoft. https://www.newtonsoft.com/json.

[32] PostgreSQL. https://www.postgresql.org/.

[33] EntityFrameworkCore. https://docs.microsoft.com/en-us/ef/core/.

[34] Inkscape. https://inkscape.org/.

[35] PDF-XChange Editor. https://www.pdfxchange.cz/.

Přílohy

Obsah příloh práce

- `/binaries` – složka s binárními soubory a spustitelnou aplikací SZDC, vyžadující nastavené připojení k databázi
- `/solution` – solution obsahující zdrojový kód knihovny GTTG a aplikace SZDC
- `/tutorials` – složka s jednotlivými tutoriály, každý obsahující v `/bin` složce spustitelnou aplikaci
- `/szdc` – složka s datovými soubory a dokumenty obsahu aplikace SZDC
- `/documentation` – složka s PDF dokumentací balíčků GTTG.Core [16] a GTTG.Model [17] (ve verzi 1.0.4)
- `/tex` – obsah práce vytvořený pomocí LaTeX

  `/cs` – složka s `.tex` soubory textu práce

  `/img` – pdf obrázky a původní svg obrázky vytvářené v Inkscape [34], převedené do PDF/A-2u pomocí PDF-XChange Editor [35]
- `/prace.pdf` – soubor obsahující tuto práci
- `/README.txt` – soubor s tímto seznamem příloh
- `/LICENSE.txt` – licence knihovny GTTG a aplikace SZDC