Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Nástroj pro uchování a zobrazení dat o silniční dopravě z různých zdrojů

Plzeň 2019 Petr Laštovka

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 1. května 2019

Petr Laštovka

Poděkování

Rád bych poděkoval panu Ing. Tomáši Potužákovi, PhD., za jeho odborné rady, poskytnuté materiály a konstruktivní připomínky při vedení této práce.

Abstract

The submitting bachelor work deals with making an application, which will store data for traffic simulation. The main goal of this work was to create a program, which will support import and export data via a plugin. These plugins will not be part of the application, but it will be added to the application as a separate part.

Abstrakt

Text předkládané bakalářské práce se zabývá vytvořením aplikace, jež bude uchovávat data pro dopravní simulace. Hlavním cílem práce bylo vytvoření programu, který bude podporovat import a export dat skrze pluginy. Tyto pluginy nebudou součástí aplikace, ale bude je možné do aplikace dodat jako samostatnou část.

Obsah

1 Úvod

Silniční doprava se v dnešní době stala velmi důležitým aspektem života. Dopravní komplikace, které nejčastěji představují dopravní zácpy, znepříjemňují situaci lidem, kteří se v tu chvíli přemísťují v dotčené oblasti, brzdí přepravu zboží a mají také vliv na spotřebu pohonných hmot. Stále se zvyšující počty osobních i dopravních automobilů spolu s pomalu se rozvíjející infrastrukturou kladou na řešení této problematiky stále větší požadavky. Není snadné určit, jaké dopravní řešení je pro konkrétní místo nejvhodnější, protože existuje pro každý případ několik proměnných. Odpovědi na tyto otázky mohou přinést simulace dopravy, které zahrnou všechna potřebná specifika.

Pro provádění těchto simulací potřebujeme nejdříve získat veškerá data o dané problematice. Jejich sběr není jednoduchý. To má za následek, že výsledný soubor dat je nejednotný, a proto musí být zpracován a uložen pro pozdější snadné využití. Právě tato problematika se stala předmětem mé práce.

V teoretické části představím dopravní simulace a data nezbytná pro jejich realizaci. Praktická část se zabývá mnou naprogramovanou aplikací, která je schopna tato data uchovávat.

2 Simulace

Simulovat znamená předstírat, napodobovat. Slovník [40] o simulaci mluví jako o napodobování reálných činností, dějů stavů a procesů. Samotný akt simulace pak znamená zobrazení (napodobení) zkoumaných klíčových vlastností nebo chování vybraných fyzikálních nebo abstraktních systémů. V odborné literatuře zabývající se simulací existuje hned několik definic:

- Simulace je proces tvorby modelu reálného systému a provádění experimentů s tímto modelem za účelem dosažení lepšího pochopení chování studovaného systému či za účelem posouzení různých variant činnosti systému [1].
- Numerická metoda, která spočívá v experimentování s matematickými modely dynamických reálných systémů na číslicových počítačích [2].

V současnosti existuje několik druhů simulací, které ale nejsou předmětem této práce. Bude-li se někdy zmiňovat v této práci simulace, bude tím myšlena počítačová simulace. Tato simulace se pokouší vymodelovat pomocí počítače skutečný svět, který následně může zkoumat. Dokáže také studovat, jak se tento svět mění, jsou-li změněna vstupní data. Příklad počítačové simulace může být např. trenažér pro piloty. Změnu ve vstupních datech představuje přímý vstup od uživatele (pilota) a reakce takového trenažéru reprezentuje změnu modelového světa na vstupní data.

2.1 Základní pojmy

Pro další popis simulace je nutné si nejdříve uvést a vysvětli několik následujících pojmů.

2.1.1 Systém

Soubor elementárních částí (prvků systému) a vazeb mezi nimi, který jako celek má určité vlastnosti. Nemusí se vždy jednat pouze o reálný systém, ale může to být i zcela fiktivní ještě neexistující systém [19].

2.1.2 Model

V modelování a simulaci je termín model použit pro analogii mezi dvěma systémy. Jednoduché příklady nabízí mapa (model části země na papíře), socha

(model osoby, zvířete atd. v neživém materiálu) nebo dětský vláček (model skutečného vlaku ve zmenšeném měřítku). Není samozřejmě možné jakoukoliv realitu popsat do těch nejmenších podrobností (podobně jako mapa nikdy nebude zobrazovat všechny detaily), proto je nutné se při průběhu analýzy reálného či fiktivního systému soustředit pouze na ty části, které jsou z hlediska cíle analýzy důležité [3].

2.1.3 Experiment

Jedno aktivní provedení simulace. Začíná zadáním vstupních údajů do simulace, pokračuje jejím provedením a končí následným zhodnocením/vyhodnocením získaných výsledků [11].

2.1.4 Abstraktní model

Zjednodušený popis zkoumaného systému [19].

2.1.5 Simulační model

Abstraktní model zapsaný formou programu [19].

2.1.6 Validace

Validace modelu představuje ověření, zda vytvořený model je skutečně modelem zkoumaného systému, tedy že se shoduje s realitou [10].

2.1.7 Verifikace

Verifikace spočívá v ověření, zda simulační model je v souladu s původním abstraktním modelem. Jde o ověření, že představa o fungování systému byla správně formálně zapsána [10].

2.2 Modelování

Modelování je proces vytváření modelů nezbytných pro simulaci. Během tvorby musí být brán ohled na následující dvě pravidla:

- Pokud zjednodušíme skutečnost příliš, model bude zkreslený a výsledky, které získáme na základě analýzy, budou nereálné až nesmyslné [4].

- Jestliže model bude usilovat o co nejvěrnější napodobení reality, budeme mít sice velmi přesný model, ale jeho následná analýza bude téměř neuskutečnitelná a výsledky poté nedosažitelné [4].

Na základě předchozích dvou pravidel je potřeba mít na paměti, že při vytváření modelu musíme najít určitý kompromis mezi věrnou kopií skutečnosti a snadnou řešitelností úlohy vyjádřené daným modelem [4]. Účelem modelování je tedy vytvoření modelu konkrétního systému a pomocí experimentů s tímto modelem získat informace o původním systému.

Samotný proces modelování lze popsat v několika krocích. Nejdříve po zvolení zkoumaného systému, pomocí verbálního nebo matematického popisu typických vlastností daného systému vytvoříme abstraktní model. Následně tento model transformujeme do počítačového prostředí, ve kterém můžeme simulovat chování systému. Vytvoříme tak simulační model. Následně musí přijít dvoukrokové ověření modelu pomocí verifikace a validace. Verifikací se v tomto případě rozumí, zda vytvořený simulační model reprezentuje v zadané míře přesnosti abstraktní model. Při validaci kontrolujeme, zda se simulační model dostatečně shoduje s reálným systémem a zda ho lze použít k zamýšlenému použití [19].

2.3 Výhody a nevýhody simulace

2.3.1 Výhody simulace

Simulace se dnes velmi často využívá pro svojí možnost relativně snadného získání a ověření znalostí v těch oblastech, kde by vynaložení zdrojů pro získání dat bylo neúnosně vysoké, nemluvě o nákladech na odstranění špatného rozhodnutí.

Mezi další velké výhody simulace patří změna rychlosti toku času během chodu simulace. Je možné tak v relativně krátké chvíli získat data, která by bylo možné nashromáždit například až za celou směnu. Naproti tomu lze i čas simulace zpomalit a je tak možné zkoumat jev, který v reálném světe trvá pouhý okamžik, neomezeně dlouho [20].

2.3.2 Nevýhody simulace

Simulace mají i některé nevýhody. Především kladou vysoké nároky na tvorbu počítačového modelu. Dále při nedostatečné kvalifikaci hrozí chybná

interpretace výsledků. Z obou faktů plyne další nevýhoda, a to že simulační modelování a analýzy mohou být časově i finančně náročné.[6]

2.4 Dělení simulace

Simulaci obecně můžeme dělit do mnoha skupin. Jedním z možných způsobů klasifikace dělení je na:

- Účel simulace
- Reprezentace času

2.4.1 Dělení podle účelu simulace

Analytické simulace

Analytické simulace se využívají pro napodobení reálného systému nebo systému, na kterém se aktuálně pracuje z reálného světa. Slouží tak ke zjištění nových informací o simulovaném systému. Pro příklad ve vztahu k dopravě si lze analytické simulace představit jako simulaci chování dopravní sítě při nehodě, uzavření některé části sítě (uzavírka ulic) nebo přesměrování přes jinou část sítě (objízdná trasa). Zjištěné výsledky lze pak interpretovat pro vylepšení zkoumaného systému [8].

Čas při analytické simulaci většinou neběží v reálném čase. Často se pracuje s diskrétním rozdělením času (viz Kap. 2.4.2).

V rámci této práce budou data uchovávány především pro vývoj a práci v analytických simulacích.

Virtuální prostředí

Dalším typem simulace je virtuální prostředí. Tento typ se od přecházejícího liší především předpokládanou přítomností uživatele. Uživatel je zde nezbytný, protože systém na základě uživatelových vstupů vyhodnocuje situaci a upravuje tak další tok simulace. Cílem této simulace není na rozdíl od analytické simulace získání nových informací.

Virtuální prostředí obvykle pracuje v reálném čase. Díky tomuto přístupu je tak jeden z klíčových bodů minimalizace doby odezvy při průběhu simulace.

Typickým představitelem simulace ve virtuálním prostředí jsou trenažery pro výuku řízení motorového vozidla nebo výukové simulátory pro vojáky. Do této kategorie simulací též spadají počítačové hry [7].

2.4.2 Dělení podle reprezentace času

Reprezentací času v simulaci se rozumí způsob, jakým bude čas v průběhu simulace ubíhat. V jejich případě lze simulace dělit na diskrétní a spojitou.

Spojité simulace

Při spojitých simulacích je systém často popsán pomocí diferenciálních rovnic. Při běhu simulace je díky těmto rovnicím možné dopočítat aktuální hodnoty v libovolném čase simulace. Čas v simulačním modelu je opakovaně zvyšován s konstantním malým krokem a vždy se přepočítají všechny děje, které v systému probíhají.

Diskrétní simulace

Při simulacích s diskrétním rozdělením času se stav simulace mění skokově. Děje se tak buď v pravidelně přicházejících intervalech, v tom případě mluvíme o tzv. *time-stepped* (časově-krokované) simulacích, nebo se tok času mění podle toho, jak nastávají různé situace. V tomto případě se jedná o *event-driven* (událostně-řízené) simulace. Oba použité výrazy budou dále vysvětleny.

Při *time-stepped* simulacích se celkový čas rozdělujeme do stejně dlouhých časových intervalů. Po uplynutí každého intervalu jsou přepočítány všechny parametry a hodnoty funkcí, pomocí nichž je popsán stav simulace [16].

Při *event-driven* simulaci není čas daný pevnými časovými kroky, ale je řízen spuštěním událostí. Tyto události jsou naplánovány a zaznamenány v kalendáři. Každá událost tak má časovou známku a čas se mění od jedné časové známky události ke druhé [16].

3 Dopravní simulace

V následujícím textu se zaměříme na simulaci dopravy.

3.1 Dělení dopravní simulace podle úrovně detailu

Základní způsob dělení dopravních simulací se odvíjí podle míry abstrakce dopravního modelu. Nelze vytvořit pouze jeden univerzální model, který by byl využitelný pro všechny modelované případy. Proto zavádíme různou míru abstrakce modelu, a to podle rozsahu modelované sítě, míry přiblížení reálnému stavu a zobrazení detailu.[15] Na základě zmíněných kritérií dopravní simulace dělíme na:

- **Makroskopické**
- **Mesoskopické**
- **Mikroskopické**

3.1.1 Makroskopické simulace

Makroskopické modely mají největší míru abstrakce. Nejčastěji jsou využívány k modelování velmi rozsáhlých dopravních sítí. Makroskopické simulace nemají reprezentaci jednotlivých vozidel. Jejich nejmenší jednotkou bývá reprezentace silnic jako dopravního proudu, jež má své parametry. Tyto parametry mohou nabývat různých hodnot (počet průjezdů vozidel za daný časový interval, čas potřebný k průjezdu, průměrná rychlost). Vzhledem k velikosti modelované sítě je potřeba velkého množství vstupních dat. Makroskopické modely bývají využívány především pro prognostické účely [5].

3.1.2 Mesoskopické simulace

Mesoskopické simulace leží uprostřed mezi makroskopickými a mikroskopickými simulacemi. Tyto simulace již většinou reprezentují všechny entity jako mikroskopické, ale interakce mezi nimi popisují na menší úrovni detailu [8].

3.1.3 Mikroskopické

Při mikroskopických simulacích je nejmenším modelovaným objektem již samotné vozidlo. Auta se pohybují po samostatných trasách vlastní rychlostí. Jsou tak zohledněny nejen důležité vlastnosti infrastruktury ale i vlastnosti dopravních prostředků [9].

3.2 Dopravní data

V této sekci budou představena hlavní sledovaná data, která jsou často využívaná k zrealizování dopravní simulace.

3.2.1 Tok

Tok popisuje veličinu, podobně jako ve fyzice, kolik vozidel projelo daným místem za určitou jednotku času [18].

3.2.2 Střední rychlost

Tato veličina popisuje, jakou rychlostí projelo auto danou oblastí či místem. Výsledek takového měření se pak nazývá *střední rychlost*. Samotné měření je prováděno dvěma způsoby.

První způsob měří rychlost jako průměr rychlostí jednotlivých vozidel, která danou oblastí projela. Druhý způsob měří čas, který vozidlo strávilo ve sledované oblasti [18].

3.2.3 Obsazenost

Obsazenost vyjadřuje část doby, kdy vozidlo zakrývá detektor. Tato hodnota bývá uváděna v procentech [18].

3.2.4 Koncentrace

Koncentrace je definována jako počet vozidel na nějakém úseku vozovky. [18].

3.2.5 Geometrické uspořádání

Geometrické uspořádání popisuje jednotlivé prvky z reálného prostředí. Příkladem může být tvar a směr křižovatky, počet pruhů komunikace, údaje o signalizačních zařízeních a geografické umístění.

3.3 Dopravní průzkumy

Dopravní průzkumy a s nimi spojené sběry dopravních dat představují důležitý zdroj informací pro rozvoj a udržitelnost dopravy. Právě tato data mohou být použita pro dopravní simulace.

V závislosti na typu dopravního průzkumu se také používá různá metodologie a technologie. Mezi nejčastěji používané metody se řadí profilové průzkumy. Využívají se pro zjištění intenzity dopravy na konkrétním profilu komunikace s možným rozlišením vozidel na kategorie a směr jízdy.

Dalším typem jsou směrové průzkumy. Ty se pak dále dělí na křižovatkové a oblastní. Křižovatkové směrové průzkumy zjišťují míru intenzity dopravy v jednotlivých směrech. Naopak při oblastním směrovém průzkumu je cílem zjistit směřování dopravy v rámci většího územního celku [12].

3.4 Technologie sběru dat

Existuje mnoho způsobů sběru dat z dopravy. Ty nejčastěji používané budou v následujícím textu představeny, popsány a rozděleny do 3 kategorií podle technologického zpracování.

3.4.1 Ruční sběr

Jedná se o nejjednodušší a nejčastěji používaný způsob sběru dopravních dat. Probíhá tak, že pracovník, jenž provádí sčítání, ručně zapisuje své pozorování do papírového záznamového archu nebo mobilní aplikace.

Bohužel tento způsob sběru je často zatížen lidskou chybou. Čítač (člověk) podléhá stresu, únavě a svým biologickým potřebám. Navíc z personálního pohledu nejsou snadno realizovatelné. Na jednopruhové komunikace je zapotřebí jednu osobu na každý jízdní pruh. V případně velkého dopravního zatížení ještě více. Další pracovní sílu je třeba v případě využití papírových archů pro přepis výsledků měření do digitální formy.

Mezi výhody patří relativně nízké náklady při krátkodobém pozorování ve srovnání s pořizovací cenou technických prostředků. V případě méně zatížených křižovatek zvládne pozorování jedna osoba.

3.4.2 Detektory zasahující do vozovky

Detektory zasahující do vozovky jsou detektory, které přímo zasahují do vozovky. Jejich instalace vždy omezí provoz a některé z nich nejsou přenosné. Liší se především svojí pořizovací cenou, množstvím zaznamenávaných dat,

možností mobility a přesností. Lze je využít jak pro řízení světelné křižovatky, tak pro sčítání vozidel. Při umístění více čidel za sebou lze detekovat obsazenost komunikace i rychlost projíždějících vozidel [13]. Typickými představiteli jsou:

Indukční smyčka

Zařízení slouží k zjištění přítomnosti vozidla. Princip jejího fungování spočívá v tom, že během průjezdu či přítomnosti vozidla na smyčce dochází ke snížení její indukčnosti, a tím se zvyšuje frekvence oscilátoru. V případě, že změna frekvence dosáhne předem daného prahu, je tato změna vnímána jako přítomnost automobilu na smyčce. Smyčka se instaluje pod povrch vozovky [13].

Magnetický detektor

Toto zařízení pracuje na principu měření hustoty siločar magnetického pole Země. Kovová masa projíždějícího vozidla zvýší v prostoru senzoru hustotu siločar. Na základě toho může přístroj detekovat přítomnost vozidla. Magnetický detektor je instalován pod porvrch vozovky [13].

Pneumatický detektor

Jedná se o první typ automobilového detektoru. Pracuje na principu měření změny tlaku v trubce položené na vozovce, přes kterou projíždějí auta. Jeho velkou výhodou je snadná instalace, mobilita i pořizovací cena. Naopak nevýhoda spočívá ve špatné možnosti rozpoznat mezi stojícím a pomalu se pohybujícím vozidlem [13].

3.4.3 Detektory nezasahující do vozovky

Tato zařízení představují opak pro předcházející detektory. Dají se instalovat bez zásahu do vozovky a stávají se tak snadno odstranitelnými. Princip jejich fungování je bezdotykový. Existuje několik druhů:

Pasivní detektor hluku (zvuku)

Pracuje na principu detekce akustické energie (hluku) evokované projíždějícím vozidlem. Je schopný detekovat přítomnost, průjezd a rychlost vozidla [14].

Ultrazvukový detektor

Využívá principu zvukových vln. Detektor v pravidelných intervalech vyšle vlnu a měří čas, za který se odražená vlna vrátí zpátky k detektoru. Je-li tento čas jiný, než který odpovídá vzdálenosti k povrchu vozovky a zpět, je vyhodnocen jako přítomnost vozidla. Ultrazvukový detektor umí měřit počet, přítomnost, obsazenost, rychlost, délku a výšku vozidel [14].

Aktivní infračervený detektor

Funguje na stejném principu jako předcházející detektor. Využívá však infračervené záření namísto ultrazvuku [14].

Pasivní infračervený detektor

Snímá tepelné záření, které vyzařuje každý objekt, jenž nedosahuje teploty absolutní nuly (0 K, -273.149 °C). Na základě rozdílných teplot dokáže rozhodnout, jestli zaměřil vozidlo nebo vozovku [14].

Video-detekce (zpracování obrazu)

Tyto detektory využívají k detekci počítačové vidění. To může být založené na detekci tvarů v obraze (např. auta, chodci) nebo detekci pohybu [14].

4 Rozšířitelná Java Aplikace

Tato kapitola bude zaměřená na možnosti přidání nové funkcionality do již stávající aplikace bez její úpravy.

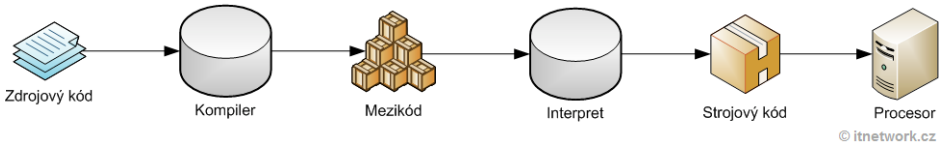
4.1 Java

Java je objektově orientovaný jazyk, který byl vyvinut firmou Sun Microsystems, a ta ho představila 23. května 1995. Od té doby se stal jedním z nejpoužívanějších programovacích jazyků na světě [29].

Jedním z důvodů tohoto úspěchu je skutečnost, že se jedná o jazyk pracující nad virtuálním strojem. Spojuje tak výhody jazyků interpretovaných a kompilovaných.

Zdrojový kód aplikace se nejprve přeloží (kompiluje) do mezikódu (tzv. bajtkód /byte code), který je velmi podobný strojovému jazyku, ale má mnohem jednodušší instrukční sadu a navíc přímo podporuje objektové programování. Následně je bajtkód interpretován do strojového kódu, který je již spustitelný na procesoru. V případě Javy se takovému interpretu říká Java Virtual Machine (JVM) [25].

Přidáním mezikroku v podobě bajtkódu přináší výhodu přenositelnosti aplikace. Je-li jednou aplikace přeložena do bajtkódu, může být spuštěna nad kterýmkoliv interpretem, který je již vytvořen pro danou počítačovou architekturu [25]. Postup od zdrojového kódu až k procesoru je znázorněn na Obr. 4.1.



Obrázek 4.1: Od zdrojového kódu k procesoru. Zdroj [24]

4.1.1 Rozšiřitelnost Java aplikace

Některé Java aplikace jsou distribuovány jako spustitelný bajtkód bez dostupných zdrojových kódů. Nastane-li situace kdy je po stávající aplikaci vyžadována nová funkcionalita, musel by ji přidat pouze autor aplikace. Jiné

autor vyvíjí jako open-source (otevřený software) a kdokoliv tak může nahlédnout do toho, jak přesně fungují. V tomto případě pak může přidat novou funkci kdokoliv, kdo by prostudoval celý zdrojový kód aplikace a porozuměl mu.

Z posledních uvedených vět by se mohlo zdát, že přidání další funkcionality do již stávající Java aplikace, je obtížný úkol. To by bylo však velké omezení a znesnadnilo by to vývoj mnoha aplikací. Naštěstí během let byly vyvinuty mechanizmy, jak rozšířit již stávající aplikace o další funkčnost bez nutnosti změny a následné kompilace původní aplikace.

Samotná Java nabízí pouze prostředky pro rozdělování do metod, z nichž se následně skládají třídy, které se sdružují do balíků, tzv. package. Package se po sloučení zabalí do `.jar` souboru v němž jsou distribuovány. Soubor `.jar` se obecně považuje za jednotku jen do chvíle, kdy je přidán na `classpath`. `Classpath` je v prostředí Javy označení pro nastavení cest, ve kterých se JVM (Java Virtual Machine) nebo Java překladač snaží nalézt požadované třídy v souborech typu jar nebo class [27]. Soubor `.jar` je tak jen jakýsi archiv a vytváří obal pro třídy, rozhraní a zdroje, které obsahuje. Po umístění na classpath mizí veškeré ohraničení a všechny veřejné třídy jsou dostupné v rámci běžícího programu.

Před dalším pokračováním je třeba si představit a vysvětlit některé možné typy softwarové architektury.

- Monolitická – u této architektury neexistuje žádné vnější rozhraní a software funguje jako jeden celek, jehož části jsou neoddělitelné.
- Modulární – tato architektura vychází z rozdělení softwaru na jednotlivé moduly, kdy každý modul vykonává určitou konkrétní část programu [41].
- Využívající pluginy – aplikace s touto architekturou využívá pluginů. V této architektuře existuje hlavní hostitelská aplikace, do které jsou dodávány nové funkce jako pluginy.

Právě poslední architektura využívající pluginy přináší možnost doplnit další funkce do již existujícího softwaru bez žádných úprav toho stávajícího, a proto další text bude věnován právě pluginům.

4.2 Plugin

Když programátor vyvíjí novou aplikaci, dostane vypracované zadání od zadavatele nebo si ho vypracuje sám. Toto zadání obvykle bývá analýza daného problému, jež by měla aplikace řešit. Obsahuje též i funkce, které jsou

od daného programu vyžadovány. Pokud se programátor nebo kdokoliv, kdo analyzoval daný problém, pokusí popsat všechny možné případy užití, nelze vyloučit, že jednou nenastane situace, kdy daný program nebude obsahovat funkci, která bude potřeba. Neznamená to, že by původní analýza byla špatná, ale mohly se pouze změnit požadavky na aplikaci. Vzniká tak potřeba doplnit nějakou funkci. To je právě případ, který se dá vyřešit pomocí pluginů [21].

Dalším důvodem, proč by programátor měl zvážit možnosti využití pluginů ve svém programu, je fakt, že v dnešní době již není vývoj monolitických aplikací příliš častý. Taková aplikace je náročná na údržbu. Každá změna bývá obtížná a není jednoduché přidat další funkčnost bez narušení těch stávajících. Místo toho jsou aplikace skládány z menších částí, jež propojuje hostitelská aplikace. Takto definovaná aplikace je pak snazší na údržbu, testování a další rozšiřování [21].

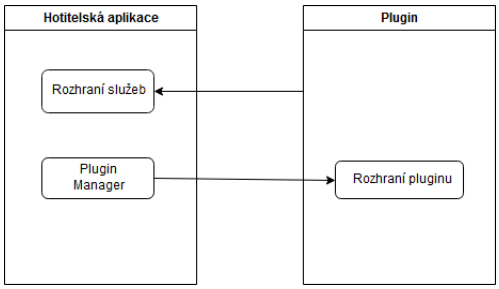
4.2.1 Mechanizmus pluginu

Dospěje-li programátor k názoru, že v budoucnu bude třeba do jeho aplikace přidávat další funkce, potřebuje si připravit nástroje/prostředí jakým tento nový kód bude komunikovat s jeho hostitelskou aplikací. Právě toto komunikační rozhraní se nazývá API nebo-li Application Programming Interface. Je to rozhraní, pomocí kterého si hostitelská aplikace a plugin vyměňují data, případně má plugin skrze API možnost využít služby hostitelské aplikace [22].

Plugin jako samostatná jednotka obvykle nepracuje sám, ale potřebuje pro svou funkci hostitelskou aplikaci. Na rozdíl od toho hostitelská aplikace je často schopna fungovat samostatně. Pro samotnou aplikaci může být implementace API dobrým způsobem, jak prodloužit dobu její životaschopnosti. Protože jak se budou měnit nároky na její funkce, bude je možné dodávat ve formě pluginů. Navíc stabilní otevřené API dovoluje vytvářet pluginy i třetím stranám a umožňuje tak další vývoj aplikace [21]. Příklad možné komunikace mezi pluginem a hostitelskou aplikací je na obrázku 4.2.

4.2.2 Výhody pluginů

Výhody využití pluginů v aplikaci byly již několikrát nepřímo zmíněny, ale zde se je zkusím sepsat podrobněji. První vychází již z jejich samotné podstaty. Pluginy jsou měnitelné. Mají-li dobře navržené API, nebývá problém vyměnit kus za kus, který může být lépe implementován a zvýší tak výkon daného programu. Další výhodou je bezpochyby fakt, že jednotlivé pluginy



Obrázek 4.2: Vztah hostitelské aplikace a pluginu.

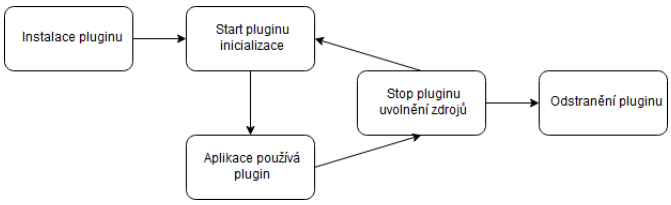
jsou mnohem menší části vykonávající jen danou činnost a jsou tak mnohem jednodušší na pochopení [22].

Dalším přínosem použití pluginů je, že jsou na sobě i na aplikaci nezávislé a lze je tím vyvíjet paralelně. V neposlední řadě je to již zmíněné prodloužení životnosti aplikace, protože je možné do ní neustále přidávat nové funkce [22].

4.2.3 Životní cyklus pluginu

Vzhledem k tomu, že plugin je jakási malá aplikace sama o sobě, ačkoliv samostatně zůstává nespustitelná, musí mít implementován životní cyklus. Aplikace přesně neví, co všechno plugin potřebuje inicializovat pro svoji funkci předtím, než může začít hostitelské aplikaci poskytovat služby. Naopak plugin nikdy neví, kdy ho aplikace začne nebo přestane používat. Může tak uvolnit zdroje, o které si řekl během své inicializace.

Jako příklad může posloužit plugin s funkcí čtení ze souboru. Na začátku musí hostitelská aplikace inicializovat plugin a předat mu informaci, kde se soubor nachází. Následně plugin soubor najde a pokusí se ho otevřít. Poté je připraven ze souboru číst informace, které hostitelský program požaduje. Pro ukončení čtení je třeba plugin informovat, že již není třeba jeho služeb. Ten poté může soubor uzavřít a uvolnit zdroje, jež si alokoval pro svoji práci. Možný životní cyklus takového pluginu je znázorněn na obrázku 4.3.



Obrázek 4.3: Obecný životní cyklus pluginu.

4.3 Správa pluginů

Má-li aplikace podporovat systém pluginů, musí mít nástroj, který je bude spravovat. Bude je schopný odebírat, přidávat, pracovat s nimi a v neposlední řadě i komunikovat. Sama o sobě Java obsahuje pouze reflexi, která umožňuje dynamické načítání tříd [26]. Není to přímá podpora pluginů, ale nástroj, který dovoluje vytvořit prostředí pro práci s nimi.

Jedno z těchto prostředí se pokusím dále představit.

4.3.1 OSGi

Frameworků podporující pluginovou architekturu je mnoho např. Java Plugin Framework[1], pf4j[2] nebo OSGi[3]. Fungování posledního zmíněného se pokusím v následujícím textu představit jako příklad.

OSGi je komponentový model, jež má několik implementací a jeho implementacím se říká OSGi framework. Důvodem vzniku bylo, že pokud programátor vyvíjí modulární aplikaci, musí si zvolit, v jakém runtime prostředí budou jeho moduly existovat a skládat tak celou aplikaci. Samozřejmě je možné, aby programátor takové prostředí vyvinul sám, ale není to příliš efektivní. Navíc pokud by chtěl využít nějaký modul případně komponentu třetích stran, pravděpodobně by byla nekompatibilní. Toho se OSGi specifikace snaží vyvarovat.

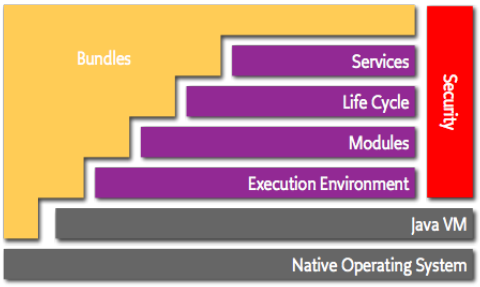
Zvolí-li vývojář nějakou implementaci OSGi specifikace, má takové prostředí, které je kompatibilní se všemi jejími jinými implementacemi.

Architektura

OSGi má vrstvený architektonický model, který je znázorněn na následujícím obrázku 4.4.

- Bundles – základní jednotkou modularity je v OSGi Bundle. Je to skupina tříd zabalená do archivu `.jar`. Standartní součástí archivu `.jar` bývá soubor s názvem `manifest.mf`, který obsahuje dodatečné informace o obsahu archivu. Toho využívá i OSGi, aby s takovým archivem mohlo pracovat. Jsou zde například uvedeny informace o názvu pluginu, verzi a závislostech, které plugin potřebuje pro svůj běh atd. [28].

[1] http://jpf.sourceforge.net/
[2] https://github.com/pf4j/pf4j
[3] https://www.osgi.org/



Obrázek 4.4: Architektura specifikace OSGi.Zdroj [28]

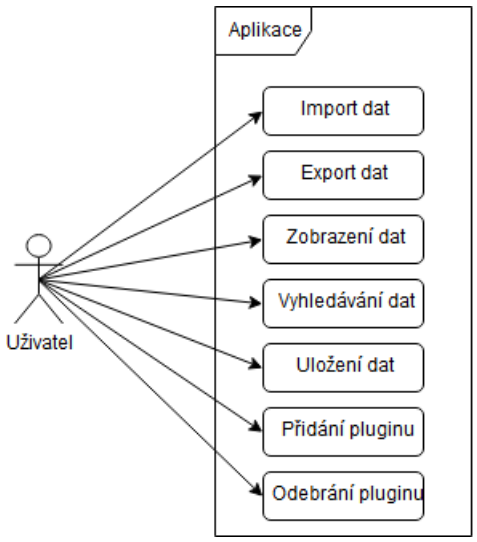
- Life cycle – life cycle nebo-li životní cyklus je vrstva zodpovědná za správu jednotlivých bundlů. Life cycle určuje, v jakém stavu se nachází konkrétní bundle. Díky této vrstvě mohou být bundly dynamicky přidávány/odebírány, spouštěny/zastavovány nebo upravovány [28].
- Services – importy a exporty balíčků v Javě definují statické závislosti mezi jednotlivými moduly. Proto OSGi obsahuje službu service. Ta umožňuje bundlům vytvořit a zaregistrovat objekt pod specifickým rozhraním. Následně si může jiný bundle tento objekt z registrů OSGi vyzvednout [28].
- Modules – protože jednotlivé *Bundle* v OSGi představují archivy `.jar`, byl by obsah těchto balíků ve standartní Java viditelný všem. OSGi proto tyto `.jar` archivy reprezentuje jako moduly, jež musí explicitně definovat závislosti, které chce importovat. Takové chování má mnoho výhod např. dovoluje využití několika verzí té samé knihovny na tom samém JVM [28].
- Security – tato vrstva má na starosti aspekty zabezpečení. Neboli řeší omezení funčnosti bundle na předem definované funkce [28].
- Execution enviroment – úroveň definující metody a třídy, které budou dostupné na konkrétní platformě. Definuje, na jaké platformě může daný Bundle pracovat [23].

5 Analýza

Tato kapitola se bude zabývat analýzou požadavků na aplikaci.

5.1 Případy užití

Diagram 5.1 ukazuje, jak bude uživatel s aplikací pracovat a jaké funkce by měla aplikace podporovat.



Obrázek 5.1: Případy užití aplikace

5.2 Popis případů užití

5.2.1 Import dat

Data, která bude aplikace zpracovávat, budou pocházet z různých nesourodých zdrojů. Proto musí podporovat variabilní import. Tato množina typů dat se může kdykoliv změnit. Je nutné vytvořit mechanizmus, který bude umožňovat rozšíření množství podporovaných formátů dat.

5.2.2 Export dat

Data z aplikace budou využívána k různým účelům a nelze předem definovat formát dat, ve kterém by měla být exportována. Proto je nutné umožnit

variabilní export dat.

5.2.3 Zobrazení a vyhledávání dat

Aplikace musí podporovat náhled na data, která spravuje. Tento pohled by měl umožnit omezit množinu zobrazených dat. Současně by aplikace měla umět v těchto datech podporovat vyhledávání a v důsledku zprostředkovat uživateli možnost exportovat pouze jím vybraná data.

5.2.4 Uložení dat

Aplikace bude pracovat s velkým množstvím dat o dopravě. Výsledný program bude muset tyto data také uchovávat. Nebude tak činit pouze po dobu spuštění programu, ale také mezi jednotlivými běhy programu.

5.2.5 Přidání a odebrání pluginu

Aby mohla aplikace podporovat variabilní export i import skrze pluginy, musí vzniknout mechanizmus, jak tyto pluginy do aplikace přidat resp. odebrat.

5.3 Analýza ukládaných dat

Z první části této práce vyplývá, že existuje mnoho typů dat o dopravě. Každé čidlo poskytuje trochu jinou množinu dat a zároveň nikdy ne všechny možné údaje. Tato množina se navíc může kdykoliv měnit podle toho, jak se budou měnit nároky na sledování dopravy. Není proto možné dopředu specifikovat všechny možné atributy, které by měly být ukládány.

Ačkoliv není možné specifikovat všechny sledované atributy, můžeme určit alespoň některé společné.

Při sbírání dat o dopravě sleduje vždy čítač (stroj nebo člověk) konkrétní místo nebo úsek. K datům se také připojuje nějaká časová značka. Tato značka může být jeden konkrétní čas měření nebo časový interval, po který měření probíhalo. Z toho plyne, že jedinými společnými atributy jsou čas nebo časový interval měření a označení místa měření. Tohoto faktu by mělo být využito při návrhu databáze.

5.4 Variabilní export a import

Podle zadání má aplikace umožnit variabilní import i export dat. Zároveň by tato variabilita neměla být omezena pouze neměnným výčtem implementovaných možností. Musí existovat možnost přidat další způsob exportu nebo importu bez úpravy stávající aplikace. Toho lze docílit využitím pluginů viz Kap. 4.2.

Tyto pluginy musí být možné přidávat do aplikace, která s nimi pracuje a umí je využívat. Budoucí aplikace by je měla být schopna rozpoznat a určit, jestli je lze použít na import nebo export dat.

5.5 Analýza GUI

Pro snadné použití by měla být aplikace ovládána skrze grafické uživatelské prostředí. Toto prostředí musí podporovat všechny operace, které jsou od aplikace vyžadovány jako např. umožnit náhled nad spravovanými daty nebo definovat export/import.

Z důvodu zpracovávání velkého množství dat je třeba, aby aplikace umožňovala uživateli paralelní práci. To znamená souběžný import dat a jejich ukládání z více zdrojů zároveň, definování více pohledů na data a paralelní export spravovaných dat.

Zároveň je třeba zajistit, že i při 2 současných zápisech do databáze nebude porušena integrita databáze. Navíc poběží-li databáze na stejném fyzickém disku jako aplikace, může vést takový paralelní zápis ke zpomalení ukládání dat.

6 Technologie

V této kapitole jsou popsány použité technologie.

6.1 Návrhové vzory

6.1.1 MVC

MVC nebo-li model-view-controller je návrhový vzor, který rozděluje aplikaci do tří hlavních částí. MVC bylo použito pro celkovou architekturu aplikace.

Model

Model označuje část programu, v níž je ukryta logika programu např. napojení na databázi nebo načítání dat ze souboru [30].

View

View (v předkladu pohled) převádí data reprezentovaná modelem do podoby vhodné k reprezentaci [30].

Controller

Controller (v překladu řadič) propojuje vrstvy model a view. Umožňuje tím jejich komunikaci [30].

6.1.2 Visitor

Návrhový vzor *visitor* umožňuje pro danou skupinu tříd dynamicky definovat nové operace, aniž by bylo nutné tyto třídy jakkoliv modifikovat. *Visitor* se řadí mezi návrhové vzory, jenž ovlivňují chování tříd a jejich instancí. Tento návrhový vzor aplikace využívá na několika místech např. pro generování databázových dotazů a parsování jejich výsledků.

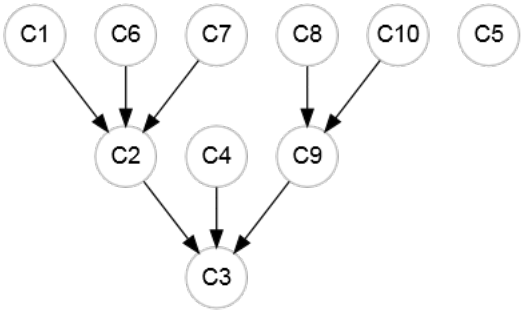
Princip

Pro každou novou akci, kterou chceme dodat původní množině tříd, vytvoříme novou třídu. Tato nová třída představuje *návštěvníka*. Instanci tohoto

návštěvníka pak předáme původní třídě a ta sama na sobě zavolá odpovídající metodu *návštěvníka*. Původní třída tedy představuje *navštíveného*. Jinými slovy návštěvník umí vykonat novou akci. *Navštívený* ho přijme a nechá ho se sebou vykonat onu novou akci. Platí, že kolik bude dodatečných akcí, tolik bude návštěvnických tříd [32].

6.1.3 Dependency injection

V objektově orientovaném programování se program skládá z jednotlivých objektů, které jsou mezi sebou propojeny. Pokud objekt C2 vyžaduje pro svou činnost objekt C1, říkáme, že objekt C2 je závislý na objektu C1. Lze si tak celý objektově orientovaný model reprezentovat jako orientovaný graf. Uzly tohoto grafu pak budou jednotlivé objekty a hrany závislosti [33] viz Obr. 6.1.



Obrázek 6.1: Struktura programu se závislostmi. Zdroj [33]

Při takovém rozložení je třeba určit, jaké konkrétní instance závislostí se použijí a kdo bude odpovědný za jejich vytváření. Nabízí se dvě možnosti. V prvním případě si o těchto závislostech objekt může rozhodnout sám. Vytvoří si v konstruktoru všechny potřebné závislosti. Výhoda takového přístupu spočívá v tom, že objekt lze použít jako samostatnou komponentu. Naopak nevýhodou zůstává, že jeho závislosti nelze použít nikde jinde a nelze je snadno změnit[33].

Druhá možnost znamená předání závislostí objektu z vnějšku. Tento způsob se nazývá dependency injection. V tomto případě může objekt spoléhat na to, že předaná závislost bude implementovat jen určité rozhraní a nemusí se starat, o jakou instanci se jedná. Takový způsob se hodí v případě testů, kdy můžeme objektu předat pouze simulaci skutečných implementací. Lze tak objekt otestovat ještě dříve, než vznikne implementace dané závislosti [33].

Pro tyto, ale i další důvody, bylo DI použito při vývoji aplikace.

6.2 Nástroje

6.2.1 Programovací jazyk

Dle zadání bude k vývoji této aplikace použit jazyk Java. Všechny nadále zmíněné a použité technologie budou kompatibilní právě s tímto jazykem.

6.2.2 Grafické uživatelské prostředí

Grafické prostředí programu je vytvořeno pomocí frameworku JavaFx. Jde o moderní framework pro tvorbu vizuálně bohatých okenních aplikací. JavaFX přináší podporu obrázků, videa, hudby, grafů, CSS stylů a dalších technologií. Důraz je kladen na jednoduchost tvorby. JavaFX se hodí jak pro desktopové aplikace, tak pro webové applety nebo mobilní aplikace [34].

JavaFX dovoluje použití nástroje JavaFX Scene Builder, pomocí něhož lze navrhovat uživatelské prostředí. Výstupem tohoto nástroje je FXML [34].

FXML

Jazyk FXML je založen na XML. Dovoluje definovat strukturu uživatelského prostředí. JavaFX umí tuto strukturu následně přečíst a vytvořit jednotlivé komponenty [35].

6.2.3 Správa projektu

Pro správu projektu a jeho sestavování je použitý nástroj Maven. Pomocí tohoto nástroje lze popsat, jak se má výsledná aplikace sestavit a zároveň definovat závislosti na ostatních projektech. To umožní použití externích knihoven v projektu [36].

6.2.4 Google Guice

Guice je dependecy injection (DI) framework s otevřeným kódem vyvinutým ve společnosti Google. Prezentuje se jako lehký (lightweight) framework pro verzi Java 6 a vyšší. Vznikl v roce 2008 a byl první DI framework, který používal Java anotace [37].

7 Implementace

7.1 Jazyk zdrojového kódu

Aplikace byla vyvinuta jako open-source a její zdrojové kódy jsou umístěny na Gitlabu[1] ve veřejném repozitáři. Z toho důvodu existuje kód, komentáře v kódu i samotná aplikace pouze v anglickém jazyce.

7.2 Databáze

Z analýzy dat v Kap 5.3 vyplývá, že ačkoliv databáze bude muset uchovávat různorodá data, mají přeci jen něco společného. Tento společný průnik představuje *čas* a *místo*. Ty společně tvoří první dvě entity databáze, ve které je reprezentují tabulky *Time* a *Location*.

Vztah mezi těmito dvěma entitami je *M:N*, neboli k jednomu místu mohou existovat údaje z různého času a zároveň pro jeden čas lze mít údaje z více míst. Databázový model však neumí s takovou vazbou pracovat. Vznikla proto tabulka s názvem *TrafficData*, jež tento vztah rozkládá.

Databáze musí umět uchovávat i naměřené hodnoty. Problém nastává v tom, že množina typů/druhů těchto hodnot není předem známa a může se měnit. Přestavíme-li si entitu *TrafficData* jako záznam jednoho měření, tak k tomuto měření bude náležet pravě N typů hodnot. Z tohoto vztahu pak vzniká další entita *Type*. Ten představuje jednotlivé měřené hodnoty jako rychlost, hustota provozu apod. V databázi tuto entitu reprezentuje tabulka *Type*.

Máme-li definované jedno měření a typy dat, které obsahuje, musíme ještě zaznamenat naměřenou hodnotu. *Hodnota* představuje poslední entitu databáze. V databázi ji představuje tabulka *Value*. Tato entita rozkládá vazbu *M:N* mezi entitami *TrafficData* a *Type*. Jedno měření obsahuje N typů dat a jeden typ dat náleží k M měření.

7.2.1 Popis atributů tabulek

Samotné entity mají své vlastní atributy, které dále specifikují jich vlastnosti. Záměrně v popisu chybí atributy využívané databází pro primární a cizí klíče, které používá databáze pouze pro svou funkci.

[1] https://gitlab.com/jokertwo/traficcontentmanagment

Time

Tato entita obsahuje atribut *Begin* (datový typ DATETIME), který označuje začátek měření a atribut *End* (datový typ DATETIME), který uvádí konec měření. Tyto dva atributy dovolují uložit informaci o tom, že měření probíhalo v intervalu.

Location

Tato entita má atribut *Place* (datový typ TEXT) pro identifikaci místa, kde měření probíhalo a atribut *Direction* (datový typ TEXT) pro upřesňující identifikaci místa.

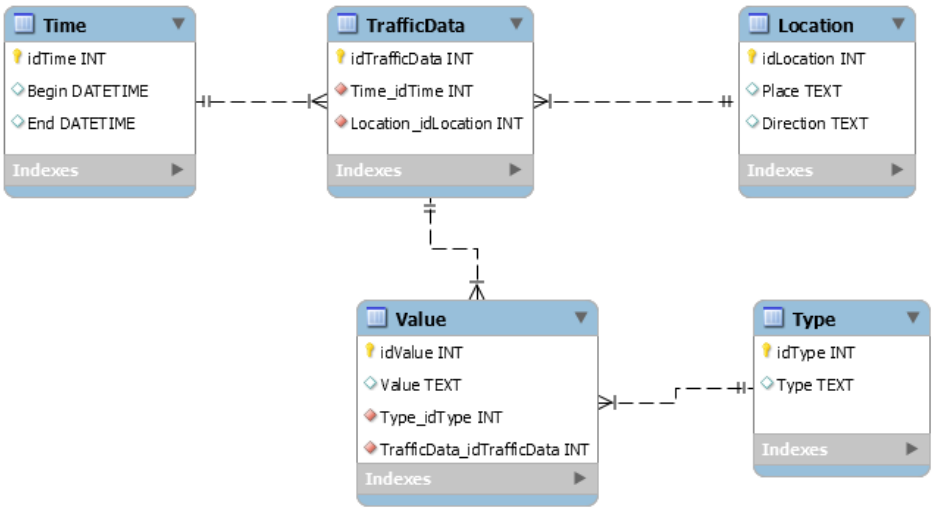
Type

Entita Type vlastní pouze stejnojmenný atribut *Type* (datový typ TEXT) označující typ dat.

Value

Stejnojmenný atribut *Value* (datový typ TEXT) entity Value reprezentuje naměřené hodnoty.

7.2.2 ERA model databáze



Obrázek 7.1: ERA model databáze

7.3 Plugin

Aplikace podle zadání musí umožňovat podporu dodatečných pluginů, které budou využívány pro import a export dat. Aby toto bylo možné, musí existovat rozhraní, které umožní komunikaci aplikace s takovým modulem. Dále je nutné vyvinout službu, která bude s těmito pluginy schopna pracovat.

Jedním z takových možných řešení by bylo OSGi, které bylo představeno v kapitole 4.3.1. Toto řešení ale nebylo nakonec použito. OSGi framwork je příliš komplexní služba přesahující potřeby této aplikace, a proto bylo implementováno vlastní řešení, které však přebírá některé vlastnosti od OSGi.

7.3.1 Společné vlastnosti pluginů

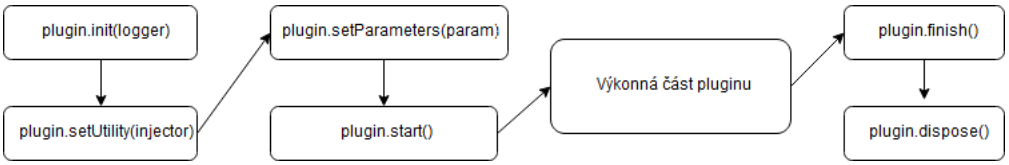
Ze zadání vyplývá, že musí existovat dva druhy pluginů (jeden pro export a druhý pro import). I přes tuto skutečnost budou mít některé části společné např. název pluginu (jak se bude plugin v aplikaci reprezentovat), popis jeho vlastností a v neposlední řadě i jeho životní cyklus.

V jazyce Java mohou rozhraní dědit vlastnosti svých předků. Na tomto principu vzniklo rozhraní `Plugin`. Od tohoto rozhraní pluginy pro export a import dědí vlastnosti. Rozhraní `Plugin` tak může definovat části, jež jsou společné pro oba druhy pluginů.

7.3.2 Životní cyklus

Životní cyklus pro oba typy pluginů je téměř totožný a liší se jen v malé výkonné části. Plugin nejdříve iniciuje metoda `Plugin.init()`. Následně mu správce pluginů předá objekt, ze kterého může získat jednotlivé služby hostitelské aplikace (metoda `Plugin.setUtility()`). Další krok nastává předáním parametrů od uživatele (metoda `Plugin.setParameters()`). Poslední akcí před výkonnou částí se stává volání metody `Plugin.start()`. Poté již následuje výkonná část pluginu, která bude popsána u jednotlivých pluginů samostatně.

Po ukončení výkonné části následuje volání metody `Plugin.finish()`. Poslední krok představuje zavolání metody `Plugin.dispose()` pro uvolnění všech zdrojů, které si plugin alokoval pro svoji práci. Společný životní cyklus pluginu viz Obr. 7.2.



Obrázek 7.2: Obecný životní cyklus pluginu

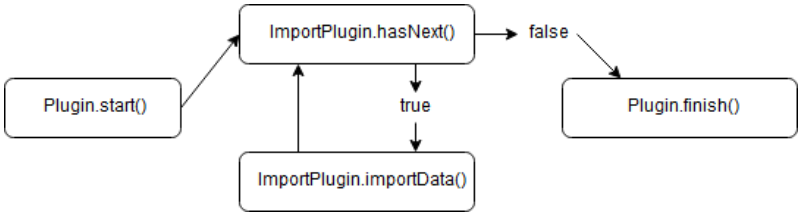
7.3.3 Import plugin

Import plugin a jeho rozhraní `ImportPlugin` má za úkol získání zdrojů a jejich předání aplikaci. Ta dopředu nikdy neví, kolik toho ještě pluginu zbývá zpracovat, a proto si nemůže být jistá, zda smí ukončit práci s pluginem. Řešením by mohlo být, že by plugin naplnil nějakou kolekci daty a tuto kolekci by pak předal najednou aplikaci. Tento přístup má však omezení v případě, že by plugin zpracovával data, jež jsou vetší než dostupná paměť, do které by je ukládal. V takovém případě by pravděpodobně následoval pád aplikace. Ta by se tím stala v podstatě nepoužitelná.

Implementovaný přístup, který řeší tento problém, byl inspirován kolekcemi. Konkrétní předlohou se staly metody `next()` a `hasNext()` z rozhraní `Iterator`. Metoda `hasNext()` vrací logickou proměnnou `true/false` podle toho, jestli existuje další prvek, který by metoda `next()` měla vrátit [38]. Obdobu těchto dvou metod nalezneme v rozhraní `ImportPlugin`.

Ekvivalentem pro metodu `Iterator.hasNext()` je stejnojmenná metoda `ImportPlugin.hasNext()`. Metoda vrací `true`, dokud může plugin poskytovat další data. V opačném případě vrací `false`. Obdobou metody `Iterator.next()` je metoda `ImportPlugin.importData()`, která vrací strukturu dat, s níž se dále v aplikaci pracuje.

Diagram výkonné části import pluginu viz Obr.7.3.

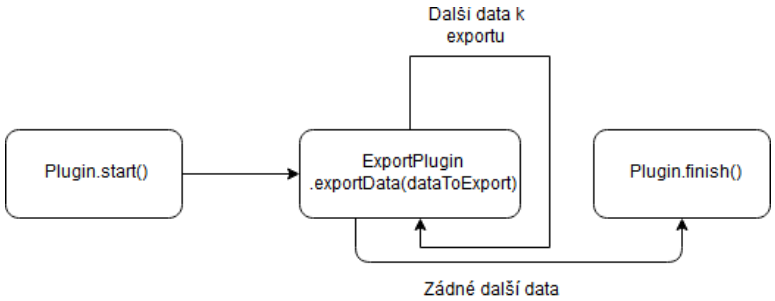


Obrázek 7.3: Výkonná část `ImportPlugin`

7.3.4 Export plugin

Plugin pro export dat má své rozhraní `ExportPlugin`. U tohoto pluginu je situace s výkonnou částí o poznání snazší. Aplikace ví, kolik dat bude skrze

plugin exportovat. Postačí jí zavolat metodu `Plugin.start()`, a následně opakovaně volat metodu `ExportPlugin.exportData()`, které jsou předány v parametru data, jež má plugin exportovat. Pro ukončení výkonné části se volá metoda `Plugin.finish()` a dále se pokračuje ve standardním životním cyklu. Diagram výkonné části export pluginu viz Obr. 7.4.



Obrázek 7.4: Výkonná část `ExportPlugin`

7.3.5 Použití pluginu aplikací

Pro použití pluginu musí mít aplikace mechanizmus, který ho umí nahrát. To umožní pouze pluginy, u kterých je dodržena tato jednotná forma.

Plugin musí implementovat právě jedno z možných rozhraní (`Import-Plugin` nebo `ExportPlugin`). Autor nového pluginu bude muset pravděpodobně implementovat i další třídy. V Kap 4.1 bylo uvedeno, že více tříd může být zabaleno do `.jar` archivu, který lze distribuovat jako jeden celek. Případnému autorovi nového pluginu tak stačí pouze implementovat jedno z možných rozhraní pluginu. Může si k tomu vytvořit libovolný počet pomocných tříd a zabalit všechny tyto implementace do jednoho `.jar` archivu.

Při načítání takového archivu by bylo obtížné zjistit, která ze tříd obsažená v `.jar` souboru implementuje rozhraní pluginu. Tento problém řeší uložení této informace do souboru `manifest.mf`, který je součástí `.jar`. Jedná se o soubor s doplňujícími informacemi o daném archivu. Tato informace se ukládá jako dvojice klíč a hodnota. Klíčem je *Plugin-Class* a jeho hodnotu tvoří plnohodnotná cesta ke třídě, která implementuje jedno z rozhraních pluginů.

8 Struktura aplikace

Jak již bylo zmíněno v Kap 6.1.1, aplikace se dělí na tři hlavní části – model, view a controller. U všech těchto částí budou popsány nejdůležitější části kódu.

8.1 View

Tato část aplikace má na starosti samotnou grafickou reprezentaci. Protože pro vývoj grafické části aplikace byla zvolena JavaFx a nástroj Scene Builder, vznikly všechny soubory reprezentující view aplikace ve formátu `.fxml`.

8.1.1 Koncept

Jelikož aplikace by měla podporovat paralelní práci na několika úkolech současně, byl zvolen koncept jednotlivých záložek. Při spuštění aplikace se zobrazí prázdné okno aplikace. Do tohoto okna jsou pomocí aplikačního menu vkládány různé záložky. Obsah záložek je uzpůsoben jejich účelu. To dovoluje spustit dvě dlouho trvající akce zároveň např. dva importy do databáze současně (pro každý import jedna záložka). Avšak taková akce zpomalí zápis do databáze. Kvůli udržení integrity databáze je aplikace navržena tak, že do ní může zapisovat pouze jedno vlákno. Takže v případě spuštění dvou současných importů se takové akce budou střídavě dělit o přístup k databázi. Avšak to nebrání uživateli tyto akce spustit a naplánovat dvě dlouhotrvající akce, které mohou běžet bez jeho přítomnosti.

8.1.2 Umístění souborů

Jako nástroj pro sestavování aplikace byl zvolen Maven, proto jsou všechny tyto soubory `.fxml` uloženy v resource (zdrojích) projektu. Konkrétní cesta je `src/main/resources/FXML`. V podsložce `src/main/resources/FXML/-setting` se nachází soubory s grafickou reprezentací aplikace.

8.2 Model

Model se stará o logickou vrstvu aplikace. Patří sem práce s databází, práce s pluginy a distribuce událostí napříč aplikací. Konkrétně se jedná o balíky

(dále jako package) `core`, `database` a `logger` včetně všech jejich podbalíků (subpackage).

Všechny níže uvedené package jsou v projektu obsaženy v nadřazeném package `my.bak.trafic`. Pro zjednodušení a přehlednost nebude tato předpona již nadále psána.

8.2.1 Package .logger

Tento package definuje dvě třídy `Log4JMembersInjector` a `Log4JTypeListener` i jednu anotaci `InjectLogger`.

Pomocí této anotace muže být injektován logger do libovolné instance spravované pomocí Google Guice 6.2.4. Není tak nutné explicitně vytvářet logger v každé takové třídě, ale stačí pouze uvést tuto anotaci nad proměnnou, do které má být logger injektován.

Zbylé dvě třídy jsou již implementace toho, jak má být logger injektován.

8.2.2 Package .core

Package core zastřešuje několik částí programu. Sám o sobě obsahuje třídu `GuiceFXMLLoader`, která vznikla jako potomek `FXMLLoader`. Tato třída zodpovídá za načítání FXML souborů a jejich parsování. Tento potomek byl vytvořen, aby controller 8.3 vznikal pomocí Google Guice 6.2.4. Dále obsahuje rozhraní `ThreadPool` a jeho implementaci `ThreadPoolImpl`. Díky tomuto rozhraní lze v aplikaci spouštět paralelní úlohy.

8.2.3 Package .core.event

V package event je implementováno rozhraní `Bus`, pomocí kterého lze distribuovat zprávy napříč aplikací. `Bus` tvoří spojení mezi dvěma komponentami, aniž by o sobě tyto dvě komponenty věděly. Jedna komponenta tak může vytvořit událost bez ohledu na to, kolik poslouchajících komponent tuto událost zachytí a zareaguje na ní.

Posluchačem se stává komponenta, která implementuje rozhraní `EventHandler` a zaregistruje si pod určitým klíčem toto rozhraní do `Bus`. Tvůrce události může být kdokoliv, kdo pomocí metody `Bus.publishEvent()` událost publikuje.

Samotná událost je reprezentovaná implementací rozhraní `Event`.

8.2.4 Package .core.plugin

Tento package obsahuje rozhraní jednotlivých typů pluginů, které byly spolu s jejich vlastnostmi uvedeny v Kap. 7.3.

8.2.5 Package .core.plugin.loader

Aby mohla aplikace začít pluginy používat, musí je umět načítat. Princip načítání popisuje Kap. 7.3.5. Samotné načítání pluginu má na starosti rozhraní `PluginProvider` a jeho implementace `PluginLoaderFactory`.

Její nejdůležitější metoda `PluginLoaderFactory.loadPlugin(pluginFile)` provádí samotné načítání pluginu. Jako parametr je jí předán `.jar` archiv s pluginem, který načte Classloader. Ten přestavuje komponentu JRE (runtime prostředí Javy), která se stará o načítání tříd a dalších potřebných zdrojů za běhu JVM [39]. Poté metoda v `manifestu` vyhledá proměnnou *Plugin-Class*. Její hodnota odkazuje na implementaci jednoho z pluginů. Posledním krokem metoda vytváří instanci tohoto pluginu, která se pak vrací volajícímu.

8.2.6 Package .core.plugin.transport

Třídy v tomto balíku implementují životní cyklus pluginů popsaných v Kap. 7.3.2. Aplikace tak může skrze tyto třídy pracovat s pluginy. Důležité v tomto balíku jsou dvě třídy. Obě slouží k výměně dat mezi pluginem a aplikací, ale každá má jinou strukturu.

Třída `ImportPluginBuilder` slouží k přenosu dat z pluginu do programu. Struktura této třídy reflektuje podobu databáze. Obsahuje atributy pro začátek a konec měření, označení místa a jeho upřesnění. Další atribut představuje kolekce typu `Map<String, String>`, kde je klíčem typ dat a hodnotou reprezentují naměřená data. To dovoluje předat aplikaci libovolný počet typů naměřených hodnot pro jedno měření, které je určeno časem a místem.

Třída `ExportPluginBuilder` přenáší data z aplikace do pluginu. Má podobnou strukturu, avšak místo kolekce obsahuje pouze dva atributy pro uložení hodnoty a typu dat. Data jsou z databáze exportovaná po řádcích, a tudíž by kolekce obsahovala pouze jeden prvek. Proto nemusí být přítomna.

8.2.7 Package .core.preferencies

Uživatelsky přívětivá aplikace si uchová některé vlastnosti a nastavení i mezi jednotlivými spuštěními. Uložení tohoto nastavení provádějí třídy z tohoto

balíku, které uchovávají např. zda se má aplikace po spuštění připojit k databázi a nastavení připojení.

8.2.8 Package .database

Tento package a jeho subpackage obsahují třídy, jež obstarávají propojení aplikace s databází. Konkrétně se jedná o rozhraní `Database` a její implementaci `DatabaseProvider`. Toto rozhraní umožňuje připojení do databáze a komunikaci s ní.

8.2.9 Package .database.entity

Třídy obsažené tento package představují pouze Java objekty, jež reflektují entity z databáze včetně jejich atributů.

8.2.10 Package .database.query

Nejzajímavější třídou tohoto balíku je rozhraní `Query` a jeho implementace `QueryBuilder`. Rozhraní se využívá pro sestavování databázových dotazů na základě vstupu od uživatele. Tento dotaz se skládá pomocí různých implementací rozhraní `Visitor` a `WhereVisitor` z balíku `database.query.visitor`. Návrhový vzor visitor byl vysvětlen viz 6.1.2.

8.2.11 Package .main

V tomto package se nachází dvě velmi důležité třídy. První třída `Main` obsahuje metodu `main`. Stává se tak vstupním bodem do aplikace. Druhá třída `ModuleManager` reprezentuje konfigurační třídu pro framework Google Guice viz 6.2.4.

8.3 Controller

Existuje-li logická vrstva aplikace a vrstva, která umí zobrazit data, musí existovat ještě jedna vrstva, která tyto dvě propojí. Implementace této vrstvy se nachází v balíku view a v jeho dílčích balíčcích.

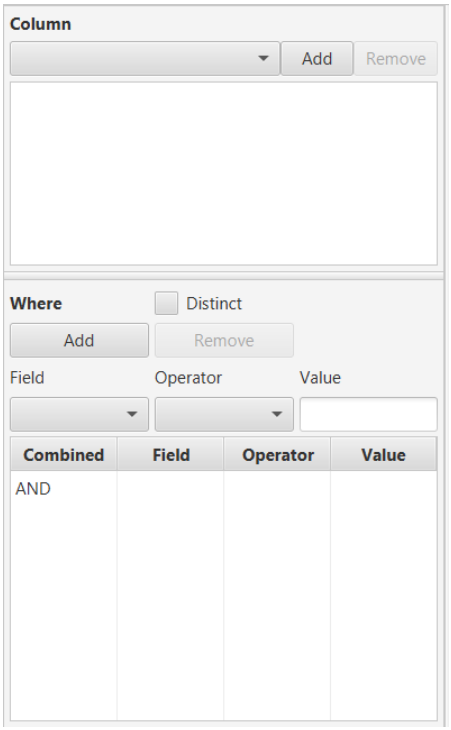
8.3.1 Rozhraní TabController

Aplikace je založena na jednotlivých záložkách, které mají různý obsah. Aby bylo možné tyto záložky spravovat a pracovat s nimi, mají společné rozhraní,

které představuje `TabController`. Implementacemi tohoto rozhraní se stávají jednotlivé funkce, které aplikace podporuje. Všechny jeho implementace se nachází v balíku `view.controler`.

8.3.2 Rozhraní QueryCotroller

Aplikace nabízí možnost nahlížet na data uložená v databázi. Aby si mohl uživatel zvolit, jaká data chce zobrazit, byla vytvořena komponenta `QueryBuilderView.fxml` a k ní odpovídající controller `QueryControlerImpl` s rozhraním `QueryController`. Díky této komponentě si může uživatel v části *Column* definovat, které hodnoty (sloupce) z databáze budou zobrazeny. V části *Where* může svůj dotaz ještě přesněji specifikovat viz Obr. 8.1.



Obrázek 8.1: Komponenta pro upřesnění databázového dotazu

9 Testování

Testování je důležitou součástí vývoje nového softwaru. Slouží pro ověření správné funkčnosti programu. Testování této aplikace probíhalo v několika krocích.

9.1 Unit testování

Při unit (jednotkovém) testování se testují malé části kódu. Takovou malou částí kódu může být např. metoda nějaké třídy. Pomocí unit testů může programátor zjistit, jestli se jím naprogramovaná metoda chová tak, jak zamýšlel. Jde o základní způsob testování [42].

Tohoto testování bylo využito i při vývoji mé aplikace. Pomohlo mi odhalit některé základní chyby (např. špatně přiřazená proměnná), ale pomohlo částečně i s návrhem aplikace. Ukázalo se, že netestovatelná komponenta je nevhodně navržená a má tak příliš těsnou vazbu s jinou komponentou. To ukazuje na nevhodně použitý DI (Kap. 6.1.3). Zároveň testy sloužily pro ověření, že byla zachována požadovaná funkčnost i při změnách v aplikaci v průběhu vývoje.

Při vytváření unit testů jsem narazil na problém s testováním částí aplikace, ve kterých jsou použity JavaFX komponenty. Aby mohly být tyto komponenty vytvořeny, musí být inicializován Toolkit pro JavaFx. Tento problém byl vyřešen pomocí anotace `@RunWith` [43]. Parametrem této anotace je třída. Před spuštěním samotného testu se vytvoří instance této třídy. Byla tak implementována pomocná třída `JfxTestRunner.class`, která inicializuje Toolkit JavaFX . U testovacích tříd, jež potřebovaly vytvářet komponenty JavaFX, tak bylo využito této anotace s uvedenou pomocnou třídou.

Pro účely unit testování bylo vytvořeno 134 testů, které pokrývají celkem 30% produkčního kódu.

9.2 Funkcionální testování

Během funkcionálního testování byly ověřeny základní funkce aplikace.

9.2.1 Scénář 1: Načtení pluginu aplikací

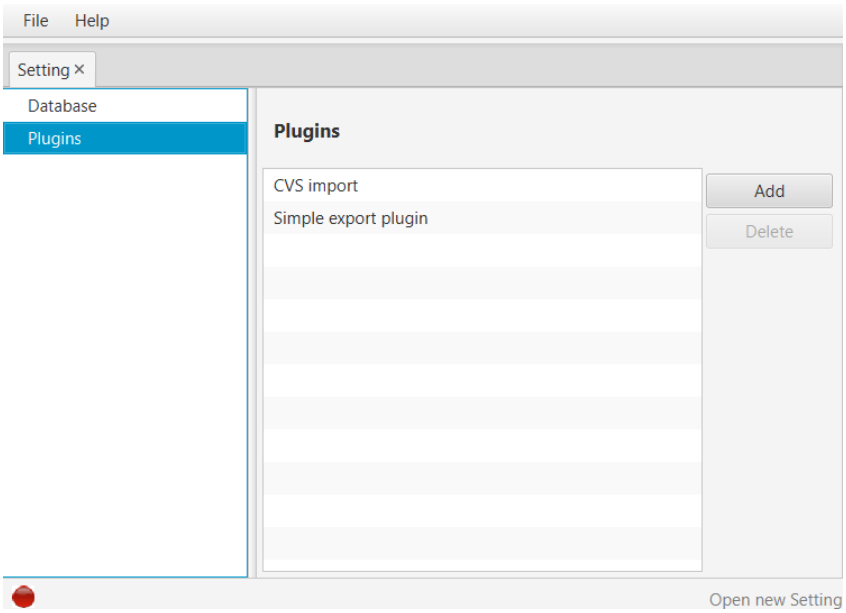
Příprava testu

Aplikace musí být schopná načíst plugin, který splňuje požadavky definované v Kap. 7.3.5. Pro tyto testovací účely lze využít kterýkoliv plugin, jenž byl vytvořen pro ověření importu a exportu. Pluginy se nachází na přiloženém CD ve složkách `importPlugin` a `exportPlugin`.

Testovací scénář

První krok představuje spuštění aplikace. Následně je nutné z aplikačního menu vybrat možnost `File→Preferencies`. Tím se otevře záložka s nastavením aplikace. Další krok znamená kliknutí na položku *Plugins* v levé části okna viz refimg.setting.test. Tato akce přepne obsah okna na správu pluginů.

V dalším kroku se pomocí tlačítka *Add* vybere a nahraje plugin do aplikace. Je-li tento krok úspěšně dokončen, přidá se tento plugin do seznamu a bude tak dostupný i při příštím spuštění. Pro ověření této skutečnosti slouží několik dalších kroků. Je nutné aplikaci vypnout a zapnout, poté znovu přejít do nastavení aplikace a nakonec zkontrolovat, že se přidaný plugin nachází v seznamu dostupných pluginů.



Obrázek 9.1: Nastavení aplikace

9.2.2 Scénář 2: Import skrze plugin

Příprava testu

Pro účely testování importu skrze plugin byl implementován plugin `SimpleImportPlugin.jar` nacházející se ve složce *importPlugin* na přiloženém CD. Tento plugin zpracovává testovací data přiložená ve složce *importPlugin/testData*. Jedná se o jeden soubor v prostém textovém formátu s názvem `ImportTestData.txt`. Soubor obsahuje fiktivní data z dopravních čidel. Data jsou řazena do řádků, kdy jeden řádek se rovná jednomu záznamu měření.

Před samotným importem musí být aplikace připojena k databázi. Pro ověření správné funkčnosti aplikace musí být vytvořena nová databáze. Tento postup spolu s konfigurací aplikace je uveden v uživatelské příručce. Je-li aplikace připojena k databázi, musí do ní být přidán plugin pro import zopakováním postupu z Kap. 9.2.1. Nyní je aplikace připravena na test importu dat.

Testovací scénář

Prvním krokem pro zahájení testu importu je vybrání položky `File → New → Import` z aplikačního menu. Otevře se záložka pro import. V následujícím kroku se z komponenty zobrazující dostupné pluginy zvolí *Simple Import Plugin* a zapíše se cesta k souboru `ImportTestData.txt` do pole s názvem *Plugin parameters*. Pro zahájení samotného importu je dalším krokem stisknuto tlačítko *Import*.

Pro ověření, zda byla data opravdu importována, vedou následující kroky. Je nutné otevřít záložku pro zobrazení dat z databáze (aplikační menu *File → New → Table*). V levé části okna musí být navolen libovolný dotaz na data. Stisknutím tlačítka *Execute* je proveden databázový dotaz a jeho výsledek se zobrazí v tabulce.

Kroky pro ověření, že si aplikace data neuchovává pouze v paměti počítače, zahrnují vypnutí a spuštění aplikace, její připojení k databázi a zopakování stejného dotazu. Pokud jsou data uložená v databázi, zobrazí se stejný výsledek.

9.2.3 Scénář 3: Export skrze plugin

Příprava testu

Pro tento způsob testování je přiložen na CD plugin `SimpleExportPlugin.jar`, který se nachází ve složce *exportPlugin*. Tento plugin během exportu

vytvoří soubor a zapíše do něj data, která mu aplikace předá.

Pro ověření správné funkčnosti je třeba přidat plugin do aplikace, vytvořit novou databázi a tu naplnit daty. Pro tyto účely je nutné zopakovat kroky z Kap. 9.2.1 a Kap. 9.2.2 .

Testovací scénář

První krok představuje otevření záložky `File → New → Export`. Dalším je definování dat, která se mají exportovat pomocí komponenty v levé části okna. Následně je nutné pomocí komponenty označené jako *Select Plugin* vybrat *Simple Export Plugin* a zadat do pole pro parametry plaginu název souboru a cestu, kam se má soubor s exportovanými daty uložit. Poté do pole pro parametry pluginu zadat cestu (včetně názvu souboru), kam se má soubor s exportovanými daty uložit. Po kliknutí na tlačítko *Export* začne aplikace exportovat data.

Po dokončení exportu následuje krok, který ověřuje, že byla data opravdu exportována. Znamená to otevřít soubor a přesvědčit se, že do něho byla data zapsána.

10 Závěr

Cílem této práce bylo seznámit se s dopravními simulacemi se zaměřením na data, která jsou pro ně nezbytná. Následně bylo nutné prozkoumat možnosti pro rozšíření aplikace, aniž by ona samotná byla upravována. Na základě získaných informací bylo třeba vytvořit aplikaci, která bude schopna uchovávat data potřebná pro dopravní simulaci a bude také umožňovat přidání nových funkcí v podobě importu a exportu dat, bez zásahu do již vytvořené aplikace.

Výsledkem této práce je program, který umí uchovávat různorodá data. Jejich import i export probíhá pomocí pluginů, které nejsou součástí programu. Představují dodatečné funkce, jenž mohou být do aplikace přidány. Program tak získal možnost zpracovávat libovolný typ dat. Stačí jen vytvořit nový plugin, který umí pracovat s konkrétním typem dat a následně ho do aplikace přidat.

Pro ukládání dat používá aplikace velmi obecný model databáze. Tento model jí dovolí uchovat velmi různorodá data. Zároveň však představuje také její slabinu. Pro zachování integrity databáze při vkládání dat je zapotřebí několika databázových dotazů navíc. Proto probíhá toto ukládání delší dobu, než jsem původně očekával. Proto bych pro vylepšení této aplikace doporučil optimalizaci databázového modelu.

Do budoucna by díky současnému návrhu aplikace bylo možné oddělit jednotlivé funkce do samostatných modulů. To by umožnilo snadné dodávání dalších funkcí např. jiné možnosti zobrazení dat (graf). Aplikace by tak data již pouze neuchovávala ve stavu pro snadné použití, ale zároveň by s nimi mohla sama pracovat.

Literatura

[1] Robert E. Shannon, *Introduction to the art and science of simulation*, Industrial Engineering, Texas A&M University

[2] Naylor, T. H., J. L. Balintfy, D. S. Burdick, and K. Chu.,1966. Computer Simulation Techniques, John Wiley.

[3] Ivan Křivý a Evžen Kindler, Simulace a modelování, Ostravská univerzita 2001

[4] Jan Fábry, *Matematické modelování*, Fakulta informatiky a statistiky,VŠE v Praze, 2011, ISBN 978-80-7431-066-9

[5] Modelování dopravy na pozemních komunikacích (ČÁST 2) [online], Dostupné online: http://projekt150.ha-vel.cz/node/95

[6] Manlig, F. - Keller, P., *Možnosti využití počítačové simulace*, Dostupné online: http://fstroj.utc.sk/journal/sk/40/40.htm

[7] FUJIMOTO, R. M.: *Parallel and Distributed Simulation Systems*, John Wiley & Sons, New York, 2000.

[8] Martin Kožíšek, *Systém pro mobilní zařízení pro manuální zaznamenávání průjezdů vozidel křižovatkou*, Fakulta aplikovaných věd, Západočeská univerzita v Plzni

[9] Tomáš Potužák, *Distributed Traffic Simulation*, University of West Bohemia

[10] Mgr. Petr Kočička, *Simulační metody jako nástroj pro rozhodování podniku*, Ekonomicko-správní fakulta, Masarykova univerzita

[11] Michal Dorda, *Úvod do modelování a simulace systémů*, Dostupné online: http://homel.vsb.cz/ dor028/Aplikace_2.pdf

[12] Michal Dorda, *Část 1. - Dopravní průzkumy*, Dostupné online: http://homel.vsb.cz/ dor028/Pruzkumy.pdf

[13] Ondřej Přibyl, *Detektory zasahující do vozovky, úvod do detekce* , [online], Ústav aplikované matematiky,ČVUT v Praze, Fakulta dopravní [cit. 2012-10-12]

[14] Ondřej Přibyl, *Neintrusivní dopravní detektory*, [online], Ústav aplikované matematiky,ČVUT v Praze, Fakulta dopravní [cit. 2016-14-04]

[15] *Dopravní inženýrství*, [online], Fakulta strojní, VŠB-TU Ostrava, 2009, Dostupné online: http://projekt150.ha-vel.cz/node/29

[16] Tomáš Potužák, *Methods for reduction of inter-process communication in distributed simulation of road traffic*,[online], Faculty of Applied Sciences, University of West Bohemia in Pilsen, 2009

[17] *Dopravní chování v datech, sborník konference*, Centrum dopravního výzkumu, v.v.i., Brno, [cit. 2018-31-10], ISBN 978-80-88074-61-8

[18] David Hartman, *Modelování dopravní situace systémy hromadné obsluhy*, Fakulta aplikovaných věd, Západočeská univerzita v Plzni

[19] Petr Peringer, Martin Hrubý, *Modelování a simulace*[online], Fakulta informačních technologií, Vysoké učení technické v Brně, [cit. 2018-22-11] Dostupné online: https://www.fit.vutbr.cz/study/-courses/IMS/public/prednasky/IMS.pdf

[20] Simulace [online], Dostupné online: http://ffa.lukasberta.com/rgb/-clanky/simulace.htm

[21] Plug-in [online], Dostupné online: https://techterms.com/definition/-plugin

[22] Plug-in Architectures, Dostupné online: https://developer.apple.com/-library/archive/documentation/Cocoa/Conceptual/LoadingCode/-Concepts/Plugins.html

[23] Execution Environments [online], Dostupné online: https://wiki.-eclipse.org/Execution_Environments

[24] ItNetowork [online], Dostupné online: https://www.itnetwork.cz/java/-zaklady/java-tutorial-uvod-do-jazyka-java

[25] Java Programming Environment and the Java Runtime Environment [online], Dostupné online: https://docs.oracle.com/cd/E19455-01/806-3461/6jck06gqd/index.html

[26] Java Reflection API [online], Dostupné online: https://docs.oracle.-com/javase/8/docs/technotes/guides/reflection/index.html

[27] Classpath (Java) [online], Dostupné online: https://docs.oracle.com/-javase/tutorial/essential/environment/paths.html

[28] OSGi [online],Dostupné online: https://www.osgi.org/developer/architecture

[29] Tiobe [online], Dostupné online: https://www.tiobe.com/tiobe-index/

[30] MVC [online], Dostupné online: https://www.tutorialsteacher.com/-mvc/mvc-architecture

[31] Visitor pattern [online], Dostupné online: https://www.algoritmy.net/-article/1643/Visitor

[32] Visitor pattern [online], Dostupné online: https://www.tutorialspoint.-com/design_pattern/visitor_pattern.htm

[33] Dependenci Injection [online], Dostupné online: http://voho.eu/wiki/-vkladani-zavislosti/

[34] JavaFX [online], Dostupné online: https://docs.oracle.com/javase/8/-javafx/get-started-tutorial/jfx-overview.htm#JFXST784

[35] FXML [online], Dostupné online: https://docs.oracle.com/javafx/2/-fxml_get_started/why_use_fxml.htm#CHDCHIBE

[36] Maven [online], Dostupné online: https://maven.apache.org/

[37] Samuel Butta, *Implementace DI kontejneru* , Fakulta informačních technologií, ČVUT v Praze, Dostupné online: https://dspace.cvut.cz/-handle/10467/69686

[38] Iterator [online], Dostupné online: https://docs.oracle.com/javase/8/-docs/api/java/util/Iterator.html

[39] Classloader [online], Dostupné online: http://voho.eu/wiki/java-classloader/

[40] Simulace [online], Dostupné online: https://dictionary.cambridge.-org/dictionary/english/simulation

[41] [online] Modular Approach in Programming , Dostupné online: https://www.geeksforgeeks.org/modular-approach-in-programming/

[42] Pavel Herout, *Testování pro programátory*, ISBN 978-80-7232-481-1

[43] Anotace `@RunWith` [online], Dostupné online: http://junit.sourceforge.-net/javadoc/org/junit/runner/RunWith.html

Seznam obrázků

Uživatelská příručka

Spuštění

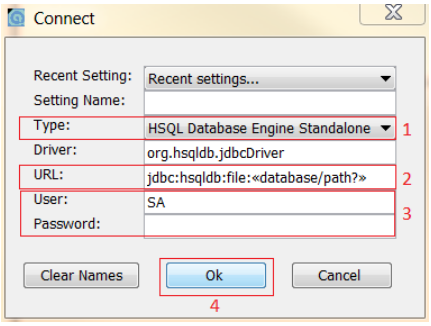
Pro spuštění aplikace musí být na počítači instalována Java[1]. Aplikace se neinstaluje. Stačí pouze spustit soubor `TrafficContentManager.jar`.

Databáze

Aplikace pro svoji činnost potřebuje databázi. Konkrétně se jedná o HSQLDB[2]. Novou databázi můžete vytvořit pomocí nástroje HSQL Database Manager. Soubor s tímto nástrojem je uložen na přiloženém CD ve složce databáze.

Vytvoření databáze

Nová databáze se vytvoří pomocí nástroje *HSQL Database Manager*. Po spuštění tohoto nástroje se objeví nové okno (Obr. 10.1). Z možností označených jako *Type:* (Obr. 10.1 – 1) vyberete HSQL Database Engine Standalone. Do pole URL (Obr. 10.1 – 2) vyplníte místo textu «database/path?» relativní nebo absolutní cestu, kde chcete, aby se databáze vytvořila. Za ni připište jméno databáze. Zbylý text v tomto poli musí zůstat. Výsledný formát může vypadat např. `jdbc:hsqldb:file:relative/path/to/db/dbName`. Do pole User a Password (Obr. 10.1 – 3) se zadávají přihlašovací údaje do databáze.



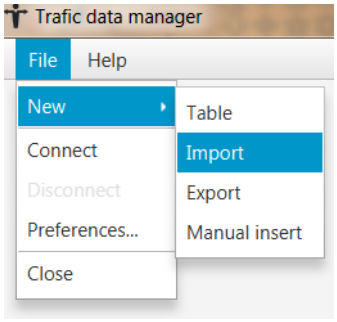
Obrázek 10.1: HSQL Database Manager

[1]https://www.oracle.com/technetwork/java/javase/downloads/jre8-downloads-2133155.html

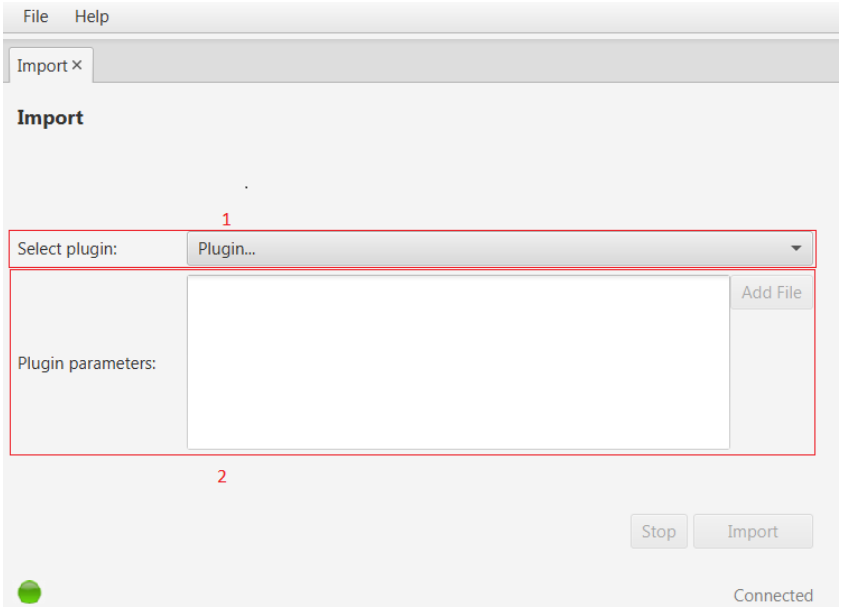
[2]http://hsqldb.org/

Import Dat

V menu vyberte a zvolte položku *Import* (Obr. 10.2). Zobrazí se nová záložka (Obr. 10.3). Dalším krokem je vybrání pluginu, kterým se budou importovat data (Obr. 10.3 – 1). Pokud plugin potřebuje pro svoji činnost nějaké parametry, zadejte je do pole *Plugin parameters* (Obr. 10.3 – 2). Samotný import spustíte tlačítkem *Import*.



Obrázek 10.2: Aplikační menu Import



Obrázek 10.3: Záložka pro import

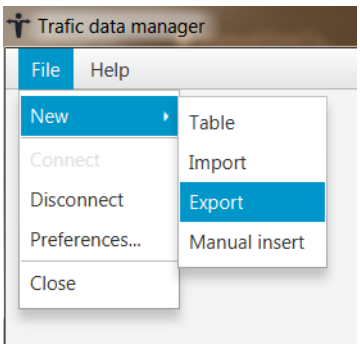
Export Dat

Pro export dat v menu vyberete položku *Export* (Obr. 10.4). Po této akci se zobrazí nová záložka (Obr. 10.5). Záložka je rozdělena na pravou a levou

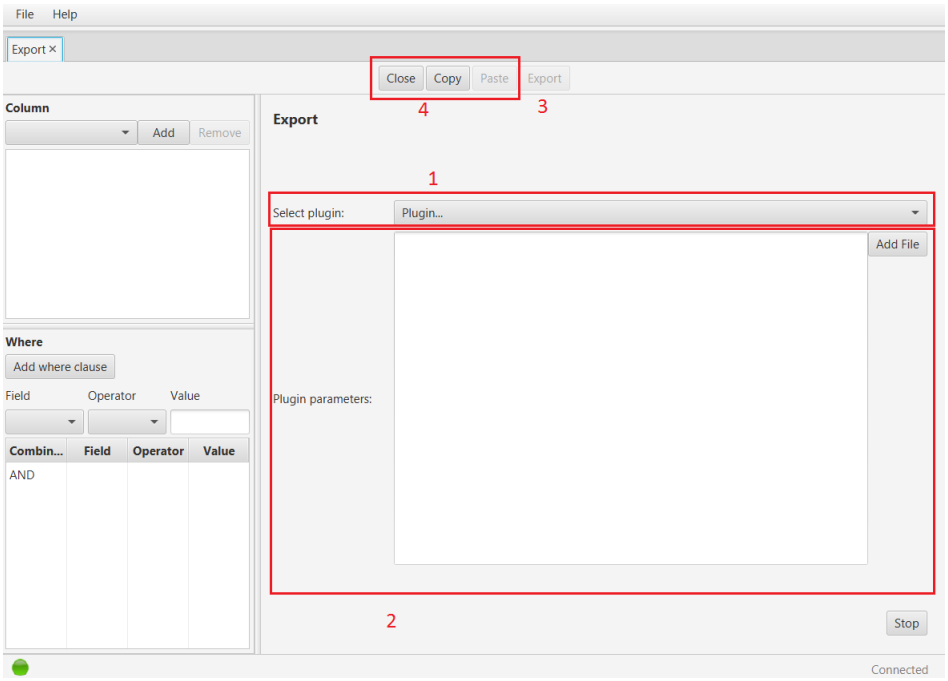
část. V levé části se definuje, jaká data budou exportována. Jak ovládat tuto komponentu popisuje v Kap. 10. V pravé části můžete definovat, jakým pluginem bude export proveden.

Nejdříve vyberete plugin, kterým budete data exportovat (Obr. 10.5 – 1). Poté zadáte parametry, které plugin potřebuje pro svůj běh (Obr. 10.5 – 2). Stisknutím tlačítka *Export* se spustí samotný export (Obr. 10.5 – 3).

Funkce tlačítek *Close, Copy a Paste* (Obr. 10.5-4) vysvětluje Kap. 10.



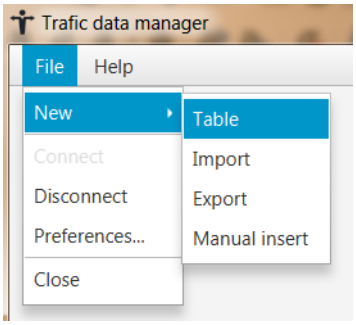
Obrázek 10.4: Aplikační menu Export



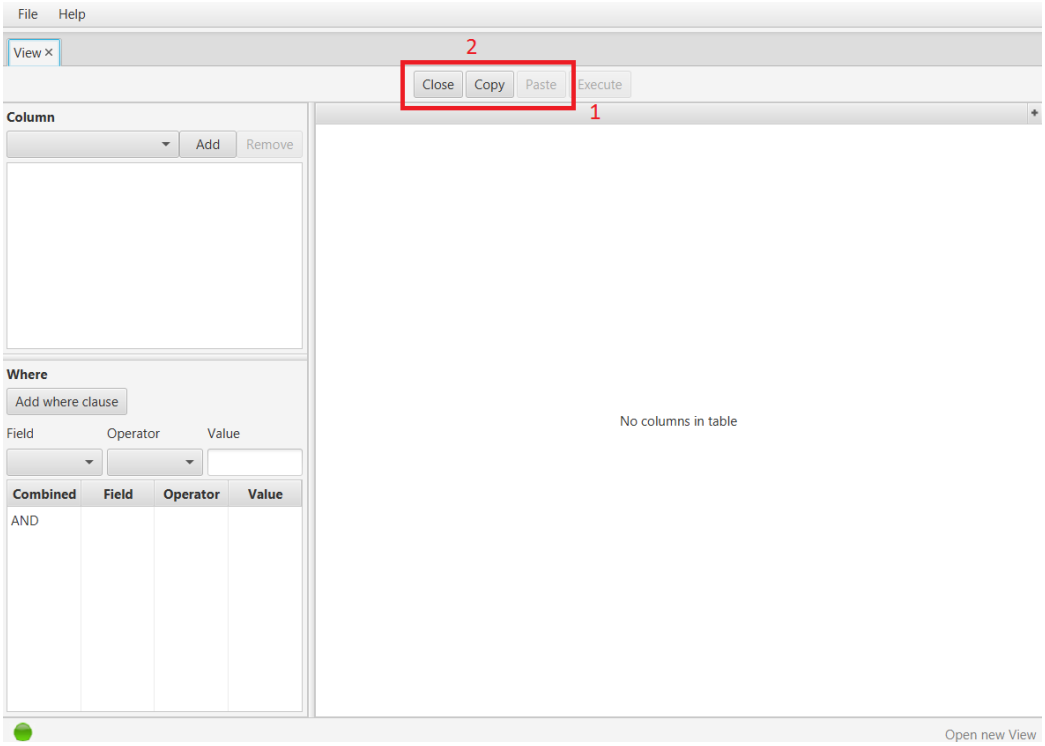
Obrázek 10.5: Záložka pro export

Zobrazení dat

Data lze zobrazit po kliknutí na položku menu Table (Obr. 10.6). Následně se otevře nová záložka (Obr. 10.7). Ta je opět rozdělena na pravou a levou část. Ovládání levé slouží pro definování databázového dotazu viz Kap. 10. Vykonání definovaného dotazu se spustí tlačítkem *Execute* (Obr. 10.7 – 1). Funkce tlačítek *Close, Copy a Paste* (Obr. 10.7 – 2) vysvětluje Kap. 10. Pravou částí je tabulka, ve které se zobrazují výsledky databázového dotazu.



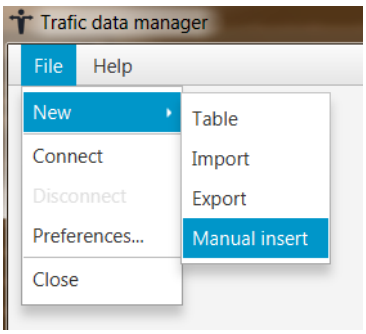
Obrázek 10.6: Aplikační menu Table



Obrázek 10.7: Záložka pro zobrazení dat

Manuální uložení dat

Pro manuální uložení dat do databáze otevřete záložku Manual insert pomocí menu (Obr. 10.8).



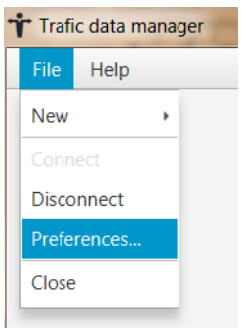
Obrázek 10.8: Aplikační menu Manual insert

Nastavení připojení k databázi

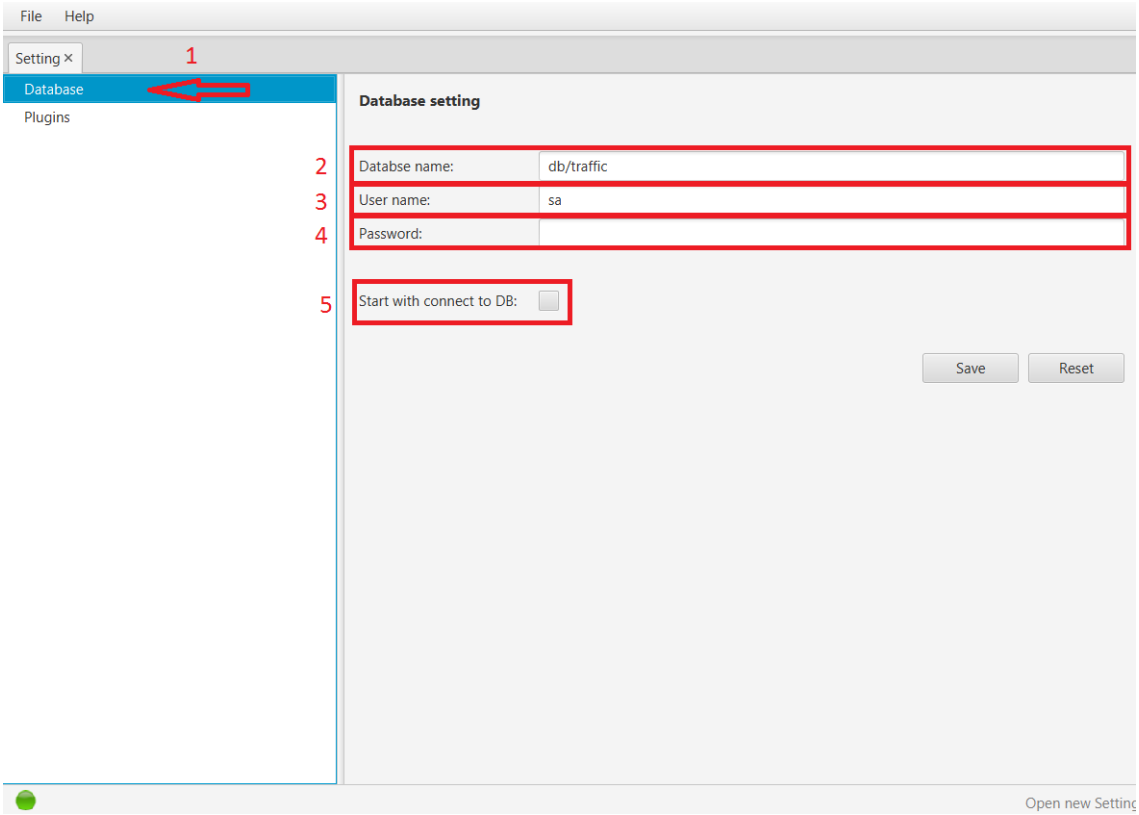
Pro nastavení připojení k databázi otevřete nastavení aplikace pomocí položky menu *Preferences* (Obr. 10.9). V otevřeném okně kliknete na položku *Database* (Obr. 10.10 – 1). V levé části se objeví nastavení připojení k databázi.

Do pole *Database name* (Obr. 10.10 – 2) se zadává (relativní nebo absolutní) cesta k vytvořené databázi včetně jejího jména (ta samá hodnota, která se vyplňuje při vytváření databáze do pole *URL*). Pole *User name* (Obr. 10.10 – 3) a *Password* (Obr. 10.10 – 4) slouží k nastavení přihlašovacích údajů.

Zaškrtnutím *Start with connect to DB* (Obr. 10.10 – 5) nastavíte aplikaci tak, aby se při příštím spuštěním rovnou připojila k databázi.



Obrázek 10.9: Aplikační menu Preferences



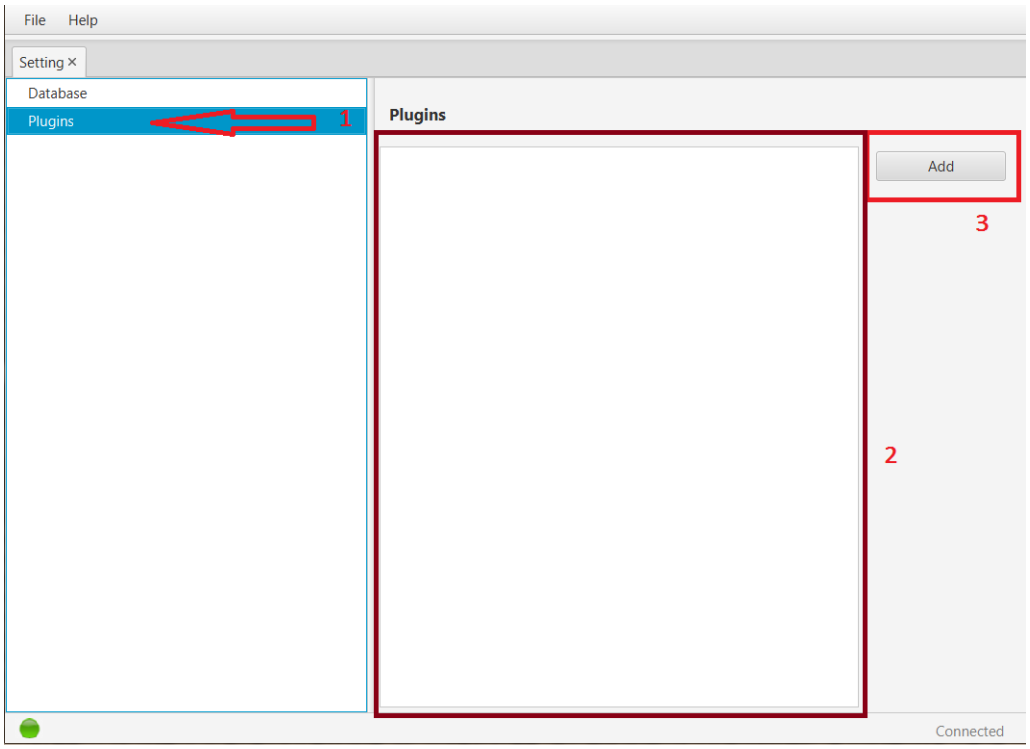
Obrázek 10.10: Nastavení databázového připojení

Přidání nového pluginu

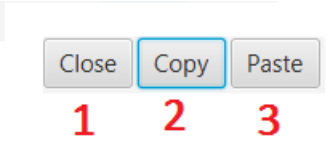
Pro přidání nového pluginu do aplikace otevřete nastavení aplikace pomocí menu (Obr. 10.9). V otevřeném okně klikneme na položku *Plugins* (Obr. 10.11 – 1). Okno se přepne na výpis dostupných pluginů (Obr. 10.11 – 2). Nový plugin lze přidat pomocí tlačítka *Add* (Obr. 10.11 – 3).

Ovládací panel

Ovládací panel (Obr. 10.12) je dostupný záložce *Table* a *Export*. Tlačítko *Close* (Obr. 10.12 – 1) umí schovat levý panel pro definování databázového dotazu. Pomocí tlačítek *Copy* (Obr. 10.12 – 2) a *Paste* (Obr. 10.12) můžete přenášet mezi těmito záložkami definovaný databázový dotaz.



Obrázek 10.11: Výpis dostupných pluginů



Obrázek 10.12: Ovládací panel

Panel pro definování dotazu

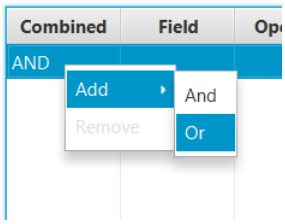
Tento panel slouží k definování databázového dotazu. Panel se skládá ze dvou hlavních částí. V části *Column* si můžete definovat, které sloupce se zobrazí po vykonání databázového dotazu. Pro přidání sloupce si stačí vybrat sloupec z nabídky a přidat ho tlačítkem *Add*. Tlačítko *Remove* odstraní z vybraných sloupců aktuálně zvolenou položku.

Druhá část tohoto panelu se nazývá *Where*. Slouží pro upřesnění databázového dotazu. Lze zde zadat logické podmínky dotazu. Tyto podmínky jsou uzavřeny do kontejnerů *AND* a *OR*, jež představují logickou funkci. Pomocí této funkce pak budou jednotlivé podmínky mezi sebou vyhodnoceny. Nový kontejner lze přidat pomocí kontextového menu vyvolaného přes pravé tlačítko myši (Obr. 10.13).

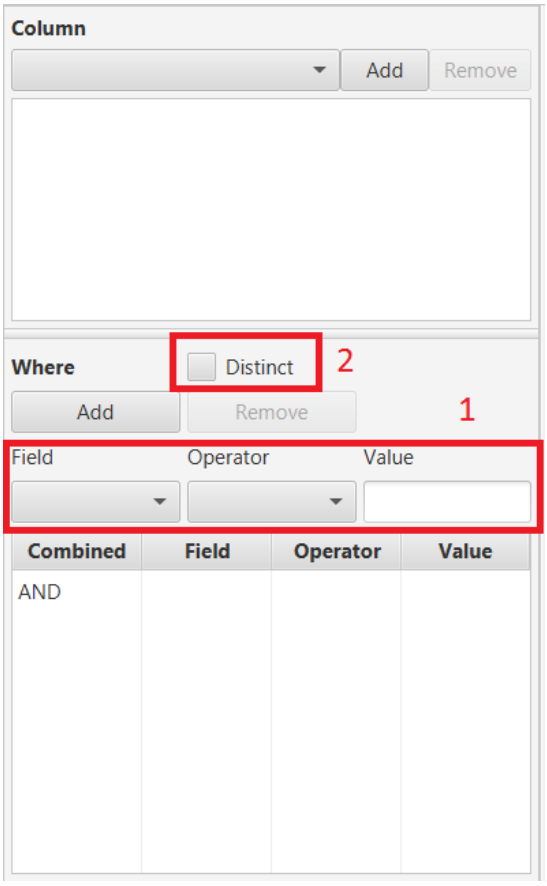
Novou podmínku lze definovat pomocí panelu na Obr. 10.14 – 1. Po vy-

plnění všech polí můžete tuto podmínku přidat do seznamu pomocí tlačítka *Add*.

Zaškrtávací možnost *Distinct* (Obr. 10.14 – 2) slouží k zapnutí funkce `distinct`[3].



Obrázek 10.13: Kontextové menu *Where*

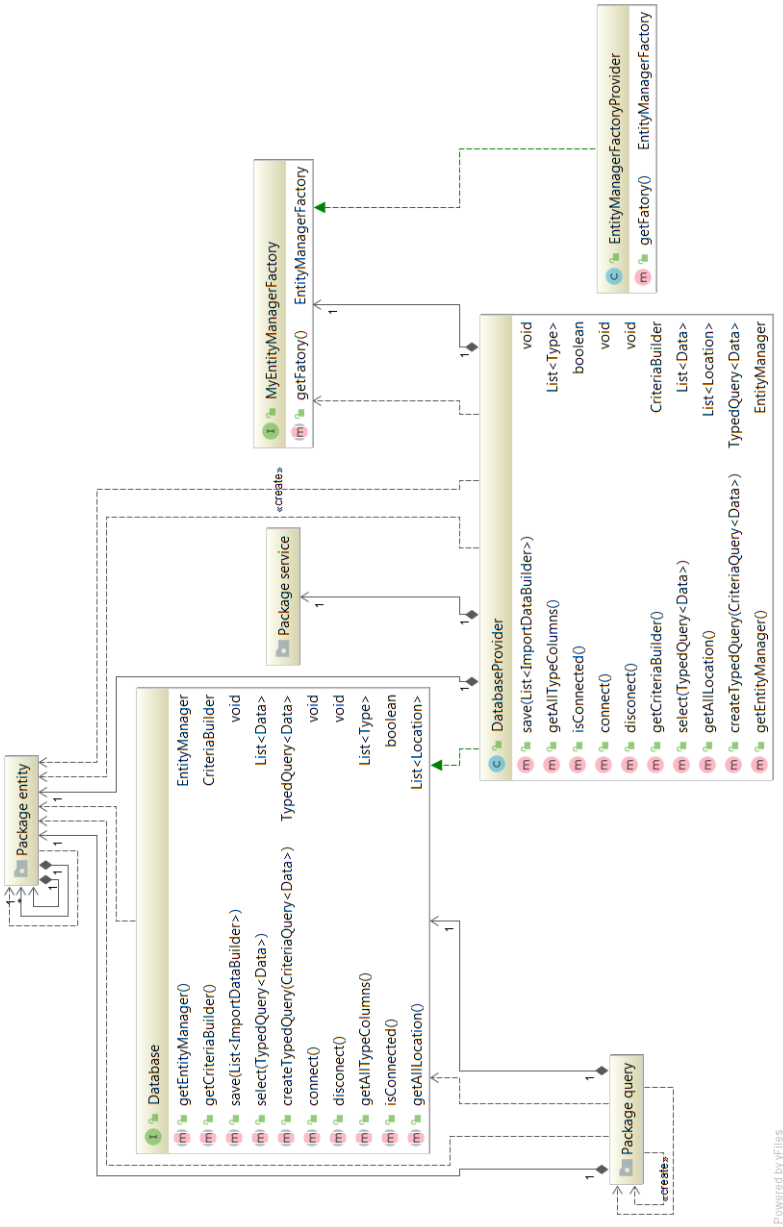


Obrázek 10.14: Panel pro definování dotazu

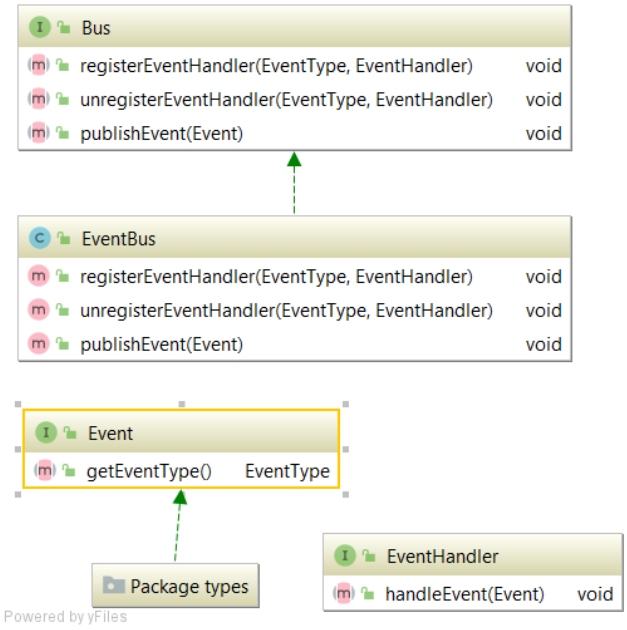
[3]https://www.w3schools.com/sql/sql_distinct.asp

UML diagramy

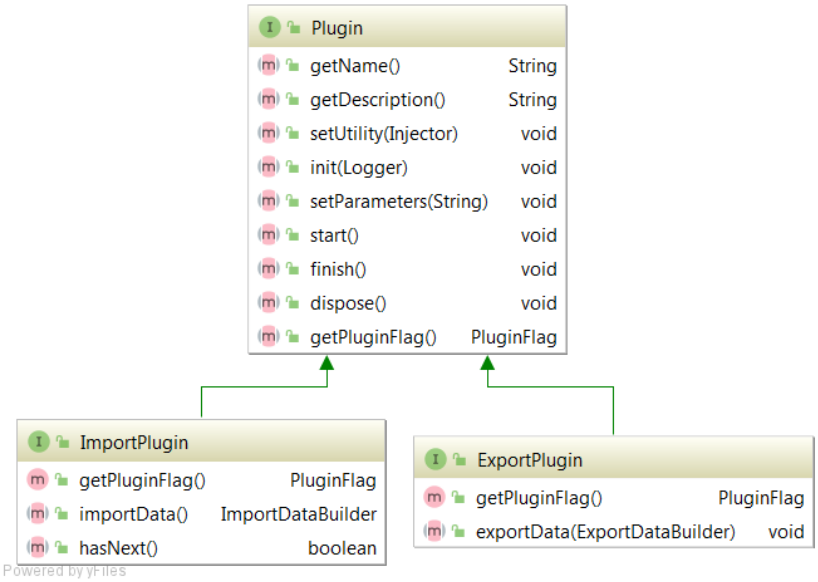
UML diagramy nejdůležitějších rozhraní a jejich implementací.



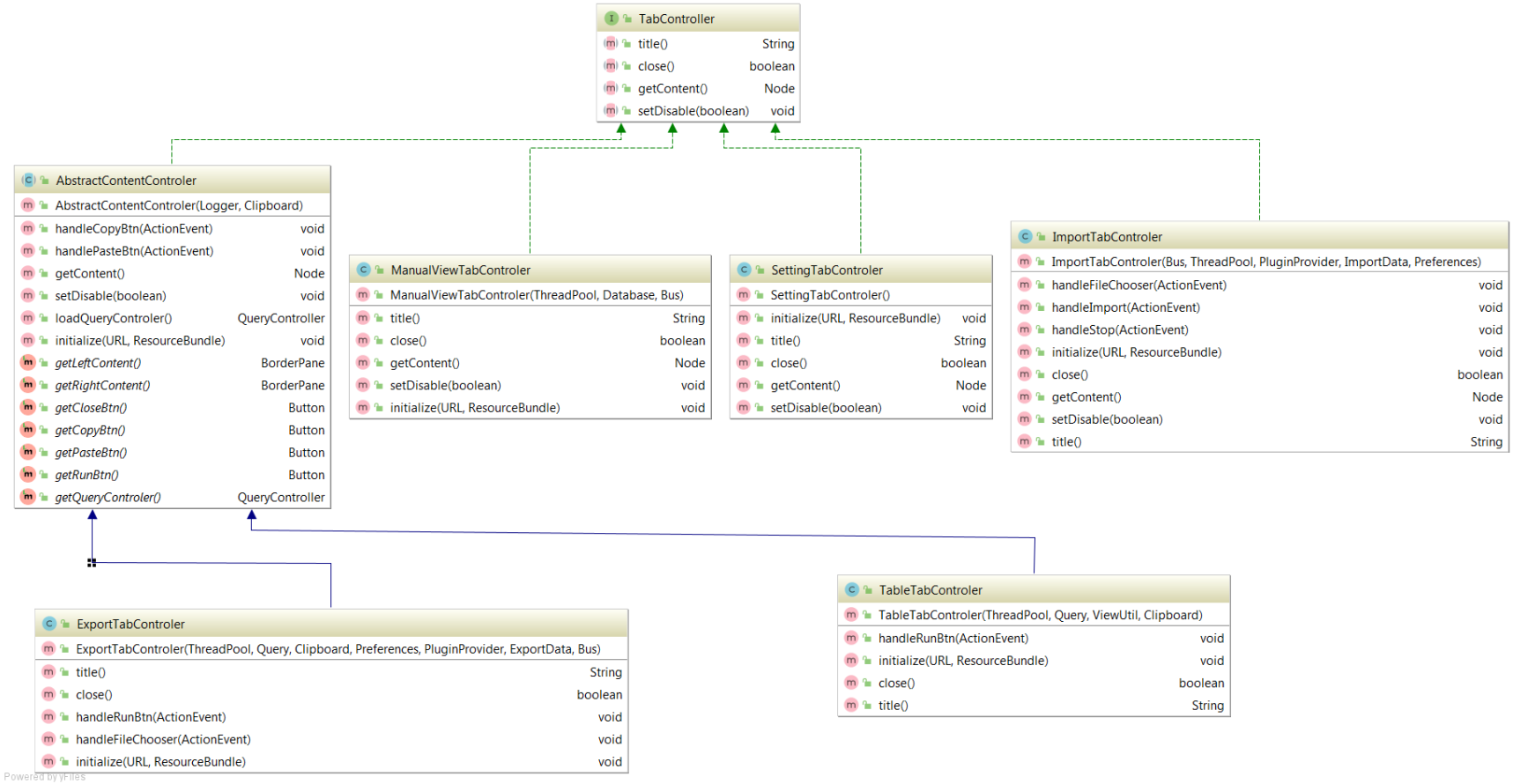
Obrázek 10.15: Diagram package .database



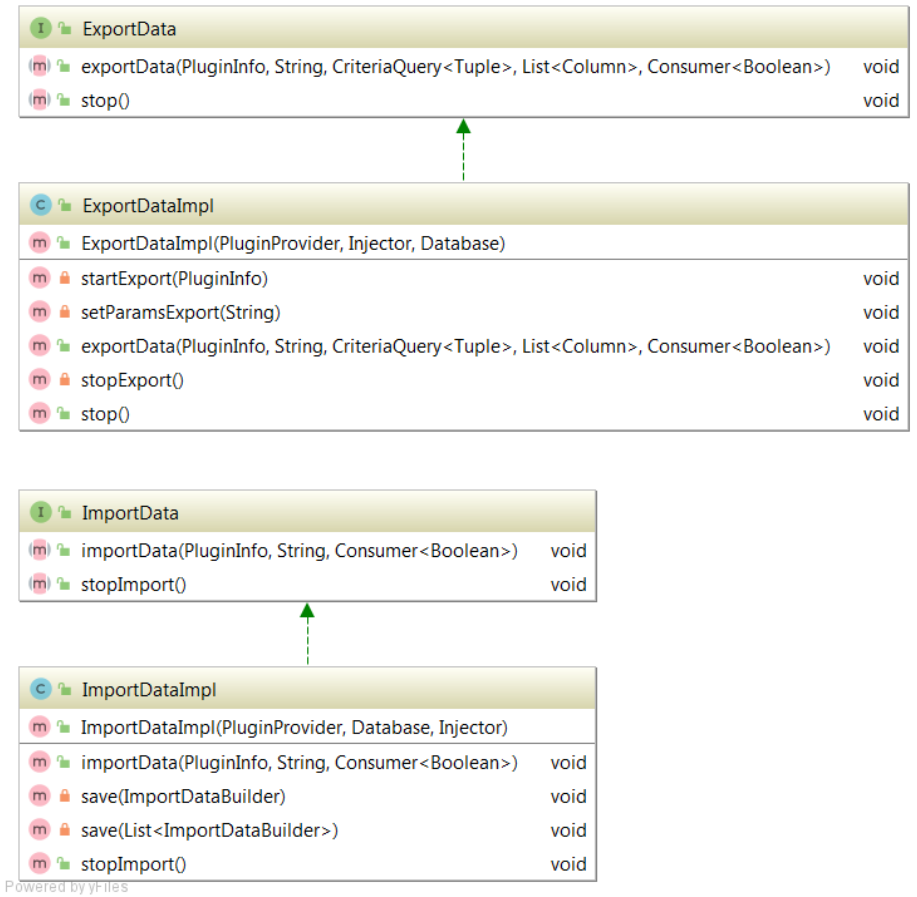
Obrázek 10.16: Diagram package .core.event



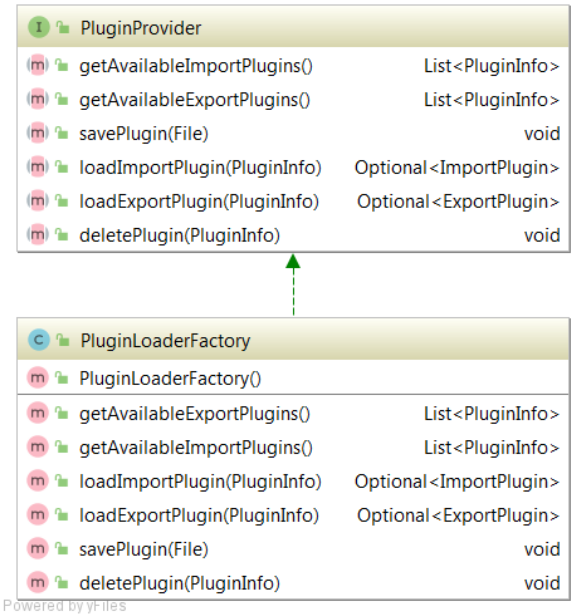
Obrázek 10.17: Diagram package .core.plugin



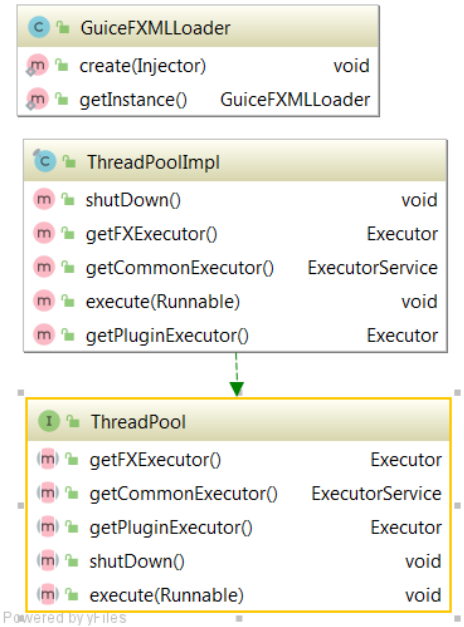
Obrázek 10.18: Diagram package .controler



Obrázek 10.19: Diagram package .core.plugin.transport



Obrázek 10.20: Diagram package .core.plugin.loader



Obrázek 10.21: Diagram package .core