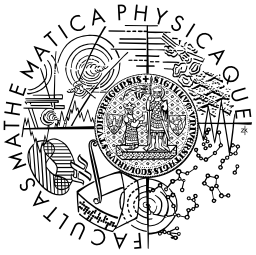


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Lukáš Kolek

Analýza letových dat

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D.

Studijní program: Programování a SW systémy

Studijní obor: IPSS

Praha 2018

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V ..... dne ..... Podpis autora

Chtěl bych poděkovat RNDr. Janu Kofroňovi, Ph.D. za vedení mé bakalářské práce, odborný dohled a cenné rady. Děkuji také Ing. Tomáši Meiserovi za dlouhodobou pomoc při implementaci práce a předání profesních zkušeností.

Název práce: Analýza letových dat

Autor: Lukáš Kolek

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: RNDr. Jan Kofroň, Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt: Cílem práce je návrh a implementace nástroje pro analýzu letových dat, která jsou produkována simulací AgentFly. Nástroj uživateli umožňuje načíst letové zprávy ve formátu CMS a dále je zpracovávat. Pro analýzu dat slouží sada statistických testů, pomocí kterých může uživatel filtrovat jednotlivé lety. Během implementace nástroje byl kladen důraz na budoucí snadné přidávání testů pro filtrování a rozšíření o další formáty zpráv.

Klíčová slova: analýza dat, letová data, filtrování dat, simulace letového provozu

Title: Flight data analysis

Author: Lukáš Kolek

Department: Department of Distributed and Dependable Systems

Supervisor: RNDr. Jan Kofroň, Ph.D., Department of Distributed and Dependable Systems

Abstract: The main goal of this thesis is a design and an implementation of a tool for analysis of flight data which are produced by AgentFly simulation. The user can load flight messages in CMS format and process them. The thesis contains a basic set of statistical tests, which are used to filter flights. The implementation focuses on an easy extension of the set of test cases and new message formats.

Keywords: data analysis flight data, data filtering, ATM simulation

Obsah

1. Úvod

1.1 Motivace

V současné době se v informatice zpracovávají stále větší objemy dat a pro člověka je již nereálné taková data kontrolovat ručně. Jedním z příkladů je simulace AgentFly, která simuluje reálný letový provoz v počítači a během jednoho běhu pracuje s tisíci lety. Po skončení simulace chybí nástroj, který by zhodnotil, zda letadla letěla podobně jako v reálném provozu a zda simulace proběhla správně nebo se zde nacházejí letadla, která měla špatnou trajektorii.

1.2 Simulace AgentFly

AgentFly je multiagentní systém umožňující rozsáhlé simulace civilní i bezpilotní letecké dopravy. Systém integruje pokročilé plánování letových tras, decentralizované řešení kolizí s využitím detailních modelů letadel i okolního prostředí. AgentFly lze provozovat v plně distribuovaném režimu na více počítačích. Tento přístup umožňuje provádět simulace zahrnující velmi vysoké počty letadel. Systém také dovoluje měřit a vyhodnocovat množství parametrů a charakteristik probíhající simulace, díky čemuž lze detailně analyzovat chování simulované letecké dopravy, jako např. spotřebu paliva, hustotu provozu či dodržování bezpečných vzdáleností.[1]

1.3 Cíl práce

Cílem práce bylo vytvořit nástroj, který bude pomocí statistických testů filtrovat lety a bude přehledně vypisovat výsledky testů na výstup. Práce je zaměřena na řešení, které je v mnoha směrech v budoucnu rozšířitelné a konfigurovatelné, protože s používáním budou růst požadavky na aplikaci. Naopak cílem práce nebylo testování konkrétních dat a vytvoření závěrů o daném souboru letů. Aplikace musí pokrývat základní funkcionalitu: načtení souborů, provedení testů, filtrování

letů a výpis výsledků. Dále musí obsahovat základní sadu testů, které bude možné v budoucnu rozšířit podle konkrétní potřeby použití.

2. Analýza

2.1 Podobné nástroje

S růstem objemů informací existuje v dnešní době mnoho nástrojů pro analýzu a filtrování dat pomocí různých podmínek. Většina ze známých nástrojů je zaměřena na obecná data a žádný z nich není specializovaný na data z letového prostoru. Výhodou specializace může být specifické načítání dat a práce nejen s proudem dat, ale s letadly jako celky trajektorií. Navíc je vhodné pracovat se všemi lety najednou, což by nebylo možné při proudovém zpracování.

2.2 Jazyk a externí knihovny

Aplikace by měla být nezávislá na platformě, proto jsou vhodné jazyky, které není potřeba pro danou platformu zvlášť kompilovat. Jako hlavní dvě možnosti se nabízí jazyky C# a Java, jež jsou překládány do mezikódu. Pro konečnou volbu jazyku Java také hovoří možné budoucí propojení na úrovni zdrojových souborů se simulací, která je napsaná rovněž v Javě.

Externí knihovny kromě JDK nejsou v práci potřebné žádné. Výjimkou je knihovna org.json [2], kterou využívá aplikace pro snadné generování .json souborů jako jeden z podporovaných výstupních formátů.

2.3 Rozhraní

Pokud je potřeba interakce s uživatelem a nestačí, aby nástroj běžel jako skript, existují dvě varianty rozhraní. První možností je grafické rozhraní, kde uživatel může snadno ovládat aplikaci pomocí myši. Druhou možností je vytvořit menu v příkazovém řádku, ve kterém může uživatel používat pouze klávesnici. V tomto případě se jedná o implementačně jednodušší variantu, která nemusí být vždy uživatelsky přívětivá.

Jelikož tato práce implementuje menší nástroj, který je primárně konfigurovatelný ze souboru, aplikace funguje jako skript, případně uživateli nabízí konzolové

menu. Celé menu je nezávislé na chodu programu. V budoucnu může být aplikace rozšířena o grafické rozhraní.

Jediné grafické rozhraní, které aplikace v současné době podporuje, je vizualizace AnalysisImagine [3]. Vizualizace není součástí práce, ale je přiložena ve formě souboru .exe a je dostupná pouze na operačním systému Windows. Data jsou předávána pomocí socketu. Vizualizace byla napsána jako zápočtový program v jazyce C#, z tohoto důvodu není větší integrace mezi nástrojem pro analýzu a vizualizací.

2.4 Zprávy ve formátu CMS

Formát zpracovávaných vstupních souborů se váže na daný letový prostor. Aplikace musí být nezávislá na konkrétním formátu. Současným požadavkem je podpora letového provozu USA, ve kterém se používají zprávy typu CMS [4]. V budoucnu však může dojít k rozšíření o další formáty.

CMS zprávy jsou automaticky generované letadly každých 12 sekund. Na tento interval by se implementace ale neměla spoléhat, neboť by někdy mohlo dojít i ke zpoždění zpráv. Data jsou generována aplikacemi podle stejné specifikace, tudíž by neměly nastat problémy s neplatností dat a formát zpráv nemusí být validován. Ve vstupním souboru se může nacházet až 13 typů zpráv. Je však potřeba, aby aplikace podporovala alespoň čtveřici základních zpráv: FH, AH, TH a CL. Ostatní typy zpráv se netýkají přímo konkrétních letů, nejsou podstatné pro zkoumání trajektorií nebo se ve vstupních souborech obvykle nevyskytují.

Pro všechny zprávy je společné, že na začátku obsahují datum a čas následované jejich typem. Za těmito daty se nacházejí specifické seznamy hodnot[1] pro každý typ zprávy. Zprávy, které jsou vázány k danému letadlu, obsahují dále flight_id na indexu 1 a computer_id na indexu 2. V aplikaci již stačí označit let tzv. full_id, což je spojení předchozích dvou id: fid_cid. Ostatní položky jsou pro jednotlivé typy zpráv různé a ne všechny jsou zatím potřebné k porovnávání.

[1] Pro přehlednost bude index položky v seznamu začínat hodnotou 1.

2.4.1 FH a AH zprávy

FH Flight Plan Information Message

AH Flight Plan Amendment Information Message

FH zprávy předcházejí jednotlivým pozicím letu a uvádějí jeho budoucí trasu. Tyto zprávy se mohou během letu vyskytovat několikrát a mohou být pozměněné pomocí AH zpráv. Z informací ve zprávě se také pozná, že se letu změnilo computer_id, na což musí aplikace správně zareagovat. Formát zpráv FH a AH je totožný, proto mohou být zprávy typu AH načtené stejně jako zprávy typu FH.

Důležité položky zprávy:

```
20130620.000148004 FH STA3410|366||B733|Q|0506|429|3628N/8416W|
E0024|360||GSO261052..CGI..DLITE..FATSS..
BOOSH..QBALL..KSTL||||||||||||||||
```

```
20130620.003048812 AH MIS2003|088||MD88|Q|5623|455|3449N/8330W|
E0030|240||GVE216051./.3449N/8330W..BYRDS..
COSEL..WORFF..KATL||||||||||||||||
```

```
1 - flight_id
2 - computer_id
4 - typ letadla
12 - plán letu
```

2.4.2 TH zprávy

TH Track Information Message

Nejdůležitějším typem zpráv jsou jednotlivé pozice letů, které aplikace musí testovat. Mezi soubory se zprávy mohou lišit časem o několik sekund až minut, tudíž se budou lišit i souřadnice zpráv. Zprávy nelze porovnávat 1:1 a s pozicemi letů by se mělo pracovat jako s celými trajektoriemi.

Pozice bez předcházející FH zprávy musí být ignorovány, protože let nebyl korektně zahájen. Před první načtenou zprávou není jasné, jak trajektorie v počátku vypadala. Zprávy obsahují také informace o sektorech. Kvůli přesnosti si aplikace sektory spravuje sama, není potřeba tyto položky načítat.

Důležité položky zprávy:

```
20130620.000212000 TH BRL3124|320|443|290|326|ZCT|033|||
352554N/0810939W|+241/+372||

1 - flight_id
2 - computer_id
3 - rychlost letadla (pozemní rychlost)
4 - přidělená výška
5 - skutečná výška
10 - poloha letadla
```

2.4.3 CL zprávy

CL Cancellation Information Message

Tento typ zprávy se stará o korektní ukončení letu. Pokud pro daný let nebyla načtena CL zpráva, měl by být tento let ignorován. Také všechny zprávy daného letu za touto zprávou musí být ignorovány.

Důležité položky zprávy:

```
20130620.003100000 CL ALT1002|798|GCV|KATL|

1 - flight_id
2 - computer_id
3 - kód výchozího letiště
4 - kód cílového letiště
```

2.5 Vstup a výstup

Při opakovaném spuštění aplikace by mělo dojít ke zjednodušení zadávání vstupních nastavení běhu. Z toho důvodu je vhodné, aby byl na vstupu aplikace konfigurační soubor, který bude popisovat základní nastavení. Jelikož bude aplikace primárně sloužit pro porovnávání letů mezi soubory, je nutná přítomnost alespoň dvou vstupních souborů s letovými zprávami.

Po načtení vstupních souborů dochází k filtrování letů. Filtry musí být uživatelem snadno popsatelné. Je potřeba, aby popis povoloval filtrování podle více hodnot najednou. Dané hodnoty budou spojovány pomocí spojek and a or. Klauzule musí jít zapsat v konfiguračním souboru, tudíž návrh zápisu klauzulí by měl odpovídat formátu tohoto souboru. Pokud aplikace již poběží a uživatel bude chtít

přidat nový filtr, je vhodné mít druhý formát zápisu klauzulí, který je vhodnější pro zadávání v konzoli.

Po načtení zpráv a filtrování letů musí být výsledky testování předány uživateli. Tento výpis výsledků bude zobrazen v konzoli, a také vypsán do souboru. Výpis by měl probíhat v několika režimech:

- Výsledky statistických testů pro lety z jednoho souboru
- Porovnání výsledků testů pro lety ze dvou souborů
- Souhrnné statistiky pro jeden soubor
- Porovnání souhrnných statistik ze dvou souborů
- Výpis vyfiltrované množiny letů s vysvětlením podmínek klauzule

Zmíněné způsoby výpisu výsledků by měly mít několik formátů. Například formát čitelný pro uživatele nebo jeden z používaných datových formátů pro následné zpracování dat (příkladem může být formát JSON[2]).

2.6 Možné způsoby filtrování letů

- Rozdílná délka trajektorie letů
- Podobná délka trajektorie, ale rozdílný horizontální profil letu
- Výchozí bod letiště Atlanta, cíl na letišti Charlotte a rozdílný čas
- Letadlo typu Airbus A320 nebo A321 a rozdílná maximální rychlost
- Maximální vzdálenost trajektorií
- Rozdílný počet stoupání
- Typ letadla Airbus A320, ale jiný úhel klesání

[2]JavaScript Object Notation

3. Návrh

3.1 Vstup

3.1.1 Konfigurace

Na vstupu dostává aplikace konfiguraci, aby mohl proběhnout automatický režim, nebo aby uživatel nemusel pokaždé zadávat znovu nastavení v ručním režimu. Podle jejího obsahu se již řídí celý chod aplikace. V konfiguraci je potřeba zadávat vstupní soubory, způsob parsování zpráv, kritéria filtrování a formát výstupu.

Pro formát konfigurace se často používají dvě varianty, tzv. soubory .properties a konfigurace ve formátu xml. Formát .properties je vhodný zejména pro interní konfiguraci, kde je potřeba nastavit hodnotu některým proměnným. Formát xml je vhodnější pro konfiguraci celých objektů a je pro uživatele čitelnější. Z tohoto důvodu byla zvolena konfigurace ve formátu xml.

Z programátorského hlediska pro práci s xml konfigurací je vhodná validace pomocí xsd schématu, ve které se pomocí xml zapíše předpokládaná struktura konfigurace. Po této validaci se již může aplikace spoléhat na validitu konfigurace bez komplikovaného kódu v Javě. Pracovat s konfigurací jako s xml dokumentem není vhodné. Lepší variantou je tento soubor převést na strukturu datových objektů, se kterými se již pomocí getterů pracuje snadněji. Pro tento proces opět existuje jednoduchý nástroj, JAXB parser, který automaticky převede soubor ve formátu xml na objekty. Parser případně podporuje i opačný proces, takže uživatel může konfiguraci vytvořit v ručním režimu.

Při psaní konfigurace si musí uživatel dávat pozor na pětici znaků a nahradit je správně jejich entitami: < (<), & (&), > (>), "("), ’ (').

3.1.2 Vstupní soubory

Kromě konfiguračního souboru aplikace potřebuje ke svému chodu alespoň dva soubory s letovými zprávami. Jeden je označen jako "master", ostatní soubory jsou označeny jako "normal". Příznak "master"slouží k tomu, aby bylo jasné, vůči

kterému souboru se budou ostatní soubory porovnávat. Typickým použitím je označit soubor z reálného provozu jako "master", dále mít několik výstupů ze simulace a pomocí aplikace sledovat rozdíly v jednotlivých bězích simulace.

Aplikace předpokládá, že jsou vstupní soubory validní, jelikož jsou generované podle určitého standardu jinými aplikacemi. Proto parser netestuje validitu na této úrovni. Aby parser mohl data načítat do paměti proudově, předpokládá, že jsou zprávy ve vstupním souboru seřazeny vzestupně podle času. Problémem dat z reálného provozu je, že některé hodnoty nejsou uvedeny správně. Například se zde vyskytují zprávy s nulovou výškou nebo vzdálenosti mezi zprávami nedopovídají rychlosti letu (rozdíl časů vynásobený průměrem rychlostí se nerovná skutečné vzdálenosti ani s rozumnou tolerancí). Testy nad jedním souborem nejsou součástí filtrů. Pouze existují moduly parseru, které upozorní uživatele na některou z chyb a jsou konfigurovatelné pro automatickou opravu. Let může být i s tímto nedostatkem porovnatelný s jiným, tudíž by bylo špatně takový let úplně odstranit.

3.1.3 Parser

Vstupní soubory se zprávami je potřeba načíst do paměti, kde již nebudou v textové podobě, ale budou reprezentovány objekty. Pro tento úkol slouží parser. V aktuální verzi stačí, aby parser podporoval zprávy ve formátu CMS. V budoucnu musí být aplikace rozšiřitelná o další formáty. Dále je vhodné, aby byl parser konfigurovatelný, protože uživatel může chtít filtrovat a upravovat textové zprávy.

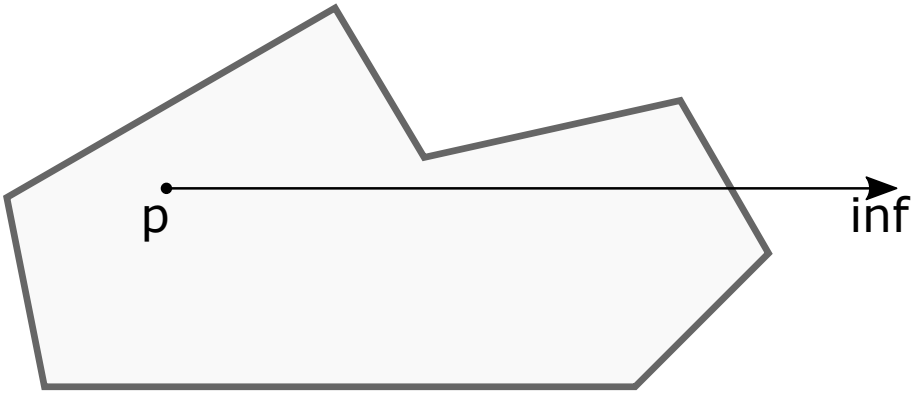
Z těchto důvodů je parser navržen tak, že se dělí do několika modulů. Moduly se postupně starají o čtyři části načítání: načtení zpráv ze souboru, převedení zpráv na objekty, upravení objektů se zprávami a finální zpracování. Každý modul je závislý na ostatních jen na úrovni rozhraní, proto je možné vyměnit jeden za druhý bez úpravy ostatních modulů.

Pro rychlejší načítání dat, která byla zpracována již v dřívějším běhu aplikace, slouží serializace do binárního souboru. Tato serializace proběhne na konci běhu aplikace. Při dalším spuštění není znovu potřeba načítat stejný textový soubor, stačí načíst binární soubor z minulého běhu.

3.2 Sektory

Vzdušný prostor daného státu se dělí na několik sektorů. V případě vzdušného prostoru USA, který je podporován aplikací, se jedná o dvacet sektorů. Pozice letů z reálného provozu jsou přibližně z daného sektoru. Trajektorie ze simulace na sektoru nezávisejí. Kvůli tomuto rozdílu aplikace načte hranice daného sektoru a provede filtr pozic přesně nad sektorem. Dále se již porovnávají takto ořezané trajektorie.

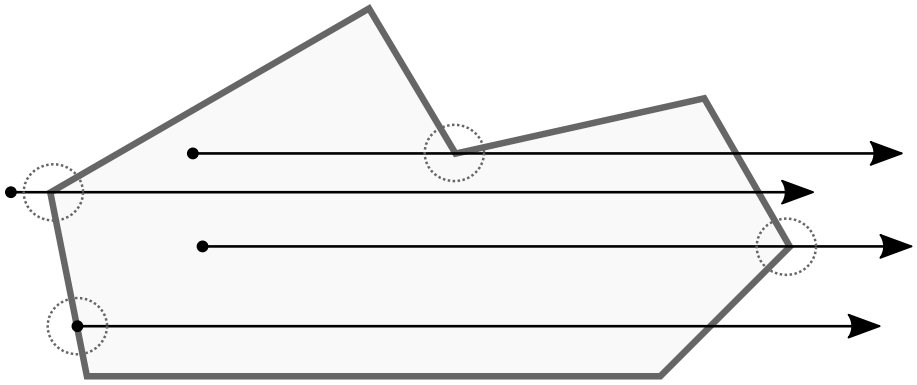
Pro filtrování pozic v sektoru je potřeba vyřešit, zda se bod (gps pozice) nachází uvnitř mnohoúhelníku (sektoru). Algoritmus je zvolen takový, že se daný bod propojí s bodem v nekonečnu (zde stačí například nulová zeměpisná délka) a provede se výpočet, kolik průsečíků má tato polopřímka s hranicemi sektoru. Pokud vyjde výsledný počet lichý, nachází se bod uvnitř tohoto mnohoúhelníku.



Obrázek 3.1: Znázornění algoritmu

V drtivé většině bodů funguje algoritmus ideálně. Existují však některé mezní hodnoty, které je potřeba ošetřit. Jedna z problémových situací je, když hraniční bod leží na testované polopřímce. V tomto případě se může správně připočítat průsečík dvakrát. Pokud bude bod uvnitř sektoru a polopřímka protne pouze hraniční bod, je potřeba, aby výsledný počet byl lichý. Těchto mezních hodnot je několik a nelze je jednoduše algoritmicky najednou všechny vyřešit.

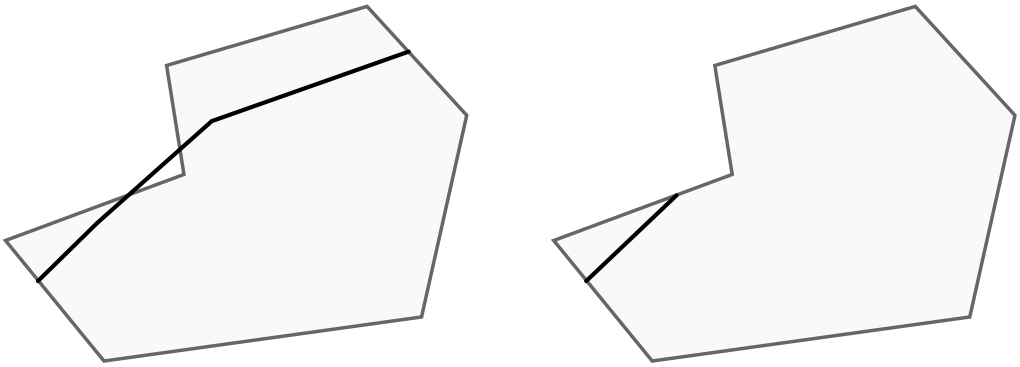
Z tohoto důvodu bylo zvoleno řešení, které vytvoří dva nové body posunuté o delta. Výpočet je pak proveden znovu. V případě, že jsou oba body uvnitř sektoru, musí být i původní bod uvnitř. Řešení je přesné, protože je delta menší než přesnost gps souřadnic. Pravděpodobnost, že tato úvaha selže, je velmi malá. Správnost celého programu by neměla být ohrožena.



Obrázek 3.2: Mezní body

Tento algoritmus byl zvolen z důvodu lineárnosti k počtu hraničních bodů, kterých je pro sektory průměrně 50. Optimalizace může proběhnout pomocí setřízení hraničních úseček podle zeměpisné šířky a výšky. Díky tomuto setřízení by byl výsledný počet testovaných hran ořezán.

Další optimalizace je v provedení výsledného rozhodnutí, zda je bod uvnitř sektoru. Pokud letadlo již jednou vyletělo ze sektoru, předpokládá se, že znovu již do sektoru nevstoupí a další pozice není třeba testovat. V reálném provozu ale existují i sektory, kdy letadlo opustí sektor pouze na několik kilometrů a optimalizace není funkční. Proto je tato funkcionalita volitelná při volbě parseru (viz kapitola 3.1.3 Parser).



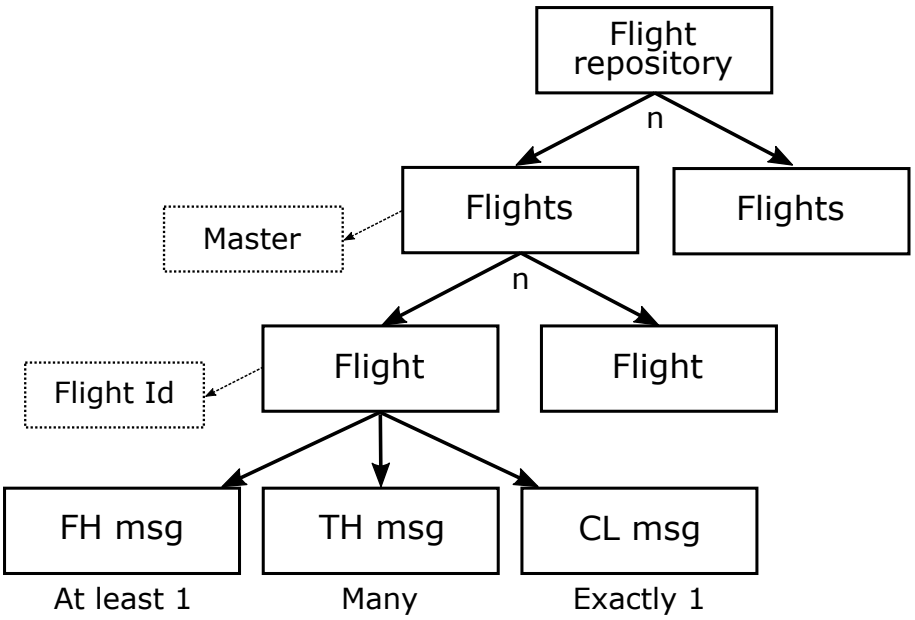
(a) Bez optimalizace (b) S optimalizací

Obrázek 3.3: Problémové situace při průletu sektorem

3.3 Datová struktura

Aplikace je navržena tak, že data nezpracovává pouze proudově, ale také je uchovává v paměti pro další použití. Uživatel se poté může rychleji dotazovat na výsledky a data se nemusí načítat ze souborů znovu.

Lety mají jednoznačný identifikátor složený z id letu a computer_id tohoto letu. Pro každý let existuje několik zpráv FH/AH, které jsou pouze informativní, a jedna CL zpráva. Hlavní zaměření je na zprávy TH, jež jsou uloženy seřazené podle času v každém letu. Lety ze stejného souboru musí být seskupeny, aby se s těmito soubory dalo pracovat jako s celkem. I jednotlivé načtené soubory je užitečné mít uložené na jednom místě. Vhodnou datovou strukturou může být kolekce Map, kde je klíčem název souboru z konfigurace.



Obrázek 3.4: Návrh datové struktury

Navíc lze v této datové struktuře ukládat další informace, jako jsou vyfiltrované lety pro určitý filtr, který soubor je master, proběhlé testy nad souborem a výsledky těchto testů.

3.4 Filtr

Hlavním použitím nástroje je filtrování letů na základě statistických testů. Tyto filtry musí splňovat dva hlavní cíle: musí být lehce konfigurovány uživatelem

a musí filtrovat lety co nejrychleji, aby uživatel mohl měnit konfiguraci a nemusel čekat na přepočítání výsledků.

3.4.1 Zápis klauzulí

Testy jsou založené na porovnávání hodnot. Proto je potřeba nabídnout uživateli pohodlný zápis filtrovacích klauzulí. Důležitou podmínkou pro návrh filtru je, aby byly možné logické operace "and"a "or". Negace stačí pouze na jednotlivých testech. Díky existenci těchto operátorů je možné pracovat pomocí výrokové logiky. Zápis klauzulí je zvolen ve dvou formátech podle prostředí, kde jsou klauzule uživatelem tvořeny.

První možností je zápis klauzulí v xml. Konfigurace filtrů se nachází v souboru společně s ostatními konfiguracemi celé aplikace. Existují tři typy elementů: <TestCase/>, <And/> a <Or/>. Spojky <And/> a <Or/> nemají žádné parametry, uvnitř ale obsahují elementy <TestCase/> nebo další spojky. <TestCase/> má název testu (name), název testované hodnoty (valueName), typ (type), který je obvykle znaménko, a hodnotu (value). Výhodou zápisu v xml je prvotní validace pomocí xsd schématu, protože uživatel snadno zjistí, že na některý parametr zapomněl. Správnost parametrů je kontrolována až při filtrování. Před hodnotu parametru type lze napsat znak !, který značí negaci daného výrazu.

Konfigurace filtrovací klauze v xml:

```
<And>
  <TestCase name="FlightGroundSpeed"
                    valueName="AVG" type=">" value="60"/>
  <TestCase name="AscDsc"
                    valueName="ASC" type="=" value="A-A"/>
  <TestCase name="AscDsc"
                    valueName="DSC" type="=" value="D-D"/>
  <Or>
    <TestCase name="FlightType"
                      valueName="FLT" type="=" value="A321"/>
    <TestCase name="FlightType"
                      valueName="FLT" type="=" value="A320"/>
  </Or>
</And>
```

Druhý způsob zápisu klauzulí je vhodný, když uživatel upravuje filtr v konzolovém rozhraní. Tento zápis se podobá zápisu funkcí v aplikaci MS Excel. Nevýhodou formátu je těžší parsování výrazu z textu a v případě delších klauzulí může

pro uživatele nastat horší čitelnost. Na druhou stranu se jedná o velmi jednoduchý zápis v případě konzole, jelikož uživatel potřebuje minimální počet znaků k vyjádření celé klauzule. I u tohoto zápisu existují tři typy elementů. Základní spojky OR() a AND() mají stejné použití jako u xml zápisu. Zápis daného testu je ve formátu: NázevTestu(valueName;type;value). Mezery, tabulátory a entery jsou ignorovány, ale zápis v konzoli podporuje pouze jednořádkové klauzule.

Konfigurace filtrovací klauze v konzoli[1]:

```
AND(
  FlightGroundSpeed(AVG,>,60);
  AscDsc(ASC,=,A-A);
  AscDsc(DSC,=,D-D);
  OR(
    FlightType(FLT,=,A321);
    FlightType(FLT,=,A322)
  )
)
```

Oba zápisy jsou velmi obecné, protože musí podporovat dynamicky načítané testy. Test obvykle nemá pouze jednu hodnotu, kterou porovnává. Hodnota lze vyjádřit pomocí valueName. Testy navíc mohou vyhodnocovat přesnou shodu nebo pouze dolní limit apod., proto neexistuje množina znaků, kteou je možné použít v parametru type. Pouze znak ! je rezervovaný pro negaci. Hodnota value je standardně typu String, který až jednotlivé testy parsují na požadovaný typ.

3.4.2 Předzpracování

Pro zajištění rychlosti chodu aplikace jsou testy předpočítány při načítání dat ze souborů. Výsledky jsou uloženy do jednotlivých letů. Poté již při filtrování projde aplikace tyto výsledky. Některé výsledky jsou spočítany zbytečně navíc, ale je předpokládané, že daný test proběhne pouze jednou při spuštění aplikace nad novými daty a při dalším spuštění již neprobíhá výpočet znovu.

V budoucnu některým testům předzpracování nemusí vyhovovat. Lze udělat kompromis, kdy neproběhne předzpracování, ale výsledky se budou počítat až při filtrování.

[1]Bílé znaky jsou připsány pouze pro přehlednost, jelikož uživatel musí klauzuli zapsat na jeden řádek.

Další nevýhodou předzpracování testů je, že algoritmy nemůžou být parametrizované z konfigurace. Tento problém by se mohl vyřešit definicí testů v konfiguraci, podobně jako je parametrizace modulů při načítání souborů. Avšak parser se vyskytuje v aplikaci pouze jednou a daný test se může vyskytovat v mnoha klauzulích. Pokud by v současné době chtěl programátor přidat test, který bude parametrizovaný, existují dvě možnosti. První možností je vytvořit několik testů. Parametrizace proběhne na základě jména testu a testy mohou dědit od jednoho výchozího, kterému předají správné parametry. Tento postup je vhodný například, pokud jsou potřeba pouze dva testy s jedním boolean parametrem v algoritmu. Další postup, jak vytvořit parametrizovaný test, je spočítat obě možnosti, a až na základě parametrů zvolit hodnotu. Nebo lze udělat kompromis a počítat hodnoty až při filtrování. Tento postup není doporučen, protože může výrazně zpomalit filtrování. Může se však stát, že se v budoucnu najde test, který se takovému chování nevyhne.

Na druhou stranu se nevýhoda předzpracování projeví pouze při filtrování s komplikovanějšími testy. V obvyklém případě by neměl nastávat problém. Myšlenkou celého předzpracování je spočítat pouze jednou všechny možné informace a být rychle připraven na dotazy uživatele. Jestliže se v budoucnu nahradí stávající grafické rozhraní, bude ještě důležitější, aby měl uživatel výsledky hned po kliknutí a nemusel by čekat na každý dotaz několik sekund.

3.4.3 Blacklist a whitelist

Před filtrováním může uživatel upravit množinu letů, která bude na vstupu. Pomocí blacklistu uživatel určí, které lety nebudou ve filtrované množině, a ve whitelistu vybere, které lety budou pouze v této množině. Zajímavá situace nastává, když by byly oba seznamy neprázdné. Řešením je, že blacklist proběhne až po whitelistu, tedy filtrování probíhá následovně:

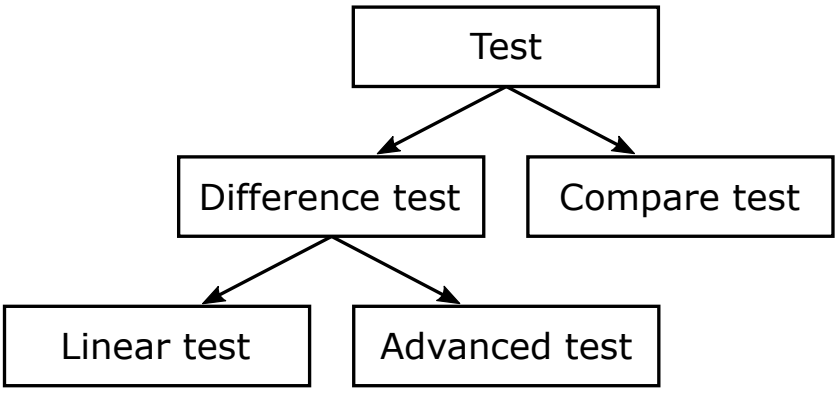
Blacklist \ Whitelist	Empty	Not on	On
Not on	—	filtered	—
On	filtered	filtered	filtered

Seznamy jsou textové soubory, kde se na každém řádku nachází jedno flight_id, které má být ve výsledném blacklistu nebo whitelistu.

3.5 Testy

Statistické testy jsou navrženy tak, aby před spuštěním filtru proběhly právě jednou. Pokud nový filtr používá testy, které již proběhly, testy nemusí být znovu spuštěny. Výpočet probíhá nad instancemi letů, které se skládají ze vstupních zpráv. Testy nesmí upravovat tyto lety, neboť k této činnosti slouží pouze moduly parseru. Po provedení výpočtu může test svou hodnotu uložit do daného testovaného letu nebo souboru. Uložená hodnota se ukládá pod zvoleným názvem do kolekce Map, tudíž tento název musí být jedinečný pro všechny testy. Z jiného místa, než z testu, se již s touto uloženou hodnotou nepracuje. Pokud některá část aplikace potřebuje získat vypočtené hodnoty, musí se dotázat na tato data pomocí testu.

Návrh počítá s hierarchií testů podle způsobu výpočtu testovaných dat.



Obrázek 3.5: Hierarchie testů

3.5.1 Lineární testy

Lineárním testům stačí předávat zprávy v pořadí jako ve vstupním souboru. Po skončení načítání zpráv je potřeba upozornit testy, aby spočítaly výsledky. Testy si ukládají mezivýsledky interně. Po skončení načítání jsou výsledky uloženy do objektů s jednotlivými lety. Příkladem testu může být výpočet délky letu.

3.5.2 Pokročilé testy

Existuje několik možností, proč všem testům nevyhovuje lineární přístup. Pro některé výsledky stačí zkoumat pouze první a poslední pozici z trajektorie. Je zbytečné, aby tyto testy dostávaly postupně celou trajektorii. Například u výpočtu délky letu se výsledná hodnota spočítá jako rozdíl časů první a poslední pozice. Další testy nepotřebují dostávat žádné pozice, neboť zkoumají ostatní zprávy, kterých je mnohem méně a není pro ně potřebný lineární přístup. Také mohou existovat testy, které pracují s trajektorií komplikovanějším způsobem (například pomocí kvadratického algoritmu), a proto potřebují celou trajektorii pro daný let. Posledním obvyklým způsobem testování je porovnávání celé množiny letů v daném souboru. Příkladem může být výpočet kolizí mezi lety.

Pokročilým testů nestačí dostávat zprávy lineárně. Na vstupu dostávají seznam všech načtených letů se zprávami a nejsou nijak limitovány.

3.5.3 Porovnávací testy

Předchozí varianty testů počítaly s tím, že testy proběhnou nad jedním souborem, a že se při filtrování letů výsledky letů ze souborů porovnávají. Může však nastat situace, kdy ani předchozí dva přístupy nejsou ideální. Příkladem je test, který porovnává jednotlivé pozice trajektorií pro danou dvojici letů. Tento typ testu dostane na vstupu lety z obou souborů a pracuje s nimi souběžně. Výsledky testu jsou uloženy pouze v jednom z porovnávaných letů, protože při filtrování již není potřeba počítat rozdíl výsledků.

3.5.4 Testování letů

Načítání probíhá pomocí čtení zpráv ze souborů. Následně se zprávy parsují a filtrují. Jelikož je tato operace zdlouhavá, parser je rozdělený do dvou vláken. Poslední část, consumer, probíhá v separátním vlákně. Během vytváření datové struktury jsou všechny pozice testovány lineárními testy. Díky tomuto rozdělení probíhá načítání a testování zpráv ze souboru souběžně. Po načtení zpráv je potřeba spočítat konečné výsledky lineárních testů a spustit ostatní testy.

Binární ukládaní načtených dat musí zohlednit, že výsledky compare testů jsou uloženy pouze v jednom ze souborů (konvence je, že jsou v non-master souboru) a nesmějí být serializovány, protože by mohlo dojít ke změně master souboru mezi běhy aplikace. Pokud by uživatel při ručním režimu změnil master soubor, musely by se přepočítat všechny compare testy.

3.6 Výstup

Po spuštění testů a následném filtrování letů je potřeba nabídnout výsledky uživateli. Nastává několik možností formátů výsledků: výsledky pro jeden soubor, porovnání dvou souborů, výsledky po filtrování, souhrny pro soubory.

Důležitou částí návrhu je, aby možné výstupy z aplikace byly snadno rozšiřitelné o další formáty, jako je tomu u testů nebo parseru. Základním formátem musí být tabulka, která je uživatelem snadno čitelná v textovém editoru nebo v konzoli. Pro snadné následné zpracování výsledků je potřeba implementovat formát, který je standardní pro tabulkové editory nebo aplikace pro vizualizaci dat. K tomuto účelu je vhodný snadno generovatelný formát CSV[2], případně formát JSON.

Speciálním případem pro výpis výsledků jsou výsledky filtrů, kde je důležité předat informaci, proč byly jednotlivé lety vyfiltrované, respektive které části klauzulí byly kladné.

Návrh výstupu z filtru:

```
-------------------------------------------------------------------
           |                          AND
           |     ?    |                OR                |GrDist-VAL
           |          |     ?    |AltMix-DSC|AltMix-ASC|
  FlightId |          |          |    > 90  |   > 90   |   < 30
-------------------------------------------------------------------
NOI0212_003|   true   |   true   |          |   100.0  |   9.99
AOA1153_973|   true   |   true   |   95.83  |          |   2.92
```

[2]Comma-separated values, hodnoty na jednom řádku jsou oddělené čárkou nebo případně středníkem.

3.7 Uživatelské rozhraní

Aktuální návrh aplikace počítá pouze s konzolovým menu. Jelikož je menu v textové formě, bude se uživatel v nabídce pohybovat pomocí klávesnice. Jednotlivé volby menu jsou očíslovány. Uživatel vybere danou možnost zadáním čísla. Tento přístup je snadný a aplikace je rozšiřitelná o další položky v menu. Struktura menu je rozdělena do několika částí, které jsou dostupné z hlavní nabídky:

Configuration poskytuje možnosti pro úpravu konfiguračního souboru.

Run tests načte nové soubory a spustí testy.

Filter spustí filtrování, uživatel má možnost rozšířit množinu spuštěných filtrů.

Results vypíše výsledky ve vybraném formátu.

Visualization spouští vizualizaci společně se serverem pro posílání dat.

Příklad menu:

```
======================
Main menu
======================
[1] - Configuration
[2] - Run tests
[3] - Filter flights
[4] - Results
[5] - Visualization

[0] - Close

Choice [0]-[5]:
```

3.8 Vizualizace

Vizualizace není součástí implementace. K programu je přiložena jako spustitelný .exe soubor. Spuštěná vizualizace se chová jako klient a pomocí socketu na portu 8000 se dotazuje na server pro data. Port lze případně změnit. Pro komunikaci slouží spustitelný server ve vedlejším vlákně. Server je schopen poslat celý seznam vyfiltrovaných letů nebo pouze jeden vybraný let. O zbytek chodu vizualizace se již aplikace nestará, uživatel ovládá vizualizaci přímo.

3.9 Příklady spuštění aplikace

3.9.1 Automatický běh

java -jar analysis.jar –a configuration.xml

Hlavním použitím aplikace bude testování výstupů mezi běhy simulace proti stejným datům z reálného provozu. V tomto případě je vhodné mít jeden konfigurační soubor, který bude obsahovat všechna data a uživatel aplikaci spustí pouze jako skript. Jakmile aplikace doběhne, uživatel může projít vygenerované soubory, případně uvidí výstup v konzoli.

3.9.2 Ruční režim

java -jar analysis.jar

Další možností pro ovládání aplikace je ruční režim, kde uživatel postupně prochází přes jednotlivá menu a testuje svá data. Výhodou tohoto režimu je, že uživatel může chtít ladit parametry filtru a nemusí pokaždé načítat data, protože pracuje s daty načtenými v paměti. Kvůli tomuto použití je důležité, aby aplikace předpočítala hodnoty testů již během načítání a po změně parametrů proběhlo pouze nové filtrování. Po filtrování má uživatel navíc možnost spustit vizualizaci jednotlivých vyfiltrovaných trajektorií. Pomocí vizualizace uživatel snáze uvidí, proč je let na výstupu, oproti čtení tabulek hodnot.

Menu by mělo být nezávislé na celé aplikaci, aby v budoucnu mohla být jeho konzolová varianta nahrazena okenní aplikací. Přechod na okenní menu, které bude propojené se simulací, je v budoucnu velmi pravděpodobný. Při implementaci se na tuto možnost musí brát ohled.

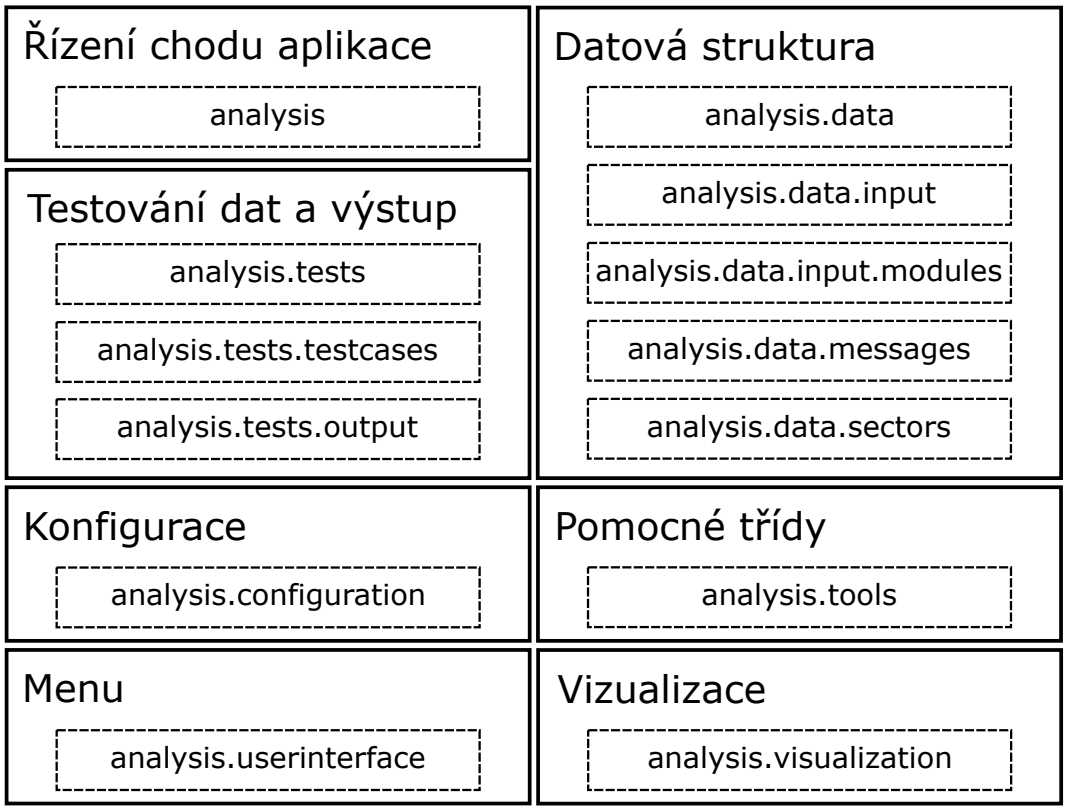
3.9.3 Validace

java -jar analysis.jar –v configuration.xml

V některých případech může chtít uživatel pouze validaci vstupní konfigurace, kterou si napsal ručně, aby nenarazil na chyby až při testování dat. Pokud validace proběhne úspěšně, bylo by vhodné vypsat celou konfiguraci do konzole, kde uživatel nalezne čitelnější formát, než je xml formát.

4. Implementace

4.1 Struktura aplikace



Obrázek 4.1: Logické rozložení aplikace

Aplikace je rozdělena na sedm logických celků, které obsahují jednotlivé balíčky s třídami:

analysis se stará o chod celé aplikace a propojuje testy s datovou strukturou. Navíc třída Main obsahuje statickou metodu main, která slouží pro spuštění celé aplikace.

analysis.data.* obsahuje třídy, které popisují načtená data ze vstupních souborů, a parser, který je rozdělený na jednotlivé moduly. Hlavním úkolem tohoto celku je načíst, uchovat a případně serializovat data.

analyis.tests.* obsahuje třídy pro testování dat a vypsání výsledků. V package testcases jsou uloženy všechny implementace testů a předpokládá se, že zde bude aplikace nejčastěji rozšiřovaná.

analysis.configuration načítá, uchovává, upravuje a ukládá konfiguraci celé aplikace.

analysis.userinterface obsahuje jednotlivé třídy pro zobrazení menu v ručním režimu.

analysis.visualization spouští vizualizaci a server pro komunikaci.

analzsis.tools obsahuje pomocné třídy pro práci s gps pozicemi, serializaci neprimitivních typů apod.

4.2 Konfigurace

Konfigurační soubor se dělí na čtyři části:

InputConfiguration jsou cesty ke vstupním souborům, jejich typ a název, který slouží pro identifikaci uživateli. Také se zde nacházejí cesty k souborům s blacklistem a whitelistem.

OutputConfiguration obsahuje výstupní adresář a formát výstupu.

ParserConfiguration konfiguruje jména a nastavení jednotlivých modulů parseru.

TestConfiguration obsahuje aktuálně použité filtry a knihovnu všech známých filtrů s jejich názvem a popisem.

Konfigurační třídy slouží pouze pro uchovávání dat, tudíž neobsahují validace dat. Pomocí anotací jsou třídy připraveny pro unmarshalling z xml souboru a marshalling do xml souboru JAXB[1] parserem [5].

Většina aplikace pouze získává hodnoty z konfigurace. Uživatel má v menu možnost konfiguraci upravit. Aby bylo zamezeno upravovat konfiguraci z jiné části, než je package analysis.configuration, jsou všechny metody kromě getterů package-private. Pro práci s konfigurací z jiných částí aplikace byl vytvořen Handler, přes který je možné konfiguraci upravovat. V Handleru jsou různé kontroly, aby nedošlo k vložení nevalidních dat do konfigurace. Navíc se Handler stará

[1] Java Architecture for XML Binding

o blacklist a whitelist, které načte ze souborů. V případě úpravy blacklistu nebo whitelistu uloží Handler seznam opět do stejných souborů. Navíc si pamatuje, zda byla konfigurace upravena během chodu aplikace, aby mohlo dojít k její aktualizaci.

analysis.configuration

(C) **Configuration**[2] je kořenem celé konfigurace a pouze drží reference na jednotlivé konfigurace různých částí aplikace. Konkrétní konfigurace částí aplikace jsou: InputConfiguration, OutputConfiguration, ParserConfiguration a TestConfiguration. Všechny konfigurační třídy obsahují mnoho vnitřních tříd, které reprezentují konkrétní hodnoty. Například InputConfiguration obsahuje třídu InputFile, ve které je název souboru a cestu k tomuto souboru.

(C) **ConfigurationHandler** slouží pro pohodlnější práci s konfigurací. Pomocí handleru je možné upravovat konfiguraci a spravovat whitelist a blacklist.

(C) **ConfigurationLoader** načítá a validuje konfiguraci ze souboru. Loader může vytvořit prázdný konfigurační soubor, který je validní.

(C) **ConfigurationSaver** ukládá konfiguraci do souboru.

(R) **configuration.xsd** obsahuje popis celého konfiguračního souboru. Pomocí xsd schématu lze jednoduše validovat strukturu xml souboru.

4.3 Datová struktura

Kořenem celé datové struktury je objekt FlightRepository, který by se v aplikaci měl vyskytovat pouze jednou. V tomto objektu jsou jednotlivé načtené soubory uložené pod identifikátorem z konfigurace. Dále je zde poznamenáno, který soubor je typu master a vyfiltrované množiny letů k použití pro výstup. Pro účely opožděného načítání souborů jsou zde také uložené parserem nenačtené soubory. Načtení těchto souborů probíhá, až uživatel tento proces zahájí, protože

[2]Konvence pro označení tříd v textu: C - třída, A - abstraktní třída, I - rozhraní, R - zdroje

je vhodné, aby načítání souborů probíhalo najednou a ne při každém přidaném souboru.

Jednotlivé vstupní soubory jsou reprezentovány objektem Flights, ve kterém jsou v mapě uloženy jednotlivé lety s klíčem flight_id. Testy si mohou v mapě uložit své předpočítané souhrny pro dané soubory. Pro zajištění spuštění každého výpočtu nad souborem pouze jednou, jsou zde poznamenány proběhlé testy.

Pro reprezentaci jednotlivých letů slouží objekt Flight. Tento objekt obsahuje zprávy, které jsou rozdělené podle jednotlivých typů, výsledky testů a také informace o názvu letu a typu letadla.

FlightRepository i Flights implementují rozhraní Iterable, a pokud programátor potřebuje projít všechny soubory (Flights), respektive lety (Flight), pracuje se s těmito objekty snadněji.

Pro zprávy existuje abstraktní třída Message, která obsahuje čas vzniku zprávy a její typ. Pro typ zprávy slouží enum MessageType. Konkrétní implementace zpráv již obsahují hodnoty, které jsou uloženy v jednotlivých textových zprávách na vstupu. Jelikož jednotlivé zprávy jsou uloženy v objektech Flight, neobsahují identifikátor letu.

analysis.data

Ⓒ **Flight** představuje jeden let, který se skládá ze vstupních zpráv. Kromě vstupních zpráv jsou zde uloženy výsledky předspracovaných testů pro daný let.

Ⓒ **Flights** pouze seskupuje lety z jednoho vstupního souboru. Stejně jako objekt Flight uchovává souhrny předspracovaných testů pro daný vstupní soubor.

Ⓒ **FlightRepository** primárně obsahuje mapu s načtenými třídami Flights s názvem souboru jako klíčem. Další uložené informace jsou: tzv. master soubor, vstupní soubory, které nebyly doposud načteny a seznam vyfiltrovaných letů pro každý spuštěný filtr.

Ⓒ **FlightPair** slouží k uchování dvojic letů se stejným flight_id ze dvou vstupních souborů. Třídy, které porovnávají data v souborech, dostávají již seznamy s lety uvnitř této třídy.

analysis.data.messages

(A) **Message** je abstraktní třída pro všechy typy zpráv. Obsahuje enum MessageType, podle kterého jsou typy zpráv identifikovány. Jednotlivé typy zpráv jsou namapované na své potomky třídy Message následovně: TH - Position, AH/FH - FlightPlan, CL - CancellationInformation.

(C) **Analyzer**

(C) **GpsPosition** reprezentuje GPS pozici společně s výškou. Pro práci s pozicemi slouží pomocná třída GpsPositionTools.

Konfigurace vstupních souborů:

```
<InputConfiguration>
    <Files serialize="true">
       <File name="Real" type="master">inputdata/real.in</File>
        <File name="Agent" type="normal">inputdata/agent.in</File>
    </Files>
    <BlackList>inputdata/blacklist.txt</BlackList>
    <WhiteList>inputdata/whitelist.txt</WhiteList>
</InputConfiguration>
```

4.4 Načítání vstupních zpráv

Načítání souborů probíhá ve dvou režimech: načtení zpráv a spuštění testů nebo se načtou binární data z předchozího běhu (viz kapitola 4.5 Binární načítání). Testy, které již proběhly v minulém běhu, se znovu počítat nemusí.

O první způsob načítání zpráv se stará modulární parser. Hlavním požadavkem na parser byla možnost jeho konfigurace. Proto byl rozdělen do čtyř fází, které jsou na sobě závislé na úrovni rozhraní a dále jsou již vyměnitelné. Jednotlivé moduly dědí od abstraktních tříd InputReader, InputParser, InputProcessor, InputConsumer.

Každý modul může mít libovolný počet parametrů, které se v konfiguraci zapisují pomocí elementu Property, kde parametr musí mít název a hodnotu. Parametry jsou převedeny do kolekce Map<Stirng, String>, klíčem je název a hodnotou je hodnota z elementu. Pokud se u modulu vyskytuje více parametrů se stejným jménem, hodnoty parametrů jsou sloučeny do jedné hodnoty pomocí

středníku jako oddělovače a poté jsou předány modulu. V případě, že uživatel neuvede některý z parametrů, moduly mají výchozí nastavení. Kvůli zachování obecnosti návrhu konfigurace parseru, nelze parametry vynutit při validaci.

Z objektového návrhu vyplývá, že je zbytečné, aby byl název letu obsažen i v objektech jednotlivých zpráv. Aby parser mohl snadno předávat modulům textovou zprávu, název letu, ke kterému zpráva patří, a zprávu ve formě objektu, moduly používají MessageWrapper, který tyto informace obsahuje. Navíc je zde zaznamenáno, zda je zpráva validní.

Parser je rozdělen do dvou vláken, protože s rostoucím počtem filtrů a testů může nastávat zpoždění. Po průchodu všemi processory se zprávy vloží do synchronizované fronty, odkud si zprávy vytahuje Consumer. Při malém počtu výpočtů nedochází k ušetření času. Aplikace je však připravena na budoucí vyšší vytížení.

analysis.data.input

Ⓒ **MessageProcessor** řídí celé parsování vstupu, vytváří instance modulů s jejich konfigurací.

Ⓒ **InputModuleLoader** načítá pomocí třídy ClassLoader instance pojmenovaných tříd z balíčku analysis.data.input.modules.

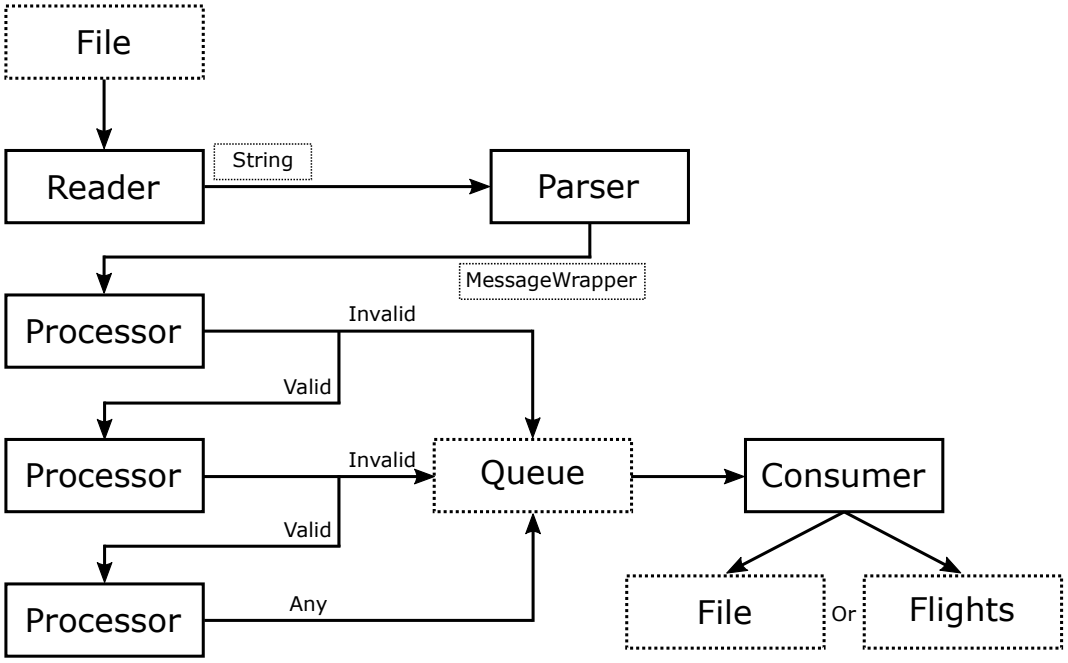
Ⓒ **MessageWrapper** je pomocná datová třída, která je předávána mezi moduly. Obsahem třídy je textová zpráva ze vstupu, zpráva převedená do objektu, flight_id letu a informaci, zda je zpráva validní.

Konfigurace parseru:

Konfigurace parseru:

```
<ParserConfiguration>
    <Producer name="LineReader"/>
    <Parser name="CMSMessageParser"/>
    <Processor name="MessageFilter">
        <Property name="lowaltitude">0</Property>
    </Processor>
    <Processor name="FlightIdRename"/>
    <Processor name="SectorFilter">
        <Property name="sector">ZTL</Property>
        <Property name="algorithm">FAST</Property>
```

```
    </Processor>
    <Consumer name="FlightsAnalyzer"/>
</ParserConfiguration>
```



Obrázek 4.2: Parsování souboru pomocí MessageProcessoru

4.4.1 InputReader

Modul reader slouží pro čtení zpráv ze vstupu. Pro budoucí rozšíření, kdy zprávy nemusí být pouze na jednom řádku, vznikl reader jako samostatný modul s vlastní konfigurací. V této konfiguraci může přibýt například kódování vstupního souboru apod. Jelikož na vstupu dostává Reader cestu ke vstupnímu souboru a sám otevírá vstupní stream, třída implementuje rozhraní Closeable. Poté lze třídu s tímto rozhraním použít v try-with-resources a usnadnit tak práci programátorovi.

analysis.data.input.modules

Ⓒ **LineReader** postupně načítá vstupní soubor, kde je každá zpráva na samostatném řádku a tyto zprávy předává dále.

4.4.2 InputParser

Modul převádí textovou reprezentaci zprávy na objekty, které jsou uloženy v paměti pro další zpracování. Parser obecně předpokládá, že jsou zprávy ve vhodném formátu. Nemusí se tudíž starat o validitu dat. Pokud aplikace dostane na vstupu data v nevalidním formátu, může se parser chovat neočekávatelně. Validace formátu není implementována kvůli zrychlení celého načítání.

Například pro zprávy ve formátu CMS existují regulární výrazy, které popisují, jak mají jednotlivé zprávy vypadat. Pokud by se ukázalo, že ve vstupních zprávách nastávají chyby, nebylo by těžké rozšířit parser o validaci, která by mohla být pomocí konfigurace zapnutá/vypnutá.

analysis.data.input.modules

Ⓒ **CMSMessageParser** z textových zpráv ve formátu CMS vytváří objekty daného typu. Podporované typy jsou mapovány následovně: FH, AH - FlightPlan, TH - Position, CL - CancellationInformation. Ostatní typy zpráv parser přeskakuje.

4.4.3 InputProcessor

Jedná se o jediný modul, který se může vyskytovat libovolněkrát. Processor má dvě použití, avšak některé implementace processoru mohou existovat pouze pro jednu variantu.

Prvním možným použitím procesoru je provádění úprav zpráv přímo v instanci MessageWrapper, kterou processor dostává v parametru funkce process. Příkladem použití je změna id letu, úprava jednotlivých dat zprávy podle podmínky apod.

Druhým použitím modulu je validace vstupních zpráv. Validace je potřebná například, když je některá hodnota neplatná. Pokud processor rozhodne, že se jedná o nevalidní zprávu, označí na zprávě hodnotu isValid false. MessageProcessor zareaguje na změnu tak, že ji nepředá dalším processorům. Konkrétním příkladem je používaný processor, který označuje zprávy s nulovou výškou za nevalidní, protože se ukázalo, že se takové hodnoty někdy vyskytují v datech z reálného provozu a jsou nežádoucí pro následné zpracování.

Jelikož nevalidní zprávy nejsou posílány k další transformaci, záleží na pořadí jednotlivých processorů. Nevhodné pořadí může ovlivnit chod aplikace, například pokud jsou lety filtrované pomocí SectorFilter. Je vhodné, aby tento filtr proběhl mezi prvními, protože zprávy mimo sektor budou zbytečně transformovány jinými processory.

analysis.data.input.modules

Ⓒ **SectorFilter** filtruje, zda se jedná o pozici uvnitř sektoru. Algoritmus a popis vnitřní struktury filtru se nachází v kapitolách [odkaz X a Y]. Z konfigurace dostává informaci, zda má proběhnout v optimalizovaném režimu nebo testovat všechny zprávy.

Ⓒ **MessageFilter** je konfigurovatelný filtr, který filtruje zprávy na základě podmínky. V současné implementaci podporuje znevalidnění pozic, které mají menší výšku, než je hodnota v konfiguraci.

Ⓒ **FlightIdRename** přejmenovává flight_id na první použité flight_id pro daný let. Celé jedinečné id letu (XYZ123_456) se skládá z id (XYZ123) a computer_id (456). Během letu může proběhnout změna computer_id. Jelikož aplikace potřebuje informaci o celé trajektorii, jsou všechny přejmenované lety přejmenovány zpět na první použité computer_id.

4.4.4 InputConsumer

Consumer dostává již upravené zprávy, které jsou navíc rozdělené na validní/nevalidní, a dokončuje celé parsování. Parsování může proběhnout ve dvou režimech. První režim (FlightsCreator) vybuduje celou datovou strukturu. Druhý režim (OutputCreator) pouze vypíše data na výstup a skončí.

Během režimu, který vytváří datovou strukturu obvykle proběhnou všechny potřebné testy. Kvůli testování, jež může trvat déle než ostatní operace a návaznosti na procesory, které také mohou trvat déle, probíhá consumer v separátním vlákně a zprávy dostává v synchronizované frontě. Z toho vyplývá, že consumer implmentuje rozhraní Runnable, aby byl snadno spustitelný ve vlastním vlákně.

Řešení s více vlákny je připravené na složitější kombinace procesorů a testů. V budoucnu by mohla přibýt volba, zda má proběhnout načtení ve více vláknech

nebo ne, neboť při menším počtu operací je zbytečným zpožděním synchronizace fronty.

Jakmile proběhne celé načítání, consumer je notifikován, aby dokončil svou činnost.

analysis.data.input.modules

Ⓒ **FlightAnalyzer** každou zprávu vloží do datové struktury. Pokud se jedná o pozici, proběhne testování pomocí lineárních testů. Po kompletním načtení zpráv jsou odebrány lety bez CL zprávy (tato zpráva ukončuje korektně let) a proběhnou ostatní testy. Poté consumer skončí.

Ⓒ **MessageWriter** pouze bere vstupní validní zprávy a ukládá je do souboru. V budoucnu může dojít k rozšíření konfigurace, kdy bude například zapisovat i nevalidní zprávy, protože jej zajímá transformace zpráv a ne validita.

4.5 Binární načítání

Pro zrychlení načítání souborů, které již aplikace načetla v jednom z předchozích běhů, slouží serializace datové struktury do binárního souboru. V konfiguračním souboru nebo při ukončení aplikace lze zvolit, zda se budou data serializovat.

Každá třída datové struktury, která podporuje serializaci, musí implementovat rozhraní Externalizable. Toto rozhraní požaduje implementaci metod read a write, které zapisují, respektive čtou ze streamu.

Výsledky testů jsou také uloženy do binárního souboru, aby nemuselo proběhnout opětovné testování zpráv. Výjimkou jsou výsledky Compare testů, u kterých nejsou výsledky uloženy v binárním souboru a testování musí proběhnout pokaždé, protože nejsou vázány pouze na jeden soubor.

Díky konfigurovatelnému parseru se mohou načíst dvě rozdílné datové struktury z jednoho vstupního souboru. Proto se na začátku binárního souboru nachází hash parseru použitého během načítání. Další kontrolou je hash originálního souboru, aby nemohlo dojít k záměně vstupního souboru se stejným jménem a hash dat v binárním souboru pro kontrolu poškození souboru.

Před samotným načítáním jsou kontroly spuštěny postupně za sebou. Pokud jedna z kontrol selže, proběhne načítání z originálního souboru.

Konvence jména souboru, který je vytvořen, je .bin za původním jménem. Podle této konvence zkusí také aplikace před načítáním z textového souboru vyhledat binární soubor ve výstupním adresáři, aby zrychlila načítání.

analysis.data

(A) **BinaryHandler** dodává potřebné funkce pro zajištění validace binárního souboru. Dále se v této třídě nachází funkce pro zápis neprimitivních typů. Například typ String je zapsán do binárního souboru jako 32-bitové celé číslo reprezentující délku textu následované jednotlivými znaky. Podobný formát je použit u všech podporovaných kolekcí. Obě následující třídy dědí od této abstraktí třídy, aby mohly využívat její metody.

(C) **BinaryReader** po validaci binárního souboru načte data a vytvoří pro daný soubor objekt Flights, který obsahuje načtené lety.

(C) **BinaryWriter** uloží do binárního souboru objekt Flights společně s kontrolními hashi.

4.6 Sektory

Hranice sektorů se načítají z interního .xml souboru, kde jsou popsané hraniční body seřazené v pořadí po obvodu. Jelikož existuje jen jeden interní .xml soubor se sektory z letového prostoru USA, aplikace podporuje pro filtrování pouze data z tohoto letového prostoru. Pokud by v budoucnu bylo potřeba rozšířit stávající sektory a formáty popisů sektorů by se lišily. Nejlepší variantou by bylo předzpracovat data do jednotného formátu, ve kterém bude aplikace, a poté data jednoduše zpracovávat. Také může být vhodná serializace, aby se urychlilo načítání sektorů.

analysis.data.sectors

(C) **Sectors** se stará o načítání sektorů ze vstupního souboru a vytváří instance třídy Sector, které si drží v paměti asociované s názvem sektoru. Pro parso-

vání třída používá xml parser DOM, pomocí tohoto parser načte potřebné hodnoty z elementů v souboru s popisem sektorů.

(C) **Sector** obsahuje jednotlivé obvodové body pro daný sektor uložené pomocí třídy GpsPosition s nulovou výškou. Hlavním použitím třídy je funkce isInside, která pro daný bod určí, zda se nachází uvnitř sektoru.

(C) **PositionFilter** poskytuje optimalizované filtrování pozic. Třída obaluje jednotlivé sektory a navíc si pro daný let pamatuje, zda vstoupil do sektoru nebo zda již sektor opustil. Pokud se jedná o případ, kdy letadlo již opustilo sektor, pak neprobíhá testování pozice, ale rovnou je rozhodnuto, že je pozice mimo sektor. Tato metoda nemusí být vždy přesná, a proto je volitelná.

(R) **secotrsUSA.xml** popisuje hranice sektorů letového prostoru USA.

4.7 Testy

Dle návrhu se testy dělí na dvě kategorie. První kategorií jsou porovnávací testy, které testují jeden soubor a při filtrování jsou výsledky porovnány mezi soubory. Druhá kategorie rovnou porovnává konkrétní dva soubory s lety a pouze tato hodnota je na výstupu. Porovnávací testy se dále dělí na lineární testy a testy, které potřebují pro spočítání výsledků celou trajektorii letu nebo všechny lety najednou.

analysis.tests

(C) **TestRepository** je hlavní úložiště načtených testů, které jsou zde uloženy podle typů. Třída navíc slouží pro analýzu testů při úpravě konfigurace v ručním režimu.

(A) **Test** je kořenovou abstraktní třídou pro všechny implementace testů. Obsahuje pouze obecné metody pro vytváření predikátů. Ostatní potřebné metody jsou specifické pro její potomky.

(A) **DifferenceTest** je v hiearchii tříd mezi Test a LinearTest nebo AdvancedTest. Pro tyto dvě třídy dodává společné metody pro práci s výsledky daného testu.

(A) **LinearTest** dodává třídě DifferenceTest metody pro lineární analýzu letů a je také abstraktní.

(A) **AdvancedTest** je na stejné úrovni jako LinearTest, ale dodává metody pro nelineární zpracování třídě DifferenceTest.

(A) **CompareTest** dodává třídě Test metody pro přímé porovnávání dvou letů a specifické získávání výsledných hodnot, protože oproti DifferenceTest je na výstupu pro každý typ hodnoty jen jeden výsledek.

(I) **NegationTest** slouží pro označení, že daný test sám vytváří predikát pro negaci.

(C) **TestLoader** dynamicky načítá instance implementací konkrétních testů podle jména.

4.7.1 Lineární testy

Lineární testy slouží k proudovému zpracování pozic. Konkrétní implementace tohoto typu musí podporovat dva režimy. Prvním režimem je optimalizované proudové načítání, kdy test dostává zprávy v pořadí, jak jsou uloženy ve vstupním souboru. Druhý režim, který je jako u AdvancedTest, slouží pro testování souborů, které jsou v paměti již načtené, ale test na nich ještě nebyl spuštěný. Pokud by měly testy pouze první implementaci, při spuštění druhým způsobem by nastával zbytečný overhead, protože již není potřeba ukládat mezivýsledky do interní paměti. Návrh nutí vývojáře k dvojí implementaci téhož algoritmu. Pro optimalizaci načítání je tento postup nevyhnutelný.

analysis.tests.testcases

(C) **FlightHeightTest** zkoumá maximální, minimální a průměrnou výšku. Maximální a minimální výška letu je aktualizována pro každou pozici. Prů-

měrná výška se dopočítává na konci testování jako suma výšek vydělena počtem pozic.

Ⓒ **AltitudeMixTest** porovnává výšky mezi dvěma sousedními pozicemi a počítá poměr mezi klesáním, stoupáním a udržováním výšky během letu. Výstup je například stoupání 20%, klesání 18% a stejná výška 62%.

Ⓒ **AscDscTest** rozhoduje pomocí vnitřní konstanty (která je neměnná z důvodu předpočítání), zda letadlo stoupalo více než 20FL a nebo klesalo více než 50FL. Klesání je definované tak, že letadlo během této akce nesmí ani jednou stoupat a může mít maximálně 10 neklesajících pozic.

Ⓒ **FlightGroundSpeedTest** počítá celkovou vzdálenost letu jako součet vzdáleností mezi navazujícími pozicemi. Pro výpočet vzdálenosti mezi dvěma body na Zemi je zvolen vzorec vycházející z Kosinovy věty pro strany [6]:

$$d = acos(sin(lat1)sin(lat2) + cos(lat1)cos(lat2)cos(long1 - long2))R$$

4.7.2 Pokročilé testy

Vstup pokročilých testů je úplný seznam letů daného souboru. Oproti lineárním testům tento případ nepodporuje analýzu jednotlivých zpráv při načítání.

analysis.tests.testcases

Ⓒ **FlightDurationTest** vypočítává délku letu pomocí rozdílu časů první a poslední načtené pozice.

Ⓒ **FlightTypeTest** nepracuje s pozicemi, ale při filtrování porovnává typ letu, který je uložený v FH nebo AH zprávě.

4.7.3 Porovnávací testy

Tento typ testů se liší od předchozích tak, že na vstupu dostává dva seznamy letadel, které porovnává a výsledky uloží do non-master souboru.

analysis.tests.testcases

[2]Flight level - letová hladina

Ⓒ **DistanceError2DTest** porovnává jednotlivé pozice mezi soubory. Výsledkem testu je průměrná, maximální, minimální, první a poslední vzdálenost mezi jednotlivými body trajektorií.

4.8 Filtrování

4.8.1 Parsování klauzulí

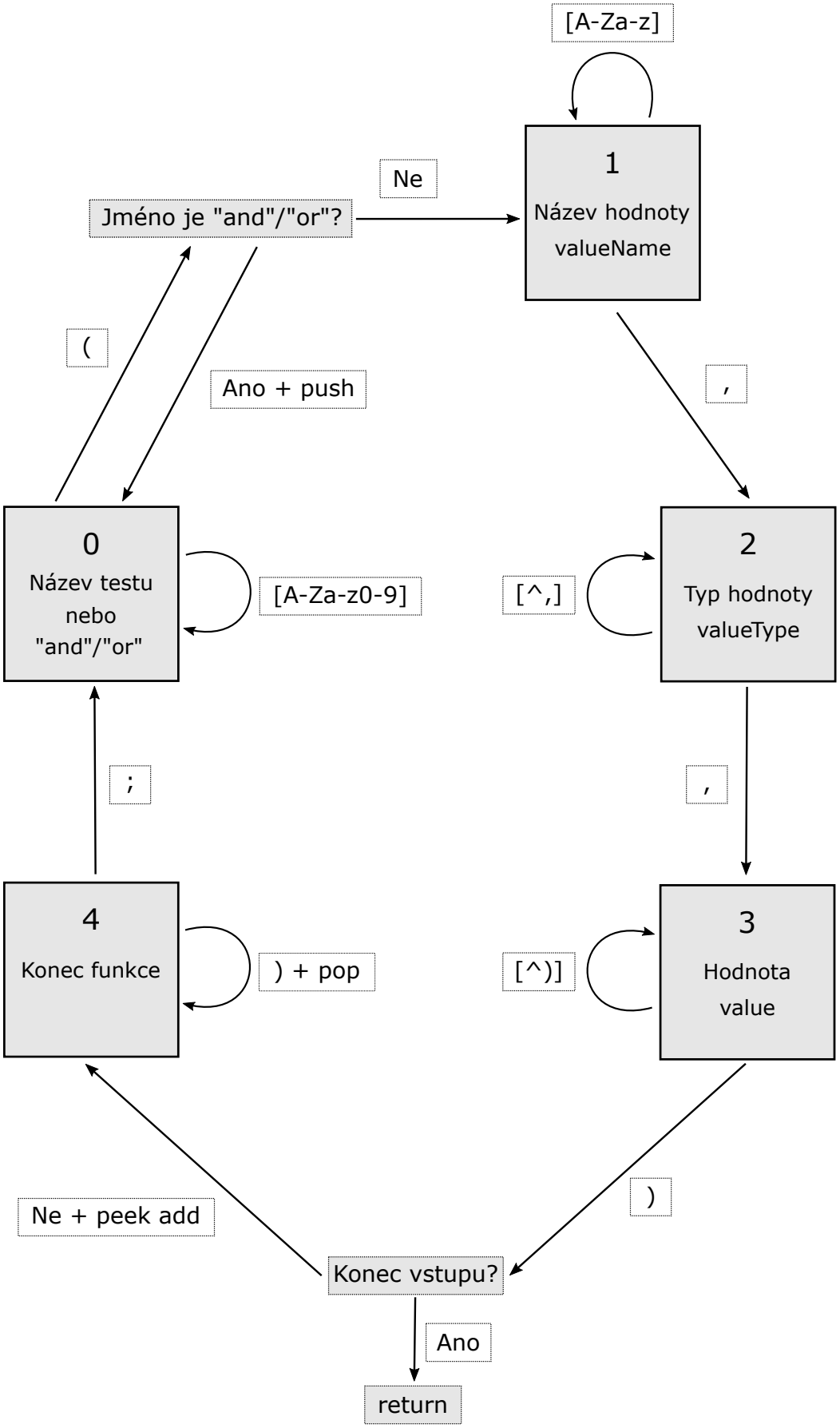
Uživatel má dvě možnosti, jak zapsat klauzule pro filtrování. Tyto klauzule se před filtrováním přeloží do objektů, které je popisují v paměti. V případě zápisu pomocí XML se o převod postará JAXB parser. Při zápisu v konzoli je však parsování výrazu zajímavější. Pro převod na objekty byl vytvořen stavový automat, který je doplněn o zásobník pro práci s vnořenými operacemi (viz obrázek 4.3).

4.8.2 Knihovna filtrů

Pro snadnější uchovávání vytvořených filtrů, které uživatel aktuálně nepoužívá, je konfigurace filtrů rozdělena na dvě části: výčet aktuálně použitých filtrů a knihovnu vytvořených filtrů. V první části uživatel pouze vybere daný filtr podle jeho identifikátoru. V knihovně jsou pojmenované filtry doplněny o popis, který slouží jako dokumentace.

Konfigurace filtrů:

```
<TestConfiguration>
  <Filters>
    <Filter name="diffDistance"/>
  </Filters>
  <Library>
    <TestFilter name="diffDistance">
      <Header>Different route length</Header>
      <Description>
      Filtered flights with distance difference &gt; 200km
      </Description>
      <Tests>
        <TestCase name="GroundDistance"
               valueName="VAL" type="&gt;" value="200"/>
      </Tests>
    </TestFilter>
  </Library>
<TestConfiguration>
```



Obrázek 4.3: Stavový automat pro parsování klauzulí

4.8.3 Filtrování letů

Filtrování letů již nepotřebuje počítat výsledky, ale pouze porovná předpočítaná data s mezními hodnotami z konfigurace daného testu. Filtrování probíhá za pomocí Java 8, která přinesla do jazyka lambda výrazy a streamy. Každý test dostane konfiguraci filtru a vytvoří predikát, který na vstupu dostává konkrétní let, a rozhoduje zda má být let filtrován. Predikáty jsou slučovány podle logických spojek z klauzule. Po průchodu stromu celé klauzule je vytvořen jeden predikát, který představuje strukturu stromu. Predikát je poté použit při filtrování letadel pomocí streamu.

Po vyfiltrování je na výstupu výsledný seznam letů, které prošly daným predikátem. Ukázalo se, že je potřeba předat uživateli informaci, proč je daný let ve výsledné klauzuli a jakou hodnotu mají jeho výsledky oproti mezním hodnotám z konfigurace. Z tohoto důvodu musí testy navíc vytvořit funkci, která z daného letu vrací testovanou hodnotu. Tato hodnota je uživateli zobrazena společně s ohodnocením jednotlivých částí klauzule.

Pro uživatele vyhodnocení neprobíhá společně s filtrováním, protože předpokládaná vyfiltrovaná množina je mnohem menší než vstupní a mnoho informací by se počítalo zbytečně. Vizualizační data jsou tudíž vytvořena až při vytváření výstupu.

Negace testů se provádí dvěma možnými způsoby. První způsob je, že se vytvoří predikát, který se pouze na konci zneguje. Pokud by tato metoda nevyhovovala některé implementaci testu nebo by pro daný test nebyla optimální, existuje druhý způsob pomocí implementace rozhraní NegationTest. Jestliže třída implementuje toto rozhraní, zavolá se na ni metoda, která má za úkol sama vytvořit daný predikát.

analysis.data

Ⓒ **FlightsFilter** filtruje lety podle klauzulí a vytváří průnik seznamů letů ze dvou souborů s pomocí whitelistu a blacklistu.

analysis.configuration

Ⓒ **TestFilterExpression** načte a uchovává filtrovací klauzuli.

4.9 Výstup

Po testování nebo filtrování letů následuje výpis hodnot pro uživatele. Implementace počítá s různými možnostmi výstupu. Všechny implementace výstupů jsou schopny zapsat výsledky do souboru nebo na konzoli.

Pro výpis hodnot slouží na testech funkce, které vracejí dané výsledky. Předpokladem je, že výsledky se pouze vypíší pomocí metody toString. Z tohoto důvodu mohou být návratové hodnoty těchto funkcí libovolného typu. Pro desetinná čísla je zavedena konvence, kdy aplikace vypisuje čísla zaokrouhlená na dvě desetinná místa. Zaokrouhlení není omezující, protože přesnost vstupních dat je mnohem menší.

Pokud se jedná o výpis rozdílů hodnot pro lety ze dvou souborů, testy vrací pro každou hodnotu tři výsledky: výsledek pro let z prvního souboru, výsledek pro let ze druhého souboru a rozdíl výsledků. Jedná-li se navíc o test, který počítá několik hodnot najednou, je těchto výsledků ve výsledném poli několik.

Pro výstup z filtru se používá speciální formát, který vypisuje testovací klauzuli, aby bylo jasné, proč je daný let ve výsledné množině. V konfiguraci lze omezit vypsání hodnot, které překračují mez z konfigurace filtru apod. Ostatní implementace výstupů nemají restrikci na dané hodnoty, které se mají přímo vypsat. U výstupu z filtru je však potřeba vypsat pouze hodnoty, podle kterých bylo filtrováno. Pro tento účel se vytváří lambda funkce pro každou konfiguraci filtru, která vrací pro daný let pouze jeden konkrétní výsledek.

Třídy tvořící výstup jsou podobně jako testy načítány dynamicky, tudíž uživatel může v konfiguraci vybrat libovolný formát. Snadno lze také přidat nový formát výstupu.

Konfigurace výstupu:

```
<OutputConfiguration file="true" console="false" pnegative="false">
    <Directory>exampleinput/output/</Directory>
    <FilterOutput>All</FilterOutput>
    <DiffOutput>All</DiffOutput>
</OutputConfiguration>
```

analysis.tests.output

Ⓘ **OutputFilterPrinter** zobrazí na výstup vyfiltrovanou množinu letů společně s popisem, proč je daný let v této množině.

Ⓘ **OutputFlightPrinter a OutputFlightDifferencePrinter** vypisuje výsledky pro každý let nebo rozdíl výsledků ze dvou souborů.

Ⓘ **OutputFileDifferencePrinter a OutputFilePrinter** zobrazí souhrnné statistiky pro jeden soubor nebo porovnání souhrnů ze dvou souborů.

Ⓒ **PrinterHandler** slouží pro dynamické načítání tříd podle jména. Třídy vyhledává v package analysis.tests.output.

4.10 Řízení aplikace

Aplikace se spustí pomocí třídy Main, kde je implementována metoda main, která dostává argumenty z příkazové řádky. Podle argumentů se aplikace rozdělí do možných režimů. Po parsování argumentů se načte konfigurace z výchozí cesty nebo z cesty zadané uživatelem. Uživatel také může v ručním režimu zvolit možnost vytvoření prázdné konfigurace, která se uloží na požadované místo. V případě ručního režimu další chod aplikace již zajišťuje třída MainMenu. V případě automatického režimu třída AutoPrcess načte lety a vypíše výsledky. Následně aplikace skončí.

analysis

Ⓒ **Main** obsahuje main metodu, která dostává argumenty z konzole a spouští celou aplikaci. Po parsování argumentů je načtena konfigurace a následně je předáno řízení chodu aplikace jiným třídám dle argumentů.

Ⓒ **AutoProcess** načte vstupní soubory společně se všemi testy, spustí filtry a vypíše výsledky.

4.11 Uživatelské rozhraní

Pokud zvolí uživatel ruční režim aplikace, po načtení konfigurace se spustí strukturované menu. Kořenové menu představuje třída MainMenu, která zobrazí

úvodní nabídku. Po výběru možnosti v tomto menu jsou volány specializované menu pro danou akci. Uživatel se v menu pohybuje pomocí zadávání čísel, která odpovídají pořadí nabídek ve vypsaném menu.

analysis.userinterface

(A) **Menu** je abstraktní base pro všechna menu dodávající metody pro jednotný vzhled nabídek a pro práci se vstupem.

(C) **MainMenu** představuje kořenové menu aplikace. Specializované implementace menu jsou: ConfigurationMenu, FilterMenu, ResultsMenu a VisualizationMenu.

4.12 Komunikace s vizualizací

Po vyfiltrování letů je možné spustit vizualizaci, která dostane vyfiltrovanou množinu letů a zobrazí dané trajektorie. Vizualizace je spuštěna jako .exe soubor. Na straně aplikace je spuštěno nové vlákno, které slouží jako server a posílá lety do vizualizace.

Komunikace probíhá pomocí oboustranného socketu. Data jsou posílána v bytovém proudu v pořadí little endian. Specifikaci implementovaného protokolu určuje vizualizace. Server pasivně čeká, až jej vizualizace požádá o data, která jsou posílána ve dvou režimech: všechny lety najednou nebo pouze konkrétní jeden let. Oba režimy lze kvůli časové náročnosti posílání větších množin letů prokládat.

analysis.visualization

(C) **ServerConnection** implementuje komunikační protokol a komunikuje s klientem.

(C) **VisualizationClient** se stará o spuštění vizualizace.

(C) **VisualizationServer** spustí server v novém vlákně, který přijímá připojení na socketu.

Závěr

Zhodnocení

Hlavním cílem práce bylo vytvořit nástroj, který bude připravený na analýzu letových zpráv a je snadno rozšiřitelný o další testy, které budou přibývat na základě konkrétních požadavků při filtrování. Aplikace je připravena na rozšíření i během načítání dat, kde může snadno přibýt podpora jiného formátu než CMS. Výstup je také připraven na další přidávání formátů.

Pro provádění analýzy dat je implementována množina základních testů pokrývající prvotní požadavky na aplikaci. Uživateli pomáhá vizualizace, která jej odstíní od konkrétních hodnot a zobrazí trajektorii.

Reference

[1] AgentFly. https://oi.fel.cvut.cz/cs/agentfly. 12.5.2018.

[2] Package org.json. http://stleary.github.io/JSON-java/index.html, 2016. Navštíveno: 12.5.2018.

[3] Lukáš Kolek. Analysis Imagine. https://github.com/LukasKolek/analysis_imagine, 2018. 12.5.2018.

[4] Volpe National Transportation Systems Center Air Traffic Management Systems Division. Java Message Service Description Document (JMSDD). https://www.faa.gov/nextgen/programs/swim/news/idw2016/media/SFDPS/NAS_JMSDD_4309-001-v2%204_20151029.pdf, Říjen 2015. 10.12.2017.

[5] Ed Ort and Bhakti Mehta. Java Architecture for XML Binding (JAXB). http://www.oracle.com/technetwork/articles/javase/index-140168.html, March 2003. 12.5.2018.

[6] Chris Veness. Calculate distance, bearing and more between Latitude/Longitude points. https://www.movable-type.co.uk/scripts/latlong.html. 12.5.2017.

A. Přílohy

A.1 Obsah přiloženého CD

/doc/ Uživatelská dokumentace a vygenerovaná programátorská dokumentace

/analysis Zdrojové soubory a přiložené soubory

/analysis/resources Přiložené soubory

/analysis/resources/exampleinput Ukázkové vstupní soubory

/analysis/src Zdrojové soubory

/analysis/pom.xml Skript pro sestavení aplikace pomocí nástroje Maven

/analysis/target Vygenerované soubory pomocí nástroje Maven

/analysis/target/analysis.jar Jar soubor pro spuštění práce

/prace.pdf Elektronická verze bakalářské práce

A.2 Příklad použití aplikace

Popis klauzule:

Ve výsledné vyfiltrované množině budou lety, které v obou souborech stoupají a klesají, jsou typu Airbus A320 nebo Airbus A321 a jejich rozdíl délky trajektorií je větší než 30km.

Klauzule v konzoli:

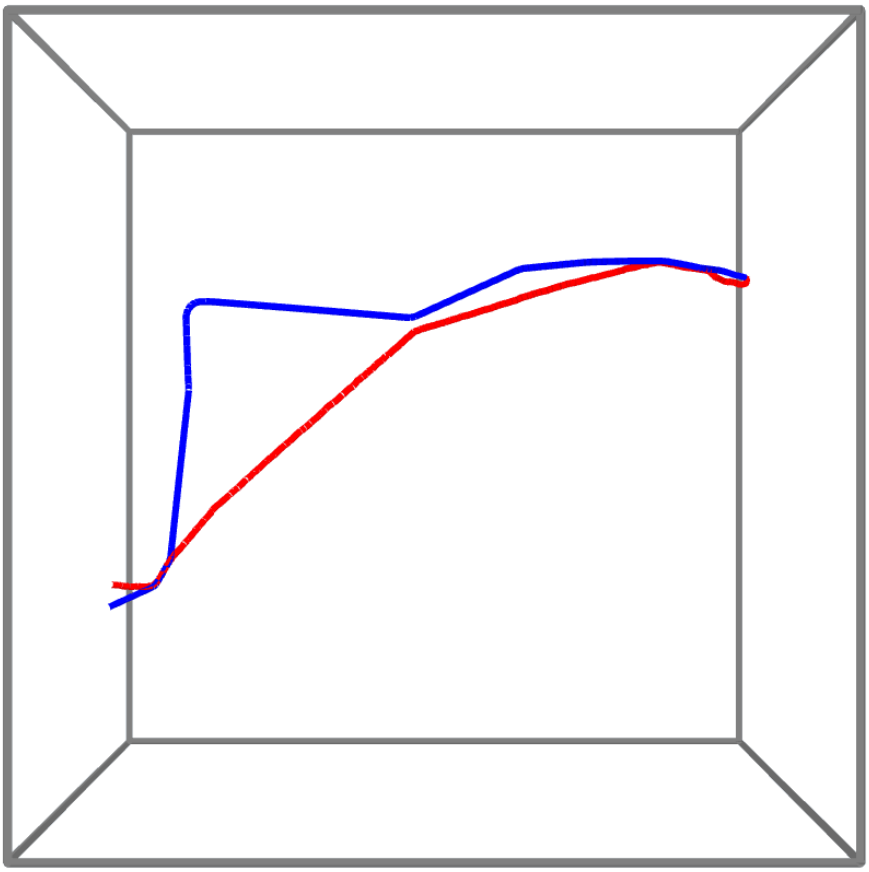
```
AND(
  GroundDistance(VAL,>,30);
  AscDsc(ASC,=,A-A);
  AscDsc(DSC,=,D-D);
  OR(
    FlightType(FLT,=,A321);
    FlightType(FLT,=,A320)
  )
)
```

Klauzule v konfiguraci:

```
<And>
  <TestCase name="GroundDistance"
           valueName="VAL"type=">" value="15"/>
  <TestCase name="AscDsc"
           valueName="ASC" type="=" value="A-A"/>
  <TestCase name="AscDsc"
           valueName="DSC" type="=" value="D-D"/>
  <Or>
    <TestCase name="FlightType"
           valueName="FLT" type="=" value="A321"/>
    <TestCase name="FlightType"
           valueName="FLT" type="=" value="A320"/>
  </Or>
</And>
```

Počty letů po filtrování:

```
Name of master file: Real
Number of mester flights: 7410
Name of second file: Agent
Number of second flights: 6655
Number of filtered flights: 4
```



Obrázek A.1: Příklad letu z vyfiltrované množiny