SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-92331

PROCESNE RIADENÝ CLI

BAKALÁRSKA PRÁCA

2021 **Juraj Puszter**

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Evidenčné číslo: FEI-5382-92331

PROCESNE RIADENÝ CLI

BAKALÁRSKA PRÁCA

Študijný program :	Aplikovaná informatika
Číslo študijného odboru:	2511
Názov študijného odboru:	9.2.9 Aplikovaná informatika
Školiace pracovisko:	Ústav informatiky a matematiky
Vedúci záverečnej práce:	Ing. Jakub Kovář

2021

Juraj Puszter

Slovenská technická univerzita v Bratislave
Ústav informatiky a matematiky

Fakulta elektrotechniky a informatiky
Akademický rok: 2020/2021
Evidenčné číslo: FEI-5382-92331

ZADANIE BAKALÁRSKEJ PRÁCE

Študent:	**Juraj Puszter**
ID študenta:	92331
Študijný program:	aplikovaná informatika
Študijný odbor:	informatika
Vedúci práce:	Ing. Jakub Kovář
Miesto vypracovania:	Ústav informatiky a matematiky

Názov práce: **Procesne riadený CLI**

Jazyk, v ktorom sa práca vypracuje: slovenský jazyk

Špecifikácia zadania:

Cieľom bakalárskej práce je vytvoriť CLI v jazyku platformy Java, ktorý bude riadený procesným modelom vytvoreným vo formalizme Petriho sietí. Aplikácia zobrazuje aktuálne spustiteľné prechody a umožňuje spúšťať dané prechody v procesnom modeli. V rámci spustenia prechodu je možné špecifikovať vykonanie príkazov Unix-like operačných systémov.
Úlohy:
1. Naštudujte problematiku rozšírených Petriho sietí.
2. Navrhnite a implementujte CLI riadený procesom namodelovanom vo formalizme Petriho sietí.
3. Otestujte a vyhodnoťte efektívnosť a použiteľnosť aplikácie v praxi.

Zoznam odbornej literatúry:

1. Juhás, G. – Mažári, J. – Mladoniczky, M. – Juhásová, A. Implementation semantics of Petriflow models. In *Algorithms and Tools for Petri nets.* Hagen: FernUniversität in Hagen, 2019, s. 45–46.
2. Mažári, J. – Juhás, G. – Mladoniczky, M. Petriflow in actions: Events call actions call events. In *Algorithms and Tools for Petri nets.* Augsburg: Universität Augsburg, 2018, s. 21–26.
3. Mladoniczky, M. – Juhás, G. – Mažári, J. – Gažo, T. – Makáň, M. Petriflow: Rapid language for modelling Petri nets with roles and data fields. In *Algorithms and Tools for Petri nets.* Kgs. Lyngby: DTU Compute, 2017, s. 45–50.

Riešenie zadania práce od: 15. 02. 2021

Dátum odovzdania práce: 04. 06. 2021

Juraj Puszter
študent

Dr. rer. nat. Martin Drozda
vedúci pracoviska

Dr. rer. nat. Martin Drozda
garant študijného programu

SÚHRN

SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

Študijný program :	Aplikovaná informatika
Bakalárska práca:	Procesne riadený CLI
Autor:	Juraj Puszter
Vedúci záverečnej práce:	Ing. Jakub Kovář
Miesto a rok predloženia práce:	Bratislava 2021

Cieľom práce je vytvoriť rozhranie príkazového riadku ktoré funguje na základe Petriho sietí. Súčasťou práce je naštudovanie problematiky Petriho sietí, vypracovanie analýzy možných spôsobov riešenia a následný návrh a implementácia programu. Implementáciu aj návrh programu sme opísali a znázornili na diagramoch. Použitie programu sme znázornili a opísali na jednoduchom príklade. Jeho funkčnosť a využitie sme otestovali na rôznych Petriho sieťach. Testovaním sme overili, že je program funkčný podľa našich očakávaní a tiež je vhodným riešením našej problematiky. Program je vhodný pre používateľa, ktorý pravidelne vykonáva určité operácie pomocou príkazového riadku. Program používateľovi zjednoduší a zrýchli túto činnosť. Implementáciu programu sme navyše navrhli tak, aby bolo možné v prípade potreby jednoducho zmeniť formát vstupných dát. V práci sme opísali, ako je tak možné urobiť. Taktiež sme navyše implementovali aby sa aktuálny stav Petriho siete počas chodu programu automaticky ukladal do súboru. Vďaka tomu vieme v prípade potreby uložiť stav Petriho siete a spustiť program od tohto uloženého stavu.

Kľúčové slová: Petriho sieť, rozhranie príkazového riadku, Java

ABSTRACT

SLOVAK UNIVERSITY OF TECHNOLOGY IN BRATISLAVA
FACULTY OF ELECTRICAL ENGINEERING AND INFORMATION TECHNOLOGY

Study Programme:	Applied Informatics
Bachelor Thesis:	Process-driven CLI
Autor:	Juraj Puszter
Supervisor:	Ing. Jakub Kovář
Place and year of submission:	Bratislava 2021

The aim of this work is to create a command line interface that works on the basis of Petri nets. This work consists of the study of Petri nets, the analysis of possible solutions and subsequent design and implementation of the program. We described and illustrated the implementation and the design of the program using diagrams. We have illustrated and described the use of the program with a simple example. We tested its functionality and use on various Petri nets. By testing, we have verified that the program is functional according to our expectations and it is also a suitable solution to our problem. The program is suitable for a user who regularly performs certain operations using the command line. The program simplifies and speeds up this activity for the user. In addition, we designed the implementation of the program so that it is possible to easily change the input data format if necessary. In this work, we described how it is possible to do so. In addition, we have also implemented that the current state of the Petri net during the program execution is automatically saved to a file. Thanks to this, we can save the state of the Petri net if needed and run the program from this saved state.

Key words: Petri net, command line interface, Java

Vyhlásenie autora

Podpísaný Juraj Puszter čestne vyhlasujem, že som Bakalársku prácu Procesne riadený CLI vypracoval na základe poznatkov získaných počas štúdia a informácií z dostupnej literatúry uvedenej v práci.

Uvedenú prácu som vypracoval pod vedením Ing. Jakuba Kováře.

V Bratislave dňa 04.06.2021

.....

podpis autora

Poďakovanie

Ďakujem vedúcemu práce Ing. Jakubovi Kovářovi za jeho pomoc, čas a konzultácie počas písania tejto práce.

Obsah

Zoznam obrázkov a zdrojových kódov

Zoznam skratiek a značiek

CLI – Command line interface

UML - Unified Modeling Language

p - Place

t - Transition

pom - Project Object Model

XML - eXtensible Markup Language

DTD - Document Type Definition

XSD - XML Schema Definition

ID - Identifier

SAX - Simple API for XML

DOM - Document Object Model

JAXB - Jakarta XML Binding

Úvod

Cieľom tejto práce je vytvoriť program, ktorý umožňuje vykonávať príkazy v príkazovom riadku operačného systému typu UNIX pomocou formálneho modelu Petriho siete. Petriho siete sú výborný nástroj pre definíciu spúšťania operácií v určitom poradí a tiež súbežne, čo vyhovuje našej problematike vykonávania príkazov v príkazovom riadku. Vďaka Petriho sieťam vieme presne definovať, kedy sa má ktorý príkaz vykonať. Nakoľko Petriho siete tvoria veľkú časť tejto práce je nutné im porozumieť. Preto sa v tejto práci venujeme aj základom teórie Petriho sietí.

Pred implementáciou práce vypracujeme analýzu dostupných riešení. Zanalyzujeme dostupné technológie, ktoré je možné v implementácií použiť a vyberieme z nich tie najvhodnejšie. Dostupné technológie v práci opíšeme a navzájom porovnáme. Na základe dosiahnutých poznatkov z analýzy navrhneme možné riešenie. Tento návrh primerane opíšeme a znázorníme na diagramoch. Návrh riešenia sa počas implementácie značne zmení, a preto slúži iba na vytvorenie hrubého obrazu programu, pred jeho implementáciou.

V časti implementácie jasne opíšeme záverečný stav programu. Pomocou diagramov znázorníme implementáciu programu a tiež opíšeme jednotlivé časti, aby bolo jasné akým spôsobom program pracuje. V programe sa pokúsime dosiahnuť čo najväčšiu možnú mieru automatizácie, aby používateľ nemusel zasahovať do programu pri spúšťaní každého príkazu. Taktiež znázorníme, ako program po jeho spustení vyzerá a ako pracuje z pohľadu používateľa.

Po implementácií program otestujeme, aby sme zaručili jeho funkčnosť. Samozrejme, nie je možné aby sme zaručili jeho funkčnosť vo všetkých operačných systémoch typu UNIX a preto pri testovaní zvolíme tie najpoužívanejšie.

Na záver práce vyhodnotíme naše dosiahnuté poznatky použitia Petriho siete na vykonávanie príkazov v príkazovom riadku a tiež použitie programu v praxi.

1 Teória problematiky

Pre porozumenie práce je nutné disponovať základnými znalosťami Petriho sietí, ktoré sú poskytnuté v tejto kapitole. Nachádza sa tu definícia Petriho siete ako aj jeho jednotlivých komponentov, spôsob použitia a značenia. Diagramy Petriho siete sú kreslené pomocou nástroja na webovej stránke http://www.biregal.com/, ktorý nám umožňuje navrhnúť a otestovať ľubovoľnú Petriho sieť.

V práci predpokladáme, že čitateľ má znalosti z programovania v jazyku Java a taktiež UML diagramov, ktoré sú využité pre grafické znázornenie funkcionality a štruktúry softvéru. Ak čitateľ tieto znalosti nemá, odporúčame aby si dané témy naštudoval z iných zdrojov, nakoľko sa týmto témam v tejto kapitole nevenujeme.

1.1 Petriho sieť

Petriho sieť je formálny model slúžiaci pre návrh súbežných a distribuovaných systémov. (1)

Súbežný systém je systém, v ktorom je viacero operácii vykonávaných v jednom čase. Takáto činnosť je možná za pomoci viacerých vlákien alebo procesov. Treba si dať pozor na zamieňanie súbežného a paralelného vykonávania, nakoľko to nie je to isté. (2)

Distribuovaný systém je systém zložený z viacerých komponentov na rôznych strojoch. Tieto stroje navzájom komunikujú s cieľom koordinácie svojej činnosti, aby sa používateľovi javili ako jeden súvislý systém. Distribuovaný systém ma tri veľké výhody. Prvou je jednoduchosť rozširovania systému o ďalšiu funkcionalitu. Druhou výhodou je vysoká tolerancia chybovosti systému, čo má za následok neporušenie funkcionality systému aj v prípade, ak jeden stroj zlyhá. Tretia výhoda je vysoká efektivita systému. Všetky tieto výhody sú možné vďaka rozdeleniu vykonávania operácií na viaceré uzly rozmiestnených na viacerých strojoch. (3)

Petriho siete majú formu orientovaného bipartitného grafu. Skladajú sa z troch komponentov a to miest, prechodov a hrán. Môžeme povedať, že Petriho sieť je pätica $N = (P, T, F, W, M_0)$, kde P je konečná množina miest, T je konečná množina prechodov, F je konečná množina hrán, W je konečná množina váh prislúchajúcich k hranám a M_0 je konečná množina počiatočného značenia miest siete. Platí že $P \cup T \neq \emptyset$, $P \cap T = \emptyset$, $F \subseteq$

(P×T) ∪ (T×P), W: (P×T) ∪ (T×P) → N je funkcia váhy kde N = { 0, 1, 2, …}, a M0: P → N je počiatočné značenie. (4)

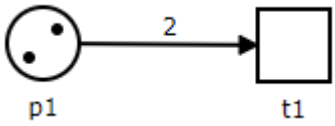
1.2 Komponenty Petriho siete

Miesta, často sa používa anglický názov places. V grafoch sa značia ako biely kruh. Ich značka je písmeno p. Slúžia pre uchovávanie tokenov, inak povedané aj značiek. Tokeny slúžia pre ovládanie spustiteľnosti prechodov a zároveň znázorňujú stav Petriho siete. Miesta sa pomenovávajú prídavnými menami, napríklad potvrdené, zrušené. Značia sa ako čierne kruhy vo vnútri miesta. Ak ich je v mieste veľké množstvo značia sa číslom, ktoré reprezentuje ich počet. Počet tokenov je prirodzené číslo vrátane nuly, čo sú čísla {0, 1, 2, …}. (5)

Prechody, často sa používa anglický názov transitions. V grafoch sa značia ako biely štvorec prípadne obdĺžnik. Ich značka je písmeno t. Slúžia pre znázornenie nejakej činnosti, operácie ktorá je vykonaná ich spustením. Prechody sa pomenovávajú slovesami, napríklad potvrdiť, zrušiť. Či je prechod spustiteľný vieme zistiť tak, že každá vstupná hrana prechodu má dostatočný počet tokenov v mieste, s ktorým je spojená. Počet tokenov v tomto mieste musí byť rovný alebo väčší ako je váha hrany. Ak je táto podmienka platná, môžeme spustiť prechod, čo vedie k zníženiu počtu tokenov miest z ktorých hrany vedú. Od pôvodnej hodnoty tokenov odpočítame váhu hrany a dostaneme novú hodnotu tokenov v danom mieste. Následne k tokenom miest ku ktorým vedú výstupné hrany prechodu pripočítame váhu hrany a dostaneme novú hodnotu tokenov daného miesta. Ak z prechodu nevedú žiadne výstupné hrany, tak tento prechod počet tokenov v žiadnych miestach nezvyšuje. Ak do tokenu nevedú žiadne vstupné hrany, vieme token spúšťať neobmedzene a počet tokenov v žiadnych miestach neznižuje. Prechod ktorý nemá žiadne vstupné ani výstupné hrany je tiež platný. Takýto prechod vieme spúšťať neobmedzene pričom nemení hodnoty tokenov žiadnych miest. (5)

Hrany, často sa používa anglický názov arcs. V grafoch majú značku čiernej, smerovanej šípky. Neoznačujú sa žiadnym písmenom. Slúžia na spojenie miest a prechodov. Ich orientácia určuje smer toku, to znamená smer v ktorom sú prideľované tokeny. Ak hrana začína v miesta a končí v prechode, môžeme povedať že je vstupná vzhľadom k prechodu a výstupná vzhľadom k miestu. Môže mať aj opačný smer, teda začínať v prechode a končiť v mieste. Hrana nesmie navzájom spájať dve miesta alebo dva

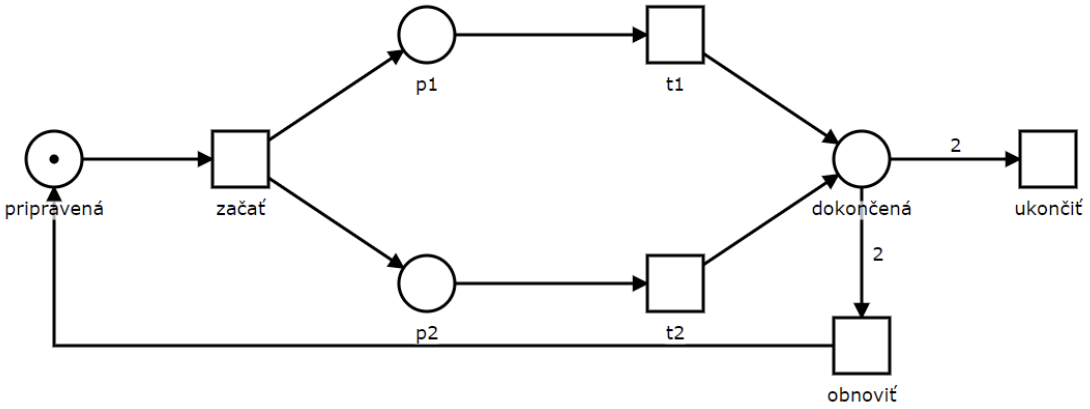
prechody. Tento stav je zakázaný. Hrana má ďalšiu dôležitú vlastnosť, a to je jej váha. Značí sa ako číslo nad hranou. Ak nie je žiadne číslo uvedené, predstavuje to váhu s hodnotou jedna. Váha hrany je prirodzené číslo okrem nuly, čo sú čísla {1, 2, 3, …}. Váha je dôležitá pri riadení spustiteľnosti prechodov. Podľa nej sa pripočítavajú alebo odpočítavajú tokeny v miestach. (5)



Obrázok 1: Značenie Petriho siete

Obrázok 1 znázorňuje značenie komponentov Petriho siete. Znázornený je prechod t1 a miesto p1 spojené hranou. Hrana smeruje z miesta do prechodu a jej váha je dva. Počet tokenov v mieste p1 sú dva. Prechod t1 je jeden krát spustiteľný.

1.3 Príklad Petriho siete



Obrázok 2: Príklad jednoduchej Petriho siete

Obrázok 2 znázorňuje príklad jednoduchej Petriho siete. Skladá sa z devätnástich komponentov. Názvy miest sú: pripravená, p1, p2 a dokončená. Názvy prechodov sú: začať, t1, t2, ukončiť a obnoviť. Prechody t1 a t2 predstavujú ľubovoľnú operáciu. Miesta p1 a p2 predstavujú ľubovoľný stav po vykonaní prechodu začať. Miesta a prechody sú prepojené dokopy desiatimi hranami. Hrany medzi miestom dokončená a prechodmi ukončiť a obnoviť majú váhu dva. Hrana medzi prechodom obnoviť a miestom pripravená má tiež váhu dva. Ostatné hrany majú váhu jedna. Počiatočné značenie siete je jeden token v mieste pripravená. V tomto stave môžeme spustiť token začať. Po jeho spustení sa z miesta pripravená odpočíta jeden token. Do miesta p1 a p2 sa pripočíta jeden token.

Následne máme na výber z dvoch spustiteľných prechodov a to prechody t1 a t2. Po spustení prechodu t1 sa odpočíta jeden token z miesta p1 a jeden sa pripočíta do miesta dokončená. Teraz je možné spustiť iba prechod t2. Jeho spustením sa odpočíta jeden token z miesta p2 a jeden sa pripočíta do miesta dokončená. Spustenie prechodov t1 a t2 sme mohli vykonať aj v opačnom poradí. Po spustený týchto dvoch prechodov budeme mat v mieste dokončená celkovo dva tokeny. Z tohto stavu môžeme spustiť prechod ukončiť alebo obnoviť. Spustením prechodu ukončiť sa odpočítajú z miesta dokončené dva tokeny a do žiadneho miesta sa ďalšie nepripočítajú. Takto sa dostaneme do stavu v ktorom sa už v žiadnom mieste nenachádzajú žiadne tokeny a žiadne prechody nie sú spustiteľné. Ak by sme miesto prechodu ukončiť spustili prechod obnoviť, odpočítajú sa z miesta dokončená dva tokeny a dva sa pripočítajú do miesta pripravená. Takto sa dostaneme do počiatočného stavu siete a môžeme začať vykonávať sieť znovu od začiatku.

2 Analýza problematiky

Táto kapitola slúži pre analýzu zadania a jeho funkcionality. Nachádza sa tu analýza možných metód riešenia z ktorých sme vybrali a ktoré sme neskôr použili pri implementácií. Taktiež sa v tejto kapitole nachádza hrubý návrh zadania pred jeho implementáciou. Tento návrh nie je finálny a ani samostatne funkčný. Počas implementácie sme množstvo častí doplnili a upravili.

V súčasnosti existuje množstvo rôznych použití Petriho sietí, ale nepodarilo sa nám nájsť rovnaké použitie ako to naše. Z tohto dôvodu táto práca nie je porovnávaná s inými prácami. Napriek tomu sme sa z časti inšpirovali nástrojom na internetovej stránke http://www.biregal.com/. Tento nástroj neposkytuje rovnaké použitie Petriho sieti ako naša práca, ale umožňuje vkladanie vstupných dát pomocou XML súborov a preto je vhodnou inšpiráciou pre časť našej práce.

2.1 Voľba programovacieho jazyka

Programovací jazyk pri tvorbe zadania sme zvolili Javu. Java je jeden z najpoužívanejších a najpopulárnejších svetových programovacích jazykov. Je jednoduchý na pochopenie vďaka čomu v nej dokáže programovať veľa ľudí. Pracuje na objektovo orientovanom princípe ktorý je veľmi vhodný pre tvorbu aplikácii a softvéru. Vďaka tomuto princípu sa programy ľahko rozširujú o ďalšiu funkcionalitu. Na rozdiel od množstva iných programovacích jazykov je nezávislá na platforme. Dokážeme ju používať na počítači s operačným systémom Windows, Mac, Linux a mnoho ďalších verzií Unix. Java sa považuje za dynamickejší jazyk ako napríklad C a C++ pretože je navrhnutá aby sa prispôsobovala vývojovému prostrediu. Ďalšie jej výhody sú vysoký výkon, odstránenie chybových situácií a bezpečnosť. Používa sa aj pre tvorbu vývojárskych nástrojov, ako sú napríklad IntelliJ IDEA, Eclipse a NetBeans. Toto zadanie sme programovali pomocou nástroja IntelliJ IDEA a použili sme najnovšiu súčasnú verziu Javy ktorá je Java SE 15. (6)

2.2 Voľba frameworku a nástroj Maven

Rozhodli sme sa, že v implementácii zadania použijeme framework. Použitie frameworku nám umožňuje uľahčenie a zrýchlenie práce využívaním rôznych funkcií ktoré nám poskytuje. Medzi najznámejšie frameworky patrí Spring, Guice, Play a

Hibernate. Našou voľbou je framework Spring. Spring je už niekoľko rokov najpopulárnejší framework pre Javu s množstvom funkcií v rôznych oblastiach. Ostatné frameworky sa zväčša špecializujú iba v určitej oblasti. Vďaka Springu je možné v prípade potreby zadanie jednoducho rozšíriť. Dovoľuje nám okrem iného napríklad vytvorenie automatizovaných testov, webového rozhrania, prácu s databázou či zvýšenie bezpečnosti pomocou autorizácie. V zadaní využívame najmä funkciu vkladania závislostí, ktorá je známa aj pod anglickým názvom Dependency Injection. (7)

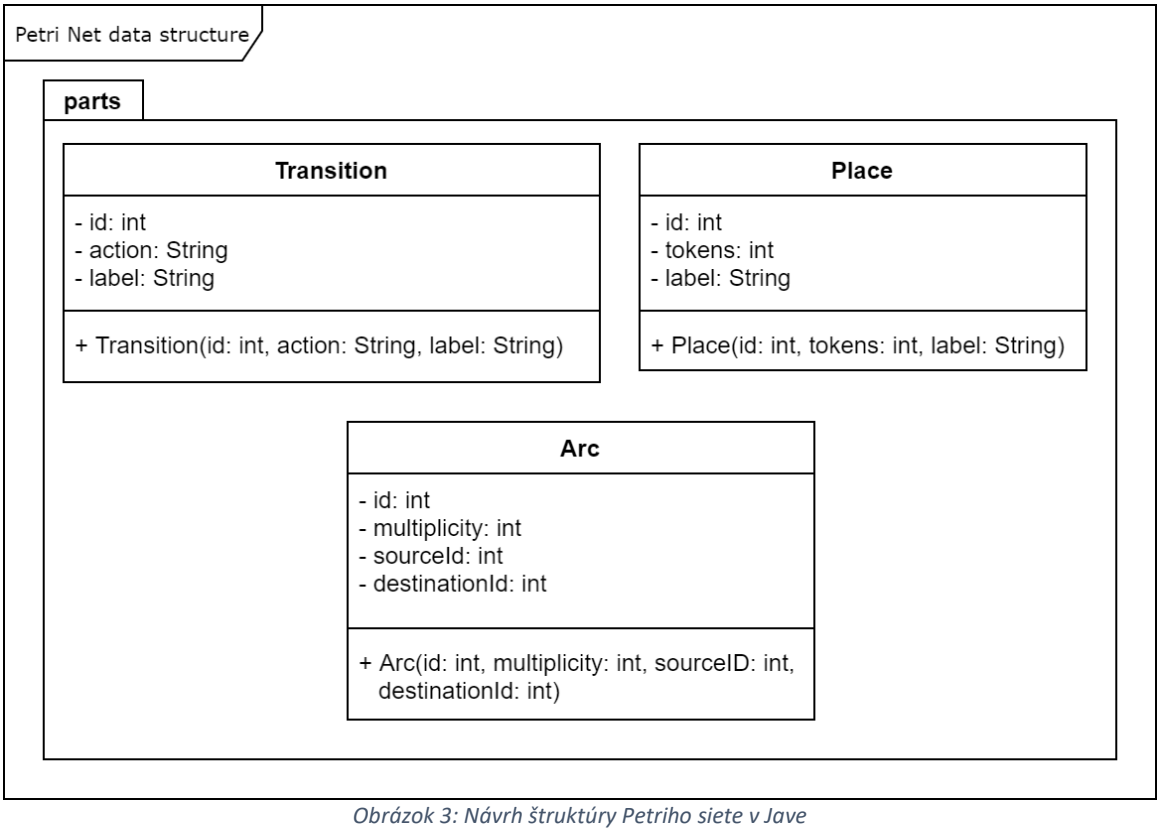
Okrem frameworku používame aj nástroj Maven. Je to nástroj pre riadenie a jednoduchšie porozumenie projektu. Maven nám uľahčuje build projektu, poskytuje jednotný systém buildovania, informácie o projekte a tiež nás podporuje dodržiavať lepšie vývojové postupy. V našom projekte sa nachádza súbor pom.xml. Je to XML reprezentácia Maven projektu. V tomto súbore sa nachádzajú všetky potrebné informácie o projekte a tiež projektové nastavenia ako sú napríklad konfiguračné súbory a závislosti, známe aj pod anglickým názvom dependencies. (8)

2.3 Reprezentácia Petriho siete v jazyku Java

```
<document>
    <place>
        <id>1</id>
        <x>347</x>
        <y>283</y>
        <label>p1</label>
        <tokens>2</tokens>
        <static>false</static>
    </place>
    <transition>
        <id>2</id>
        <x>506</x>
        <y>283</y>
        <label>t1</label>
    </transition>
    <arc>
        <id>3</id>
        <type>regular</type>
        <sourceId>1</sourceId>
        <destinationId>2</destinationId>
        <multiplicity>2</multiplicity>
    </arc>
</document>
```

Zdrojový kód 1: XML reprezentácia Petriho siete

Pri návrhu reprezentácie Petriho siete v Jazyku Java sme sa inšpirovali nástrojom na internetovej stránke http://www.biregal.com/. Okrem návrhu a testovania Petriho siete nám nástroj umožňuje aj ukladanie návrhu do súboru. Tento súbor je vo formáte XML, ktorý je možný pomocou nástroja otvoriť pre znovu načítanie uloženého návrhu. Zdrojový kód 1 znázorňuje, ako vyzerá takýto súbor, ktorý obsahuje návrh Petriho siete. Obrázok 1 znázorňuje túto Petriho sieť. Zo súboru sme vyčítali, že miesta, prechody a hrany musia mať unikátne identifikačné číslo pre jednoznačnú identifikáciu komponentov. Vďaka tomuto číslu je jasné o ktorý komponent sa jedná. Miesta ďalej obsahujú informáciu o počte tokenov v danom mieste. Hrany obsahujú informáciu o váhe danej hrany. Miesta aj prechody obsahujú informáciu, ktorá udáva ich značenie. Toto značenie nie je pre funkčnosť siete nutné, nakoľko komponenty majú unikátne ID, ale je to vhodná informácia pre používateľa. Informácie o tom, ako sú miesta a prechody prepojené hranami vieme vyčítať z parametrov hrán. Každá hrana obsahuje ID zdroja z ktorého začína a ID cieľa do ktorého vedie. Vďaka tomuto vieme zistiť aj jej orientáciu, nakoľko hrana je orientovaná vždy smerom od zdroja k cieľu. Ďalšie parametre ktoré sa v súbore nachádzajú nás pre funkčnosť našej práce nezaujímajú. Sú to parametre udávajúce pozíciu komponentu a tiež typ komponentu. V našej práci typy komponentov nerozlišujeme.



Obrázok 3: Návrh štruktúry Petriho siete v Jave

Obrázok 3 znázorňuje, ako by sme všetky potrebné informácie pre reprezentáciu Petriho siete v Jave mohli zaviesť do tried. Trieda Transition, Place a Arc predstavujú miesta, prechody a hrany so všetkými potrebnými informáciami, ktoré sme spomenuli. Tieto informácie sú reprezentované vo forme atribútov. Okrem týchto atribútov trieda Transition navyše obsahuje atribút action. Tento atribút slúži pre uchovanie príkazu pre príkazový riadok ktorý sa vykoná po spustení prechodu. Triedy pre inicializáciu využívajú parametrické konštruktory. Taktiež obsahujú metódy get a set pre každý atribút, ktoré v diagrame pre lepšiu čitateľnosť uvedené nie sú.

2.4 Obsah a formát vstupných dát

Vstupné dáta vkladáme do programu pomocou súboru vo formáte XML. Skratka XML znamená eXtensible Markup Language. XML je značkovací jazyk ktorým určujeme formát súboru. Súbor je vhodný pre ukladanie dát a tiež sa jeho využitím jednoducho komunikujú dáta medzi rôznymi platformami. Jeden zo základných princípov pri jeho vytvorení bola čitateľnosť a jednoduchá tvorba pre človeka. XML súbor poskytuje potrebné nástroje akým definovať formát dát. XML súbor tento formát priamo nedefinuje, musí o ňom rozhodnúť človek. Všetky platformy ktoré dané XML využívajú sa musia zhodnúť na jeho rovnakom formáte. Zhodná štruktúra súboru zabezpečí, že dáta úspešne prejdú parsovaním. XML súbor je definovaný štandard vďaka čomu sa s ním jednoducho pracuje a z toho dôvodu ho využíva množstvo systémov. (9)

Snažili sme sa dosiahnuť čo najvyššiu kompatibilitu s nástrojom na internetovej stránke http://www.biregal.com/ a preto je formát týchto dát podobný. Používateľ môže pre návrh Petriho siete využiť túto stránku a tak si uľahčiť prácu pri písaní XML súboru. Zdrojový kód 1 znázorňuje aký formát používa táto stránka. Zdrojový kód 2 znázorňuje formát ktorý sme navrhli pre náš program. Rovnako ako pri návrhu tried v Jave sú údaje o pozícií komponentov a typoch komponentov pre nás nepotrebné. Taktiež sme pridali parameter action pre transition.

```
<document>
    <place>
        <id>1</id>
        <label>p1</label>
        <tokens>2</tokens>
    </place>
    <transition>
        <id>2</id>
        <label>t1</label>
        <action>echo test</action>
    </transition>
    <arc>
        <id>3</id>
        <sourceId>1</sourceId>
        <destinationId>2</destinationId>
        <multiplicity>2</multiplicity>
    </arc>
</document>
```

Zdrojový kód 2: Návrh XML reprezentácie Petriho siete

2.5 Validácia vstupných dát

Súbor so vstupnými dátami potrebujeme nejakým spôsobom validovať. Pre validáciu XML súborov sú nám dostupné dve technológie a to DTD a XSD. Skratka DTD znamená Document Type Definition a skratka XSD znamená XML Schema Definition. Obe technológie slúžia pre vytvorenie definícií ktorými validujeme XML súbory. Tieto definície sú uložené vo vlastných súboroch. Obe technológie sú značkovacie jazyky podobne ako HTML. DTD využíva vlastný syntax ale XSD využíva rovnaký syntax ako XML, teda XSD súbor je tiež XML. DTD definuje elementy, ich atribúty a poradie elementov aké môžeme mať v dokumente. XSD poskytuje omnoho viac možností ako DTD, ktorými môžeme podrobnejšie definovať štruktúru XML dokumentu. Pre definovanie štruktúry používa objektovo orientovaný spôsob definovania. XSD poskytuje možnosť vytvárania menných priestorov, anglicky namespace, a tiež nám poskytuje základné typy pre prácu, ako sú napríklad string, integer, byte a float. Za pomoci týchto jednoduchých typov môžeme vytvárať vlastné, komplexné typy ktorými definujeme elementy. DTD nerozlišuje rôzne typy, všetko je string a preto je aj validácia veľmi obmedzená. Na základe týchto poznatkov sme sa rozhodli validovať vstupné dáta pomocou XSD súboru, nakoľko nám poskytuje oveľa viac možností validácie. (9)

2.6 Parsovanie vstupných dát zo súboru

Najznámejšie technológie pre parsovanie XML súborov sú SAX, DOM a JAXB parsery. Skratka SAX znamená Simple API for XML, DOM znamená Document Object Model a JAXB znamená Jakarta XML Binding.

SAX parser je založený na báze eventov ktoré vznikajú počas čítania súboru. Nevytvára žiadnu vnútornú štruktúru. Je vhodný pre čítanie veľkých súborov, nakoľko nenačíta celý obsah súboru ale číta ho po častiach. Vďaka tomu je rýchli a nezaberá veľa pamäte. Je náročnejší na programovanie ako ostatné parsery pretože treba súbor postupne čítať a spracovávať. Počas čítania prejde súbor iba raz a to od začiatku do konca. Nedokáže čítať ľubovoľnú časť súboru. SAX parser dokáže súbor iba čítať, nedokáže ho upravovať. Elementy ktoré zo súboru nepotrebujeme môžeme vynechať. (10)

DOM parser je založený na báze stromovej štruktúry. Poskytuje triedy pre prácu s parsovaným súborom. Je vhodný pre čítanie menších súborov nakoľko ukladá celý obsah súboru do pamäti. Je pomalší ako SAX parser. Z objektu ktorý obsahuje dáta zo súboru môžeme tieto dáta získavať v ľubovoľnom poradí. DOM parser dokáže súbor čítať a tiež upravovať. Pracuje sa s ním jednoduchšie ako so SAX. Na rozdiel od SAX nedokážeme vynechávať elementy v súbore, nakoľko je nutné načítať celý súbor. (10)

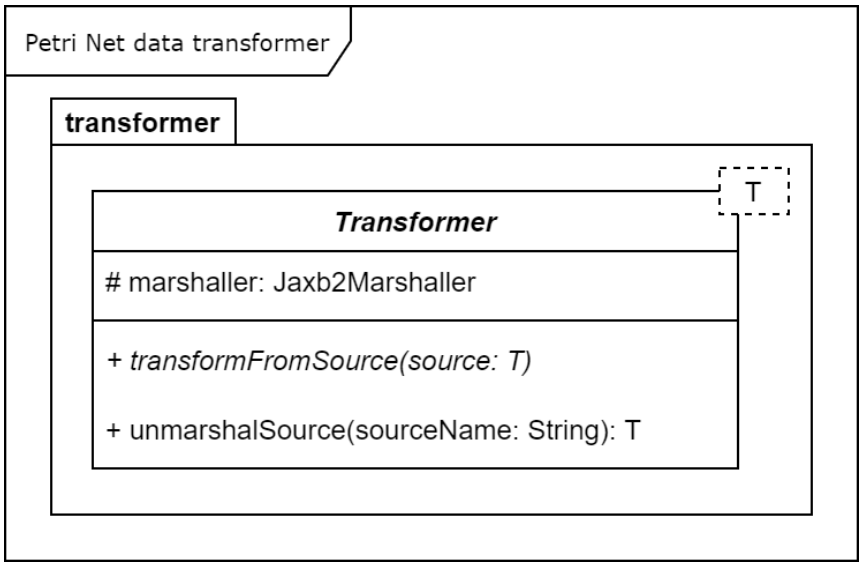
JAXB umožňuje čítanie XML do Java objektov a tiež opačne, zápis objektov do XML. Pre mapovanie XML elementov a atribútov na Javové parametre využíva Java anotácie. Pre čítanie XML do tried poskytuje metódu s názvom unmarshal. Metóda ktorá zapisuje Java triedy do XML sa nazýva marshal. JAXB2 Maven plugin nám navyše umožňuje generovať anotované Java triedy z XSD a tiež opačne, generovať XSD z anotovaných tried. Výsledok parsovania súboru budú vygenerované triedy naplnené dátami. Spôsobom, akým sú triedy generované je možné nastaviť. Základné nastavenie je jedna trieda v ktorej sa nachádzajú ostatné vnorené triedy. Je možné nastaviť aby tieto triedy neboli vnorené, alebo boli samostatné. Pre naše zadanie však vyhovuje toto základné nastavenie a preto ho nebudeme meniť. Z uvedených možností parsovania sme si zvolili JAXB2 Maven plugin. Pretože sme sa rozhodli validovať XML podľa XSD môžeme z neho pomocou JAXB2 vygenerovať triedy a parsovať XML. Parsovanie a validácia je veľmi jednoduchá. Stačí uviesť ktorý XML súbor chceme podľa ktorého XSD súboru parsovať a validovať. Pre parsovanie nám stačia metódy marshal a unmarshal ktoré JAXB2

poskytuje a nemusíme programovať žiadne parsovacie metódy, ako by sme museli pri SAX a DOM. (11)

2.7 Oddelenie vrstvy vstupných dát

Triedy, ktoré sú vygenerované a uchovávajú vstupné dáta sa budú líšiť od nami vytvorených tried ktoré reprezentujú Petriho sieť. Je to tak preto, aby bolo možné poskytovať vstupné dáta v rôznych formátoch. Chceme, aby mal programátor možnosť si vybrať formát vstupných dát taký, aký mu vyhovuje. V zadaní naprogramujeme formát vstupných dát tak, ako sme si my vybrali vo vyššie uvedenej podkapitole. Tento formát je možné zmeniť bez ovplyvnenia chodu programu. Formátom vstupných dát máme na mysli štruktúru XML súboru. Informácie ktoré tento dokument obsahuje musia byť stále rovnaké, ale štruktúra v ktorej sú podávané sa môže zmeniť.

Aby toto bolo možné, potrebujeme triedu, ktorá bude využívať JAXB2 Maven plugin a poskytovať metódy pre parsovanie súboru. V triede je tiež nutné vytvoriť metódu, ktorá nám prevedie dáta medzi vygenerovanými triedami a našimi triedami, ktoré reprezentujú Petriho sieť. Pri zmene formátu vstupných dát je tiež nutné vytvoriť nový XSD súbor, ktorý tento formát definuje.



Obrázok 4: Návrh triedy pre prácu so vstupnými dátami

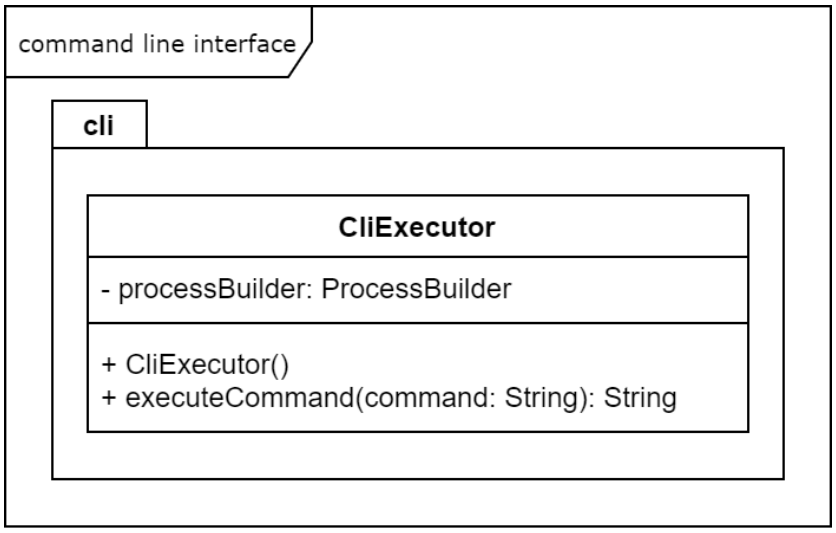
Obrázok 4 znázorňuje návrh vyššie spomínanej triedy Transformer pre prácu so vstupnými dátami. Trieda Transformer je abstraktná generická trieda. Triedu je nutné zdediť a implementovať v inej triede, ktorej implementácia je prispôsobená formátu vstupných dát. Typ T je typ objektu, ktorý JAXB2 vygeneruje. Transformer obsahuje

protected atribút marshaller, ktorý je objekt typu Jaxb2Marshaller. Tento objekt je z JAXB2 Maven pluginu. Jeho metódy použijeme pri validácií a parsovaní vstupných dát ktoré sa nachádza v implementácii public metódy unmarshalSource. Táto metóda vráti zvalidované vstupné dáta zo súboru. Vstup metódy je názov tohto súboru. Výstup metódy je objekt typu T, čo je trieda vygenerovaná pomocou JAXB2. Metóda unmarshalSource nie je abstraktná, bude implementovaná už v triede Transformer a zdedené triedy ju budú mať rovnaké. Implementáciu metódy nie je nutné meniť pri zmene formátu vstupných dát. Public abstraktná metóda transformFromSource je metóda, ktorá vytvorí triedy ktoré náš program využíva pre reprezentáciu Petriho siete a prekopíruje do nich validované a parsované vstupné dáta z vygenerovaných tried. Vstup do metódy je vygenerovaná trieda ktorú chceme transformovať na našu triedu. Výstup budú naše triedy reprezentujúce Petriho sieť. Presný formát výstupu nám zatiaľ nie je jasný preto nie je v diagrame uvedený. Transformer má defaultný konštruktor, geter a seter pre atribút marshaller ktoré v diagrame pre lepšiu čitateľnosť nie sú uvedené.

2.8 Vykonávanie príkazov v príkazovom riadku

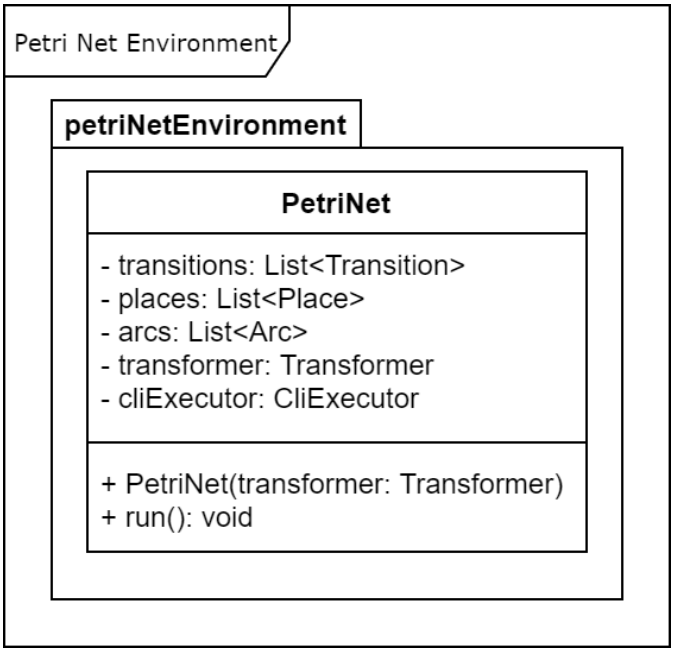
Pre vykonávanie príkazov v príkazovom riadku máme v Jave dostupné dva spôsoby. Prvý spôsob je pomocou triedy Runtime a druhý spôsob je pomocou vytvorenej inštancie triedy ProcessBuilder. Obe triedy fungujú pre operačný systém Windows aj pre Linux. Z týchto dvoch spôsobov býva preferovaný ProcessBuilder pretože poskytuje možnosť nastaviť rôzne detaily. Nastavenie ktoré môžeme využiť je napríklad nastavenie pracovného adresára v ktorom sa príkazy vykonajú. Taktiež môžeme ľubovoľne presmerovať konzolový vstup a výstup. Z týchto dôvodov sme sa v našej implementácií rozhodli použiť ProcessBuilder. (12)

Obrázok 5 znázorňuje návrh triedy CliExecutor ktorá bude vykonávať príkazy v príkazovom riadku. Obsahuje private atribút processBuilder triedy typu ProcessBuilder, ktorej metódy využívame pri implementácií public metódy executeCommand. Táto metóda vykoná príkaz ktorý má na vstupe formou String. Výstup metódy bude vo forme String a jeho obsah bude konzolový výstup po vykonaní príkazu zo vstupu metódy. Trieda obsahuje aj geter a seter pre atribút processBuilder a konštruktor, v ktorom je tento atribút inicializovaný.



Obrázok 5: Návrh triedy ktorá riadi vykonávanie príkazov

2.9 Riadenie činnosti programu



Obrázok 6: Návrh triedy ktorá riadu činnosť programu

Obrázok 6 znázorňuje návrh triedy PetriNet ktorá bude riadiť činnosť nášho programu. Obsahuje privátne atribúty transitions, places a arcs. Tieto atribúty sú typu List. List je typu prislúchajúcich tried a to Transition, Place a Arc. Tieto atribúty sú naše vstupné dáta ktoré obsahujú všetky informácie potrebné pre činnosť Petriho siete. Trieda PetriNet obsahuje aj private atribút transformer ktorý je typu Transformer. Tento atribút trieda použije v konštruktore pre parsovanie vstupných dát na vygenerované triedy a následne ich z týchto vygenerovaných tried prevedie na naše triedy a uloží do

prislúchajúcich atribútov. Vstupný parameter konštruktora je transfromer typu Transformer ktorý v konštruktore používame. Private atribút cliExecutor typu CliExecutor trieda použije vo funkcii run pre vykonávanie príkazov v príkazovom riadku. Public metóda run riadi celú činnosť programu pomocou atribútov dostupných v tejto triede. V tejto metóde sa načítajú a spracujú vstupné dáta, načítajú sa tu vstupy od používateľa a tiež sa tu riadi spúšťanie prechodov a prepočítavanie stavu celej Petriho siete. Metóda tiež poskytuje používateľovi všetky dôležité informácie ako napríklad spustiteľné prechody vo forme konzolového výstupu. Vstup od používateľa ako napríklad voľba prechodu, ktorý chce spustiť, metóda získava z konzolového vstupu. Návratová hodnota metódy je void. Všetky atribúty majú get a set metódy, ktoré v diagrame nie sú uvedené.

3 Implementácia zadania

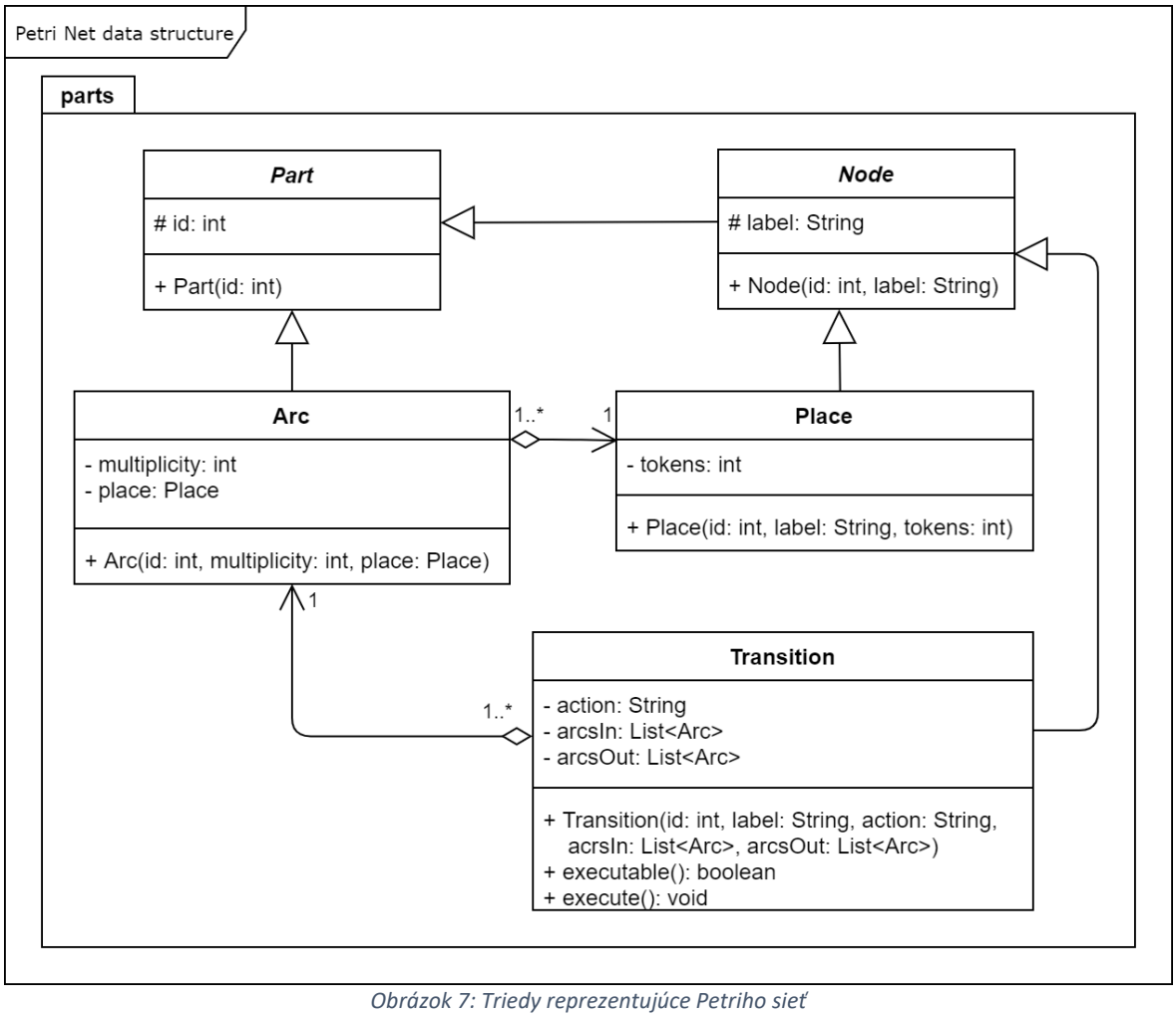
V tejto kapitole sa nachádza popis implementácie finálnej verzii programu. Nachádza sa tu opis tried, metód a atribútov, ich použitie a funkcia. Vysvetlenie používaných technológií sa nachádza v kapitole 2 Analýza problematiky a preto v tejto kapitole nebudú vysvetlené. Implementácia zadania v tejto kapitole sa značne líši od pôvodného návrhu v analýze. V implementácii pribudli funkcie, atribúty, triedy a ujasnilo sa mnoho nejasných častí. Pre kompletnosť riešenia budú v tejto kapitole opísané aj časti, ktoré sa od analýzy nezmenili. Implementácia programu je opísaná pomocou UML diagramov. Všetky triedy obsahujú metódy get a set pre svoje atribúty, ale pre lepšiu čitateľnosť ich v diagramoch neuvádzame. Taktiež ak v diagrame nie je uvedený konštruktor, znamená to že trieda používa defaultný konštruktor.

3.1 Validácia vstupných dát

Dáta validujeme pomocou súboru XSD v ktorom je zadefinovaný formát vstupných dát. Formát týchto dát sme navrhli v kapitole 2.5 Validácia vstupných dát a zostáva rovnaký aj v implementácii. Zdrojový kód 2 znázorňuje príklad Petriho siete v XML. XSD súbor ktorým tieto dáta definujeme má názov PetriNetDataDefinition.xsd a je umiestnený v adresári resources/xsd. V súbore je zadefinované, že celá sieť je obalená značkou <document>. V nej sa nachádzajú značky <place>, <transition>, <arc> v ktorých sa nachádzajú atribúty a ich hodnoty prislúchajúcich komponentov Petriho siete. Značka <place> obsahuje značky <id>, <label>, <tokens>. Značka <transition> obsahuje <id>, <label>, <action> a značka <arc> obsahuje <id>, <sourceId>, <destinationId>, <multiplicity>. Všetky tieto značky sú zadefinované ako povinné. Taktiež sme v súbore zadefinovali určité vlastnosti všetkých vstupných hodnôt atribútov. ID musí byť unikátne naprieč všetkým komponentom. Je to kladné číslo okrem nuly a jeho dĺžka môže byť maximálne päť číslic. Meno miesta môže byť dlhé v rozsahu nula až päťdesiat znakov. Meno prechodu môže byť dlhé jeden až päťdesiat znakov a musí byť unikátne aby bolo používateľovi jasné, ktorý prechod spúšťa. Whitespace znaky čo sú napríklad medzery a znak nového riadku sú zo začiatku a konca mena automaticky odstránené. Počet tokenov miesta je kladné číslo vrátane nuly. Príkaz v prechode môže mať od nula do dvesto znakov. Váha hrany je kladné číslo okrem nuly. ID zdroja a cieľa hrany musia byť kladné čísla

okrem nuly dĺžky maximálne päť číslic. Tieto ID čísla musia byť platné ID referencie miesta alebo prechodu. V triede Transformer sa nachádza metóda pre dodatočnú validáciu ktorú nebolo možné vykonať pomocou definície v XSD súbore. Validujeme, že ID zdroja a cieľa hrany nesmú byť rovnakého typu komponentu, teda miesto nesmie byť spojené s miestom a prechod nesmie byť spojený s prechodom. Triedu podrobnejšie opíšeme v neskoršej kapitole.

3.2 Triedy reprezentujúce Petriho sieť



Obrázok 7: Triedy reprezentujúce Petriho sieť

Obrázok 7 znázorňuje diagram tried ktoré reprezentujú Petriho sieť. Všetky triedy sa nachádzajú v balíku parts. Trieda Arc, Place a Transition predstavujú hranu, miesto a prechod. Abstraktná trieda Part predstavuje základ komponentu a obsahuje protected atribút id typu int. Toto id je unikátne identifikačné číslo ktoré vlastnia všetky komponenty. Id je nutné, aby bolo jasné o ktorý komponent sa jedná. Abstraktná trieda

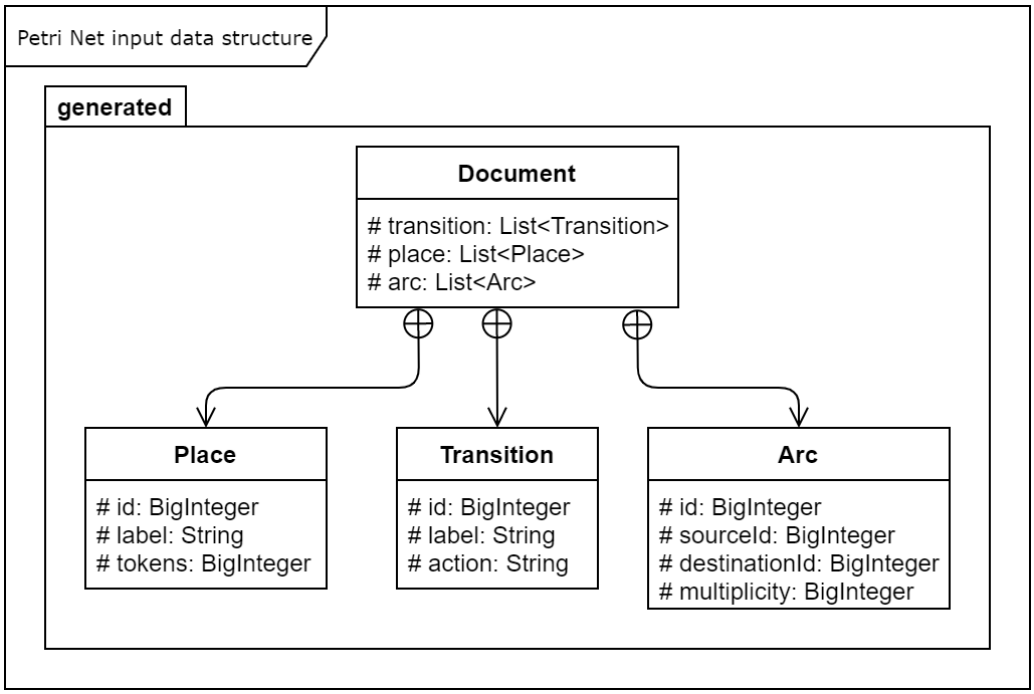
Node predstavuje uzol a obsahuje protected atribút label typu String. Tento atribút uchováva meno komponentu. Trieda Node dedí od triedy Part nakoľko je uzol komponent. Uzle Petriho siete sú miesta a prechody a preto triedy Place a Transition dedia od tejto triedy Node. Hrana nie je uzol ale je komponent a preto trieda Arc dedí od triedy Part. Triedy Part a Node obsahujú parametrické konštruktory.

Trieda Place obsahuje private atribút tokens typu int ktorý predstavuje počet tokenov daného miesta. Konštruktor triedy je parametrický.

Trieda Arc obsahuje private atribút multiplicity typu int ktorý uchováva hodnotu váhu hrany. Privátny atribút place typu Place obsahuje referenciu na miesto, s ktorým je hrana spojená. Trieda obsahuje parametrický konštruktor.

Na rozdiel od analýzy kde jadrom Petriho siete bola hrana, v implementácií sme sa rozhodli že bude vhodnejšie keď jadrom bude prechod. Toto rozhodnutie značne zrýchli a zjednoduší chod programu. To že je prechod jadrom znamená, že Petriho sieť je budovaná okolo prechodu a tiež že prechod riadi chod Petriho siete. Z toho vyplýva, že trieda Transition obsahuje private atribúty arcsIn a arcsOut typu List ktorý je typu Arc. To znamená, že každý prechod obsahuje zoznam vstupných a výstupných hrán, ktoré do neho vchádzajú alebo z neho vychádzajú. List obsahuje referencie na hrany, nie len ich id vďaka čomu vieme pri spúšťaní prechodov jednoduchšie pracovať s Petriho sieťou. Trieda Transition obsahuje aj private atribút action typu string ktorý uchováva príkaz, ktorý vykonáme v príkazovom riadku po spustení prechodu. Trieda obsahuje public metódy executable a execute. Metóda executable vráti true ak je prechod spustiteľný a false ak nie je. Metóda execute vykoná spustenie prechodu, čo znamená že prepočíta tokeny v miestach s ktorými sú spojené vstupné a výstupné hrany na základe váh hrán. V prípade vstupných hrán sa z miesta na ktoré hrana obsahuje referenciu odpočíta od atribútu tokens atribút multiplicity. V prípade výstupných hrán sa k atribútu tokens pripočíta atribút multiplicity. Trieda Transition obsahuje parametrický konštruktor. O spúšťanie príkazov v príkazovom riadku sa stará iná trieda a metóda, ktorá bude opísaná neskôr.

3.3 Triedy generované pomocou JAXB2



Obrázok 8: Vygenerované triedy

Obrázok 8 znázorňuje diagram tried generovaných pomocou JAXB2 Maven pluginu. Tieto triedy sú generované na základe nášho XSD súboru petriNetDataDefinition.xsd. Triedy sú generované do adresára target/classes/generated. Ako môžeme vidieť tak triedy sú zhodné s formátom nášho XML súboru ktorým vkladáme vstupné dáta. Trieda Document slúži ako obal tried Place, Transition a Arc. Obsahuje protected atribúty transition, place, arc typu List ktorý je prislúchajúceho typu Place, Transition alebo Arc. To znamená, že trieda Document predstavuje celý náš XML súbor a tým aj celú Petriho sieť. Trieda Place obsahuje protected atribúty id typu BigInteger, label typu String a tokens typu BigInteger. Trieda Transition obsahuje protected atribúty id typu BigInteger, label typu String a action typu String. Tieto atribúty uchovávajú rovnaké hodnoty ako atribúty tried ktoré sme vytvorili v balíku parts pre reprezentáciu Petriho siete. Atribúty triedy Place sa od triedy Place z balíka parts líšia tým, že trieda Place neobsahuje zoznamy vstupných a výstupných hrán. Trieda Arc má protected atribúty id typu BigInteger, sourceId typu BigInteger, destinationId typu BigInteger a multiplicity typu BigInteger. Atribúty id a multiplicity predstavujú rovnaké hodnoty ako tieto atribúty v triede Arc z balíka parts. Na rozdiel od triedy v balíku parts má táto trieda atribúty sourceId

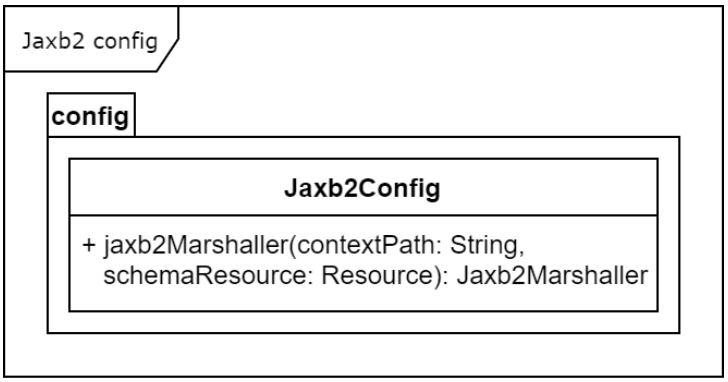
a destinationId. Tieto atribúty obsahujú id komponentu zdroja a cieľa hrany. Triedy Place, Transition a Arc sú vnorené triedy v triede Document.

Podmienky dát ktoré sme v XSD súbore definovali sa preniesli na vygenerované triedy a ich atribúty formou anotácií. Nad každou triedou sa nachádza anotácia @XmlType ktorá obsahuje údaj o atribútoch a ich poradí. Nad triedou Document sa navyše nachádza anotácia @XmlRootElement ktorej údaj určuje, že Document je koreňový element ostatným triedam. Nad atribútmi v triedach Place, Transition a Arc sa nachádza anotácia @XmlElement ktorej údaj hovorí o tom, či je atribút povinný. Taktiež sa nad týmito atribútmi nachádza anotácia @XmlSchemaType ktorej údaj definuje jeho typ v XSD resp. XML dokumente. Všetky tieto triedy slúžia iba pre uchovávanie dát a preto okrem set a get metód iné metódy neobsahujú.

3.4 Konfigurácia triedy Jaxb2Marshaller

Triedu Jaxb2Marshaller používame pre parsovanie vstupných dát z XML súboru do vygenerovaných tried a tiež pre opačnú operáciu, parsovanie dát z vygenerovaných tried do XML súboru. Taktiež ju používame pre validáciu parsovaných dát podľa XSD súboru. Opis Jaxb2 sa nachádza v kapitole 2.6 Parsovanie vstupných dát zo súboru.

Balík Spring oxm nám značne zjednodušuje prácu s JAXB2. Vďaka Spring bean factory dokážeme jednoducho konfigurovať marshaller bez toho, aby sme museli vytvoriť JAXB2 context, JiBx factories atď. Marshaller môžeme konfigurovať ako akýkoľvek iný bean v kontexte našej aplikácie. Spring oxm poskytuje množstvo ďalších výhod ktoré tu ale spomínať nebudeme nakoľko nie sú pre nás dôležité. (13)

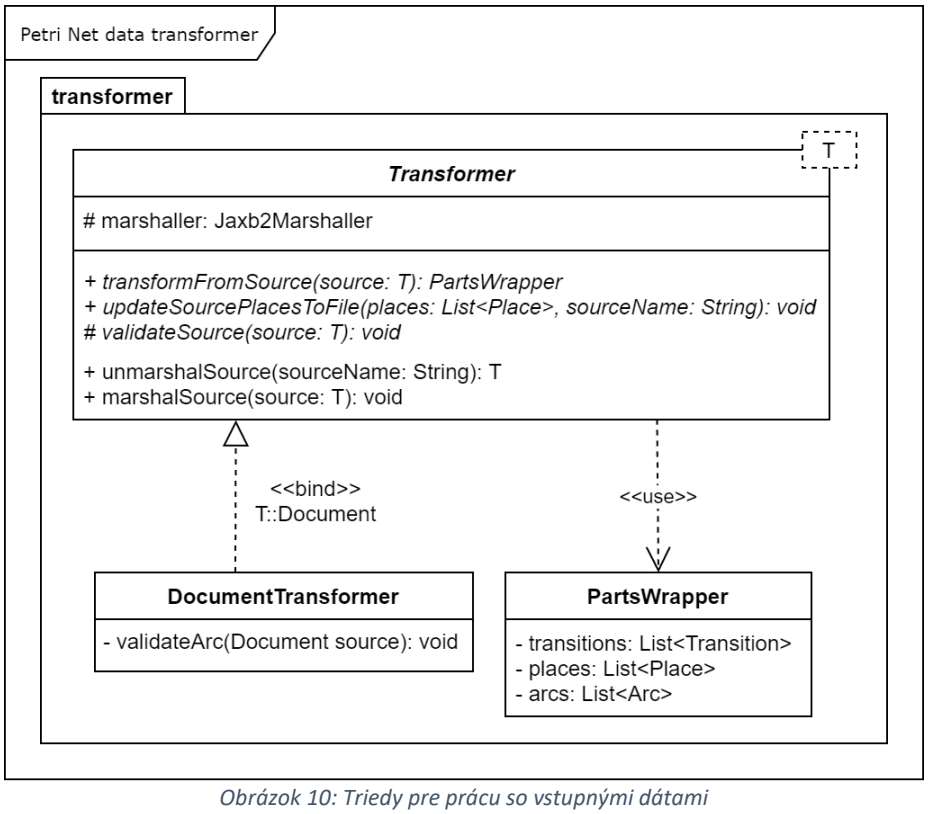


Obrázok 9: Trieda pre konfiguráciu Jaxb2

Obrázok 9 znázorňuje diagram triedy Jaxb2Config. Trieda sa nachádza v balíku config. Trieda slúži pre konfiguráciu triedy Jaxb2Marshaller. V public metóde

jaxb2Marshaller sa nachádza táto konfigurácia Jaxb2Marshaller beanu a preto nad metódou musí byť anotácia @Bean. Vstupné parametre metódy sú contextPath typu string a schemaResource typu Resource. Parameter contextPath je hodnota ktorá obsahuje názov adresára v ktorom sú vygenerované triedy do ktorých budeme parsovať XML súbor so vstupnými dátami. Parameter schemaResource je XSD súbor podľa ktorého budeme hodnoty parsovať a validovať. Hodnoty oboch parametrov sa nachádzajú v súbore application.properties, ktorý sa nachádza v adresári resources. V našom prípade súbor application.properties obsahuje nasledujúce hodnoty parametrov: context.path=generated a schema.location=xsd/PetriNetDataDefinition.xsd. Pred vstupnými parametrami sa nachádza anotácia @Value pomocou ktorej získame hodnoty zo súboru application.properties. Výstup metódy jaxb2Marshaller je inštancia triedy Jaxb2Marshaller ktorá je v metóde nakonfigurovaná. Naša konfigurácia spôsobí, že po použití anotácie @Autowired na parameter typu Jaxb2Marshaller v akejkoľvek časti našej aplikácie Spring automaticky tento parameter pomocou tejto konfigurácie inicializuje. Tento proces sa nazýva dependency injection.

3.5 Triedy pre prácu so vstupnými dátami



Obrázok 10: Triedy pre prácu so vstupnými dátami

Obrázok 10 znázorňuje diagram tried Transformer, DocumentTransformer a PartsWrapper. Tieto triedy slúžia pre prácu so vstupnými dátami a nachádzajú sa v balíku transformer. Trieda Transformer je abstraktná generická trieda. Má jediný atribút a to protected marshaller typu Jaxb2Marshaller. Nad atribútom sa nachádza anotácia @Autowired čo znamená, že sa o jeho inicializáciu stará Spring. Trieda je inicializovaná podľa metódy jaxb2Marshaller z triedy Jaxb2Config ktorá je podrobnejšie opísaná v predchádzajúcej kapitole. Metódy z triedy Jaxb2Marshaller ktoré používame sú metóda marshal a unmarshal. Metóda unmarshal prevedie vstupné dáta z XML súboru na triedy a metóda marshal prevedie dáta z tried do XML súboru. Počas tohto prevodu sú dáta validované.

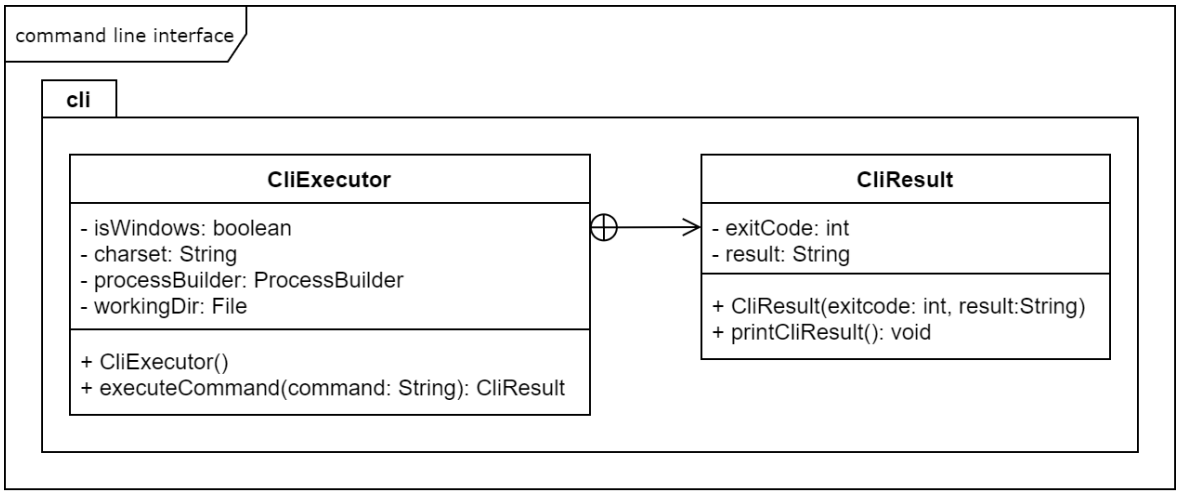
Trieda PartsWrapper obsahuje tri private atribúty a to List typu Transition, Place a Arc. Tieto atribúty sú triedy z balíka parts ktoré slúžia pre reprezentáciu Petriho siete. Trieda PartsWrapper slúži ako obal týchto tried aby sa nám s nimi jednoduchšie pracovalo.

Trieda Transformer obsahuje niekoľko metód pre prácu so vstupnými dátami. Public abstraktná metóda transformFromSource prevedie dáta z objektu source typu T ktorý je vstup metódy na objekt typu PartsWrapper, ktorý je jej výstupom. Public abstraktná metóda updateSourcePlacesToFile aktualizuje počty tokenov miest v súbore podľa ich hodnôt ktoré sú v Liste typu Place. Metóda má dva vstupy. Prvý vstup metódy je parameter places typu List ktorý je typu Place. Je to List prechodov ktoré chceme v súbore aktualizovať. Druhý vstup je parameter sourceName typu String. Je to názov súboru v ktorom chceme hodnoty zodpovedajúce prechodom z prvého parametra aktualizovať. Predpokladáme, že tento súbor je rovnaký súbor, z ktorého sme parsovali vstupné dáta. Zápis do súboru je riešený aktualizovaním prechodov z dôvodu zvýšenia efektívnosti programu. Ak by sme do súboru znovu parsovali celú sieť bolo by to menej efektívne. Treba brať do úvahy, že v čase volania tejto metódy musí existovať pôvodný súbor s dátami ktoré aktualizujeme. Výstup metódy je void. Protected abstraktná metóda validateSource slúži pre validáciu vstupných dát. Je to validácia ktorú nebolo možné vykonať pomocou XSD súboru a preto ju treba vykonať programovo. Vstup metódy je objekt source typu T. Výstup metódy je void. Vyššie uvedené metódy sú abstraktné pretože sa ich implementácia bude líšiť podľa typu objektu ktorý uchováva vstupné dáta. Tento objekt je generovaný a jeho formát sa môže zmeniť. Public metóda unmarshalSource prevedie vstupné dáta zo súboru na objekt. Vstup metódy je parameter sourceName typu

String. Je to názov súboru, ktorý prevádzame. Výstup metódy je objekt typu T. Public metóda marshalSource prevedie dáta z objektu do súboru. Vstup metódy je parameter source typu T ktorý predstavuje prevádzaný objekt. Názov súboru do ktorého sú dáta prevedené je PetriNetDataProgressBackup.xml. Metódu marshalSource používa metóda updateSourcePlacesToFile ktorú používame pri zálohovaní súčasného stavu Petriho siete. Výstup metódy je void. Všetky metódy ktoré čítajú alebo zapisujú súbory tak robia z adresára v ktorom je spustený program. Treba dávať pozor na správne umiestnenie súboru aby nebol problém s jeho otvorením prípadne vytvorením. Vyššie uvedené neabstraktné triedy sú implementované ale napriek tomu ich treba zdediť pred ich použitím, pretože používajú generický typ.

Trieda DocumentTransformer je trieda ktorá dedí od triedy Transfromer. Táto trieda predstavuje implementáciu triedy Transformer pre vstupné dáta vo forme objektu typu Document. To znamená, že všetky objekty typu T z triedy Transformer budú typu Document. V triede DocumentTransformer je nutné implementovať všetky abstraktné metódy z triedy Transformer. Trieda navyše obsahuje private metódu validateArc ktorej vstupom je parameter source typu Document a výstup je void. V metóde validujeme, či nejaká hrana z objektu source nespája dve miesta alebo dva prechody. V prípade takejto chyby metóda vyhodí výnimku. Implementácia metódy validateSource volá metódu validateArc. Žiadne iné programové validácie Petriho siete nevykonávame.

3.6 Triedy pre vykonávanie príkazov v príkazovom riadku



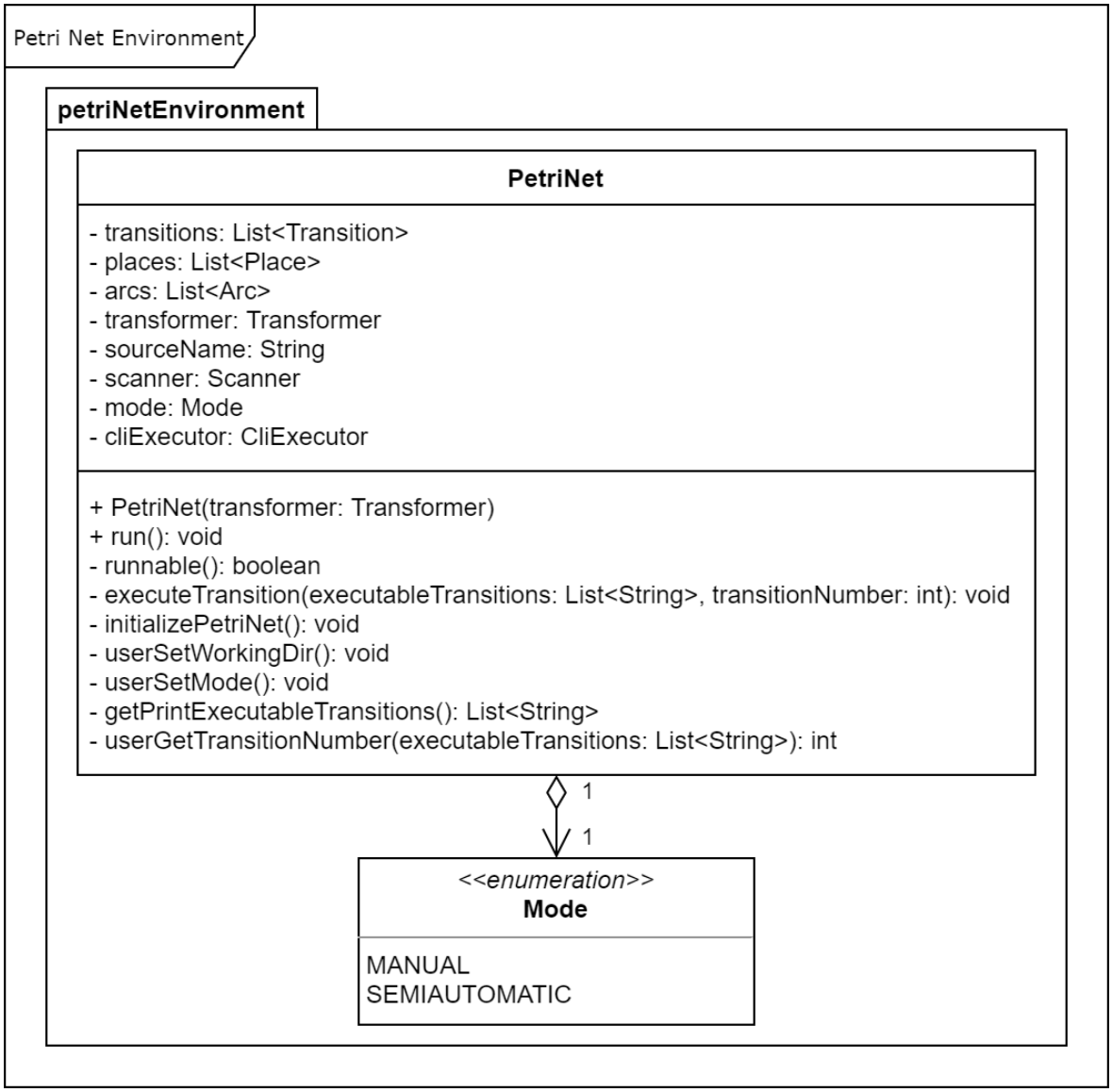
Obrázok 11: Triedy ktoré vykonávajú príkazy v príkazovom riadku

Obrázok 11 zobrazuje diagram tried, ktoré používame pre vykonávanie príkazov v príkazovom riadku. Tieto triedy sa nachádzajú v balíku cli. Trieda CliExecutor má private atribút isWindows typu boolean ktorý uchováva hodnotu true ak je program spustený v operačnom systéme Windows a false ak nie je. Je to z dôvodu aby sme mohli prispôsobiť implementáciu funkcií operačnému systému, vďaka čomu bude program spustiteľný vo Windows aj v Linux. Private atribút charset uchováva hodnotu znakovej sady, ktorý náš operačný systém využíva. Potrebujeme ho, aby sme výstup do konzoly správne formátovali a zobrazili. Private atribút processBuilder typu ProcessBuilder nám umožňuje vykonávať príkazy vo zvolenom príkazovom riadku. Pre Windows je to cmd a pre Linux je to sh. Tieto príkazy vykonávame pomocou funkcie command ktorú ProcessBuilder poskytuje. Ďalší opis triedy ProcessBuilder sa nachádza v kapitole 2.4 Vykonávanie príkazov v príkazovom riadku. Posledný private atribút workingDir typu File uchováva adresár v ktorom chceme vykonávať príkazy. Všetky vyššie uvedené atribúty inicializujeme v konštruktore triedy. Pri inicializácii atribútu processBuilder nastavíme jeho pracovný adresár pomocou atribútu workingDir. Hodnota atribútu charset pre operačný systém Linux je UTF-8. Pri operačnom systéme Windows musíme získať presnú hodnotu používanej stránky znakov, anglicky character page. Túto hodnotu získame vykonaním príkazu chcp v príkazovom riadku. Výstup príkazu je napríklad hodnota Active code page: 852. Túto hodnotu spracujeme a do atribútu charset uložíme iba hodnotu, ktorá sa nachádza za dvojbodkou, v tomto prípade je to 852. Public metóda executeCommand vykoná v príkazovom riadku príkaz, ktorý sa nachádza vo vstupnom parametri command typu String. Výstup metódy je vo forme objektu typu CliResult. Metóda formátuje výstup podľa atribútu charset aby mal správny tvar pri jeho vypisovaní.

Trieda CliResult uchováva výstup po vykonaní príkazu v príkazovom riadku. Private atribút exitCode typu int uchováva hodnotu nula ak sa príkaz vykonal úspešne a inú hodnotu, ak zlyhal. Private atribút result typu String obsahuje výstup po vykonaní príkazu. V prípade chyby obsahuje túto chybu. Trieda používa parametrický konštruktor. Trieda má iba jednu metódu a to public metódu printCliResult. Metóda nemá žiadne vstupné parametre a jej výstup je void. Metóda vypíše hodnotu result do konzoly. Pred jej vypísaním vypíše Operation execution: SUCCESS ak exitCode je nula a Operation execution: FAILURE ak je exitCode iná hodnota. Vďaka tomu bude jasné, či vypísaná

hodnota obsahuje výsledok po vykonaní príkazu alebo hlášky chyby v prípade zlyhania. Trieda CliResult je vnorená trieda triedy CliExecutor.

3.7 Trieda riadiaca chod programu



Obrázok 12: Trieda riadiaca chod programu

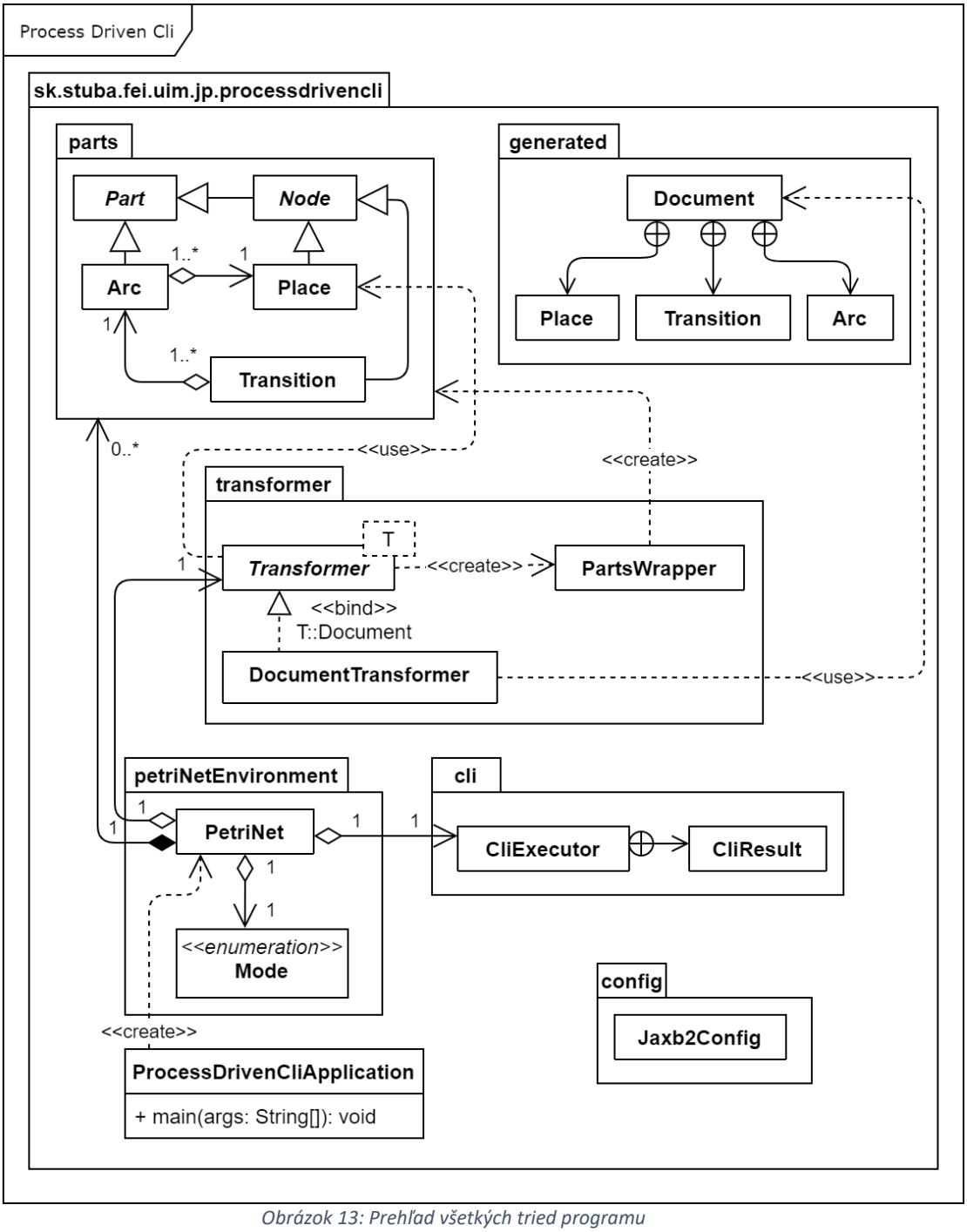
Obrázok 12 zobrazuje diagram ktorý znázorňuje triedu, ktorá riadi činnosť programu. Táto trieda sa volá PetriNet a nachádza sa v balíku petriNetEnvironment. Private atribúty transitions, places a arcs sú objekty typu List ktoré sú prislúchajúceho typu Transition, Place a Arc. Tieto atribúty tvoria reprezentáciu Petriho siete. Private atribút transformer typu Transformer používame pre spracovanie súboru so vstupnými dátami, ktorými naplníme atribúty transitions, places a arcs. Private atribút sourceName typu

String obsahuje meno súboru z ktorého berieme vstupné dáta. Atribút scanner typu Scanner je private a používame ho pri získavaní vstupov od používateľa. Private atribút mode typu Mode obsahuje hodnotu ktorá určuje mód behu programu. Enumeračná trieda Mode obsahuje dve možnosti hodnôt. Prvá hodnota MANUAL znamená, že pre každé spustenie prechodu sa očakáva vstup od používateľa. Druhá hodnota SEMIAUTOMATIC znamená, že ak existuje iba jeden spustiteľný prechod tak sa spustí automaticky. Používateľ ho nemusí spúšťať zadaním vstupu. Private atribút cliExecutor typu CliExecutor slúži pre vykonávanie príkazov v príkazovom riadku. Nad týmto atribútom sa nachádza anotácia @Autowired čo znamená, že o jeho inicializáciu sa stará Spring. Konštruktor triedy PetriNet má vstupný parameter transformer typu Transformer. Nad konštruktorom sa nachádza anotácia @Autowired čo znamená, že o inicializáciu vstupných parametrov sa stará Spring. V našom prípade do tohto objektu Spring vloží inštanciu objektu typu DocumentTransformer nakoľko je to jediná implementácia abstraktnej triedy Transformer. To znamená, že atribút transformer triedy PetriNet bude obsahovať inštanciu triedy DocumentTransformer. Ďalší atribút, ktorý v tomto konštruktore inicializujeme je scanner. Ostatné atribúty inicializujeme pomocou iných metód na začiatku chodu programu.

Public metóda run je metóda ktorá riadi celý chod programu pomocou ostatných private metód triedy PetriNet a tiež pomocou metód atribútov cliExecutor a transformer. Metóda nemá žiadne vstupné parametre a jej výstup je void. Po spustení metódy run sa na jej začiatku zavolá metóda initializePetriNet. Metóda si vypýta od používateľa názov súboru s dátami Petriho siete. Tento súbor spracuje pomocou metód atribútu transformer a inicializuje atribúty places, transitions a arcs. Metóda initializePetriNet nemá žiadne vstupné parametre a jej výstup je void. Nasledujúca metóda ktorá sa zavolá je metóda runnable. Metóda vráti true ak existuje aspoň jeden spustiteľný prechod. Ak neexistuje, vráti false a program sa skončí. Métóda nemá žiadne vstupné parametre. Ďalšie metódy v metóde run ktoré sa zavolajú sú metódy userSetWorkingDir a userSetMode. Obe metódy nemajú žiadne vstupné parametre a ich výstup je void. Metóda userSetWorkingDir si od používateľa vypýta názov adresára, v ktorom chceme vykonávať príkazy v príkazovom riadku. Vstupom od používateľa sa nastaví atribút workingDir v triede cliExecutor. Ak používateľ nezadá žiadnu hodnotu, tak sa tento atribút nastaví na hodnotu domovského adresára používateľa. V operačnom systéme Windows 10 to zvyčajne býva

C:\Users\UserName. Metóda userSetMode sa používateľa spýta, v ktorom móde chce aby program pracoval. Metóda touto hodnotou nastaví atribút mode. Používateľ musí zadať číslo jedna pre MANUAL a číslo dva pre SEMIAUTOMATIC. Nasledujúca volaná metóda je getPrintExecutableTransitions, ktorá nemá žiadne vstupné parametre. Metóda vypíše používateľovi všetky spustiteľné prechody. Výstup metódy je List typu String ktorý obsahuje mená všetkých spustiteľných prechodov. Následne sa zavolá metóda userGetTransitionNumber. Metóda slúži pre získanie voľby používateľa ktorý prechod chce spustiť. Vstup metódy je parameter executableTransitions typu List ktorý je typu String. Tento parameter obsahuje mená spustiteľných prechodov z ktorých má používateľ na výber. Používateľ si vyberá spustiteľný prechod zadaním poradového čísla ktoré funkcia prechodu pridelí, aby používateľ nemusel zadávať celé meno prechodu. Výstup metódy je typu int. Je to poradové číslo prechodu ktorý chce používateľ spustiť. Ďalšia volaná metóda je executeTransition. Táto metóda spustí používateľom zvolený prechod pomocou metód ktoré poskytuje atribút cliExecutor. Jeho spustením sa prepočítajú tokeny v miestach s ktorými je tento prechod spojený hranami. Taktiež sa v príkazovom riadku vykoná príkaz uložený v prechode v atribúte action a výsledok po tejto operácii sa vypíše na konzolu. Prvý vstupný parameter metódy je parameter executableTransitions typu List ktorý je typu String a obsahuje mená spustiteľných prechodov. Druhý parameter je transitionNumber typu int ktorý predstavuje poradové číslo prechodu ktorý spúšťame. Nasledujúca metóda ktorá sa volá je metóda updateSourcePlacesToFile z triedy Transformer. Táto metóda uloží aktuálny stav Petriho siete do súboru PetriNetDataProgressBackup.xml vďaka čomu vieme v prípade ukončenia programu po jeho opätovnom spustení pokračovať, odkiaľ sme skončili. Na záver sa znovu zavolá metóda runnable ktorá kontroluje ukončenie programu.

3.8 Prehľad všetkých tried programu



Obrázok 13: Prehľad všetkých tried programu

Obrázok 13 zobrazuje diagram všetkých tried, ktoré v programe využívame. Všetky triedy sa nachádzajú v balíku sk.stuba.fei.uim.jp.processdrivencli. Na diagrame sú zobrazené vzťahy medzi triedami. Každý balík a triedy v ňom sú opísané podrobnejšie vo vlastnej kapitole. Tieto kapitoly môžeme nájsť vyššie. Nasleduje opis vzťahov medzi

triedami ktoré sa nachádzajú v rozdielnych balíkoch. Trieda ktorou začína náš program je trieda ProcessDrivenCliApplication. Táto trieda obsahuje main metódu, v ktorej vytvoríme triedu PetriNet a zavolaním jej metódy run spustíme chod programu. Z diagramu môžeme vyčítať, že pre triedu PetriNet sú triedy z balíka parts neoddeliteľnou súčasťou. Bez týchto tried by trieda PetriNet nemala zmysel. Tieto triedy slúžia pre reprezentáciu dát Petriho siete. Trieda PetriNet využíva triedu Transfromer z balíka transformer a tiež triedu CliExecutor z balíka cli. Pomocou triedy Transformer spracovávame vstupné dáta a pomocou triedy CliExecutor vykonávame príkazy v príkazovom riadku. Trieda Transformer z balíka transformer využíva triedu Place z balíka parts. Trieda PartsWrapper z balíka transformer vytvára triedy z balíka parts. Trieda DocumentTransformer z balíka transformer je implementáciou abstraktnej triedy Transformer. Táto trieda využíva generovanú triedu Document z balíka generated.

3.9 Zmena formátu vstupných dát

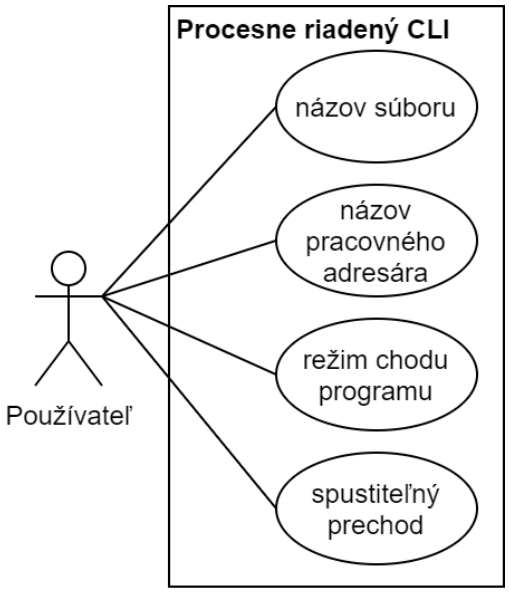
Program sme navrhli tak, aby bolo možné zmeniť formát vstupných dát. Dosiahli sme to tak, že sme oddelili vrstvu ktorá sa a zaoberá vstupnými dátami od vrstvy, ktorá riadi logiku programu. V tejto kapitole opíšeme, čo všetko je v implementácii nutné zmeniť pre zmenu formátu vstupných dát.

Prvé čo treba zmeniť je implementáciu súčasného XSD súbor za druhý, ktorý definuje novú štruktúru vstupných dát. Je nutné aby tento súbor definoval minimálne tie atribúty, ktoré sú potrebné pre funkčnosť Petriho siete. Tieto atribúty môžeme vidieť v triedach v balíku parts. Súbor ktorý treba zmeniť sa nazýva PetriNetDataDefinition.xsd a nachádza sa v adresári resources/xsd. Pokiaľ chceme zmeniť názov alebo umiestnenie tohto XSD súboru, treba zmeniť hodnotu schema.location v súbore application.properties. Súbor application.properties sa nachádza v adresári resources. Na základe XSD súboru sa vygenerujú triedy, ktoré uchovávajú vstupné dáta po ich parsovaní pomocou JAXB2. Umiestnenie vygenerovaných tried nastavíme pomocou hodnoty context.path v súbore application.properties.

Druhá a posledná vec ktorú treba zmeniť je implementácia abstraktnej generickej triedy Transformer. Súčasná implementácia sa nazýva DocumentTransformer ktorá dokáže pracovať iba so súčasnou vygenerovanou triedou, ktorej názov je Document. Táto trieda sa nachádza v balíku generated. Objekt typu T v triede Transformer predstavuje objekt typu

Document v triede DocumentTransformer. V prípade zmeny formátu XSD súboru sa zmení aj formát vygenerovaných tried. Nová implementácia abstraktnej generickej triedy Transformer musí pracovať s novou vygenerovanou triedou. Samozrejme treba zmeniť implementáciu abstraktných metód z triedy Transformer. Sú to nasledujúce metódy: transformFromSource, updateSourcePlacesToFile a validateSource. Tieto triedy sú abstraktné pretože pracujú s vygenerovanou triedou a v prípade zmeny tejto triedy sa ich implementácia značne zmení.

4 Ukážka chodu programu



Obrázok 14: Vstupy používateľa

Obrázok 14 znázorňuje všetky vstupné hodnoty ktoré musí používateľ zadať na začiatku programu. Používateľ musí zadať názov súboru v ktorom sa nachádzajú vstupné dáta Petriho siete. Ďalej musí zadať názov pracovného adresára, v ktorom sa budú vykonávať príkazy v príkazovom riadku. Nakoniec je nutné ešte vybrať, v akom režime chce používateľ spustiť program. Používateľ ma na výber dva režimy a to MANUAL a SEMIAUTOMATIC. Po zadaní týchto hodnôt sa začne vykonávať chod Petriho siete a používateľ si môže začať vyberať, ktorý spustiteľný prechod chce spustiť.

Obrázok 15 znázorňuje ako vyzerá program po jeho spustení v príkazovom riadku cmd operačného systému Windows 10. Program je spúšťaný pomocou .jar súboru programu. Spúšťame ho pomocou príkazu java -jar processDrivenCli.jar. Po spustení programu sú od používateľa pýtané tri dôležité vstupné hodnoty, ktoré sme už spomenuli vyššie. Ako vidíme, v našom prípade názov súboru so vstupnými dátami je PetriNetData. Obrázok 1 znázorňuje Petriho sieť, ktorá sa nachádza v tomto súbore. Zdrojový kód 2 znázorňuje presný obsah tohto súboru. Aby bolo možné súbor otvoriť, musí sa nachádzať v rovnakom adresári z ktorého spúšťame .jar súbor programu. Nasledujúca vstupná hodnota je názov pracovného adresára, ktorú sme v našom prípade nechali prázdnu nakoľko nám vyhovuje defaultný adresár C:\Users\Juraj. Názov adresára musí byť vo forme celej cesty aby bolo jasné, o ktorý adresár sa jedná. Ďalšia vstupná hodnota je výber režimu chodu programu. V našom prípade sme zvolili režim MANUAL. Po zadaní

všetkých troch vstupných hodnôt sa spustí vykonávanie Petriho siete a vypíšu sa spustiteľné prechody. V našom prípade je to iba jeden prechod a to prechod s názvom t1. Program si vyžiada voľbu prechodu ktorý chceme spustiť. Nakoľko je iba jeden spustiteľný prechod tak ten sme zvolili. Po jeho spustení sa vykoná v príkazovom riadku prechodu prislúchajúci príkaz, v našom prípade je to echo test. Program vypíše hlášku Operation execution: SUCCESS ktorá znamená, že sa príkaz vykonal úspešne. Následne sa vypíše výstup po vykonaní príkazu ktorý je v našom prípade výpis slova test. Po spustení prechodu sa aktuálny stav Petriho siete uloží do súboru PetriNetDataProgressBackup.xml. Umiestnenie tohto súboru je v rovnakom adresári z ktorého spúšťame .jar súbor programu. Následne program skončí nakoľko neexistuje žiadny ďalší spustiteľný prechod.

```
Microsoft Windows [Version 10.0.19042.985]
(c) Microsoft Corporation. Všetky práva vyhradené.

C:\Users\Juraj>cd Desktop

C:\Users\Juraj\Desktop>java -jar processDrivenCli.jar

  .   ____          _            __ _ _
 /\\ / ___'_ __ _ _(_)_ __  __ _ \ \ \ \
( ( )\___ | '_ | '_| | '_ \/ _` | \ \ \ \
 \\/  ___)| |_)| | | | | || (_| |  ) ) ) )
  '  |____| .__|_| |_|_| |_\__, | / / / /
 =========|_|==============|___/=/_/_/_/
 :: Spring Boot ::        (v2.3.4.RELEASE)

2021-05-29 23:50:10.351  INFO 5824 --- [           main] s.s.f.u.j.p.ProcessDrivenCliApplication  : Starting ProcessDriv
enCliApplication v0.0.1-SNAPSHOT on DESKTOP-87ETPH1 with PID 5824 (C:\Users\Juraj\Desktop\processDrivenCli.jar started b
y Juraj in C:\Users\Juraj\Desktop)
2021-05-29 23:50:10.351  INFO 5824 --- [           main] s.s.f.u.j.p.ProcessDrivenCliApplication  : No active profile se
t, falling back to default profiles: default
2021-05-29 23:50:10.915  INFO 5824 --- [           main] s.s.f.u.j.p.ProcessDrivenCliApplication  : Started ProcessDrive
nCliApplication in 0.82 seconds (JVM running for 1.088)

Zadajte nazov suboru ktory chcete otvorit: PetriNetData
Zadajte nazov adresara v ktorom chcete pracovat, alebo nezadajte nic pre defaultnu hodnotu 'C:\Users\Juraj':
Vyberte rezim chodu programu, zadajte cislo 1. Manualny, 2. Semiautomaticky: 1
Spustitelne prechody:
1. t1

Zadajte cislo prechodu ktory chcete spustit: 1
Operation execution: SUCCESS

test


C:\Users\Juraj\Desktop>
```

Obrázok 15: Ukážka chodu programu vo Windows 10 cmd

5 Testovanie implementácie programu

Program sme testovali v dvoch rôznych operačných systémoch typu UNIX a to v operačnom systéme Windows 10 a Linux Ubuntu 20.04. Ako prvé sme testovali či sa vstupné dáta Petriho siete zo súboru správne validujú a tiež či sa správne parsujú do programu. Tiež sme testovali či sa správne spracovávajú vstupné dáta ktoré používateľ zadáva na začiatku programu. V prípade nesprávnych vstupných dát sme testovali či je používateľ o ich nesprávnosti informovaný a tiež či sú od neho vyžiadané správne dáta. Všetky testy ohľadom vstupných dát prebehli úspešne. Následne sme testovali funkčnosť chodu Petriho siete. Testovali sme, či rôzne návrhy Petriho sietí fungujú podľa očakávaní. To znamená, že sme testovali či sú prechody spustiteľné len ak vyhovujú všetky podmienky a tiež či sa po ich spustení správne pripočítajú a odpočítajú hodnoty tokenov v prislúchajúcich miestach.

Ďalšie čo sme testovali bolo vykonávanie príkazov v príkazovom riadku. Pre vykonávanie príkazov v príkazovom riadku používame už existujúcu Java knižnicu ProcessBuilder. Nakoľko je naša funkcionalita vykonávania príkazov závislá od funkcionality tejto knižnice, počítame s tým že je jej funkčnosť zaručená. Z tohto dôvodu sme testovali najmä či je naše použitie tejto knižnice správne. Testy našej implementácie použitia tejto knižnice prebehli úspešne, príkazy do nej správne vkladáme a tiež správne získavame a vypisujeme výstupné hodnoty.

Napriek tomu že sa spoliehame na funkčnosť knižnice ProcessBuilder, otestovali sme aspoň niektoré základné najpoužívanejšie príkazy. Sú to napríklad príkazy pre prácu so súbormi, príkazy ktoré vypisujú údaje ako napríklad obsah adresára a tiež spúšťanie .exe súborov vrátane spúšťania s argumentami. Všetky testy prebehli úspešne, okrem testu príkazu cd. Tento príkaz slúži pre zmenu pracovného adresára príkazového riadku. V prípade použitia tohto príkazu je konzolový výstup chybová hláška a zmena adresára sa neuskutoční. Nakoľko trieda ProcessBuilder obsahuje vlastnú hodnotu pracovného adresára v ktorom sa vykonávajú príkazy, nie je možné túto hodnotu zmeniť konzolovým príkazom. Z tohto dôvodu odporúčame používateľom sa tomuto príkazu vyhýbať. V prípade potreby zmeny pracovného adresára počas chodu programu, je možné program ukončiť a spustiť z iného adresára. Súbor PetriNetDataProgressBackup obsahuje posledný

stav Petriho siete a preto je možné pokračovať v Petriho sieti otvorením tohto súboru. Nie je nutné začínať od začiatku otvorením súboru PetriNetData.

Posledné čo sme testovali bolo vykonávanie viacerých príkazov v rámci jedného prechodu. V operačnom systéme Windows 10 môžeme pomocou znaku & spájať viacero príkazov dokopy. Príkazy sú vykonávane postupne. Pomocou znakov && môžeme tiež spájať viacero príkazov dokopy, ale nasledujúci príkaz sa vykoná, iba v prípade že predchádzajúci príkaz sa vykonal bez chyby. Spojením príkazov pomocou znakov || sa nasledujúci príkaz vykoná iba v prípade, že predchádzajúci príkaz skončí s chybou. V operačnom systéme Linux Ubuntu 20.04 dokážeme spájať príkazy pomocou znaku ;. V oboch operačných systémoch dokážeme spájať príkazy pomocou znaku |. Týmto znakom sa vezme výstup predchádzajúceho príkazu a vloží sa na vstup nasledujúceho. Všetky testy ktoré vykonávajú viacero príkazov naraz prebehli úspešne, ale vykonávanie viacerých príkazov v jednom prechode neodporúčame. Ak niektorý z príkazov zlyhá, môže to spôsobiť neočakávané problémy, nakoľko nám nemusí byť jasné ktorý z príkazov zlyhal. Taktiež nedokážeme znovu vykonať iba príkazy ktoré zlyhali, nakoľko dokážeme vykonať príkazy uložené v prechode iba ako celok. Testovali sme aj presmerovanie výstupu z príkazu pomocou znakov >, >> a tiež vkladanie vstupu do príkazu pomocou znaku <. Nakoniec sme testovali vykonávanie príkazov pomocou podmienok if. Všetky tieto testy tiež prebehli úspešne.

Záver

V tejto práci sme úspešne splnili všetky plánované ciele. Na začiatku práce sme si naštudovali problematiku Petriho sietí a z týchto poznatkov sme napísali základy teórie, ktoré sú dôležité pre porozumenie práce. Následne sme analyzovali dostupné riešenia a technológie, z ktorých sme vybrali tie najvhodnejšie a vypracovali sme hrubý návrh riešenia. Po vypracovaní návrhu sme vypracovali implementáciu programu, ktorú sme podrobne opísali a znázornili na diagramoch. Program sme navrhli tak, aby bolo možné v prípade potreby zmeniť formát vstupných dát. V práci sme opísali akým spôsobom je tak možné urobiť.

Použitie programu sme znázornili a opísali na jednoduchom príklade použitia. Implementáciu sme otestovali a zistili sme, že funguje správne a podľa našich očakávaní. Výnimkou je nefunkčnosť zmeny pracovného adresára pomocou konzolového príkazu cd, ktorý nie je funkčný vzhľadom na funkcionalitu využívanej knižnice ProcessBuilder. Usúdili sme, že tento problém nie je závažný a dá sa mu vyhnúť. V programe sme navyše implementovali aby sa aktuálny stav Petriho siete automaticky ukladal do súboru po každom spustení prechodu. Vďaka tomu vieme v prípade potreby uložiť stav Petriho siete. To nám umožňuje pokračovanie vykonávania Petriho siete z iného adresára a tiež možnosť predčasného ukončenia a znovu spustenia programu kde sme skončili.

Po vyskúšaní a testovaní rôznych Petriho sietí sme dospeli k záveru, že naše riešenie implementácie rozhrania príkazového riadku je efektívne a spoľahlivé. Náš program je schopný uľahčiť používateľovi robotu, ak pravidelne vykonáva určité operácie pomocou príkazového riadku. Používateľ si môže navrhnúť Petriho siete podľa jeho potreby a spúšťať ich kedykoľvek potrebuje. Program poskytuje manuálny aj poloautomatický režim chodu. Takýmto spôsobom sme zvýšili úroveň automatizovanosti, nakoľko používateľ nemusí voliť spustiteľný prechod ak je iba jeden. Program dokáže spustením prechodu vykonať jeden alebo viacero programov v príkazovom riadku. Napriek tomu že dokáže vykonať aj viacero príkazov v rámci spustenia jedného prechodu, odporúčame aby každý prechod obsahoval iba jeden príkaz. Spúšťanie viacerých príkazov z jedného prechodu môže viesť k rôznym nejasnostiam, ktoré sme opísali v predchádzajúcej kapitole.

Program aj túto dokumentáciu je možné stiahnuť na internetovej adrese https://github.com/j-puszter/Process-driven-CLI.

Zoznam použitej literatúry

1. **Chen, Wai-Kai.** *The Electrical Engineering Handbook.* s.l. : Academic Press, 2004. ISBN 978-0-12-170960-0.

2. **Smith, Graeme.** *Concurrent Systems.* Boston : Springer, 2000. ISBN 978-1-4613-7401-5.

3. **Gibb, Robert.** StackPath. *What is a Distributed System?* [Online] 26. Júl 2019. [Dátum: 21. Marec 2021.] https://blog.stackpath.com/distributed-system/.

4. **Pawlewski, Pawel.** *Petri Nets - Manufacturing and Computer Science.* Rijeka : InTech, 2012. ISBN 978-953-51-0700-2.

5. **Desel, Jörg a Juhás, Gabriel.** *"What Is a Petri Net?" Informal Answers for the Informed Reader.* Berlin; Heidelberg : Springer, 2001. ISBN 978-3-540-43067-4.

6. **HACHINET.** HACHINET. *What is Java? Why choose Java?* [Online] 30. 06 2020. [Dátum: 25. 4 2021.] https://hachinet.com/blogs/what-is-java-why-choose-java.

7. **Baeldung.** Baeldung. *Why Choose Spring as Your Java Framework?* [Online] 13. 12 2019. [Dátum: 25. 4 2021.] https://www.baeldung.com/spring-why-to-choose.

8. **The Apache Software Foundation.** The Apache Software Foundation. *What is Maven?* [Online] 25. 4 2021. [Dátum: 25. 4 2021.] https://maven.apache.org/what-is-maven.html.

9. **Jervis, Michael.** SitePoint. *XML DTDs Vs XML Schema.* [Online] 26. 11 2002. [Dátum: 3. 5 2021.] https://www.sitepoint.com/xml-dtds-xml-schema/.

10. **Satyabrata_Jena.** GeeksforGeeks. *Difference Between SAX Parser and DOM Parser in Java.* [Online] 17. 3 2021. [Dátum: 4. 5 2021.] https://www.geeksforgeeks.org/difference-between-sax-parser-and-dom-parser-in-java/.

11. **Baeldung.** Baeldung. *Guide to JAXB.* [Online] 17. 8 2019. [Dátum: 4. 5 2021.] https://www.baeldung.com/jaxb.

12. —. Baeldung. *How to Run a Shell Command in Java.* [Online] 13. 1 2020. [Dátum: 9. 5 2021.] https://www.baeldung.com/run-shell-command-in-java.

13. **VMware.** Spring Framework. *Chapter 8. Marshalling XML using O/X Mappers.* [Online] 23. 8 2013. [Dátum: 14. 5 2021.] https://docs.spring.io/spring-ws/site/reference/html/oxm.html.