EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

Evidenčné číslo: 103003/I/2017/36080377334281476

SIMULÁCIA VYBRANÉHO PODNIKOVÉHO PROCESU

Diplomová práca

2017 **Bc. Tomáš Ballon**

EKONOMICKÁ UNIVERZITA V BRATISLAVE
FAKULTA HOSPODÁRSKEJ INFORMATIKY

SIMULÁCIA VYBRANÉHO PODNIKOVÉHO PROCESU

Diplomová práca

Študijný program: Informačný manažment

Študijný odbor: Kvantitatívne metódy v ekonómii

Školiace pracovisko: Katedra operačného výskumu a ekonometrie

Vedúci záverečnej práce: Ing. Marian Reiff, PhD.

Bratislava 2017 **Bc. Tomáš Ballon**

Ekonomická univerzita v Bratislave
Fakulta hospodárskej informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bc. Tomáš Ballon
Študijný program: Informačný manažment (Jednoodborové štúdium, inžiniersky II. st., externá forma)
Študijný odbor: 3.3.24 Kvantitatívne metódy v ekonómii
Typ záverečnej práce: Inžinierska záverečná práca
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Simulácia vybraného podnikového procesu

Anotácia: Diplomová práca má za cieľ modelovať vybraný proces pomocou metodiky dynamickej simulácie diskrétnych udalostí v programovom nástroji Java a Simul8. V prostredí Java je cieľom naprogramovať simulačný nástroj, aplikáciu na modelovanie vybraného procesu obsluhy.

Vedúci: Ing. Tomáš Domonkos, PhD.
Vedúci: Ing. Marian Reiff, PhD.
Katedra: KOVE FHI - Katedra operačného výskumu a ekonometrie FHI
Vedúci katedry: prof. Mgr. Juraj Pekár, PhD.

Dátum zadania: 09.11.2015

Dátum schválenia: 11.11.2015

prof. Ing. Michal Fendek, PhD.
vedúci katedry

Čestné vyhlásenie

Čestne vyhlasujem, že záverečnú prácu som vypracoval samostatne a že som uviedol všetku použitú literatúru.

Dátum: 27. apríla 2017 .....

Poďakovanie

Týmto by som chcel poďakovať vedúcemu mojej diplomovej práce, Ing. Marianovi Reiffovi, PhD. , za rady a usmernenia pri tvorbe diplomovej práce.

Abstrakt

BALLON, Tomáš: Simulácia vybraného podnikového procesu – Ekonomická univerzita v Bratislave. Katedra operačného výskumu a ekonometrie FHI. – Ing. Marian Reiff, PhD. - Bratislava: FHI EU, 2017, počet strán 78.

Diplomová práca má za cieľ modelovať vybraný proces pomocou metodiky dynamickej simulácie diskrétnych udalostí v programovom nástroji Java a Simul8. V prostredí Java je cieľom naprogramovať simulačný nástroj, aplikáciu na modelovanie vybraného procesu obsluhy. Práca je rozdelená do troch kapitol.

V prvej kapitole sa popisuje základný prehľad problematiky simulácie. V ďalšej kapitole sa popisuje návrh a samotná implementácia programu v Jave. V poslednej kapitole sa aplikácia verifikuje pomocou programu SIMUL8 a hodnotí sa jej možné využitie a vylepšenie v budúcnosti.

Výsledkom práce je aplikácia SimulApp, v ktorej je možné vykonávať simuláciu ekonomických procesov. Práca popisuje kroky nutné pre vytvorenie danej aplikácie.

Kľúčové slová: dynamická simulácia, počítačová simulácia, simulačný nástroj, Java, SIMUL8

Abstract

BALLON, Tomáš: Simulation of the selected corporate process. University of Economics in Bratislava. Operational research and econometrics department FHI. – Ing. Marian Reiff, PhD. - Bratislava: FHI EU, 2017, 78 pages.

The aim of the thesis is to model a selected process by the methodics of dynamic simulation of discrete events in a programming tool Java and SIMUL8. The aim in Java environment is to program a simulation tool, application for modelling of the selected operation process. The thesis is divided into 3 chapters.

The first chapter deals with basic overview of simulation issues. The next one describes suggestion and implementation of the program in Java. In the last chapter the application is being verified by the programme SIMUL8 and its possible use and improvement in the future is being evaluated.

The result of the thesis is an application SimulApp, in which it is possible to perform simulation of economic processes. It describes steps necessary for creation of the application.

Key words: dynamic simulation, computer simulation, simulation tool, Java, SIMUL8

Obsah

Úvod

Simulácia má v dnešnej dobe mnoho využití v rôznych vedných disciplínach. Nesie v sebe obrovský potenciál, práve kvôli tomu, že môžeme simulovať reálne situácie zo života bez toho, aby sme akokoľvek riskovali. So simuláciou sa stretávame v bežnom živote, napríklad aj pri jazde v trenažéri v autoškole, či pri hraní niektorých počítačových hier.

Simulácia hrá v podnikovom prostredí veľmi dôležitú úlohu. Dovoľuje nám vyskúšať si rôzne situácie cvične, bez ohrozenia financií podniku. Pomocou simulácie dokážeme odhaliť slabé miesta v podniku, alebo v konkrétnom podnikovom procese. Okrem toho vieme odstrániť nadbytočné prvky a pripraviť sa na rôzne smery vývoja ekonomického prostredia.

Existuje veľké množstvo simulačných metód a nástrojov. My sa zameriame na simulačný softvér umožňujúci vytvárať dynamické simulácie a na možnosti vývoja vlastného simulačného softvéru. Navrhneme a naprogramujeme vlastný nástroj v ktorom budeme schopní vytvoriť a spustiť základnú simuláciu. Prispôsobíme ho na možné rozšírenie v budúcnosti. Tak isto sa zameriame na to, aby bol náš nástroj použiteľný na najznámejších platformách. Vytvoríme ho pomocou bezplatného softvéru, zahŕňajúc operačný systém Linux, ale aj bezplatné vývojové prostredie. Tým poukážeme na to, že pre vytvorenie takéhoto programu nám postačia aj bezplatné nástroje.

V prvej časti práce spracujeme prehľad danej problematiky, popíšeme počítačovú simuláciu, jej históriu. Ďalej sa zameriame na výber vhodného programovacieho nástroja a vývojového prostredia pre programovanie nášho riešenia.

V hlavnej časti práce sa zameriame na návrh aplikácie a na jej samotné programovanie. Rozdelíme ju do viacerých logických celkov a každému z nich sa budeme venovať zvlášť. Ďalej zhodnotíme najväčšie problémy s ktorými sa pri programovaní stretneme.

Po naprogramovaní funkčnej aplikácie navrhneme netriviálnu simuláciu, ktorá bude vhodná pre náš simulačný program, ale aj pre platenú aplikáciu SIMUL8. Pomocou porovnania viacerých spustení týchto simulácii v oboch programoch overíme funkčnosť nášho vytvoreného simulačného nástroja.

1 Prehľad danej problematiky

Táto časť diplomovej práce sa zaoberá prehľadom danej problematiky. V prvom rade popíšeme, čo je to simulácia a ako sa tvorí. Ďalej sa zameriame na najpoužívanejší simulačný software. Nakoniec si priblížime možnosti programovania simulácie.

1.1 Počítačová simulácia

Pri fungovaní každej organizácie sa stretávame s problémami, ako správne rozvrhnúť zdroje organizácie čo najefektívnejšie, aby nám priniesli čo najväčší prospech. Najlepšie by bolo otestovať si niekoľko nastavení organizácie a porovnať výsledky hospodárenia pri daných nastaveniach. Ak by sme to robili za behu s reálnymi jednotkami, mohlo by nás to stáť veľa času, ale aj peňazí. Pre tento účel bola vymyslená počítačová simulácia. Využívajú ju najmä manažéri a dokážu pomocou nej predvídať a optimalizovať podnikové procesy vo firme. Jednou z najväčších výhod simulácie je, že sa deje v počítači bez nutnosti zásahu do fungovania podniku. Takto sa vyhneme chybám v nastavení podniku, ktoré nás môžu pripraviť o nemalé finančné čiastky.

Najjednoduchšie by sme základnú myšlienku simulácie mohli popísať ako napodobnenie zložitého podnikového systému pomocou počítačového modelu. Následne jeho využitie na experimentovanie a pozorovanie zmien a chovania systému. Čím je modelovaný systém zložitejší, tým viac vyniknú výhody simulácie.[1]

1.1.1 História

Na začiatku tu bola metóda Monte Carlo, ktorá je založená na využití náhodných veličín a teórie pravdepodobnosti. Najviac sa používala v druhej polovici dvadsiateho storočia, pri nástupe počítačov. Táto metóda vyžadovala veľmi dobrý generátor náhodných čísel a vhodný algoritmus riešení.[2] O generovaní náhodných čísel budeme písať neskôr.

Metódu Monte Carlo si môžeme rozdeliť do troch základných krokov:

1. Rozbor problému a návrh modelu;
2. Generovanie náhodných veličín a ich transformácia na veličiny s daným stochastic-

1 Dlouhý, Fábry, Kuncová, Hladík. 2007. Simulace podnikových procesů. Brno: Computer Press, a.s., s. 5

2 Hradec Králové: Univerzita Hradec Králové, Pedagogická fakulta, Katedra fyziky a informatiky, [cit. 2017-03-30]. Dostupné na internete: http://www.black-hole.cz/soubory/mc.pdf.

kým rozdelením;

3. Štatistické spracovanie výsledkov.[3]

Od metódy Monte Carlo bola časom odvodená simulácia ako samostatná vedná disciplína. Dodnes ich považujeme za veľmi príbuzné. Za autora metódy Monte Carlo je považovaný poľský matematik Stanislaw Ulam. Údajne ju vymyslel pri skúmaní pravdepodobnosti výhry v kartách.

Simuláciu je možné definovať ako metódu štúdia zložitých pravdepodobnostných dynamických systémov pomocou experimentov s počítačovým modelom. Simuláciu môžeme chápať ako napodobňovanie reálnych javov a procesov za pomoci počítačových modelov. Simulácia sa využíva v mnohých disciplínach a nie len vo vede. Máme napríklad letecké či vojenské simulátory, ale aj simulátory, s ktorými sme sa mohli stretnúť osobne. Takým je napríklad jazdný simulátor v autoškole. Medzi simulácie môžeme takisto zaradiť aj počítačové hry, ako napríklad podnikové alebo ekonomické hry, ktoré simulujú fungovanie podnikov, marketingových stratégií a pod.[4]

1.1.2 Simulačný projekt

Cieľom simulačného projektu je zlepšenie podnikových procesov. Takýto projekt je zložený z niekoľkých fáz, podobne ako je to spomenuté v knihe *Simulace podnikových procesů*. Dané rozdelenie na fázy nie je jediné možné a jediné správne. Takisto môžeme niektoré fázy preskočiť za účelom šetrenia času, alebo vzhľadom na typ simulovaného procesu.

Fázy simulačného projektu:

1. Rozoznanie problémov a stanovenie cieľov;
2. Vytvorenie konceptuálneho modelu – konceptuálny model je základnou predstavou o modelovanom systéme;
3. Zber dát;
4. Tvorba simulačného modelu – konceptuálny model pretransformujeme do modelu

3 TESAŘ, Jiří; BARTOŠ, Petr. České Budějovice: Pedagogická fakulta Jihočeské univerzity v Č. Budějovicích, Katedra fyziky, [cit. 2017-03-30]. Dostupné na internete: http://dsp.vscht.cz/konference_matlab/MATLAB06/prispevky/tesar_bartos/tesar_bartos.pdf.

4 Dlouhý, Fábry, Kuncová, Hladík. 2007. Simulace podnikových procesů. Brno: Computer Press, a.s., s. 8 - 10

simulačného;

5. Verifikácia a validácia modelu – overíme, či je vytvorený počítačový model v súlade s pôvodným konceptuálnym modelom;
6. Experimenty a analýza výsledkov;
7. Dokumentácia modelu;
8. Implementácia modelu – implementácia získaných výsledkov do praxe.[5]

1.2 Náhodné čísla a variabilita procesov

Čísla v podnikových procesoch nie sú ekonomicky dané. Nemáme povedané, aký dlhý čas bude trvať konkrétna činnosť a je tu veľká účasť náhody. Napríklad sa pozrieme na príklad z reálneho života, akým je vybavovanie občanov na niektorom z úradov. Takmer každý človek rieši unikátne problémy, a tak vybavenie každého z nich trvá iný čas. Pri rôznych procesoch môžu nastať rôzne situácie, ktoré trvajú odlišný čas. Vieme ich modelovať pomocou distribúcií náhodných čísel. Preto je dôležité sa zamerať pri simulácii aj na generovanie náhodných čísel. Existuje niekoľko typov rozdelení náhodných čísel. Jedná sa o spojité a diskrétne rozdelenia, ktoré sa ďalej rozdeľujú na konkrétne typy rozdelení, ktorým sa budeme venovať samostatne.

Náhodné čísla chápeme ako nezávislé hodnoty rovnomerného rozdelenia na otvorenom intervale (0, 1).[6] Pre generovanie náhodných čísel existujú dva druhy generátorov, a to generátory mechanické a generátory aritmetické. Mechanickým generátorom môžeme nazvať aj kocku zo spoločenskej hry Človeče nehnevaj sa. Vždy, keď ju hodíme, padne nám náhodné číslo. Aritmetické generátory sa zas používajú v počítačoch. Tieto aritmetické generátory negenerujú reálne náhodné čísla, ale čísla pseudonáhodné. Znamená to, že procesor počítača nejakým spôsobom odvodzuje číslo na základe algoritmu, ktorý musí byť vhodne zvolený. Pri dobrom algoritme sú charakteristiky týchto pseudonáhodných čísel veľmi podobné náhodným číslam.

1.2.1 Spojité rozdelenia

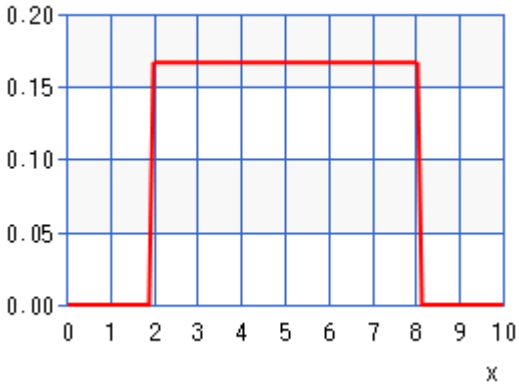
V tejto časti si popíšeme spojité rozdelenia pravdepodobnosti. Najviac používaným

5 Dlouhý, Fábry, Kuncová, Hladík. 2007. Simulace podnikových procesů. Brno: Computer Press, a.s., s. 11
6 Dlouhý, Fábry, Kuncová, Hladík. 2007. Simulace podnikových procesů. Brno: Computer Press, a.s., s. 17

rozdelením v simulácii je exponenciálne rozdelenie. Najčastejšie sa používa na generovanie intervalov medzi po sebe nasledujúcimi príchodmi požiadaviek, generovanie dĺžky trvania požiadaviek alebo náhodných porúch. Exponenciálne rozdelenie má jediný parameter a tým je λ. Pri modelovaní príchodov môže byť nazvaný ako intenzita príchodov za jednotku času.

Obrázok 1: Exponenciálne rozdelenie

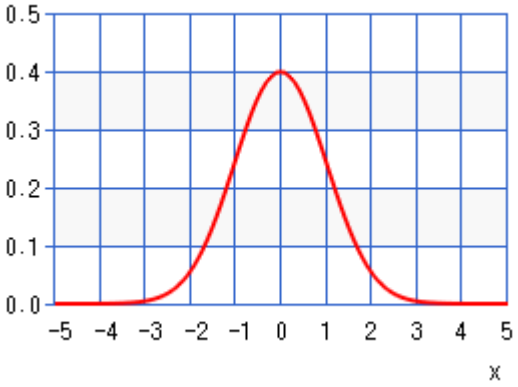
Ďalším spojitým rozdelením je rovnomerné rozdelenie. Rovnomerné rozdelenie je dané dvoma parametrami. Tie reprezentujú minimálnu a maximálnu hodnotu náhodného čísla. Pre každé číslo z tohto intervalu platí, že môže byť vygenerované s rovnakou pravdepodobnosťou.



Obrázok 2: Rovnomerné rozdelenie

Posledným spojitým rozdelením, ktoré si popíšeme je normálne rozdelenie. Graf tohto rozdelenia má zvonový tvar. Najčastejšie sa používa pre zachytenie chyby pri

ekonomických pozorovaniach, ale aj na generovanie dôb trvania činností. Normálne rozdelenie má dva parametre. Prvým je μ, teda stredná hodnota a druhým je σ^2, ktorým je definovaný rozptyl.[7]

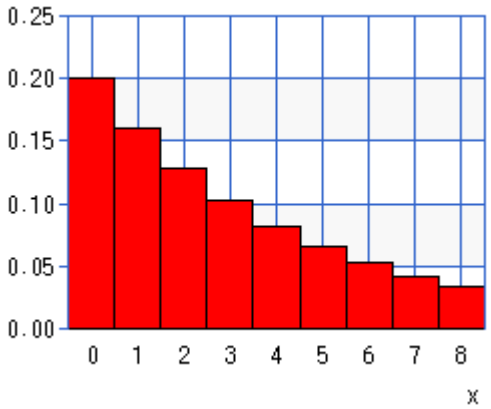


Obrázok 3: Normálne rozdelenie

1.2.2 Diskrétne rozdelenia

Pri diskrétnych rozdeleniach je treba spomenúť geometrické a binomické rozdelenie. Pri diskrétnych rozdeleniach môže mať nejaký jav hodnotu pravda alebo nepravda. Zjednodušene povedané, ten jav nastať môže, ale nemusí.

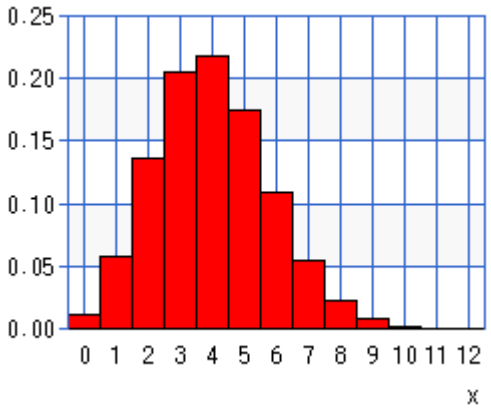
Geometrické rozdelenie slúži na popis rozdelenia počtu úspechov javu, ktorý nastane v n pokusoch. Toto rozdelenie má jediný parameter p, ktorý hovorí o nastúpení priaznivého javu.



Obrázok 4: Geometrické rozdelenie

7 Dlouhý, Fábry, Kuncová, Hladík. 2007. Simulace podnikových procesů. Brno: Computer Press, a.s., s. 25 - 29

Ďalším diskrétnym rozdelením je binomické rozdelenie. Náhodná premenná X v binomickom rozdelení nám popisuje, ako sa rozdelí priaznivý jav v n nezávislých realizáciách náhodného pokusu. Napríklad koľkokrát padne hlava pri desiatich hodoch mincou.[8]



Obrázok 5: Binomické rozdelenie

1.3 Možnosti simulácie

V tejto kapitole si popíšeme spôsoby, ktorými sme schopní vytvoriť počítačovú simuláciu. Popíšeme si dva najpoužívanejšie simulačné programy, ale aj možnosť tvorby vlastného simulačného softvéru. Existujú tri druhy simulačných modelov, a to diskrétny, spojitý a kombinovaný simulačný model. My sa zameriame na diskrétnu simuláciu a softvér pre diskrétnu simuláciu.

1.3.1 Existujúci softvér pre tvorbu počítačovej simulácie

Existuje niekoľko počítačových programov na tvorbu simulácie. Disponujú dôležitou funkcionalitou, ktorú môžeme potrebovať. Obsahujú rôzne objekty, jednotky a procesy generovania a vizualizácie výsledkov. Ich najväčšou výhodou je to, že nás odbremeňujú od nutnosti vytvárania celých systémov a od začiatku sa môžeme zamerať na tvorbu nášho modelu. No obsahujú aj veľké množstvo funkcionality, ktorú možno nebudeme nikdy potrebovať. Spomenieme si z toho dva komplexné programy. Prvým bude SIMPROCESS a druhým SIMUL8.

SIMPROCESS vyvíja americká firma CACI Products. Je to hierarchický modelovací nástroj, ktorý kombinuje diskrétnu simuláciu a kalkuláciu nákladu založenú na činnos-

8 Dlouhý, Fábry, Kuncová, Hladík. 2007. Simulace podnikových procesů. Brno: Computer Press, a.s., s. 30 - 32

tiach v jednom prehľadnom používateľskom prostredí. Jeho funkciami sú hierarchické mapovanie procesov, objektovo orientované modelovanie, animácia procesu, reportovanie výsledkov, grafov a mnoho atribútov pre konkrétne objekty simulácie. Jednou z jeho najväčších výhod je multiplatformovosť, keďže je dostupný pre Windows, MAC OS, ale aj Linuxové operačné systémy.

Základné stavebné prvky simulácie v programe SIMPROCESS:

- **Procesy a aktivity**, ktoré reprezentujú podnikové činnosti;
- **Entity** nám môžu reprezentovať ľudí, či pohybujúci sa tovar a iné informácie;
- **Zdroje**, ktoré v sebe nesú entity. Zdroje sú väčšinou obmedzené, čo je významným limitujúcim faktorom v podnikových procesoch.[9]

SIMUL8 je produkt firmy SIMUL8 Corporation. Slúži na tvorbu diskrétnej simulácie. Podobne ako SIMPROCESS je založený na základných objektoch simulácie. Na svojom webe je prezentovaný ako výkonný, intuitívny simulačný softvér, ktorý môže ktokoľvek použiť na rýchle získanie výsledkov. Zlepšuje procesy, zvyšuje efektivitu a pomáha znížiť náklady. SIMUL8 používajú veľké spoločnosti ako American Airlines, GM alebo aj NASA.

Základnými prvkami simulácie v SIMUL8 sú:

- **Entita**, alebo účastník simulácie;
- **Vstup**, je miesto kde do systému vstupujú entity;
- **Pracovné stredisko**, cez ktoré prechádzajú entity a sú tu spracovávané. Prichádzajú o zdroje alebo nadobúdajú nové;
- **Zásobník** je bodom, kde sa zhromažďujú entity a čakajú, kým sa dostanú do pracovného strediska;
- **Výstup** je bodom, kde entity opúšťajú simuláciu;
- **Zdroj** slúži na modelovanie jednotiek, akými sú napríklad peniaze a pod.;
- **Cesta** nám prepája jednotlivé vstupy, výstupy, pracovné strediská a zásobníky.[10]

9 CACI Products, [cit. 2017-03-30]. Dostupné na internete: http://simprocess.com/about-simprocess/.
10 SIMUL8 Corporationn, [cit. 2017-03-30]. Dostupné na internete:https://www.simul8.com/.

1.3.2 Tvorba vlastného modelovacieho nástroja

Vyššie sme sa zoznámili s najpoužívanejším simulačným softvérom, spomenuli sme si hlavne jeho výhody. Každý z týchto simulačných nástrojov má používateľské hranice, za ktoré sa nedostaneme. Všetky objekty a funkcie sú v nich pevne dané a nemáme možnosť pridávať nové alebo upravovať ich. Nemáme možnosť dopracovať s k dokumentácii a zdrojovému kódu. V neposlednom rade je ich veľkou nevýhodou vysoká cena (najdostupnejšia licencia SIMUL8 basic stála v marci 2017 1995 $).

Programovanie vlastného simulačného softvéru môže byť náročné a zložité. No ak vytvoríme dostatočne jednoduchý a rozšíriteľný simulačný program, ktorý by mohol byť dostupný aj pre verejnosť, každý, kto má skúsenosti s programovaním by bol schopný rozšíriť si toto riešenie pre vlastné potreby. A toto by bolo jednou z jeho najväčších výhod. Od začiatku je potrebné zamerať sa na jednoduchosť, prehľadnosť a možnosť budúcej úpravy a rozšírenia. Takto získame obrovskú flexibilitu simulátora, ktorý nemusí byť prepchatý všetkou možnou funkcionalitou, ale bude pripravený na integráciu iba tých funkcií, ktoré sú pre daného používateľa alebo pre daný druh simulácie potrebné. V základnej verzii by nám mal umožniť modelovať a simulovať základné podnikové procesy bez toho, aby sme museli zasahovať do zdrojového kódu.

1.4 Prehľad vývojových prostredí a podporného softvéru

Najzložitejšie bolo navrhnúť multifunkčnú aplikáciu, ktorou by sme boli schopní simulovať rôzne ekonomické procesy. Základ pre programovanie aplikácie je výber vhodného jazyka, v ktorom bude aplikácia naprogramovaná a výber IDE (Integrated Development Environment), teda integrované vývojové prostredie. Dôležité je aj to, na akom operačnom systéme chceme našu aplikáciu používať.

1.4.1 Operačné systémy

Operačný systém je základný systém, ktorý beží na počítači, spravuje jeho zdroje a poskytuje k nim rozhranie. Medzi základné zdroje môžeme zaradiť správu pamäte počítača, ukladací priestor, procesor, ktorý nám poskytuje výpočtový výkon a vstupné a výstupné zariadenia. Pod operačným systémom bežia aplikácie programované vo vyšších programovacích jazykoch, prípadne ďalšie aplikácie na báze serverov a interpreterov skriptovacích jazykov. Takýmto príkladom servera je napríklad webový server Apache2, v ktorom bežia rôzne skriptovacie jazyky, akým je aj jazyk PHP, slúžiaci na vytváranie dynamických we-

bových aplikácií. Najpoužívanejšími operačnými systémami sú Windows, Mac OS, Linuxové a Unixové operačné systémy.

	WINDOWS	MAC OS	LINUX
Cena	Platený, bezplatný pri kúpe s PC	Bezplatný pri kúpe s PC	Bezplatný
Otvorený zdrojový kód	Nie	Nie	Áno
Platforma	X86, AMD64	Iba pre vlastný hardvér	Všetky
Online podpora	Áno	áno	Možnosť platenej, iba pri niektorých distribúciách

Tabuľka 1: Základné porovnanie operačných systémov

1.4.2 Programovacie a skriptovacie jazyky

Programovací jazyk vieme jednoducho popísať ako systematický opis postupu pri riešení konkrétneho problému. Možno ho porovnať s receptom na prípravu jedla. Recept je tak isto zjednodušene povedané „program“. Programovacie jazyky pre vytváranie aplikácií sme schopní rozdeliť do dvoch základných skupín, a to na skriptovacie jazyky a programovacie jazyky, priamo interpretované kompilátorom do strojového jazyka.

Programovacie jazyky priamo kompilované do strojového jazyka

Strojový jazyk je súbor inštrukcií priamo vykonávaných procesorom v počítači bez toho, aby museli byť akýmkoľvek spôsobom menené alebo prekladané do iného jazyka. Miernym rozdielom pri týchto jazykoch je Java. Tá sa kompiluje do bajtkódu. O tom bližšie pojednáva kapitola Java. Hoci vie procesor vykonávať strojový kód v reálnom čase, človek ho nebude nikdy schopný písať priamo, ale len prostredníctvom vyššieho programovacieho jazyka. Ten je potom kompilátorom prevedený na strojový kód, ktorému procesor rozumie. Pri tomto druhu programovacích jazykov sa spúšťa kompilátor po každej zmene kódu. Tak sme potom schopní takto skompilovanú aplikáciu spúšťať na počítači. Vyššie programovacie jazyky sú veľmi efektívne, no častokrát sú zložitejšie na pochopenie a naučenie. Ich veľkou výhodou je, že sa v nich jednoduchšie hľadajú syntaktické chyby, lebo kompilátor nedokáže skompilovať chybne zapísaný kód a väčšinou nám vráti aj

správu o tom, kde sa v kóde táto chyba nachádza.

Skriptovacie jazyky

Skriptovacie jazyky pôvodne slúžili na zjednodušenie a zautomatizovanie niektorých operácií v počítači. Najčastejšie len volali rôzne skompilované programy a nie priamo samostatné inštrukcie. Neskôr sa z nich vyvinuli oveľa sofistikovanejšie programovacie jazyky, ktoré sa už v mnohom dokážu vyrovnať tým klasickým. Skriptovacie jazyky väčšinou nedokážeme kompilovať, aj keď už existujú kompilátory na niektoré skriptovacie jazyky, preto sú skriptovacie jazyky prekladané za behu interpreterom skriptovacieho jazyka do strojového kódu. Interpreter vždy prekladá iba aktuálne spustenú časť kódu, nie celý kód naraz, ako to robia kompilátory. Tu nám vzniká medzikrok, ktorý nie je pri klasických jazykoch prítomný. Tým pádom sú skriptovacie jazyky menej efektívne, ako jazyky priamo skompilované do strojového kódu. Skriptovacie jazyky sú na rozdiel od klasických oveľa jednoduchšie na naučenie, no ťažšie sa v nich hľadajú programátorské chyby (kód je prekladaný len po častiach, nie celý naraz a chybu vyvoláme, iba keď je spustená chybná časť kódu).

1.4.3 Výber programovacieho jazyka

Pri výbere jazyka sme sa zamerali na jeho jednoduchosť a použiteľnosť na rôznych platformách. Nezanedbateľným kritériom bolo aj to, ktoré jazyky už ovládame alebo akou rýchlosťou sme schopní sa ich naučiť.

Webové jazyky PHP/HTML v kombinácii s JavaScriptom

Vyššie menované technológie sa používajú na programovanie webových aplikácií. Základom simulátora je vizualizácia informácií. Tu pripadal do úvahy jazyk HTML spolu s JavaScriptom. Na serverovej strane by mohol byť použitý jazyk PHP pre ukladanie výsledkov a simulácií. V tejto kombinácii jazykov by sme boli schopní naprogramovať požadované rozhranie, ale pri pohľade na efektivitu JavaScriptu nám táto kombinácia až tak nevyhovovala. JavaScript je totiž skriptovací jazyk.

C++

C++ je vyšší programovací jazyk, ktorý je prekladaný kompilátorom priamo do strojového kódu. V rámci efektivity je teda vhodným kandidátom pre našu aplikáciu. Jeho syntax je jednoduchá ľahko pochopiteľná. C++ je objektový programovací jazyk, čím je

vodný pre naše využitie, no beží len na systéme a architektúre, na ktorej je daný kód skompilovaný. Riadi sa pravidlom „Píš raz, skompiluj kdekoľvek.“

Java

Riadi sa pravidlom „Píš raz, spusti kdekoľvek.“, čo je oproti C++ a Pascalu, „Píš raz, skompiluj kdekoľvek.“ výhodnejšie. Aplikáciu totiž nemusíme kompilovať na každej platforme, na ktorej ju chceme spustiť. Syntakticky vychádza z jazyka C, čo je veľmi používaná syntax aj v iných programovacích a skriptovacích jazykoch. Narozdiel od C++ a Pascalu obsahuje automatickú správu pamäte (garbage collection), ktorá sa automaticky stará o vymazávanie starých a nepoužívaných údajov z pamäte, čo uľahčuje prácu programátorom (v predchádzajúcich jazykoch sa o to musel starať programátor).

V Jave sa aplikácia spúšťa vo virtuálnom stroji, čo je jednak medzikrok naviac oproti Pascalu a C++, ale umožňuje jej to multiplatformovosť bez nutnosti prekompilovania na každej platforme zvlášť. Java sa nekompiluje priamo do strojového kódu, ale do bajtkódu. Bajtkód je vysoko optimalizovaná sada inštrukcií, ktoré sú navrhnuté pre spravovanie virtuálnym strojom Javy (JVM), ale nebráni v tom, aby sa Java kompilovala do strojového kódu. Javu nekompilujeme do bajtkódu pri vývoji (ako to je v C++ a Pascal). O jej kompiláciu sa stará Java Virtual Machine, ktorá musí obsahovať technológiu JIT (Just In Time – v správny čas). Táto technológia skompiluje časti bajtkódu postupne na základe toho, ako sú dané časti kódu vyžadované. Tento proces možno nazvať „dynamická kompilácia“.[11]

Vyhodnotenie

Pri programovaní mojej aplikácie by som chcel poukázať aj na to, že programovanie podobnej aplikácie ide aj bez použitia drahých nástrojov. Aj preto som sa rozhodol pre vývoj v Jave pod operačným systémom Linux. Výsledná aplikácia bude spustiteľná na všetkých najpoužívanejších operačných systémoch pre stolné počítače a pritom bude dostatočne efektívna pre prácu s veľkým množstvom dát v reálnom čase.

11 Schild Herbert. 2012. Java 7. Brno: Computer Press, a.s., s. 24 – 27

	PHP/HTML	C++	Object Pascal	Java
Garbage Collection	Áno	Nie	Nie	Áno
Typový jazyk	Nie	Áno	Áno	Áno
Použitie	Web	Natívne aplikácie	Natívne aplikácie	Natívne aplikácie / Web
Multiplatformová aplikácia	Áno	Nie	NIe	Áno

Tabuľka 2: Porovnanie programovacích jazykov[12]

1.4.4 Prehľad vybraného programovacieho jazyka

Základné informácie o Jave sme si povedali už skôr. V tejto časti sa jej budeme venovať podrobnejšie. Povieme si niečo o jej histórii, ale aj o tom, ako sama funguje a dopodrobna si preberieme, aké možnosti programovania a dátových štruktúr nám poskytuje.

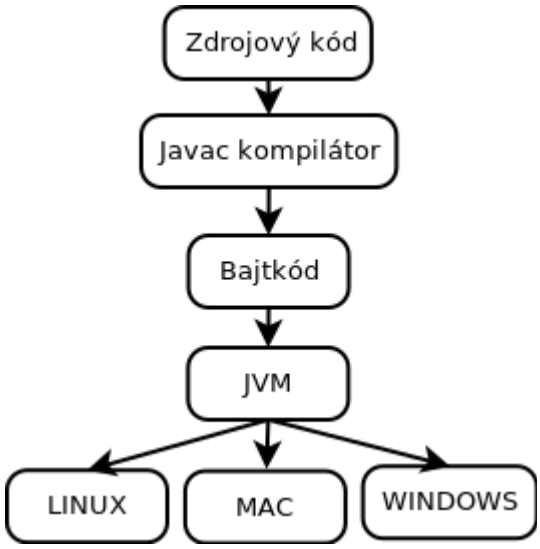
Prvá verzia Javy vznikla v roku 1995. Pôvodne bola nazývaná Oak (dub), ale zistilo sa, že jazyk s týmto menom už existuje. Pôvodne slúžila na tvorbu appletov, čo boli drobné programy bežiace na webe. V roku 1996 bol vydaný prvý Java Development Kit (JDK 1.0), v ktorom boli obsiahnuté všetky nástroje potrebné pre programovanie appletov. Pri vývoji Javy sa kládol dôraz na to, aby to bol jazyk ľahko naučiteľný a ľahko pochopiteľný. Vychádzal zo syntaxe jazyka C++, ale bol značne zjednodušený. Jednou veľkou výhodou je odstránenie používania smerníkov a automatická správa pamäte Garbage Collection. Za pomoci Garbage Collection sa Java zbavila častého problému iných programovacích jazykov, ktoré neuvoľňujú pamäť automaticky, ale musí sa o to starať programátor. Programátori často zabúdali vymazávať objekty z pamäte, a tak prichádzalo k jej zaplneniu (memory leak).[13]

Najväčším rozdielom oproti C++ a ostatným klasickým jazykom je jeho multiplatformovosť. Keďže Java beží v JVM, je absolútne nezávislá na platforme, na ktorej beží. Týmto krokom priniesla do sveta programovania niečo, čo tu ešte nebolo. V minulosti bolo možné naprogramovať softvér na jednej platforme a skompilovať a spúšťať na inej, ale pri

12 Rosetta code, [cit. 2017-04-02]. Dostupné na internete: https://rosettacode.org/wiki/Language_Comparison_Table.

13 NOVOTNÝ, Luděk. Masarykova Univerzita, Fakulta Informatiky, Historie a vývoj jazyka Java, [cit. 2017-04-02]. Dostupné na internete: http://www.fi.muni.cz/usr/jkucera/pv109/2003p/xnovotn8.htm.

Jave nutnosť tejto kompilácie úplne odpadla.



Obrázok 6: Štruktúra JVM

Postupom času Java prenikala na web vo forme appletov. Najväčším problémom týchto appletov bola časom ich bezpečnosť, a tak boli postupne z webu takmer úplne vytlačené. Dnes je už problém applet spustiť v prehliadači. Java sa postupne presunula od webov k aplikáciám pre stolné počítače a mobilné zariadenia.

Edície platformy Java:

- JavaCard – programovanie drobných appletov pre chytré karty;
- Java ME (Micro Edition) – programovanie spotrebičov (TV, PDA, tlačiarne a pod.) a programovanie aplikácii pre „hlúpe“ mobilné telefóny;
- Java SE (Standard Edition) – programovanie aplikácií pre stolné počítače;
- Java EE (Enterprise Edition) – programovanie systémov pre rozsiahle podnikové nasadenie.[14]

Java je objektovo orientovaný programovací jazyk. Objektovo orientované programy sú usporiadané okolo dát. Na týchto dátach definujeme funkcie, tzv. rutiny, ktoré s týmito dátami môžu pracovať. Dátový typ nám definuje presne to, aké druhy operácií môže-

14 ORACLE, [cit. 2017-03-31]. Dostupné na internete: http://www.oracle.com/technetwork/java/index.html.

me na dané dáta aplikovať. Základné vlastnosti objektovo orientovaných jazykov sú zapuzdrenie, polymorfizmus a dedičnosť.

Zapuzdrenie nám umožňuje v Jave zväzovať dáta a kód. Existujú mechanizmy, pomocou ktorých vieme dosiahnuť, aby dáta videla iba daná trieda alebo daná skupina tried a pre ostatné by boli neviditeľné. Trieda môže zvonku vystupovať ako skupina funkcií, ktoré nám nesprístupňujú samotné surové dáta. Keď sú dáta a kód takýmto spôsobom prepojené, hovoríme, že nám vznikol objekt.

Polymorfizmus nám hovorí o možnosti použitia jedného rozhrania na viacero všeobecných tried. Môžeme si to povedať na veľmi jednoduchom príklade niektorej z vstavaných tried v Jave. Napríklad zásobník. Môže obsahovať viacero druhov objektov. Či sú to už celé čísla, čísla s desatinnou čiarkou alebo reťazce znakov, pre všetky máme jedno rozhranie na vyberanie a vkladanie prvkov bez ohľadu na to, akého dátového typu sú. Polymorfizmus nám výrazne uľahčuje používanie programovacích jazykov, keďže jedno rozhranie podporuje mnoho dátových typov a nemusíme sa pre každý dátový typ učiť používať, alebo nutne programovať iné rozhranie.

Dedičnosť je veľmi užitočná vlastnosť, ktorá umožňuje jednému objektu získavať a pozmeňovať vlastnosti iného. Pre pochopenie si môžeme popísať objekt bod. Ten má nejakú pozíciu a farbu. Berieme ho ako základný rodičovský objekt. Z neho vytvoríme objekt kruh, ktorý dedí jeho vlastnosti, ktorými sú pozícia a farba, ale rozširujú ho o tvar a namiesto bodu nakreslíme kruh. Tým pádom nemusia všetky podobjekty definovať všetky vlastnosti pôvodného objektu, ale len tie, ktoré sú nové alebo zmenené.[15]

Preťažovanie metód je jednou z najlepších metód používaných v Jave. V princípe to znamená, že si môžeme zadefinovať niekoľko metód s rovnakým názvom, ale s inými vstupnými parametrami. V tomto prípade hovoríme, že sú preťažené. Takýmto spôsobom sa v Jave implementuje polymorfizmus. Pri definícii sa musí líšiť typ alebo počet parametrov, nestačí, aby sa líšili len v návratových typoch. O spracovanie sa už postará kompilátor.[16] Príklad je znázornený nasledujúcim zdrojovým kódom.

```
public static final double parseDoubleFromString(String str, double def){
   ...
}
```

15 Schild Herbert. 2012. Java 7. Brno: Computer Press, a.s., s. 29 – 30
16 Schild Herbert. 2012. Java 7. Brno: Computer Press, a.s., s. 222 – 223

```
public static final double parseDoubleFromString(String str){
    return parseDoubleFromString(str, 0);
}
```

Generické typy sú ďalšou veľmi využiteľnou súčasťou Javy. Generické typy, môžeme popísať ako parametrizované typy. Umožňujú nám vytvárať rozhrania a metódy, ktorých typ nie je pevne daný, ale je daný vstupným parametrom. Generické typy nám umožňujú používať kód opakovane pre viacero typov bez toho, aby sme pre každý typ museli programovať nové funkcie.[17] Ako príklad uvádzame nasledujúcu funkciu, kde je použitý generický typ <T>.

```
public void add(T item, int priority){
    Queue<T> subStack = multiStack.get(priority);
    if(subStack == null){
        subStack = new LinkedList<T>();
        multiStack.put(priority, subStack);
    }
    subStack.add(item);
}
```

Pre ukladanie stavu aplikácie je v Jave prítomná možnosť serializácie. Serializácia objektu znamená prevedenie objektu na byte stream takým spôsobom, že tento byte stream môže byť spätne prevedený na Javový objekt. Týmto spôsobom môžeme stav aplikácie ukladať aj do súboru.[18]

Java obsahuje vstavané primitívne typy boolean, byte, char, double, float, int, long, short a polia. Týmto typom sa nebudeme nejak zvlášť venovať. Skôr sa zameriame na zložitejšie dátové štruktúry, alebo kolekcie, ktoré nám jednoznačne uľahčia programovanie aplikácie.

Prvou z týchto štruktúr je ArrayList. ArrayList môžeme v jednoduchosti chápať ako pole s dynamickou veľkosťou. Pridávanie elementov sa realizuje pomocou volania metód, nie ako pri klasickom poli. Veľkosť ArrayListu sa počas pridávania a odstraňovania prvkov dynamicky prispôsobuje ich počtu. ArrayList narozdiel od poľa nemôže obsahovať primitívne typy, akými je napríklad int. Pre nás to v podstate znamená pri definovaní ArrayListu namiesto primitívneho typu definovať jeho objektový ekvivalent. Napr. na miesto int zadefinujeme jeho objektový ekvivalent Integer, do ktorého môžeme bez problémov ukladať premenné typu int. Ale aj tak pravdepodobne využijeme ArrayList na ukladanie zložitej-

17 Schild Herbert. 2012. Java 7. Brno: Computer Press, a.s., s. 476
18 ORACLE, [cit. 2017-03-31]. Dostupné na internete:https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html.

ších údajov a dátových štruktúr. Prístup k dátam prebieha pri ArrayListe podobne ako pri poli cez číselný index. Čo sa týka výkonu, pole je pri týchto operáciach rýchlejšie.

HashMap nám narozdiel od ArrayListu poskytuje prístup k dátam pomocou vopred definovaného indexu. Index môžeme definovať ako Integer, pri tom budeme mať veľmi podobnú štruktúru ako pri ArrayListe, ale napríklad môžeme definovať kľúče pomocou Stringu, čo nám ArrayList neumožňoval. Narozdiel od ArrayListu tu nemôžeme využívať duplicitné kľúče.

V neposlednom rade si spomenieme štruktúru LinkedHashMap. Pri klasickej Štruktúre HashMap sa nám poradie prvkov mení počas každého prístupu do HashMap. V LinkedHashMap je zabezpečené to, že poradie prvkov sa nám od ich vloženia nemení, a teda vieme, v akom poradí boli vkladané. Rýchlosť HashMap a LinkedHashMap je rovnaká, avšak kvôli linkovaniu prvkov LinkedHashMap spotrebováva viac pamäte.

TreeMap je ďalším typom, ktorý je podobný predchádzajúcim, no na rozdiel od LinkedHasmap usporadúva indexy v poradí od najmenšieho.

Queue je v preklade do slovenčiny rad. To nám napovedá budúce použitie tejto dátovej štruktúry. Queue je vlastne interface, ktoré dokážeme aplikovať napríklad aj na LinkedList a pod. V princípe ide o to, že nám poskytuje funkcie optimalizované na prácu s radom. Teda primárne využijeme pridanie objektu na koniec radu a odobranie objektu zo začiatku radu.

V simulácii budeme mať použitý extrémne zrýchlený čas, počas ktorého budeme chcieť simulovať pohyb a grafické objekty. Pri klasických dátových typoch spomenutých vyššie, ale môže nastať problém, že počas simulácie budú v jednom čase do týchto objektov pristupovať viaceré algoritmy, a to spôsobí chyby, prípadne pád celej aplikácie. Tu existujú dva pre nás pravdepodobne dôležité typy, ktoré tento problém eliminujú. Sú to ConcurrentSkipListMap a ConcurentHashMap. Zjednodušene môžeme povedať, že ConcurentHashMap je obdobou HashMap a ConcurrentSkipListMap je obdobou LinkedHashMap a oba sú zabezpečené proti chybám pri viacnásobnom prístupe do tohto dátového typu.

1.4.5 Výber vývojového prostredia (IDE)

Výber správneho IDE patrí medzi nezanedbateľné úlohy každého vývojára. Prog-

ram vieme napísať aj v bežnom textovom editore a kompilovať ho cez konzolu, to by však nebolo veľmi efektívne. Textový editor nám totiž neumožňuje debugging a refactoring. V pokročilom IDE máme množstvo nástrojov pre urýchlenie programovania, ako napríklad automatické dopĺňanie kódu, vyznačovanie syntaxe aj syntaktických chýb a pokročilé krokovanie kódu. Na porovnanie sme vybrali tri najpoužívanejšie IDE – Eclipse, NetBeans a IntelliJ IDEA.[19] [20]

Eclipse

V roku 2001 vydalo IBM Eclipse ako Open Source projekt. Vtedy to bolo úplne jednoduché IDE na editáciu zdrojových kódov písaných v Jave. Neskôr sa do neho pridala podpora iných jazykov ako napríklad Go, Scala, a pod.

Eclipse je technicky zložený z mnohých pluginov, čo z neho vytvára mimoriadne prispôsobivý nástroj na programovanie. Pre človeka, ktorý má radšej hotové funkčné riešenie, nie je až taký vhodný. Môžno konštatovať, že Eclipse nie je iba IDE, ale celá technologická platforma.

NetBeans

NetBeans bol vyvíjaný v tieni Eclipse ako čisté IDE (Integrated Development Environment – Integrované Vývojové Rozhranie). Existuje 5 rôznych verzií na stiahnutie, a to od najmenšej verzie iba s podporou jazyka C++, až po verziu podporujúcu veľké množstvo používaných jazykov. NetBeans bol vždy braný ako menší a slabší brat po boku Eclipse, no v posledných rokoch ho začal dobiehať. Pohľad na NetBeans je veľmi podobný ako na Eclipse. Oba majú dobré rozhranie s potrebou inštalovania mnohých pluginov.

IntelliJ IDEA

Idea je veľmi šikovné IDE, ktoré funguje pod všetkými najpoužívanejšími operačnými systémami určenými pre stolné počítače. Taktiež pre ňu existuje mnoho pluginov a rozšírení. Má veľmi prepracovaný mód pre refactoring kódu. Idea narozdiel od Eclipse a Netbeans indexuje celý projekt, na ktorom aktuálne pracujete, čo je skvelé pre automatické dopĺňanie kódu, návrhy pre opravu rôznych chýb, či návrhy pre zjednodušenie kódu.

IDEA je poskytovaná v dvoch edíciách, a to v Open Source edícii, ako aj v platenej

19 BOLTON, David.Three Java IDEs Compared, [cit. 2017-03-28]. Dostupné na internete: http://insights.dice.com/2013/10/24/three-java-ides-compared-147/.

20 VLATKO, Natali. An overview of the top Java IDEs, [cit. 2017-03-28]. Dostupné na internete: https://jaxenter.com/the-top-java-ides-114599.html.

uzavretej edícii s podporou mnohých programovacích jazykov a pokročilých funkcií. Narozdiel od predchádzajúcich dvoch IDE, sú v základnej bezplatnej verzii prítomné všetky potrebné funkcie pre vývoj Java projektu (bez nutnosti inštalovania pluginov).

Vyhodnotenie

Na základe porovnania môžeme konštatovať, že nám všetky 3 prostredia vyhovujú podobne. Z dôvodu poskytnutia komplexného riešenia aj bez potreby inštalovania rôznych pluginov, sme si nakoniec vybrali IntelliJ IDEA.

	IntelliJ IDEA	Eclipse	NetBeans
Indexovanie celého projektu	Áno	Prostredníctvom pluginu	Prostredníctvom pluginu
Pluginy	Áno	Áno	Áno
Multiplatformovosť	Áno	Áno	Áno
Podpora Java SE	Áno	Áno	Áno
Edícia	Bezplatná	Bezplatná	Bezplatná / Enterprise

Tabuľka 3: Porovnanie IDE s podporou Javy

1.4.6 Správa zdrojových kódov a verziovanie

K vývoju aplikácií neodmysliteľne patrí aj správa, verziovanie a zálohovanie zdrojových kódov. Správa zdrojových kódov bez verziovacieho systému je veľmi nepraktická a neprehľadná. Na správu zdrojových kódov sa najčastejšie používajú dva systémy Subversion (SVN) a GIT.

Subversion (SVN)

SVN je centralizovaný systém slúžiaci na správu verzií. To znamená, že všetci členovia tímu pracujú na jednom centrálnom repozitári bežiacom na serveri a zo servera si stiahnu „pracovnú kópiu“ celého projektu (snímka určitej verzie aplikácie).

Úložisko SVN je organizované do niekoľkých adresárov. Je to adresár pre hlavnú líniu vývoja, vetvy alternatívnych línií a značky alebo tagy, ktoré označujú jednotlivé revízie projektu.

GIT

Git je na rozdiel od SVN distribuovaný systém. Namiesto kopírovania snímky aplikácie na počítač klienta sa klonuje celý repozitár zo servera. Tým pádom má programátor na svojom PC plnohodnotný klon s celou históriou zmien a všetkými vetvami. S touto kó-

piou môže pracovať offline na svojom počítači, bez potreby pripájať sa k serveru.

Git sa skladá z jednej jedinej zložky „.git“, v ktorej sa nachádza celý repozitár. Na rôzne vetvy tu nie sú použité samostatné priečinky, ale len jeden základný priečinok aplikácie a pri ňom samotný git repozitár v priečinku „.git“. Adresár aplikácie teda obsahuje iba aktuálne aktívnu vetvu aplikácie.[21]

Výber

V tomto prípade poskytuje SVN aj GIT veľmi podobnú funkcionalitu, predovšetkým, ak aplikáciu programuje iba jeden programátor, ako je to aj v našom prípade. GIT má lepšiu správu vetiev, jednoduchšiu štruktúru a je použiteľný aj bez pripojenia k internetu.

21 Fournova software, [cit. 2017-03-28]. Dostupné na internete: https://www.git-tower.com/learn/git/ebook/en/desktop-gui/appendix/from-subversion-to-git.

2 Implementácia

Jednou z najťažších úloh pri vývoji aplikácií je samotný návrh aplikácie. Implementácia našej aplikácie pozostáva z niekoľkých krokov. V prvom kroku je nutné zamyslieť sa nad ekonomickým procesom z pohľadu programátora. Ďalším krokom je teoretický výber reprezentácie týchto objektov a nakoniec uplatnenie tohto výberu a samotné programovanie aplikácie.

Pri týchto krokoch sa budeme zameriavať aj na prehľadnosť kódu a možnosť budúceho rozšírenia a úpravy aplikácie pre ďalšie špecifické účely. Zdrojový kód bude otvorený a voľne dostupný každému, komu by mohol pomôcť pri vytváraní vlastného ekonomického simulátora.

2.1.1 Základné objekty

Na začiatku je potrebné stanoviť, aké objekty budú v našej simulácii vystupovať. Tu nemáme na mysli objekty ako programátorské triedy, ale ako reálne objekty, od ktorých sa odrazíme a neskôr navrhneme triedy, ktoré budeme programovať.

Každý ekonomický proces má svojich aktérov. Aktéri vstupujú do procesu cez vstupnú bránu a pri splnení určitých podmienok proces opúšťajú. Každý aktér nesie v sebe nejaké jednotky, či už je to obnos peňazí, množstvo paliva, či čas, ktorý má k dispozícii. Takýto aktér prechádza bodmi, v ktorých tieto jednotky spotrebováva alebo prijíma. Všetci aktéri sú závislí na čase.

Tieto jednotky si teraz popíšeme z programátorského hľadiska.

Čas

Základnou jednotkou simulácie je čas. Každá akcia je vykonávaná v konkrétnom čase a je možné ich usporiadať na časovú os. Každý aktér je závislý okrem iného na čase, takže bude nutné naprogramovať objekt, pomocou ktorého budeme schopní merať a stanoviť čas v každom momente simulácie.

Aktér (crawler)

Aktér je základným účastníkom simulácie, ktorý v sebe nesie informácie o rôznych jednotkách a je závislý na čase. Prechádza procesom určitou rýchlosťou a v určitom momente môže simuláciu opustiť. Tak isto simuláciu opúšťa, ak minie niektorú zo svojich

jednotiek, či už je to čas alebo peniaze. Každý aktér bude môcť mať inú prioritu podľa ktorej bude spracovávaný.

Trasa pohybu

Trasa a smer pohybu určujú, odkiaľ kam sa môžu alebo musia pohybovať aktéri simulácie.

Vstupná brána (start point)

Vstupné brány budú body, v ktorých budú do simulácie vstupovať aktéri simulácie. Môžu vstupovať v určitom čase a v určitom počte, buď po jednom alebo naraz v dávkach.

Miesto spracovania (processing point / end point)

Miesto spracovania je bod v simulácii, do ktorého vstupujú aktéri. Miesto spracovania má danú kapacitu, po ktorej naplnení aktéri čakajú na vstup v zásobníku daného bodu. Vo vnútri sú spracovávaní a môžu míňať svoje priradené jednotky. Po ich minutí opúšťajú simuláciu. Ak z daného miesta simulácie nevedie žiadna trasa do iného bodu, v tom momente opúšťajú simuláciu. Nastavenia miesta spracovania budú dynamické a toto miesto spracovania bude možné použiť ako zásobník.

2.1.2 Používateľské rozhranie aplikácie

V tejto kapitole sa zaoberáme predstavou, ako by malo vyzerať používateľské rozhranie našej aplikácie.

Používateľské rozhranie aplikácie je rozhodujúce pre jednoduché ovládanie. Dizajn je potrebné navrhnúť tak, aby sa používateľ nemusel zdĺhavo zaoberať samotnou aplikáciou a hľadaním jej ovládacích prvkov, ale aby bol od prvého momentu zameraný na jej používanie a tvorbu simulácie.

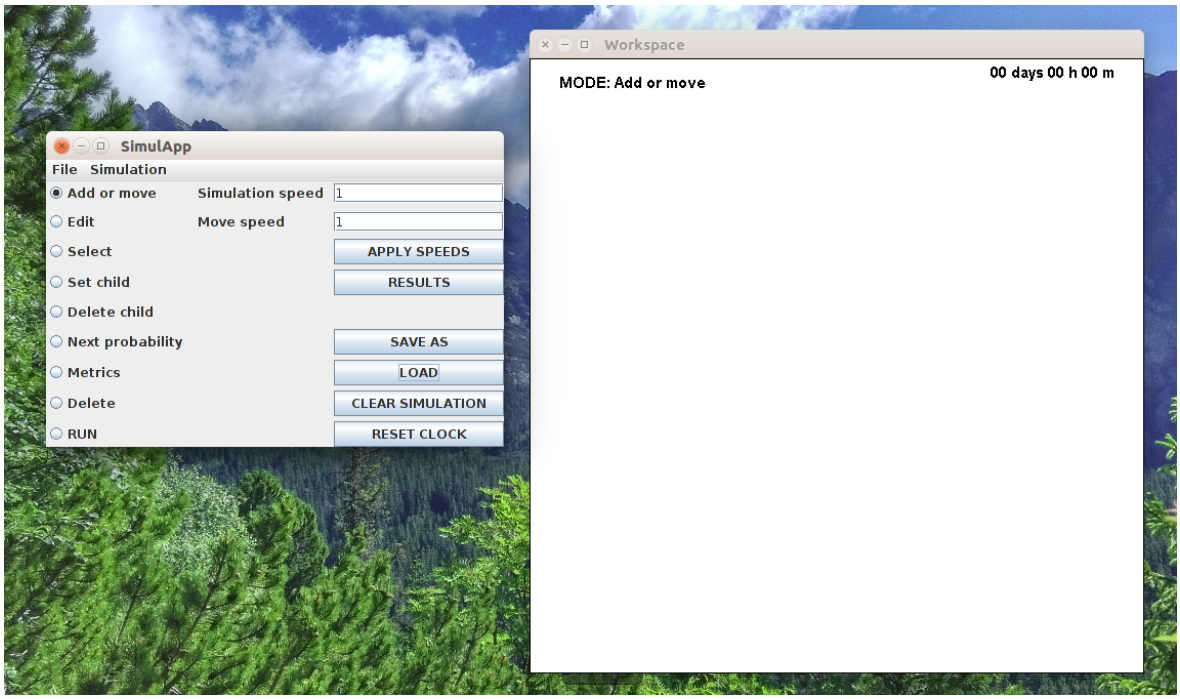
Budeme potrebovať menu pre nastavenie aktuálneho módu a samotnú pracovnú plochu, na ktorej budeme modelovať podnikový proces.

Aplikácia bude disponovať niekoľkými základnými módmi:

- vkladanie objektu;
- presúvanie objektu;
- editácia objektu;

- vytvorenie cesty medzi objektami;
- zrušenie cesty medzi objektami;
- mazanie objektov;
- nastavenie rýchlosti simulácie a pohybu;
- zobrazenie a vizualizácia nameraných výsledkov;
- spustenie simulácie.

Zvolili sme dvojoknové rozloženie, pričom prvé okno bude slúžiť na prepínanie aktuálneho módu a v druhom okne budeme modelovať a sledovať spustenú simuláciu. Keďže v simulácii bude viac objektov, výsledky pre každý objekt budeme prezentovať v samostatných oknách.



Obrázok 7: Dvojoknové rozloženie aplikácie

Okno pre výber módu aplikácie

Okno pre výber módu aplikácie bude zároveň hlavným oknom. Bude to jediné okno, po ktorého zavretí sa ukončí celá aplikácia. Výber módu bude realizovaný pomocou navzájom previazaných radiobuttonov, z ktorých bude môcť byť aktívny vždy iba jeden.

Pracovná plocha

Pracovná plocha bude v samostatnom okne, ktoré sa bude dať maximalizovať a minimalizovať, ale nie zatvárať. Vkladanie objektov bude prebiehať klikaním myšou do plátna, pričom musí byť aktívny mód pridávania objektu. Pri kliknutí myšou zistíme presné súradnice kliknutia, vygenerujeme okno s výberom objektu a priradením jeho parametrov. Po zadaní parametrov objektu v bode kliknutia vytvoríme požadovaný objekt. Objekty sa budú dať presúvať a upravovať aj po pridaní (mód editácie a mód presúvania objektov).

Výsledky simulácie

Okno výsledkov simulácie budeme vyvolávať po kliknutí na tlačidlo „Results“ v hlavnom okne pre výber módu aplikácie. Výsledky aplikácie budú prezentované ako tabuľka so základnými údajmi, nazbieranými počas behu simulácie s možnosťou zobrazenia detailov pre každý objekt.

2.2 Programová reprezentácia základných objektov

Pred samotným programovaním je nutné zamyslieť sa nad nasledujúcimi problémami:

- akého správcu okien využiť pri programovaní rozhrania;
- ako správne reprezentovať simuláciu z dátového hľadiska;
- akým spôsobom zaznamenávať a neskôr vhodne vizualizovať namerané údaje simulácie.

2.2.1 Správa okien a vizualizácia objektov

V našej aplikácii je nutné programátorsky zabezpečiť zobrazovanie samotných okien a objektov v nich. Pre vizualizáciu môžeme využiť knižnicu AWT (Abstract Window Toolkit) alebo modernejšiu knižnicu Swing.

Knižnica AWT využíva priamo komponenty platformy, na ktorej aplikácia beží a nie je čisto definovaná Javou. Knižnice AWT používajú vstavené kódové prostriedky platformy, na ktorej aplikácia beží, a preto sa označuje ako ťažkotonážna a veľmi sa nepoužíva.

Knižnica Swing je zoskupením tried a rozhraní, ktoré priamo v Jave definujú to, ako bude aplikácia vyzerať. Je nezávislá na platforme, a preto je pre nás vhodnejším kandi-

dátom na programovanie rozhrania. Dovoľuje nám vytvárať samostatné okná, formuláre a umožňuje nám vykresľovať a animovať rôzne grafické objekty, ktoré vyzerajú a správajú sa na každej platforme rovnako. [22]

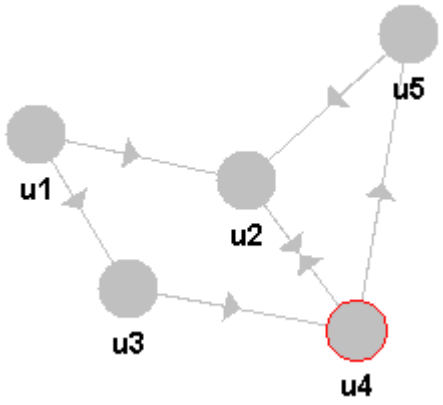
2.2.2 Dátová štruktúra simulácie

Ďalšou dôležitou otázkou pri programovaní aplikácie bude, ako reprezentovať prepojenie objektov v simulácii. Je dôležité vedieť určiť, medzi ktorými objektami simulácie sa môžu pohybovať aktéri simulácie. Nie je podstatné len to, aby boli objekty prepojené, ale aj to, aby sme vedeli určiť smer pohybu medzi objektami, a to nie len jednosmerne, ale dať možnosť objektom pohybovať sa aj obojsmerne. Najviac nám v tomto vyhovuje dátová štruktúra orientovaný graf.

Dátová štruktúra graf

Graf je dátová štruktúra zložená z nasledujúcich dvoch komponentov:

- konečná množina uzlov – vrcholy grafu (u_1, u_2, … u_n);
- konečná množina usporiadaných dvojíc vrcholov – hrany grafu ((u_1, u_2), (u_{n-1}, un), (u_x, u_y)), pričom pri orientovanom grafe je dôležité poradie hrán. To nám určuje smer prechádzania grafu.[23]



Obrázok 8: Grafické znázornenie orientovaného grafu

22 Schild Herbert. 2012. Java 7. Brno: Computer Press, a.s., s. 7

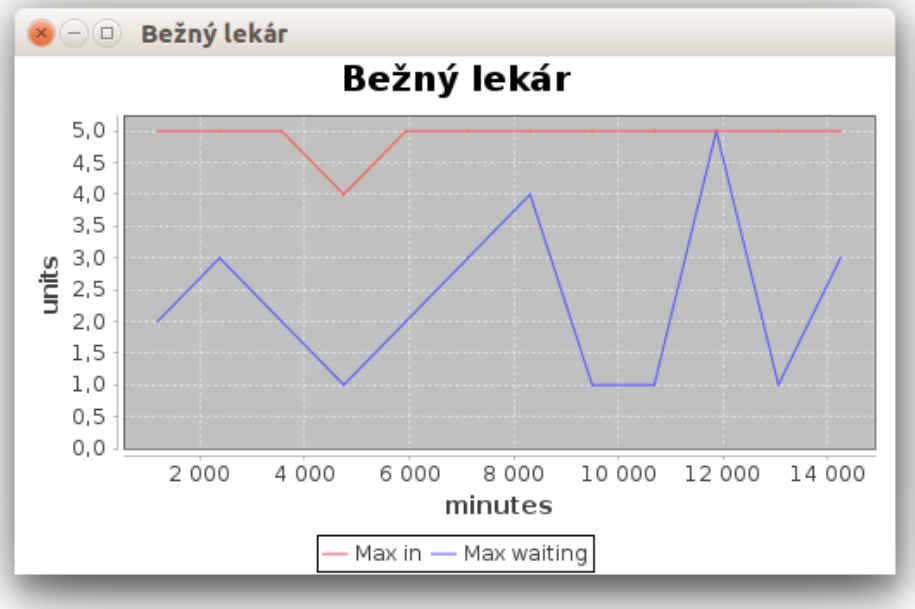
23 Geekforgeeks, [cit. 2017-03-28]. Dostupné na internete: http://www.geeksforgeeks.org/graph-and-its-representations/.

2.2.3 Meranie hodnôt a ich vizualizácia

V simulácii je jednou z nezanedbateľných úloh zaznamenávanie hodnôt závislých na čase. V prvom rade treba nejakým spôsobom reprezentovať simulovaný čas, ktorý bude priamo úmerný reálnemu času, teda bude jeho násobkom. V aplikácii bude pre tento účel prítomný timer (časovač), ktorý bude tikať presne určenou rýchlosťou a pri každom tiknutí timera budeme zaznamenávať náš virtuálny čas.

V každom časovom intervale simulácie budú vystupovať aktéri. Nás bude zaujímať, kedy vstúpia do simulácie, aké parametre majú nastavené pri vstupe, kedy vstúpia do jednotlivých hrán grafu, aký vplyv bude mať vstup a výstup do hrany na ich parametre a kedy zo simulácie vystúpia. Na to použijeme vlastnú dátovú štruktúru, ktorú bližšie popíšeme v kapitole venujúcej sa programovaniu aplikácie.

Počas behu simulácie, ale aj po jej ukončení nás budú zaujímať namerané hodnoty. Na to vytvoríme tabuľkové rozhranie, v ktorom budeme zobrazovať priemerné namerané hodnoty pre každú vstupnú bránu a pre každé miesto spracovania. Po kliknutí na konkrétny element budeme mať k dispozícii vygenerovanie grafu a v ňom zobrazenie hodnôt vzhľadom na čas.



Obrázok 9: Vygenerovaný graf (jednotky závislé na čase)

2.3 Programovanie

V tejto časti pristúpime k reálnemu programovaniu a premieňaniu vyššie spomenutej teórie na reálny zdrojový kód v Jave.

2.3.1 Java balíčky (packages)

Pri programovaní je užitočné zoskupiť súvisiace časti kódu dohromady. V Jave na tento účel slúžia balíčky (packages). Balíček slúži ako prostriedok na pomenovanie kolekcie tried. Tak isto je dobrým prostriedkom na zapuzdrovanie tried. Okrem toho slúži ako mechanizmus prístupu k danej kolekcii tried. Pre malé aplikácie stačí len jeden základný balíček, avšak pre skutočné veľké aplikácie by to bolo veľmi neprehľadné. Preto sme aj mi pristúpili k vytvoreniu viacerých balíčkov v rámci našej aplikácie.[24]

Hierarchický prehľad balíčkov aplikácie:

- **simulApp** – základný balíček s celou našou aplikáciou;
  - **main** – bude obsahovať hlavnú triedu a správu času v aplikácii;
  - **graphics** – zoskupenie základných grafických elementov v aplikácii;
    - **canvas** – plocha, na ktorú budeme vykresľovať;
    - **canvasObjects** – objekty, ktoré sa starajú o vykresľovanie;
      - **members** – základné tvary pre vykreslenie;
      - **manipulation** – triedy, ktoré budú mať na starosti manipuláciu so základnými objektami;
  - **EconSimul** – balíček, ktorý zoskupuje triedy pre ekonomickú simuláciu;
    - **members** – obsahuje triedy pre vykresľovanie ekonomických objektov (dedí vlastnosti z balíčka graphics);
    - **manipulation** – triedy, ktoré budú mať na starosti manipuláciu so základnými objektami (dedí vlastnosti z balíčka graphics);
    - **forms** – okná a formuláre potrebné pre ovládanie a získavanie výsledkov ekonomickej simulácie;

24 Schild Herbert. 2012. Java 7. Brno: Computer Press, a.s., s. 7

- **graphing** – bude mať za úlohu vykresľovanie grafov s výsledkami simulácie;
- **savers** – ich úlohou bude serializácia a deserializácia aplikácie a ukladanie stavu aplikácie do súboru;

- **metrics** – triedy, ktoré nám budú slúžiť na rôzne merania v rámci simulácie;
- **util** – rôzne pomocné triedy, ktoré nám uľahčia programovanie simulácie;

2.3.2 Základné balíčky simulácie

Do tejto skupiny balíčkov zaradíme hlavný balíček main, ktorý bude dôležitý pre samotný beh aplikácie. Ďalej sem zaradíme balíčky util a metrics. O každom si napíšeme detailnejšie ďalej v texte.

Balíček main

Balíček main bude obsahovať tie najzákladnejšie triedy aplikácie. Bude spravovať zobrazenie základného rozhrania, spúšťať timer na meranie času a operácie závislé na čase. Okrem toho bude prepájať všetky triedy na nižšej úrovni. Tento balíček je zložený z dvoch základných tried, a to simulApp a tick. Obe sú závislé na triede Run.

Trieda Run sa síce nebude nachádzať v balíčku main, ale bude pre beh aplikácie najdôležitejšia. Bude spúšťať celú aplikáciu a preväzovať funkcionalitu jednotlivých tried v balíčku main. V prvom rade vytvorí inštanciu simulApp a inštanciu objektu tick.

SimpulApp bude základnou triedou aplikácie, z ktorej sa po spustení vytvorí jedna inštancia. V prvom rade sa postará o vykreslenie formulára s hlavnými ovládacími prvkami aplikácie, mainGUI, a potom spustí manipulátor, ktorý bude mať na starosti manipuláciu s grafickými objektami. Oboma spomenutými triedami sa budeme zaoberať ďalej.

V Jave je prítomný timer, pomocou ktorého dokážeme spúšťať zvolený kód v striktne daných časových intervaloch. Za pomoci tohto timera vytvoríme triedu tick. Timer funguje teda ako presné hodiny. To môže mať za následok, že máme v jednom čase spustených viac častí kódu súčasne. Ak si programátor na túto možnosť nedá pozor, môže to vyústiť až do pádu aplikácie. Aby sme sa tomu vyvarovali,vytvorili sme si vlastný timer, rozširujúci pôvodný TimerTask. Ten náš však zabezpečuje to, že už nehľadíme striktne na to, aby bol každý interval rovnako dlhý, ale aby sa vždy vykonal všetok kód a až potom sa

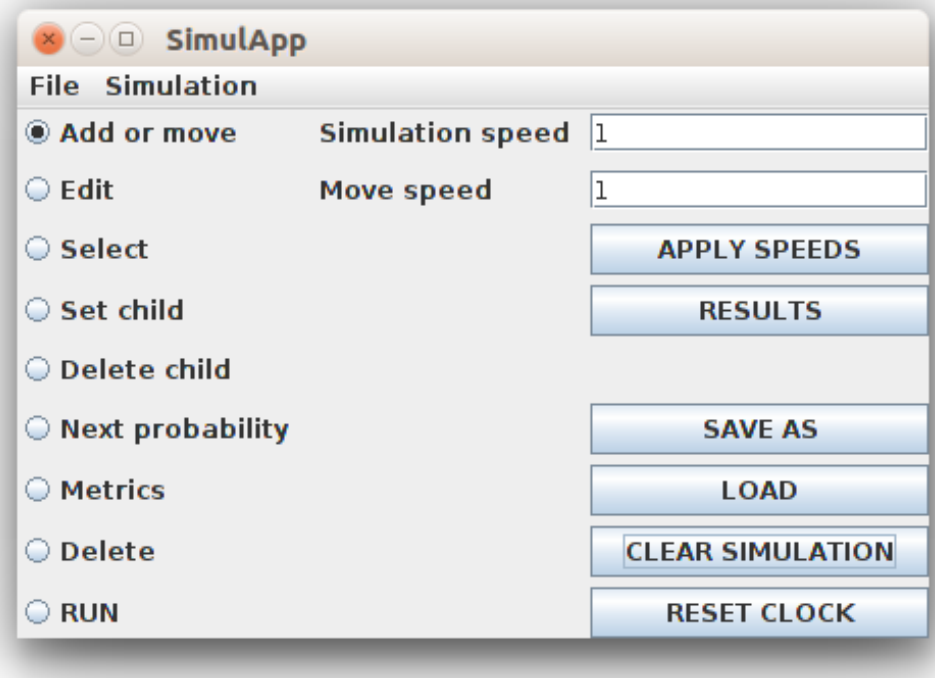
spustil ďalší interval. Nás nebude zaujímať presné meranie reálneho času, ale presné meranie virtuálneho času simulácie. To sme dosiahli nasledujúcim kódom.

```
public class tick extends TimerTask {
    private simulApp app;
    protected boolean canTick;
    public tick(simulApp app){
        this.app = app;
        canTick = true;
    }
    public void run() {
        if(canTick) {
            canTick = false;
            app.tick();
        }
        canTick = true;
    }
}
```

Poslednou triedou v tomto balíčku bude mainGUI. MainGUI bude okno s formulárom na výber módu aplikácie so základnými ovládacími prvkami. Okrem zobrazenia a obsluhy ovládacích prvkov bude mať táto trieda na starosti predávanie aktuálne nastaveného módu aplikácie manipulátora, ktorý sa postará o manipuláciu s grafickými objektami.

Hlavné módy budú reprezentované skupinou radiobuttonov. Pre zistenie aktuálneho módu aplikácie tu bude prítomná jedna verejná metóda getCurrentMode, ktorá bude vracať mód vo forme reťazca znakov. Jednotlivé módy budú definované ako konštanty v manipulátore.

Cez tento formulár budeme schopní ovplyvňovať rýchlosť pohybu aktérov simulácie, ale aj rýchlosť celej simulácie. Pre tento účel tu budeme mať prítomné dve textové polia.



Obrázok 10: Formulár mainGUI

Balíček util

Tento balíček bude zoskupením pomocných tried, ktoré môžeme potrebovať kdekoľvek v aplikácii. Meno má odvodené od anglického slova utilities, čo v preklade znamená pomôcky. Bude obsahovať osem tried, z ktorých si šesť rozoberieme podrobnejšie.

Trieda geometry

Prvou z tried v tomto balíčku bude trieda geometry. V tejto triede budú zoskupené geometrické funkcie. Poslúžia nám buď na geometrické výpočty alebo na vykresľovanie geometrických objektov, ktoré Java neobsahuje.

Pri kreslení ciest a ich smeru v simulácii bude potrebné vizualizovať tento smer. V Jave neexistuje žiadna jednoduchá funkcia, ktorá dokáže nakresliť šípku na základe niekoľkých základných parametrov, preto si jednu takúto funkciu budeme musieť naprogramovať. Pomenujeme ju drawArrow a fungovať bude veľmi podobne ako kreslenie oby-

čajnej čiary. No čiara bude smerovo orientovaná, a tým pádom sa šípka vykreslí v smere z prvého zadaného bodu do druhého.

Ďalšou pomocnou funkciou bude funkcia midPoint. Je veľmi pravdepodobné, že budeme musieť často vypočítať stred dvoch bodov a na zjednodušenie počítania tohto bodu nám poslúži práve táto funkcia.

```
public static final Point midPoint(Point p1, Point p2){
    return new Point(((p1.x + p2.x) / 2), ((p1.y + p2.y) / 2));
}
```

Väčšina objektov nebude zobrazovaná ako jednoduchý bod, ale budú skôr reprezentované nejakým tvarom ako napríklad štvorec, obdĺžnik alebo elipsa. Takéto tvary sa v Jave bežne vykresľujú pomocou funkcie, ktorej sa ako vstupný parameter zadá bod, od ktorého sa má daný objekt vykresľovať a následne výška a šírka objektu. No my ich chceme mať nakreslené so stredom v mieste kliknutia myši. Táto funkcia nám pri zadaní bodu kliknutia, výšky a šírky vykresľovaného objektu zabezpečí vrátenie pozície, z ktorej sa má daný objekt vykresľovať. Nazveme ju getStartXY.

```
public static final Point getStartXY(int x, int y, int w, int h){
    return new Point((x - (w / 2)), (y - (h / 2)));
}
```

Pri pohybe aktéra simulácie budeme musieť vyriešiť jeho pohyb po priamke. Naprogramujeme funkciu s názvom getNextPos, ktorá nám s tým pomôže. Ako vstup budú použité dva body (aktuálna pozícia a konečná pozícia, do ktorej sa chceme dopracovať) a rýchlosť pohybu. Táto funkcia nám vráti vypočítaný nasledujúci bod pohybu aktéra.

```
public static final Point getNextPos(int x1, int y1, int x2, int y2, int
speed){
    int i = 0;
    int resX = x1;
    int resY = y1;
    while(i < speed) {
        int deltaX = x2 - x1;
        int deltaY = y2 - y1;
        double distance = Math.sqrt((deltaX * deltaX) + (deltaY * del-
taY));
        double velX = (deltaX / distance) * 5;
        double velY = (deltaY / distance) * 5;
        resX = x1 + (int) velX;
        resY = y1 + (int) velY;
        if(resX == x2 && resY == y2){
            break;
        }
```

```
            x1 = resX;
            y1 = resY;
            i++;
        }
        return new Point(resX, resY);
}
```

Trieda math

V tejto triede budeme mať zoskupené pomocné matematické alebo číselné metódy, ktorými samotná Java nedisponuje, alebo sú v inej podobe, ako to my vyžadujeme. Bude to pre nás užitočné napríklad vtedy, keď budeme programovať distribúciu aktérov z jedného bodu do viacerých bodov. Pri tom budeme potrebovať generovať náhodné čísla pomocou rôznych funkcií distribúcie.

V prvom rade sa zameriame na vhodné riešenie funkcií pre generovanie náhodných čísel. Budú to funkcie randomUniform, randomNormal a randomExponential. Tieto funkcie vygenerujú náhodné číslo na základe uniformnej, normálnej a exponenciálnej distribúcie.

Pri parsovaní čísel z textu využijeme túto pomocnú parseDobleFromString a parseIntFromString, ktoré nám zabezpečia, aby sa nám vrátil iba očakávaný číselný formát. Na vstupe sú dva parametre, prvý je stringový reťazec znakov a ten druhý je hodnota, ktorú funkcia vráti, ak sa jej nepodarí vyparsovať z textu čísla. ParseIntFromString nám vždy vráti integerovské, teda celé čislo a parseDoubleFromString nám môže vrátiť aj číslo s desatinnou čiarkou.

```
public static final double parseDoubleFromString(String str, double def){
    double res;
    try{
        res = Double.parseDouble(str);
    }catch(NumberFormatException e){
       res = def;
    }
    if(res > 999999){
        res = 999999;
    }
    return res;
}
```

Trieda PriorityStack

Java disponuje základnou dátovou štruktúrou zásobník. Pri simulácii nám pravdepodobne nemusí stačiť tento klasický zásobník. V niektorých situáciách budeme chcieť rôznym prvkom v zásobníku priradiť inú prioritu. Na to nám poslúži táto dátová štruktúra.

Bude to vlastne zásobník zásobníkov usporiadaných podľa priority. Najskôr sa budú vyberať prvky zo zásobníkov s vyššou prioritou a až keď sa minú tieto prvky, začneme využívať zásobníky s nižšou prioritou.

```
private TreeMap<Integer, Queue<T>> multiStack;

...

public void add(T item, int priority){
    Queue<T> subStack = multiStack.get(priority);
    if(subStack == null){
        subStack = new LinkedList<T>();
        multiStack.put(priority, subStack);
    }
    subStack.add(item);
}

...

public T poll(){
    int greathestKey = -1;
    for(Map.Entry<Integer, Queue<T>> item: multiStack.entrySet()){
        if(item.getKey() > greathestKey && item.getValue().size() > 0){
            greathestKey = item.getKey();
        }
    }
    Queue<T> tmpItem = multiStack.get(greathestKey);
    if(tmpItem == null){
        return null;
    }
    return tmpItem.poll();
}
```

Trieda timeDepender a timeDependerVal

Pri programovaní simulácie budeme potrebovať simulovať na časovo viazaných hodnotách. Napríklad otváracie hodiny, silnú a slabú prevádzku zariadenia a pod. Tým pádom budeme musieť naviazať hodnoty na časové intervaly. Každá hodnota bude reprezentovať čas od kedy platí a bude platiť do času, v ktorom bude zadaná nasledujúca hodnota, prípadne do polnoci. Budeme sa pri tom zameriavať iba na hodiny a minúty dňa. V tomto bode sa musíme zamyslieť nad tým, ako najjednoduchšie reprezentovať túto štruktúru pre čo najrýchlejšie vyhľadávanie algoritmom, ale aj pre čo najjednoduchšiu prácu s dátami pre programátora. V tomto bode sme dospeli k vytvoreniu dvoch tried, pričom jedna bude reprezentovať tieto dáta pre čo najrýchlejšie algoritmické vyhľadávanie a tá druhá bude slúžiť pre našu prácu s dátami.

Základná štruktúra timeDependera bude TreeMap zložený z ďalších TreeMap, ktoré budú niesť už samotné hodnoty. Prvý TreeMap nám bude reprezentovať hodiny a druhý minúty danej hodiny. Celý timeDepender bude niesť generický typ, tým pádom do neho budeme vedieť ukladať ľubovoľné dáta.

```
private TreeMap<Integer, TreeMap<Integer, T>> vals = new TreeMap<>();

...

public T getByTime(int fromHour, int fromMinute){
    if(fromHour > 23){
        fromHour = 23;
    }
    if(fromMinute > 59){
        fromMinute = 59;
    }
    int startHour = fromHour;
    int startMinute = fromMinute;
    boolean exist = false;
    TreeMap<Integer, T> mins = null;
    T val = null;
    for (Map.Entry<Integer, TreeMap<Integer, T>> h: vals.entrySet()){
        if(h.getKey() <= startHour && h.getValue().size() > 0){
            mins = h.getValue();
            exist = true;
        }
    }
    if(!exist){
        return null;
    }
    for (Map.Entry<Integer, T> m: mins.entrySet()){
        if(m.getKey() <= startMinute){
            val = m.getValue();
        }
    }
    return val;
}

...

public ArrayList<timeDependerVal<T>> getAll(){
    ArrayList<timeDependerVal<T>> res = new ArrayList<>();
    for (Map.Entry<Integer, TreeMap<Integer, T>> h: vals.entrySet()){
        for (Map.Entry<Integer, T> m: h.getValue().entrySet()){
            res.add(new timeDependerVal<T>(h.getKey(), m.getKey(), m.get-
Value()));
        }
    }
    return res;
}
```

Výstupom pre programátora musí byť prehľadná dátová štruktúra timeDependerVal, ktorá bude obsahovať hodinu, minútu a hodnotu k nim patriacu. Môžeme ju mať uloženú napríklad v ArrayListe, ako môžeme vidieť vo funkcii getAll v predchádzajúcom kóde. Nižšie je znázornená štruktúra timeDependerVal.

```
public class timeDependerVal<T> implements Serializable {
    private int hour;
    private int minute;
    private T value;
    public timeDependerVal(int h, int m, T v){
        hour = h;
        minute = m;
        value = v;
    }
    public int getHour(){
        return hour;
    }
    public int getMinute(){
        return minute;
    }
    public T getValue(){
        return value;
    }
    public void setHour(int h){
        hour = h;
    }
    public void setMinute(int m){
        minute = m;
    }
    public void setValue(T v){
        value = v;
    }
}
```

Balíček metrics

Meranie spotreby jednotiek bude vzhľadom na čas spracované práve prostredníctvom tried v balíčku metrics. Metrics sa bude skladať z dvoch jednotlivých tried metricUnit a triedy Measure.

Trieda metricUnit

Pomocou metricUnit budeme zaznamenávať konkrétne meranie danej jednotky. Trieda bude schopná pracovať v dvoch režimoch. Bude zaznamenávať hodnoty a ich súčet alebo priemernú hodnotu v závislosti od času. Každá jednotka bude pomenovaná unikátnym stringovým indexom.

Funkcia setAvg bude nastavovať mód merania priemeru. Pomocou parametra nastaveného touto funkciou bude trieda vedieť, že pri generovaní výsledkov treba vypočítať priemer a nie súčet zaznamenaných hodnôt za časový interval. Podobnou funkciou bude funkcia setMaxGetter. SetMaxGetter nám zas poslúži na nastavenia módu merania maximálnej hodnoty v danom čase.

Ďalšou funkciou v tejto triede bude funkcia add. Pomocou nej vložíme čas merania a nameranej hodnoty do zoznamu meraní. Neskôr si budeme vedieť vyžiadať záznam priradený k danému času, alebo záznamy z daného časového intervalu.

Pomocou funkcie get sa zas dostaneme ku hodnote z daného času. Táto funkcia jednoducho vráti hodnotu patriacu k danému času. Ak nebude k danému času priradená žiadna nameraná hodnota, vráti prázdnu hodnotu (null). Nadväzujúcou funkciou bude getTotal. Pokiaľ nie je nastavený mód merania priemeru, metóda getTotal vráti súčet všetkých hodnôt nameraných za dané obdobie. Ak je zadaný mód merania priemeru, vráti hodnotu z funkcie getTotalAvg. Pomocou funkcie getTotalAvg získame spriemerované hodnoty. Funkcia počíta s tým, že je v každom čase uložená priemerná hodnota. Celkovú hodnotu získa ako súčet všetkých hodnôt vydelenú počtom meraní. Pokiaľ je nastavený mód maxGetter, na získanie všetkých výsledkov bude použitá funkcia getTotalMax. Tá vráti najväčšiu zaznamenanú hodnotu.

Pre získanie výsledku z daného časového intervalu nám poslúži funkcia getCountInTimeInterval. Takisto pozerá na to, či pracujeme s priemernými hodnotami, so súčtom hodnôt alebo s najväčšou zaznamenanou hodnotou.

Pri generovaní grafov pre výsledky budeme potrebovať funkciu, ktorá nám vráti výsledky zoskupené podľa pravidelného časového intervalu. Na vstupe jej zadáme čas, od ktorého chceme hodnoty; čas, po ktorý má vrátiť hodnoty a časový interval, po akom má zoskupovať výsledky. Na získanie výsledkov z časových krokov využije funkciu getCountInTimeInterval.

Trieda Measure

Trieda Measure nám bude zoskupovať zoznam inštancií triedy metricUnit. K jednotlivým inštanciám triedy metricUnit budeme pristupovať podľa kľúča, ktorý bude pre jednoduchšie používanie zároveň aj názvom daného merania. Všetky tieto názvy si zadefinujeme ako konštanty v hlavičke tejto triedy. Popíšeme si tri základné funkcie na zaznáme-

návanie meraní.

Funkcia addAvg dostane na vstupe štyri parametre, a to názov meranej jednotky, čas merania, súčet nameraných hodnôt a počet meraných hodnôt. Overí existenciu danej meranej jednotky. Ak pre ňu neexistuje inštancia objektu metricUnit, vytvorí ju. Zo zadaných hodnôt vypočíta priemer a uloží ho. Samozrejme, v metóde je zabezpečené, aby inštanciu metricUnit nastavila do módu merania priemeru. Aby nedošlo ku skresľovaniu výsledkov, budeme do tejto inštancie zapisovať už len pomocou tejto funkcie.

Funkcia addInc funguje veľmi podobne ako addAvg. No nezapisujeme priemerné hodnoty. Inc nám napovedá, že v tejto metóde budeme hodnoty pripočítavať. Pripočítavanie nastane iba v prípade, ak sa už v danom časovom intervale zapísala niektorá z hodnôt. Inak ju iba jednoducho zapíšeme.

Funkcia addIfMax nám do daného času uloží hodnotu iba ak pre tento čas nie je uložená žiadna hodnota, alebo je uložená hodnota menšia ako aktuálne ukladaná. Pre addIfMax je dobré raz za beh aplikácie zavolať funkciu setMaxGetter, aby sme nastavili korektný mód pre danú meranú jednotku.

Funkcia add bude najjednoduchšou z trojice funkcií. Na rozdiel od addInc ju nebude zaujímať, či už bola zaznamenaná hodnota pre daný časový interval a ak náhodou bola, tak ju jednoducho prepíše tou najnovšou.

2.3.3 Grafické balíčky

V tejto skupine balíčkov máme zoskupené balíčky a triedy, ktoré majú niečo spoločné s grafikou a vykresľovaním objektov. Každému balíčku sa budeme venovať podrobne.

Balíček graphics/canvas

Canvas, v preklade z angličtiny maliarske plátno, bude mať pre nás veľmi podobný význam. Bude to totiž plátno pre naše vykresľované objekty. Tento balíček bude obsahovať dve triedy, canvasPanel a simuCanvas.

CanvasPanel je panel, ktorý bude vykreslený v okne s pracovnou plochou. Jeho hlavnou úlohou bude odchytávanie stlačenia myši a odovzdávanie tejto informácie manipulátoru aplikácie. Bude odovzdávať informácie ako 2D súradnice pozície, na ktorú sme klikli a zároveň informáciu o tom, ktoré tlačidlo myši bolo stlačené. CanvasPanel bude ob-

sahovať jednu veľmi dôležitú metódu paintComponent, na ktorej vstupe prichádza grafický element, do ktorého je možné vykresliť objekty. V tejto metóde bude našou úlohou poslať referenciu na tento element manipulátoru, ktorý ju následne posunie jednotlivým elementom, aby mohli byť vykresľované do hlavného plátna.

Druhou triedou v tomto balíčku bude simuCanvas. Táto trieda bude jednoduchšia a bude mať za úlohu vykreslenie okna a vytvorenie inštancie canvasPanel. Vycentruje okno na stred monitora a vykreslí ho. Prepája inštanciu sumulApp s canvasPanelom. Okrem iných metód bude obsahovať jednu dôležitú verejnú metódu getMousePosition, ktorá nám vráti pozíciu myši v danom čase. Pomocou tejto metódy sa dostaneme k súradniciam myši bez ohľadu na to, či stlačíme niektoré tlačidlo myši alebo nie.

Balíček graphics/canvasObjects/members

V aplikácii musíme nejakým spôsobom reprezentovať grafické objekty. Na to nám poslúži nasledujúci balíček. Obsahuje triedy, ktoré nám reprezentujú základné objekty pre vykreslenie na pracovnej ploche. Základ bude tvoriť objekt cShape, ako nosič základných dát pre rozširujúce objekty. Všetky ďalšie objekty budú tieto vlastnosti dediť a dopĺňať vlastnými rozšíreniami. Pri pomenovaní niektorých objektov budeme používať prefix „c“, ako napríklad cShape, cEllipse a pod. Toto „c“ nám reprezentuje slovíčko canvas a bude nám pripomínať, že dané objekty budú priamo previazané canvasom aplikácie.

Trieda cShape

Trieda cShape nám reprezentuje základný grafický objekt. Ten je nosičom všetkých univerzálnych informácii o grafickom objekte, ktoré budú dostupné pre všetky rozširujúce objekty.

V tejto triede budeme mať prítomné aj základné konštanty, ktoré nám pomôžu pri pretypovávaní objektu cShape na objekt, ktorý ho rozširuje. To neskôr využijeme napríklad v manipulátore. Okrem týchto konštánt bude táto trieda obsahovať aj ďalšie pomocné konštanty.

Za pomoci tejto triedy bude možné do každého grafického objektu vložiť obrázok, ktorý sa vykreslí miesto základného tvaru, ktorý daný objekt reprezentuje. Tomu sa budeme venovať detailnejšie pri popise jednotlivých funkcií tejto triedy.

Základné parametre, ktoré budú mať všetky grafické objekty spoločné:

- **index** – reťazec znakov, ktorý jednoznačne definuje objekt;
- **text** – textová hodnota uložená v objekte;
- **type** – reťazec znakov, ktorý nám hovorí o aký typ objektu sa jedná;
- **X, Y** – hodnota X a Y nám definuje umiestnenie objektu na pracovnej ploche;
- **width, height** – rozmery objektu máme definované ako šírku a výšku objektu;
- **crawlerStart** – booleanovská hodnota, ktorá nám hovorí o tom, či sa jedná o štartovaciu pozíciu aktérov simulácie alebo len o bod spracovania;
- **selected** - booleanovská hodnota definujúca, či je daný objekt vybraný alebo nie;
- **color** – farba objektu je definovaná ako farba samotného objektu, ale aj ako farba písma popisu objektu;
- **currentDistribution** – definuje akou funkciou budú aktéri z tohto objektu smerovaní do detí objektu;
- **imgSrc** – umiestnenie obrázka na disku, ak je nastavené, tak sa na miesto objektu vykreslí obrázok;
- **image** – bufferovaný (uložený vo vyrovnávacej pamäti) obrázok v pôvodných rozmeroch;
- **resizedImage** – bufferovaný obrázok, naškálovaný na aktuálne rozmery objektu (pre efektívnejšie vykresľovanie);
- **meter** – metriky patriace danému objektu, tieto metriky rozoberieme ďalej v texte;
- **children** – zoznam susedských vrcholov, do ktorých môžu byť distribuovaní aktéri, používané iba pri statických objektoch;

Pre všetky vyššie vymenované parametre budú existovať funkcie na ich získavanie a úpravu. Ďalej sa budeme zaoberať len špecifickými funkciami.

Vo funkcii budeme setImgSrc mať možnosť nastaviť obrázok, ktorým sa pri vykresľovaní nahradí náš objekt (štvorec, elipsa a pod.). Ak nastavíme vstupný parameter na prázdnu hodnotu, resp. null, zmažeme pôvodný obrázok a objekt sa začne vykresľovať v pôvodnom tvare. Ak zadáme ako parameter cestu k obrázku v PC, táto funkcia overí jeho

existenciu, uloží jeho bufferovanú verziu do objektu a následne vytvorí zoškálovanú verziu, ktorú uloží do ďalšej premennej, pre efektívnejší beh celej aplikácie.

Funkcia setDimensions nastaví rozmery daného objektu. Ak je k objektu priradený obrázok, tak sa originálny obrázok preškáluje. V objekte bude uložený aj v zmenšenej podobe, čo síce zvýši spotrebu pamäte, lebo budeme chcieť mať uložený obrázok aj v pôvodných rozmeroch pre budúce spracovanie. No táto spotreba pamäte je pri dnešných počítačoch zanedbateľná. Zoškálovanou verziou obrázka dosiahneme efektívnejší beh celej aplikácie, lebo ho nebude treba škálovať v každom momente vykresľovania.

Pre výpočet pravdepodobnosti návštevy susedného bodu si naprogramujeme funkciu getNextShapeToVisitByProbability. Bude používaná pri bodoch, ktoré majú viac ako jedného suseda. Na toto nám poslúžia tri možnosti nastavenia pravdepodobnosti návštevy suseda. Bude to cicular, čo bude znamenať pravidelné striedanie, ďalej to bude percentual, kde nastavíme percentuálnu hodnotu pravdepodobnosti návštevy suseda a poslednou možnosťou bude shortestQueue, čo znamená, že aktéri budú nasmerovaný do bodu v ktorom je najmenej čakajúcich aktérov.

```
public cShape getNextShapeToVisitByProbabilityCircular(){
    if(currentCircular >= children.values().size()){
        currentCircular = 0;
    }
    int i = 0;
    for(cShape s: children.values()){
        if(i == currentCircular){
            currentCircular++;
            return s;
        }
        i++;
    }
    return null;
}

public cShape getNextShapeToVisitByProbabilityShortestQueue(){
    int shortest = -1;
    cShape sh = null;
    for(cShape s: children.values()){
        if(shortest == -1 || s.getWaintingPersonsCount() < shortest){
            sh = s;
            shortest = s.getWaintingPersonsCount();
        }else if(s.getWaintingPersonsCount() == shortest){
            if(randomUniform(0, 1) == 0){
                sh = s;
                shortest = s.getWaintingPersonsCount();
            }
        }
```

```
    }
    return sh;
}

public cShape getNextShapeToVisitByProbabilityPercentual(){
    recalculatePercentualValues();
    double totalProp = 0;
    for(cShape s: children.values()){
        totalProp += getVisitProbability(s.getIndex());
    }
    double res = randomUniform(0.0, totalProp);
    totalProp = 0;
    for(cShape s: children.values()){
        totalProp += getVisitProbability(s.getIndex());
        if(res <= totalProp){
            return s;
        }
    }
    return null;
}
```

Pre overenie, či sme klikli na daný objekt, nám poslúži funkcia isClicked. Táto funkcia bude na vstupe očakávať aktuálnu pozíciu myši. Následne overí, či bolo kliknuté do objektu.

```
public boolean isClicked(int x, int y){
    int minX = X - (width / 2);
    int minY = Y - (height / 2);
    int maxX = minX + width;
    int maxY = minY + height;
    if(x >= minX && x <= maxX && y >= minY && y <= maxY){
        return true;
    }
    return false;
}
```

Budeme potrebovať funkciu, ktorá vykreslí jednotlivé prepojenia medzi objektami. Pomenujeme ju paintChildrenConnections. Táto funkcia prejde všetky deti objektu a vykreslí do nich čiaru so šípkou. Na kreslenie sa tu bude využívať funkcia drawArrow z vyššie spomínanej pomocnej triedy geometry.

```
public void paintChildrenConnections(Graphics g){
    g.setColor(connectionColor);
    for (cShape s: getChildren().values()) {
        Point p1 = new Point(X, Y);
        Point p2 = new Point(s.getX(), s.getY());
        Point midpoint = geometry.midPoint(p1, p2);
        g.drawLine(X, Y, s.getX(), s.getY());
        geometry.drawArrow(g, X, Y, midpoint.x, midpoint.y);
    }
}
```

Funkcie paintShape a paintImage budú mať na starosti samotné vykresľovanie objektu. Budú veľmi podobné, avšak vždy bude volaná iba jedna z nich podľa toho, či má daný objekt nastavený obrázok alebo nie. V oboch funkciách budeme musieť graficky znázorniť viacero atribútov. Prvý z nich je, či je daný objekt vybraný. Ak áno, vykreslíme okolo neho červený štvorec. Ďalším atribútom je vykreslenie crawlerStart. Ak je tento objekt zároveň označený atribútom crawlerStart, vykreslí okolo neho väčší zelený štvorec. Následne nám zostáva úloha vykreslenia samotného objektu a textu pod daný objekt. V tejto základnej triede sa vykreslí objekt ako bod, pričom táto metóda sa u jednotlivých rozširujúcich tried mení podľa toho, aký objekt reprezentujú.

Trieda cCrawlerPoint

CCrawlerPoint nám bude reprezentovať aktéra simulácie, bude teda pohybujúcim sa bodom. Je rozšírením objektu cShape, teda od neho zdedí všetky vlastnosti. Niektoré z týchto vlastností bude mierne pozmeňovať a dopĺňať.

Doplnené atribúty:

- **startShape** a **endShape** – reprezentujú začiatočný a konečný bod, od ktorého ku ktorému sa aktér pohybuje.
- **MovingEnabled** – tento atribút nám hovorí o tom, či sa tento aktér môže pohybovať.

Prvou pridanou funkciou bude funkcia move. V tejto funkcii využijeme pomocnú funkciu getNextPos z už skôr spomínanej triedy geometry. Touto funkciou získame novú pozíciu objektu a uložíme ju do jeho atribútov.

Ďalšou pridanou funkciou bude funkcia inFinish. Táto funkcia nám poslúži na zistenie, či sa aktér nachádza v cieli. Za cieľ považujeme okruh 5 bodov okolo stredového bodu cieľového objektu. Týmto spôsobom dostaneme mierne rozhádzané usporiadanie na danom bode, nebudeme preto vidieť viac bodov ako jeden, ale budú jeden cez druhý mierne vyčnievať.

```
public boolean inFinish(){
    if(endShape == null){
        return false;
    }
    if(Math.abs(X - endShape.getX()) < 5 &&
       Math.abs(Y – endShape.getY()) < 5){
```

```
            return true;
        }
        return false;
    }
```

Obrázok 11: Usporiadanie aktérov

Funkcia paint bude obmieňať pôvodnú funkciu paint zdedenú z objektu cShape. Narozdiel od cShape tu nie je priestor pre vykreslenie obrázka a samotný cCrawlerPoint sa vykreslí ako malý kruh o priemere 15 bodov. Hlavná farba je zelená, ale je možné ju meniť.

Triedy cEllipse a cRectangle

Tieto dve triedy sú taktiež rozšírením cShape, no reprezentujú už konkrétne tvary. V cEllipse sú upravené funkcie paintShape aj paintImage tak, aby reprezentovali elipsu. Pri cRectangle stačilo pozmeniť iba funkciu paintShape, keďže v cShape sa už obrázok vykresľuje v požadovanom formáte.

```
public void paintShape(Graphics g){
    g.setColor(color);
    Point startDraw = geometry.getStartXY(X, Y, width, height);
    g.fillRect(startDraw.x, startDraw.y, width, height);
    if(selected){
        g.setColor(Color.RED);
        g.drawRect(startDraw.x, startDraw.y, width, height);
    }
    if(crawlerStart) {
        Point startDravC = geometry.getStartXY(X, Y, (width + 10), (he-
ight + 10));
        g.setColor(Color.GREEN);
        Graphics2D g2 = (Graphics2D) g;
        g2.setStroke(new BasicStroke(2));
        g2.drawRect(startDravC.x, startDravC.y, (width + 10), (height +
10));
    }
    g.setColor(fontColor);
    Font font = new Font("Arial", Font.BOLD, (height / 2));
    FontMetrics metrics = g.getFontMetrics(font);
    int textX = (X - (metrics.stringWidth(value) / 2));
```

```
    int textY = startDraw.y + height + metrics.getHeight();
    g.setFont(font);
    g.drawString(value, textX, textY);
}
```

Trieda cText

Trieda cText nám bude modifikovať cShape na vykresľovanie jednoduchého textu bez rôznych tvarov. Pozmenené sú funkcie isClicked a paint. Vo funkcii isClicked sa musí použiť mierne odlišné porovnanie ako v predchádzajúcich dvoch funkciách. Tu musíme overiť, či sa kliklo na text a vo funkcii paint sa už vykresľuje text centrovaný na bod kliknutia, namiesto centrovaného geometrického objektu.

Balíček graphics/canvasObjects/manipulation

Tento balíček bude zameraný na manipuláciu s objektami. Bude obsahovať základné funkcie na manipuláciu, ktoré budú neskôr rozšírené a upravené rozširujúcou triedou z balíčka econSimul. Základnou myšlienkou tohto balíčka je dostať objekty balíčka members pod kontrolu, to znamená vytvoriť intuitívne rozhranie medzi používateľom a vlastnosťami jednotlivých objektov.

Trieda cShapes

CShapes nám bude slúžiť ako úložný priestor a základné rozhranie pre statické objekty, teda pre všetky objekty okrem objektov cCrawlers. Základné rozhranie je chápané ako výber konkrétneho objektu, jeho presúvanie, zobrazenie a zmena jeho parametrov. Ďalej sa budeme venovať každej dôležitejšej funkcii samostatne.

Funkcia addCShape vloží daný objekt do zoznamu objektov. Zoznam je linkovaný, čo znamená, že objekty budú systematicky usporiadané podľa toho, v akom poradí boli do zoznamu pridávané. Každý z objektov je pre ľahšie vyhľadávanie definovaný indexom, ktorý je reprezentovaný stringovým reťazcom znakov.

Ďalej budeme potrebovať funkciu, ktorá sa postará o odstránenie objektov zo zoznamu objektov. Pomenujeme ju deleteCShape. Táto funkcia skontroluje na základe indexu výskyt objektu v zozname. Ak sa tu objekt vyskytuje, zmaže ho zo zoznamu.

V mnohých situáciach sa budeme musieť dopracovať k objektu na určitej pozícii, k čomu nám poslúži funkcia getCShapeOnPos. Na vstupe dostane pozíciu z pracovnej plochy, potom bude prechádzať všetky objekty v zozname a bude volať funkciu isClicked

každého z nich. Prechádzanie skončí, keď sa nájde kandidát, na ktorého sa kliklo, resp. keď sa bez úspechu prejdú všetky objekty v zozname. Okrem vyhľadania objektu podľa jeho pozície bude pre nás užitočný aj výber objektu podľa jeho indexu. Na to nám poslúži funkcia getCShapeByIndex. Funkcia getCShapes nám umožní dostať sa ku kompletnému zoznamu objektov.

Funkcia setSelected bude použitá na vyznačenie konkrétneho objektu. Funguje na podobnom princípe ako vyhľadávanie objektu podľa pozície (getCShapeOnPos). Nájde objekt na danej pozícii a nastaví ho ako vybraný. Ak bol pred tým vybraný niektorý iný objekt, zruší jeho výber.

Pri volaní unselectAll docielime to, aby nebol vybraný ani jeden z objektov. Táto funkcia prejde všetky objekty v zozname a ak má niektorý z nich nastavený parameter selected, tak ho nastaví ako nevybraný.

Funkcia childToCurrentClick bude slúžiť na dva účely. Názov napovedá, že budeme nastavovať dieťa niektorému z elementov. No zároveň ho budeme môcť aj zrušiť, a to na základe vstupného parametra funkcie. V prvom rade overíme, či máme vybraný aspoň jeden z objektov. Bez toho by nemalo zmysel pokračovať. V ďalšej fáze nájdeme objekt na ktorý sa kliklo. Ak taký objekt neexistuje, funkcia v tomto bode skončí. V poslednom kroku prepojíme, alebo rozpojíme (podľa hodnoty vstupného parametra) dva dané objekty.

```
public void childToCurrentClick(int x, int y, boolean set){
    if(!hasSelected){
        return;
    }
    cShape shapeOnPos = null;
    HashMap.Entry<String, cShape> shapeInfo = getCShapetOnPos(x, y);
    if(shapeInfo == null){
        return;
    }
    shapeOnPos = shapeInfo.getValue();
    if(shapeOnPos == null || shapeOnPos.isSelected()){
        return;
    }
    for(cShape entry : shapes.values()) {
        if(entry.isSelected()){
            if(set) {
                entry.addChild(shapeOnPos);
            }else{
                entry.deleteChild(shapeOnPos);
            }
```

```
        }
    }
}
```

Funkcia paint nám zabezpečí zobrazenie všetkých statických objektov. Prejde zoznam objektov a postupne ich vykreslí. Vykresľovanie prebehne v dvoch krokoch. V prvom kroku sa vykreslia prepojenia objektov a v druhom samotné objekty. Je to z toho dôvodu, aby nám prepojenia objektov neprekrývali samotné objekty.

```
public void paint(Graphics g){
    try {
        for (cShape s : shapes.values()) {
            s.paintSiblingsConnections(g);
        }
        for (cShape s : shapes.values()) {
            s.paint(g);
        }
    } catch (ConcurrentModificationException e) {
        System.out.println("SKIP PAINTING (cShapes)");
    }
}
```

Trieda cShapeCrawlers

Trieda cShapes nám bude slúžiť ako úložisko pre statické objekty. CShapeCrawlers nám poslúži ako úložný priestor a správca pohybu pre pohyblivé objekty (aktérov simulácie) a bude mať za úlohu prekresľovanie a animáciu jednotlivých pohyblivých objektov. V prvom rade nám bude slúžiť ako základ pre rozširujúcu triedu, ktorej sa budeme venovať ďalej v texte. Aj v tejto triede si iba v krátkosti prejdeme jednotlivé funkcie, lebo rozširujúca trieda ich značne pozmení. V základe nám poslúži na to, aby bol prejdený kompletne celý graf.

Dôležité funkcie v triede cShapeCrawlers:

- **addShapes** – pridávanie statických objektov do simulácie (aby sme poznali celý graf simulácie);
- **startNewCrawlers** – vygeneruje prvé pohyblivé objekty v startpointoch;
- **run** – spustí pohyb, spravuje ich pohyb do jednotlivých uzlov;
- **move** – v každom tiknutí timera vykoná presunutie daného objektu na novú pozíciu;
- **paint** – vykreslenie objektu na jeho pozícii;

Trieda cManipulator

Triedou cManipulator spojíme obe vyššie spomenuté triedy. Poslúži nám ako základ pre rozširujúcu triedu a bude mať prístup k timeru. V každom časovom intervale timera bude vykonávať funkciu tick.

Funkcia tick nám bude slúžiť na odovzdanie informácie o časovom intervale pre cCravlersPoints. Zabezpečí periódu pohybu aktérov simulácie. V každej perióde sa pohnú o vopred zadanú dĺžku kroku.

Pri detekcii stlačenia pravého tlačidla myši si naprogramujeme funkciu mouseDownRight, ktorá nám bude vyberať objekt, na ktorý sa klikne. Výber objektu neovplyvní aplikáciu pri spustenej simulácii, a tak tento výber môže byť prítomný v každom móde simulácie. Zavolá funkciu setSelected v triede cShapes, ktorá prejde všetky objekty v zozname, a keď pomocou funkcie isClicked zistí, že bolo na jeden z objektov kliknuté, vyznačí ho ako vybraný.

Funkcia mouseDown bude rozoznávať kliknutie ľavým tlačidlom myši. Tu už je dôležité aký mód budeme mať v našej aplikácii nastavený.

Každý z módov si stručne popíšeme:

- **crawling** – mód, v ktorom bude aktívne prechádzanie grafu, teda simulácia spustená;
- **selecting** – výber objektov (to isté ako pravé stlačenie pravého tlačidla myši);
- **addingChild** – pridávanie prepojenia medzi vybraným objektom a objektom, na ktorý sa kliklo;
- **deletingChild** – odstraňovanie prepojenia medzi objektami;
- **adding** – adding budú dva režimy v jednom. Pokiaľ sa klikne na existujúci objekt, dá sa presúvať. Ak sa klikne na prázdne miesto na pracovnej ploche, vygeneruje sa formulár na pridávanie nového objektu.

Ako sme vyššie v texte spomínali, budeme mať možnosť meniť pozíciu už pridaných objektov. Presúvanie objektov bude aktívne v móde „adding“. Detegujeme stlačenie myši na danom objekte a pri jej presúvaní nastavujeme vybranému objektu aktuálnu pozíciu myši. Týmto docielime pohyb objektu po pracovnej ploche na základe pohybu myši.

Funkciu pomenujeme moveCShape

2.3.4 Balíčky ekonomickej simulácie

V tejto skupine balíčkov budú zoskupené balíčky a triedy slúžiace na samotnú ekonomickú simuláciu. Budú tu reprezentované ekonomické objekty, grafické rozhranie na ich ovládanie a nástroje na vizualizáciu výsledkov simulácie. Z veľkej časti budú dediť vlastnosti z predchádzajúcich balíčkov, prípadne ich budú rozširovať.

Balíček econSimul/members

V balíčku members budeme mať zadefinované objekty, ktoré už konkrétne súvisia s ekonomickou simuláciou. Pri ekonomických objektoch budeme potrebovať viac parametrov ako pri základných objektoch, o ktorých sme písali vyššie. Tieto objekty budú rozšírením objektov z balíčka graphics/canvasObjects/members.

Trieda cClock

Ako sme už skôr spomínali, základnou mernou jednotkou je čas. Objekt cClock bude práve tým objektom, ktorý bude merať čas a bude poskytovať informácie o čase, či už vizuálne, ale aj interne v rámci vnútornej komunikácie objektov aplikácie. Tento objekt je rozšírením triedy cText, aby sme vedeli jednoducho vypisovať aktuálny čas na pracovnej ploche.

Hlavné atribúty triedy cClock:

- **time** – aktuálny čas behu simulácie reprezentovaný celým číslom (počet simulovaných minút od spustenia simulácie);
- **tickSpeed** – premenná, ktorá nám pomôže urýchliť beh simulácie;
- **tickDelay** – premenná, ktorá nám pomôže spomaliť beh simulácie .

Prvou funkciou v tejto triede bude funkcia isClicked. Táto funkcia je zaujímavá tým, že bude za každých okolností vracať hodnotu false, teda, že sa na objekt nekliklo. Tak zabránime možnej manipulácii s týmto objektom na pracovnej ploche. Funkcia getTime nám zas vráti surovú hodnotu ubehnutého času simulácie z premennej time.

Funkcie getDay, getHour, getMinute, getStringTime nám budú slúžiť na prekladanie číselnej časovej premennej do ľudskej reči. Pomocou nich budeme dostávať čas vo formáte dní, hodín a minút. GetStringTime nám vráti takto naformátovaný čas v kompletnom

formáte, v ktorom sa nám bude zobrazovať na pracovnej ploche.

Trieda cPerson

Už máme pripravenú triedu cCrawler. Na reprezentáciu aktérov bude musieť táto trieda niesť viac informácii. Každý aktér bude mať k dispozícii okrem už jednotiek obsiahnutých v triede cCrawler aj čas, ktorý môže stráviť v simulácii a peniaze, ktoré môže v simulácii spotrebovať. Ďalej tu bude definovaná farba „mainColor“, ktorá bude vyjadrovať z ktorého štartovacieho bodu vyšiel. Počas simulácie bude meniť farbu podľa toho, v akom stave sa bude nachádzať, no mainColor bude mať nastavenú, keď bude na ceste z jedného bodu do druhého. Okrem mainColor bude niesť atribút „parentPoint“, v ktorom bude uložená referencia na štartovací bod, v ktorom aktér vznikol. Posledný z atribútov bude parameter „priority“, ktorý bude hovoriť akú prioritu má daný aktér vzhľadom na ostatných aktérov simulácie.

Každý aktér bude mať aj rôzne stavy v ktorých sa môžu nachádzať:

- **ON_ROUTE_STATE** – aktér je na ceste medzi dvoma statickými bodmi;
- **IN_PROCESSING_POINT_STATE** – aktér je momentálne spracovávaný v processing centre;
- **WAITING_STATE** – aktér čaká na vstup do processing centra;
- **OUT_OF_AVAILABLE_UNITS_STATE** – aktér sa už nezúčastňuje simulácie, minul čas alebo peniaze;
- **EXITED_STATE** – aktér opustil simuláciu bodom, z ktorého už nevedie cesta do žiadneho ďalšieho.

Trieda cProcessingPoint

V triede cProcessingPoint budú spracovávaní jednotliví aktéri. Musíme sa zamyslieť nad tým, akú štruktúru bude mať cProcessingPoint a aké dátové štruktúry v ňom použijeme.

Za vzor sme si zobrali situácie z reálneho života. Môžeme si predstaviť, že cProcessingPoint je napríklad niektorý z úradov alebo klientských centier. Centrum má svoju kapacitu, určitý počet priehradiek, za ktorými si môžu občania vybavovať potrebné náležitosti. Vybavovaných môže byť súčasne aj niekoľko osôb naraz. Ďalej sa v takejto organizá-

cii nachádza čakáreň, v ktorej čakajú ďalší ľudia, kým sa dostanú na rad. Niektorí občania môžu byť vopred objednaní, a tým majú vyššiu prioritu ako ostatní. Vybavení ľudia zase z organizácie odchádzajú.

Každá organizácia má svoje otváracie hodiny, časy v ktorých je viac vyťažená, časy v ktorých sú ceny nižšie, alebo vyššie, na to využijeme vyššie spomenutý timeDepender.

Podľa predchádzajúceho vzoru bude mať náš cProcessingPoint nasledovnú štruktúru atribútov:

- **moneyConsumptionMin a moneyConsumptionMax** – interval, z ktorého sa vyberie náhodné číslo, ktoré bude reprezentovať spotrebu peňazí daného aktéra;
- **časovo závislé intervaly moneyConsumption** – intervaly určené timeDependerom. Ak nie je definovaný, berie sa do úvahy hodnota z predchádzujúceho intervalu;
- **timeConsumptionMin a timeConsumptionMax** – interval, z ktorého sa vyberie náhodné číslo, ktoré bude reprezentovať spotrebu času daného aktéra;
- **časovo závislé intervaly timeConsumption** – obdobne ako časovo závislé intervali moneyConsumption;
- **capacity** – kapacita processing pointu hovorí o počte súbežne spracovávaných aktérov; tí ktorí nemôžu byť spracovaní, čakajú na vstup;
- **časovo závislá premenná capacity** – časovo závislá kapacita, ktorá bude definovaná pomocou timeDependera;
- **personsWaiting** – rad, v ktorom aktéri čakajú na vstup do preplneného centra, reprezentovaný dátovou štruktúrou PriorityStack z balíčka util;
- **personsIn** – aktuálne spracovávaní aktéri;
- **personsOut** – už spracovaní aktéri, ktorí čakajú na nasmerovanie do ďalšieho centra spracovania alebo na opustenie simulácie;
- **personsOutOfMoney** – aktéri, ktorí opustili simuláciu kvôli spotrebovaniu všetkých svojich peňazí;
- **personsOutOfTime** – aktéri, ktorí opustili simuláciu kvôli spotrebovaniu všetkého

času, ktorý mali k dispozícii;

- **personsExited** – aktéri, ktorí opustili simuláciu kvôli tomu, že z daného centra už nepokračuje cesta do ďalšieho bodu simulácie;

V prvom rade si popíšeme funkciu addPerson slúžiacu pre vstup osoby do processing centra.

Algoritmus bude nasledovný:

1. Over či existuje pre daný čas časovo závislá hodnota kapacity. Ak áno, nastav kapacitu na túto hodnotu, ak nie, tak ju nastav na pôvodnú.
2. Porovnáme, či je počet osôb v centre rovný ako je kapacita. Ak je rovný, pokračuj bodom 10, inak pokračuj bodom 3.
3. Overíme či existuje pre daný čas časovo závislá hodnota intervalov moneyConsumption. Ak áno, nastavíme podľa nich hodnoty moneyConsumptionMin a moneyConsumptionMax.
4. Obdobne overíme a spracujeme časovo závislé hodnoty pre timeConsumption.
5. Vygenerujeme dve náhodné čísla z intervalu moneyConsumptionMin, moneyConsumptionMax a z intervalu timeConsumptionMin a timeConsumptionMax. Týmto spôsobom získame čísla konzumácie jednotiek danej osoby.
6. Porovnáme vygenerovanú hodnotou spotreby peňazí s hodnotou, ktorou disponuje daný aktér. Ak je vygenerovaná hodnota väčšia ako je hodnota, ktorú v sebe nesie aktér, vyradíme aktéra zo simulácie a zapíšeme to do metrics pomocou funkcie addInc a ukončíme algoritmus.
7. Obdobne porovnáme vygenerovanú hodnotu spotreby času.
8. Aktérovi nastavíme nové hodnoty jeho jednotiek a pridáme ho do zoznamu personsIn. Zapíšeme do metrics pomocou funkcie addInc. Do metrics zapíšeme aj hodnoty spotrebovaného času a peňazí pomocou funkcie addAvg.
9. Nastavíme čas budúcej akcie osoby (aktuálny čas + spotrebovaný čas) a ukončíme algoritmus.
10. Tento bod je druhou vetvou pokračovania bodu 2. Tu len zapíšeme do metrics hod-

noty aktéra opúšťajúceho simuláciu.

Ďalej si popíšeme funkciu tick. Táto funkcia bude vykonávaná pri každom tiknutí timeru. Tu budeme spracovávať aktérov, ktorí sú vo vnútri centra, alebo čakajú na vstup do neho.

Algoritmus bude nasledovný:

1. V prvom rade skontrolujeme všetkých aktérov, ktorí sú vo vnútri centra. Ak uplynul čas, ktorý musel stráviť vo vnútri, nastavíme ho na východ z centra. Každého spracovaného aktéra zapíšeme do metrics pomocou funkcie addInc.
2. Porovnáme, či je počet aktérov v centre rovný ako je kapacita, poprípade časovo určená kapacita. Ak je rovný, ukončíme algoritmus.
3. V opačnom prípade, ak existuje na vstupe čakajúci aktér, vyberieme toho, ktorý má nastavenú najvyššiu prioritu. Ak ich je viac s touto prioritou, vyberieme toho, ktorý najdlhšie čaká (dátová štruktúra PriorityStack z balíčka util). Presunieme ho na spracovanie pomocou funkcie addPerson a vrátime sa na bod 2, inak pokračujeme bodom 4.
4. Zapíšeme nový počet čakajúcich osôb pomocou funkcie addInc (záporným číslom). Do metrics tak isto zapíšeme aktuálny počet čakajúcich aktérov a počet aktérov vo vnútri centra.

Teraz si popíšeme úpravy z hľadiska vykresľovania objektu. CProcessingPoint bude rozšírením triedy cRectangle. Jednou zo zmien bude vykresľovanie vo funkcii paint, v ktorej bude musieť vykresliť informácie o svojom stave počas simulácie. Základ sa vykreslí pomocou funkcie paint zdedenej z cRectangle. Následne sa pod vykreslený objekt vypíšu informácie o počtoch aktérov, ktorí čakajú na vstup, ktorí sú spracovávaní a ktorí ho opustili. Pred každým číslom bude skratka, ktorá bude napovedať, o ktorú informáciu ide:

- **w** – waiting, čiže áktéri aktuálne čakajúci na vstup;
- **in** – in bude reprezentovať aktuálne spracovávaných aktérov;
- **o** – out, čiže počet aktérov, ktorí opustili tento bod simulácie;
- **u** – táto skratka nám bude reprezentovať slovo units a bude nás informovať o tom, koľko aktérov tu opustilo simuláciu, lebo minuli niektoré z jednotiek, ktoré mali k dispozícii (out of available units);

Obrázok 12: Popisky pod cProcessingPointom

Trieda cStartPoint

CStartPoint bude vstupnou bránou aktérov do simulácie. Presnejšie povedané, tento bod ich bude generovať na základe vopred stanovených pravidiel. Týchto bodov bude v simulácii ľubovoľný počet, zmysel budú mať iba v prípade, že budú prepojené minimálne s jedným cProcessingPointom.

Navonok bude cStartPoint vystupovať podobne ako cProcessingPoint, ale nebude obsahovať funkcie na spracovávanie aktérov. Aktéri vstupujúci do cStartPointu, teda nevytvorení priamo daným cStartPointom v ňom budú opúšťať simuláciu, no nebude vhodné používať ho na tento účel.

Každý cStartPoint bude mať pri vytvorení vygenerovanú náhodnú farbu, aby sme vedeli pri grafickej simulácii rozoznať, ktorý aktér z ktorého cStartPointu vyštartoval. Túto farbu budú mať nastavenú aj všetci aktéri štartujúci z tohto bodu.

CProcessingPoint bude tak isto rozšírením triedy cRectangle. Tu bude pod objektom vypísané číslo reprezentované skratkou „n“ (new), reprezentujúce počet aktérov, ktorí do simulácie vstúpili.

Obrázok 13: Popisok pod cStartPointom

Hlavné atribúty triedy cStartPoint (intervaly, z ktorých sa vygenerujú náhodné čísla):

- **personsPerCreatingMin a personsPerCreatingMax** – pomocou týchto atribútov

nastavíme triede počet aktérov v dávke, ktorý môže byť vytvorený za jedno generovanie;

- **časovo závislé intervaly personsPerCreating**;
- **personMoneyAvailableMin a personMoneyAvailableMax** – také množstvo peňazí, ktoré bude mať vytváraný aktér k dispozícii;
- **personTimeAvailableMin a personTimeAvailableMax** – taký dlhý čas, aký bude môcť generovaný aktér stráviť v simulácii;
- **countTicksToCreatingMin a countTicksToCreatingMax** – počet minút, po koľkých sa môže vygenerovať ďalšia dávka aktérov;
- **časovo závislé intervaly countTicksToCreating**;
- **personsPriority** – priorita, akú budú mať priradenú aktéri, ktorí tu budú generovaní;
- **personsColor** – farba, ktorú budú mať aktéri nastavenú ako mainColor a ktorá bude aj farbou daného startPointu.
- **meter** – inštancia triedy metrics slúžiaca na meranie hodnôt generovania aktérov.

Z hľadiska zložitosti a komplexnosti bude pre nás zaujímavá iba táto funkcia tick. Počas každej simulovanej minúty bude mať za úlohu určiť správny čas generovania a vygenerovať osoby podľa zadaných pravidiel.

Generovací algoritmus bude fungovať v nasledovných krokoch:

1. Vygenerujeme náhodné číslo z intervalu personsPerCreatingMin a personsPerCreatingMax, poprípade ak existujú časovo závislé hodnoty, tak vygenerujeme z nich.
2. Vygenerujeme náhodne zadaný počet aktérov, prípadne ak existuje časovo závislá hodnota, tak použijeme tú.
    a) Vygenerujeme dve náhodné čísla z intervalu personTimeAvailableMin a personTimeAvailableMax a z intervalu personMoneyAvailableMin a personMoneyAvailable;
    b) Vygenerujeme aktéra s danými hodnotami;

c) Overíme počet vygenerovaných aktérov za daný cyklus, ak ich nie je dostatok, pokračujeme bodom 2.a), ak ich je dostatok, pokračujeme bodom 3.

3. Vygenerujeme náhodné číslo, ktoré nám zadá, po koľkých minútach majú byť vygenerovaný ďalší aktéri simulácie.
4. Budeme kontrolovať, koľko minút uplynulo od posledného generovania. Ak je tento počet rovný náhodnému číslu vygenerovanému v bode číslo 2, prejdeme na bod 1 a pokračujeme.

Pri každom generovaní budeme zaznamenávať počty generovaných aktérov, ako aj priemerné hodnoty generovaných jednotiek pomocou objektu metrics.

Balíček econSimul/forms

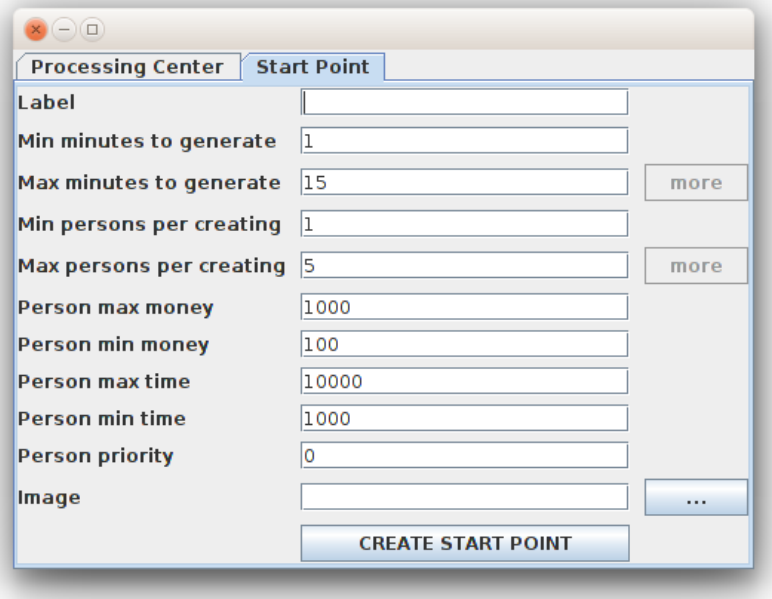
V balíčku forms sa budú nachádzať všetky formuláre potrebné na obsluhu aplikácie. Prostredníctvom nich budeme schopní komunikovať so zadefinovanými triedami, a teda budeme nové triedy vytvárať, zadávať im parametre a budeme ich môcť aj editovať a mazať.

Formulár econObjForm

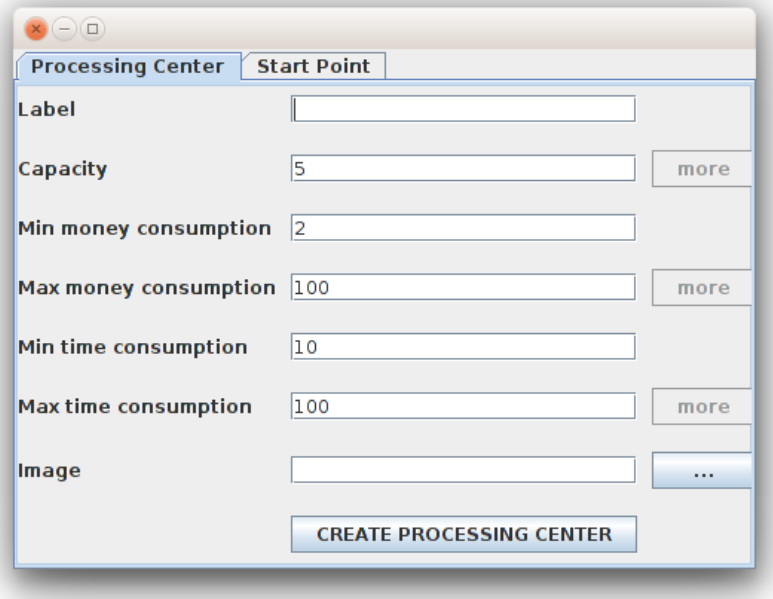
Cez tento formulár budeme vytvárať a editovať ekonomické objekty. Zvolíme tu kartové usporiadanie, aby sme cez jeden formulár vedeli vkladať a editovať oba druhy statických ekonomických objektov (cStartPoint a cProcessingPoint).

Budeme mať tri možnosti vyvolania formulára:

- **volanie iba pomocou funkcie a zadaním polohy kurzora myši** – prázdny formulár pre vytvorenie nového objektu;
- **volanie pomocou vloženia cStartPoint do parametra** – načíta hodnoty vloženého cProcessingCenter, vypíše ich a bude slúžiť ako editačný formulár;
- **volanie pomocou vloženia cProcessingCenter do parametra** – obdobne ako predchádzajúci.



Obrázok 14: Formulár econObjForm pre cStartPoint

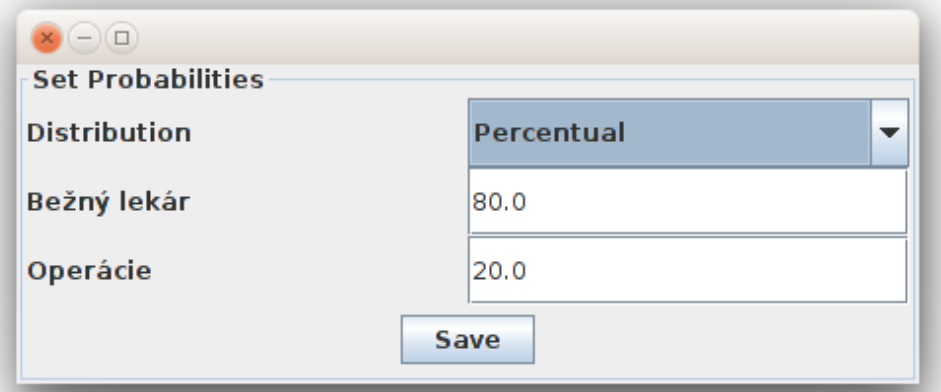


Obrázok 15: Formulár econObjForm pre cProcessingPoint

Formulár visitProbForm

V simulácii budeme môcť nastaviť určitú pravdepodobnosť návštevy statického objektu. Bude to vhodné vo chvíli, keď bude z jedného objektu viesť viacero ciest do iných objektov. Algoritmus sme si už vysvetlili skôr.

V tomto formulári sa budú textové polia generovať automaticky podľa toho, s ktorými bodmi vybraný objekt susedí. Formulár teda dostane na vstupe bod, na ktorý sa kliklo a z neho vyčíta jeho susedov. Vypíše pravdepodobnosti ich návštevy, ktoré budeme môcť upraviť a uložiť.



Obrázok 16: Formulár visitProbForm

Formulár fullResultsForm

Pre základnú vizualizáciu výsledkov simulácie nám poslúži práve tento formulár. Výsledky budú reprezentované v dvoch tabuľkách, pričom prvá bude určená pre cStartPointy a druhá pre cProcessingPointy.

Algoritmus generovania výsledkových tabuliek:

1. Prejdeme všetky cStartPointy a vygenerujeme tabuľku.
    a) Vytvoríme hlavičku tabuľky. Na prvé miesto umiestnime unikátny index daného bodu, na druhé jeho názov. Ďalej prejdeme metriky každého cStartPointu a názvy meraných jednotiek použijeme ako hlavičky;

b) Samotné hodnoty budú generované z objektov a ich metrics. Hodnoty z metrics získame volaním funkcie getTotal pre každú metrics zvlášť;

2. Prejdeme všetky cProcessingPointy a vygenerujeme tabuľku.

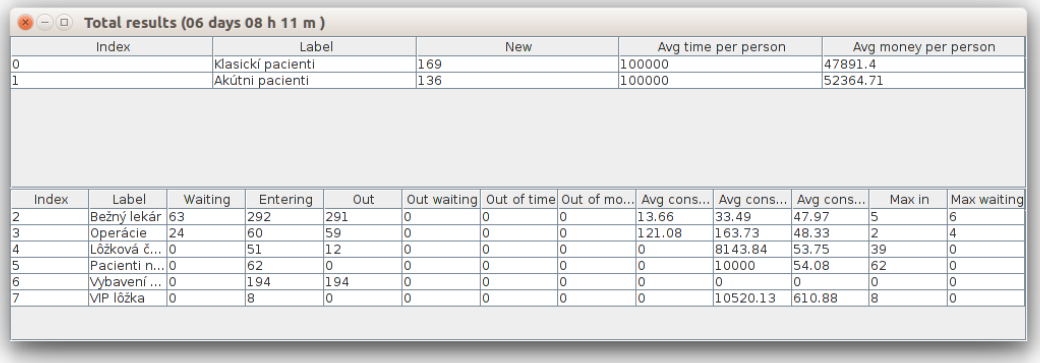
   a) Obdobne ako cStartPointy.

V tejto triede bude pre nás najzaujímavejšou funkciou funkcia addTable. Na vstupe dostane zoznam hodnôt s názvami stĺpcov a zoznam hodnôt jednotlivých riadkov, pričom každá hodnota môže byť iného typu (reťazec znakov, celé číslo alebo číslo s desatinným miestom). My by sme chceli mať výsledky vizuálne čo najprívetivejšie (pri celých číslach nemať desatinné miesta s nulami a nemať príliš dlhý desatinný rozvoj).

Dosiahneme to nasledovne:

- Ak je hodnota reťazec znakov, zapíš ju do tabuľky.
- Ak je hodnota číslom, zisti či je číslom desatinným alebo celým.
- Ak je to číslo desatinné, zaokrúhli a vypíš do tabuľky.
- Ak je to číslo celé, zmeň jeho typ na celočíselný a zapíš do tabuľky.

Ďalšou úlohou bude zabezpečiť možnosť vypísania detailov po kliknutí na konkrétny riadok. Keďže v prvom stĺpci budeme mať vypísaný unikátny index daného objektu, budeme ho môcť pri kliknutí na riadok využiť. Po kliknutí zavoláme graphDataForm pre objekt s unikátnym indexom na riadku, na ktorý sme klikli.



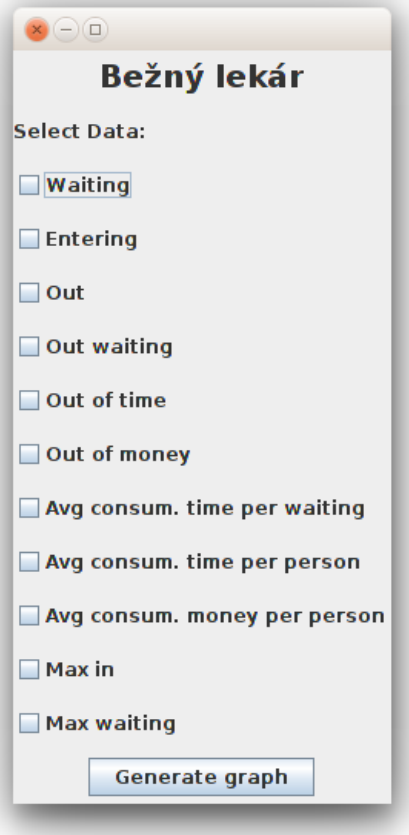
Index	Label	New	Avg time per person	Avg money per person
0	Klasickí pacienti	169	100000	47891.4
1	Akútni pacienti	136	100000	52364.71

Index	Label	Waiting	Entering	Out	Out waiting	Out of time	Out of mo...	Avg cons...	Avg cons...	Avg cons...	Max in	Max waiting
2	Bežný lekár	63	292	291	0	0	0	13.66	33.49	47.97	5	6
3	Operácie	24	60	59	0	0	0	121.08	163.73	48.33	2	4
4	Lôžková č...	0	51	12	0	0	0	0	8143.84	53.75	39	0
5	Pacienti n...	0	62	0	0	0	0	0	10000	54.08	62	0
6	Vybavení ...	0	194	194	0	0	0	0	0	0	0	0
7	VIP lôžka	0	8	0	0	0	0	0	10520.13	610.88	8	0

Obrázok 17: Formulár fullResultsForm

Formulár graphDataForm

Pri generovaní grafov bude vhodné mať možnosť vybrať si hodnoty, ktoré chceme grafom vizualizovať. Niekedy budeme chcieť zobraziť iba vývoj konkrétnej hodnoty, inokedy budeme chcieť porovnať viacero hodnôt v čase.



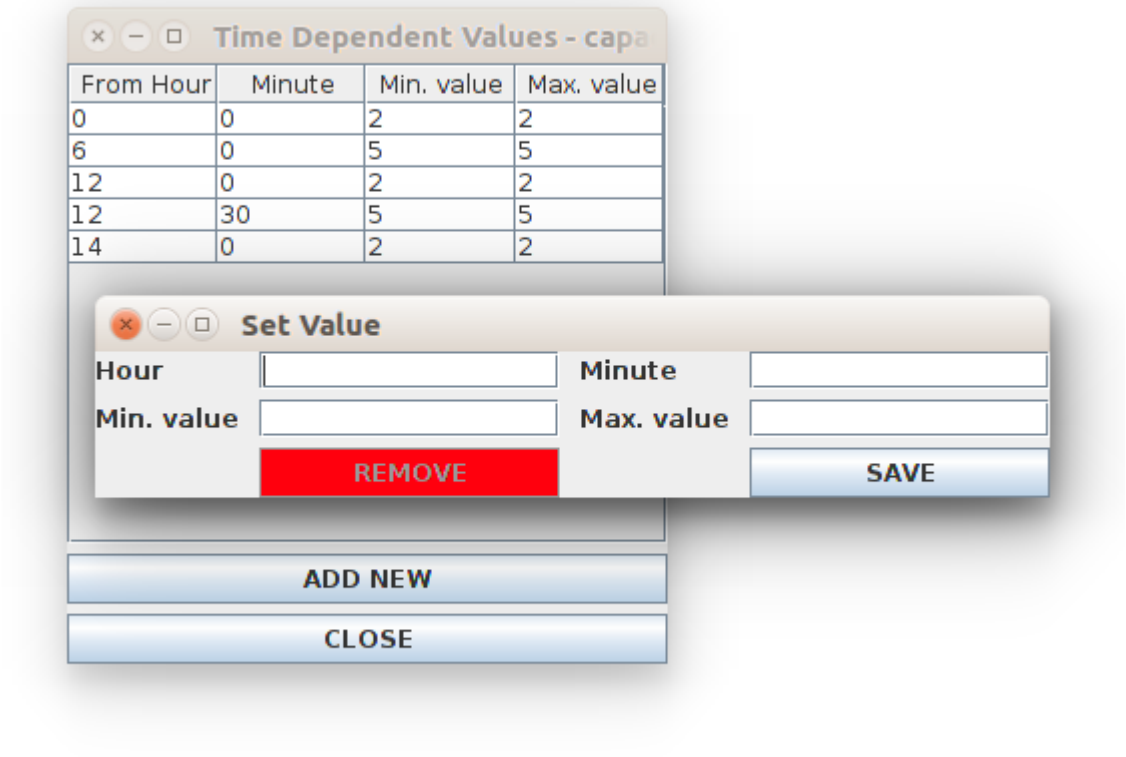
Obrázok 18: Formulár graphDataForm

Formulár dostane na vstupe zoznam s metricUnits daného objektu a vygeneruje chcekbox pre každú z meraných hodnôt. Po kliknutí na tlačidlo generate sa predajú hodnoty triede LineChart v balíčku graphing. O tejto triede si povieme ďalej v texte.

Formuláre timeDepentForm a addTimeDependForm

TimeDepend form nám bude slúžiť na zobrazovanie časovo závislých hodnôt. Budú zobrazené v jednoduchej tabuľke, usporiadané podľa času. Po kliknutí na riadok sa nám ot-

vorí ďalší formulár (addTimeDependForm), pomocou ktorého budeme môcť túto hodnotu upraviť alebo vymazať. AddTimeDependForm bude obsahovať 4 textové vstupy, ktoré nám umožnia definovať jednotlivé, časovo závislé hodnoty.



Obrázok 19: Formuláre timeDepentForm a addTimeDependForm

Balíček econSimul/graphing

Balíček obsahuje len jednu triedu, no v budúcnosti nám môže poslúžiť na prehľadné rozšírenie aplikácie a ďalšie typy grafov. Pri tomto balíčku budeme na generovanie grafov používať externú knižnicu jFreeChart. Tá nám výrazne uľahčí vizualizáciu grafov.

Trieda LineChart

Trieda LineChart nám bude slúžiť na vizualizáciu údajov pomocou čiarového grafu. Bude nadstavbou nad externú knižnicu jFreeChart, ktorá bude premieňať náš formát dát na dáta vhodné pre jFreeChart. Táto trieda bude pozostávať z funkcií setData, createDataset, createChart a createGraph.

SetData je funkcia schopná spracovať surové intervalové dáta z balíčka metrics.

Ukladá ich do zoznamu a sú pripravené na spracovanie. Tieto dáta sú posunuté funkcii createDataset, ktorá premení získané dáta na dátovú sadu vhodnú pre vizualizáciu.

Funkcia createChart nakonfiguruje vizuál grafu, nastaví popisky, jeho rozmery a iné dizajnové nastavenia.

Funkcia createGraph je prepojením funkcií createDataset a createChart. Po ich vykonaní vytvorí okno, do ktorého vygeneruje finálny graf s výsledkami.

Balíček econSimul/manipulation

Zatiaľ čo balíček graphics/canvasObjects/manipulation bol určený len na manipuláciu so základnými objektami, balíček econSimul/manipulation má za úlohu starať sa o manipuláciu s ekonomickými objektami, o ich vytváranie, editáciu, ale aj o presúvanie. V neposlednom rade bude mať celkovú kontrolu nad bežiacou simuláciou.

Trieda simulManipulator

SimulManipulator je rozšírením pôvodnej triedy cManipulatorora. Je doplnený o jeden veľmi dôležitý objekt, ktorým sú hodiny, inštancia triedy cClock, ktoré merajú simulovaný čas. Tu sa budeme venovať iba pozmeneným funkciám.

V prvom rade tu bude volaná pôvodná funkcia tick z rodičovského objektu, následne bude vykonaná funkcia tick v inštancii triedy cClock, aby sme zabezpečili beh virtuálnych hodín.

Ďalšou dôležitou funkciou v tejto triede bude funkcia mouseDown. Keďže funkcia mouseDown je taktiež len rozšírením rodičovskej funkcie, povieme si iba o nových módoch v triede simulManipulator.

Nové módy aplikácie:

- **editing** – editácia ekonomického objektu, na ktorý je kliknuté;
- **metrics** – vygenerovanie formuláru pre výber z metrics a následne zobrazenie výsledkov vizuálne v grafe;
- **visitpropability** – pre vybraný objekt sa vygeneruje formulár pre nastavenie pravdepodobností návštevy dcérskych objektov.

Trieda staticObjects

Trieda staticObjects bude len drobným vylepšením triedy cShapes. Bude doplnená o prenášanie informácií z hodín do jednotlivých objektov (vo funkcii tick). Ďalej tu bude zabezpečené generovanie formulára pre pravdepodobnosť návštevy dcérskych objektov a funkcia, pomocou ktorej vygenerujeme formulár, cez ktorý sa dostaneme ku grafovej vizualizácii výsledkov simulácie.

Trieda personObjects

Trieda cShapeCrawlers bola len základom pre prechádzanie grafu. Trieda personObjects je jej prepracovanejšou verziou. V tejto triede musíme riešiť pohyb každého jedného aktéra simulácie, jeho vygenerovanie, následný výber cesty a pohyb v simulácii. Podrobnejšie si popíšeme jednotlivé funkcie tejto triedy.

Základné premenné v triede (zdedené z cShapeCrawlers):

- **startShapes** – zoznam všetkých cStartPointov simulácie;
- **shapes** – zoznam všetkých cProcessingPointov simulácie.

Najjednoduchšia funkcia tejto triedy je funkcia run a slúži na nastavenie parametra, ktorý ju informuje o tom, že simulácia beží a má animovať objekty.

Vo funkcii startNewPersons prejdeme všetky objekty uložené v zozname startShapes a následne zistíme, či sú k dispozícii vygenerovaní noví aktéri simulácie. Ak áno, použijeme na nich funkciu setNextPoint, pre zistenie, ku ktorému objektu majú smerovať. Podobnou funkciou bude funkcia continueOldPersons. V tejto funkcii prejdeme všetky objekty uložené v zozname shapes a postupujeme obdobne ako vo funkcii startNewPersons.

Vo funkcii setNextPoint spracujeme všetkých aktérov, ktorých sme tu dostali na vstupe. Zistíme, či sa majú pohybovať v simulácii alebo ju musia opustiť.

Algoritmus bude nasledovný:

1. Vyberieme aktéra zo vstupného zoznamu (pri každom z nich vieme zistiť objekt, v ktorom sa práve nachádza). Ak je zoznam prázdny, algoritmus ukončíme.
2. Pozrieme sa na aktuálny objekt, v ktorom sa práve aktér nachádza a zistíme pomocou funkcie getChildren, či existujú nejaké dcérske objekty. Ak nie, použijeme funkciu exitPerson daného objektu a tým dáme vedieť simulácii, že tento aktér ju v

danom bode opustil a vraciame sa do bodu 1. Inak pokračujeme ďalším bodom.

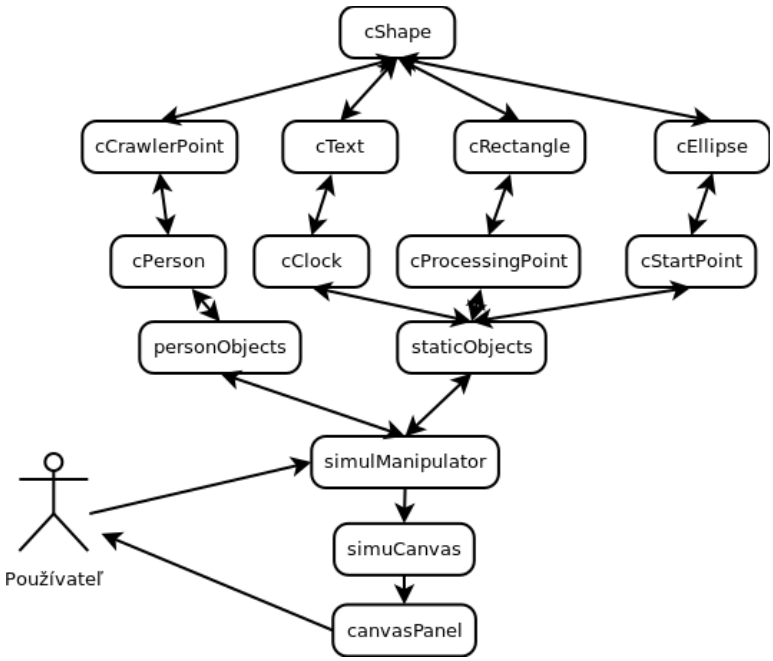
3. V aktuálnom objekte vykonáme funkciu getNextShapeToVisitByProbability, pomocou ktorej získame podľa nastavenej pravdepodobnosti bod, do ktorého má aktér smerovať. Daný bod uložíme do atribútu aktéra. Pokračujeme bodom 1.

Aj táto trieda bude obsahovať funkciu paint. Táto funkcia bude vykonaná pri každom tiknutí timera. Vždy tu budú volané funkcie startNewPersons a continueOldPersons. Taktiež tu overíme, či sa aplikácia nachádza v móde crawling. Ak nie, simulácia a pohyb budú zastavené.

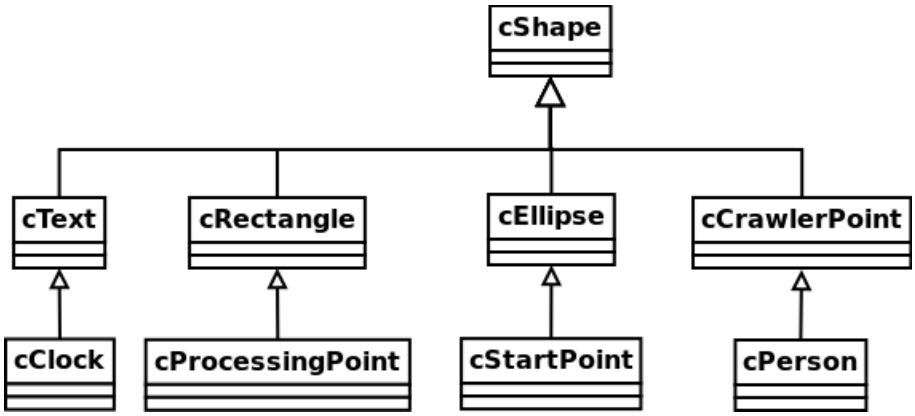
Zvyšok algoritmu si podrobne popíšeme:

1. Vyberieme aktéra zo vstupného zoznamu. Ak je zoznam prázdny, algoritmus ukončíme.
2. Pre aktuálneho aktéra zavoláme funkciu move s parametrom aktuálne nastavenej rýchlosti pohybu a následne funkciu paint pre vykreslenie. Týmto získavame animáciu pohybu aktéra.
3. Overíme, či sa nachádza v cieľovom bode pomocou funkcie aktéra inFinish. Ak je výsledok záporný, vrátime sa k bodu 1, inak pokračujeme bodom 4.
4. Pridáme aktéra do jeho cieľového bodu pomocou funkcie addPerson a pokračujeme bodom 1.

Pomocou nasledujúcich diagramov si znázorníme manipuláciu s grafickými objektami, ich vykresľovanie a dedičnosť vlastností grafických objektov od cShape až po jednotlivé ekonomické objekty.



Obrázok 20: Manipulácia s grafickými objektami a ich vykresľovanie



Obrázok 21: Class diagram pre grafické objekty

Balíček econSimul/savers

Bez možnosti ukladania simulácií by bolo používanie tohto simulátoru veľmi nepraktické. Preto sme sa rozhodli vytvoriť tento balíček, v ktorom budeme mať zjednodušenú reprezentáciu všetkých objektov. Ukladať budeme len tie najnutnejšie údaje, akými sú startPointy, processingPointy, ich parametre (aj časovo závislé parametre) a rýchlosť pohy-

bu aktérov v simulácii. Vygenerovaní aktéri a výsledky simulácie sa ukladať nebudú, keďže pri každom spustení danej simulácie by sme sa mali dopracovať k takmer zhodným výsledkom. Každá základná trieda, ktorá bude ukladaná, bude disponovať metódou save, v ktorej sa vytvorí jej obraz na základe jej ukladacej triedy. Nakoniec využijeme na uloženie stavu simulácie serializáciu a na načítanie stavu simulácie deserializáciu.

Ukladacie objekty našej aplikácie:

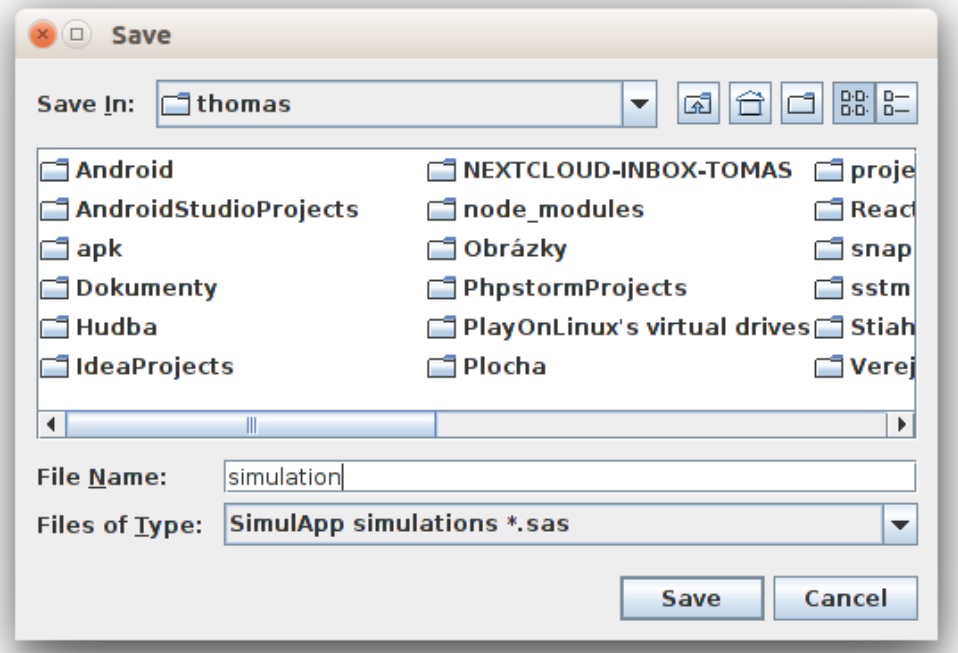
- **cShapeSaver** – ukladanie základných parametrov pre všetky grafické objekty;
- **cStartPointSaver** – ukladanie dôležitých parametrov pre startPoint, dedí parametre z cShapeSaver;
- **cProcessingPointSaver** – ukladanie dôležitých parametrov pre processingPoint, dedí parametre z cShapeSaver;
- **cShapesSaver** – základné ukladanie statických objektov;
- **staticObjectsSaver** – všetku funkcionalitu dedí od cShapeSavera, v budúcnosti poslúži na ukladanie ďalších špecifických parametrov.

Pri načítaní uloženej simulácie použijeme deserializáciu a získame obraz saverov z daného uloženého stavu aplikácie. Ten pomocou metódy load v cShapesSaver prevedieme na reálne objekty, vymeníme cManipulator novým prázdnym manipulátorom a doplníme do neho načítané objekty.

```
public void loadDialog(){
    JFileChooser fileChooser = new JFileChooser();
    FileNameExtensionFilter filter = new FileNameExtensionFilter("Simu-
lApp simulations *.sas", "sas");
    fileChooser.setFileFilter(filter);
    int returnValue = fileChooser.showOpenDialog(null);
    if (returnValue == JFileChooser.APPROVE_OPTION) {
        File selectedFile = fileChooser.getSelectedFile();
        if(selectedFile == null){
            return;
        }
        cShapesSaver shapes = null;
        try {
            FileInputStream fileIn = new FileInputStream(selectedFile);
            ObjectInputStream in = new ObjectInputStream(fileIn);
            shapes = (cShapesSaver) in.readObject();
            manipulator = new simulManipulator(this);
            manipulator.shapes.setShapes(shapes.load());
            manipulator.initClock();
            manipulator.crawler = new personObjects(this);
```

```
            mainGui.addingRadioButton.setSelected(true);
            in.close();
            fileIn.close();
        }catch(IOException i) {
            i.printStackTrace();
            showMessageDialog(null, "Error loading simulation.");
            return;
        }catch(ClassNotFoundException c) {
            showMessageDialog(null, "Error loading simulation.");
            c.printStackTrace();
            return;
        }
    }
```

Uložený stav simulácie budeme ukladať do súboru s koncovkou .sas (SimulApp simulation). Využijeme na to javový filechooser s filtrom pre danú koncovku. Ak bude súbor s rovnakým názvom pri ukladaní existovať, aplikácia upozorní používateľa na to, že súbor s rovnakým názvom už existuje.



Obrázok 22: SaveDialog

3 Verifikácia a možnosti rozšírenia

V tejto kapitole sa zameriame na možnosti použitia nášho riešenia. Otestujeme ho na základnej simulácii a porovnáme výsledky simulácie s podobným modelom v inom existujúcom simulačnom programe. Uistíme sa, či je naše riešenie funkčné a valídne. Zistíme, ako ďalej postupovať, či je potrebné opraviť nejaké časti softvéru a zameriame sa aj na možnosti rozšírenia do budúcna.

3.1 Problémy pri implementácii

Implementácia systému prebehla podľa predstáv a podarilo sa nám naprogramovať funkčné riešenie s možnosťou vizualizácie simulácie a jej výsledkov. Pri implementácii však nastalo aj niekoľko problémov.

Jedným z najväčších problémov bola implementácia prechádzania grafu. Nemali sme správne implementovaný timer a niektoré príkazy sa vykonávali duplicitne. To znamená, že sme mali spustený niektorý z algoritmov, ktorý ešte nedobehol do konca a už bol znova zavolaný timerom. V tom momente sme objekt zrušili, čo spôsobilo pád aplikácie, lebo sa snažila manipulovať s neexistujúcim objektom. Oprava tohto problému bola veľmi náročná, keďže nastával len sporadicky. Tým pádom sme na prvý pohľad nevedeli určiť, z akého dôvodu aplikácia padá. Najskôr sme museli zistiť, kde problém vzniká. Samotné riešenie už bolo jednoduché. Problém sme vyriešili opravou implementácie timera.

Ďalším problémom spojeným s časom bola chybná implementácia zrýchlenia času. Táto chyba sa prejavovala tak, že pri rôznych rýchlostiach behu simulovaného času sme v tej istej simulácii dostávali za rovnaké simulované obdobie iné výsledky. Chyba bola v tom, že pri zrýchlenom čase sa timer zvyšoval po väčších krokoch, nie len po jednotkách. Toto zvyšovanie sme následne jednotkovo spracovávali v cykle každého objektu. Takto sa nám za jeden tick mohlo v objekte vygenerovať, alebo spracovať dávkovo viac aktérov a získali sme veľmi skreslené výsledky. Riešenie nebolo až také zložité. Namiesto skokového zvyšovania timera sme implementovali jeho zvyšovanie v cykle, tým pádom sa korektne vykonala každá simulovaná minúta simulácie. Nová implementácia timera je znázornená v nasledujúcom kóde.

```
public void tick(Point mousePosition){
    if(mousePosition != null){
        moveCShape(mousePosition.x, mousePosition.y);
    }
    boolean ticked = false;
    int nextTime = clock.getTime() + clock.getTickSpeed();
    try {
        while (clock != null && clock.getTime() < nextTime && simulAp-
p.mainGui.getCurrentMode().equals(cManipulator.CRAWLING)) {
            ticked = true;
            shapes.tick();
            crawler.move();
            clock.setTime(clock.getTime() + 1);
        }
        if (ticked && clock.getTime() < nextTime && clock != null) {
            clock.setTime(0);
        }
    }catch (NullPointerException e){
        System.out.println("Clock reset");
    }
}
```

Posledným problémom v implementácii bola práca s aktérmi v cStartPointe a cProcessingPointe. Keď sme chceli odovzdať aktérov pomocou metódy getPersons a následne sme chceli vyprázdniť zásobník, v ktorom sa aktéri pôvodne nachádzali, neúmyselne sme zmazali aj aktérov, ktorí už boli spracovávaní v inej časti aplikácie. Táto situácia vyústila do pádu aplikácie. Tento problém sme vyriešili pomocou klonovania zásobníka s osobami. Vo funkciu getPersons sme si najskôr vytvorili klon týchto aktérov, ktorý sme poslali ďalej do aplikácie. Pôvodný zásobník, ktorý je súčasťou cStartPointu a cProcessingPoint zostal nezmenený a nezávislý na aktéroch odovzdaných na ďalšie spracovanie. Tento zásobník sme tým pádom mohli vyprázdniť a garbage collector sa postaral o ich odstránenie z pamäte bez toho, aby to vyústilo do pádu aplikácie.

3.2 Verifikácia aplikácie

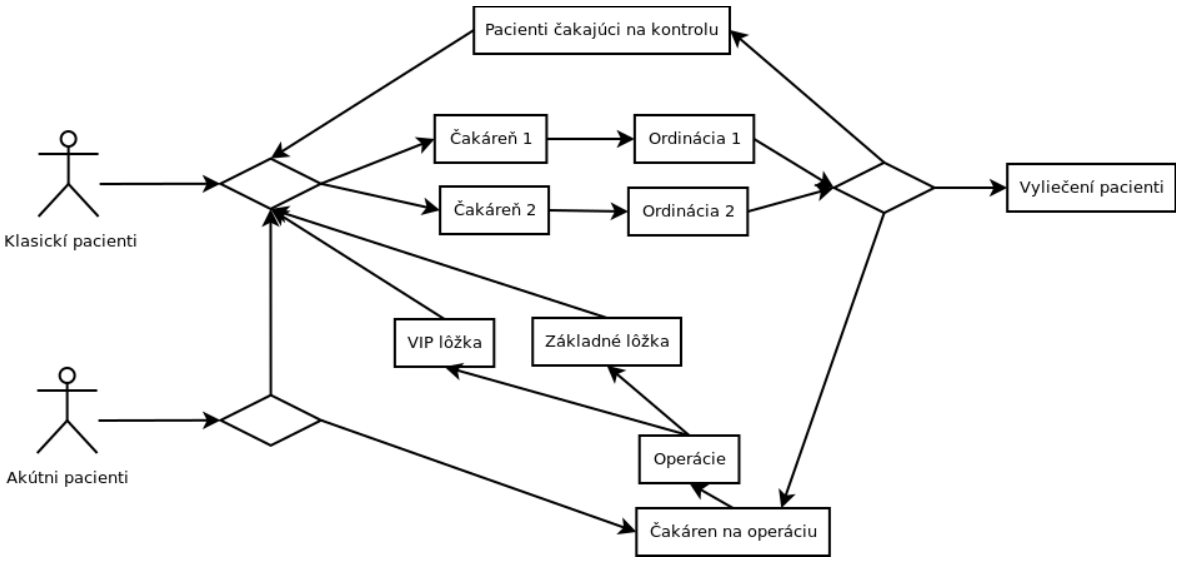
Po naprogramovaní prvej verzie aplikácie je nutné overiť jej výsledky. Preto bude nutné navrhnúť model, ktorý budeme schopní zostrojiť v našej aplikácii, ale aj v niektorej už existujúcej. Našu aplikáciu budeme verifikovať prostredníctvom aplikácie, s ktorou už máme skúsenosti, a tak sme sa rozhodli využiť SIMUL8. Pri modelovaní procesu je nutné zamerať sa aj na to, že naša aplikácia a SIMUL8 obsahujú rozdielne komponenty, no mi ich dokážeme poskladať tak, aby sme nasimulovali rovnaký proces.

3.2.1 Vytvorenie modelu pre verifikáciu

Ako model sme vybrali zjednodušený model zdravotného strediska. Zadefinujeme si základné časti strediska, otváracie hodiny, kapacitu a pod. Nasledovne vytvoríme konceptuálny model, na základe ktorého vytvoríme modely v našej aplikácii a v aplikácii SIMUL8.

Základné jednotky vystupujúce v modeli zdravotného strediska:

- dva druhy pacientov:
  - klasickí pacienti – budú prichádzať iba počas otváracích hodín;
  - akútni pacienti – budú prichádzať v menšom počte počas celého dňa;
- 3 základné časti zdravotného strediska:
  - 2 ordinácie s viacerými lekármi;
  - časť pre operácie pacientov;
  - lôžková časť:
    - základné lôžka;
    - drahšie VIP lôžka s menšou kapacitou;
- pacienti čakajúci na kontrolu;
- pacienti, ktorí sú úspešne vyliečení.



Obrázok 23: Konceptuálny model

Otváracie hodiny pre klasických pacientov budú od 06:00 do 14:00 hod. V našej aplikácii na zadefinovanie využijeme timeDepender v Start Pointe, kde si zadefinujeme, že osoby budú vstupovať po jednej a iba v čase od 06:00 do 14:00 hod. V SIMUL8 si musíme zadefinovať vlastnú časovo závislú distribúciu pre Start Point. V našej aplikácii je v súčasnosti použitý generátor náhodných čísel s uniformným rozdelením, preto pri SIMUL8 budeme voliť uniformnú distribúciu. Cez otváraciu dobu budú vstupovať pacienti v intervaloch od jednej minúty do pol hodiny. V našej aplikácii si to zadefinujeme jednoducho ako parametre Start Pointu, v SIMUL8 si vytvoríme obdobnú uniformnú distribúciu. Všetci títo pacienti budú postupovať do čakárne ambulancie, v ktorej je najmenší rad. Na to využijeme v našej aplikácii Next Probability, v SIMUL8 Routing Out s výberom možnosti Shortest Queue.

Akútni pacienti budú môcť vstupovať kedykoľvek, no bude ich o niečo menej. V tomto prípade neprihliadame na otváracie hodiny a v oboch programoch si zadefinujeme interval vstupu od 15 do 120 minút. 20% akútnych pacientov bude odoslaných rovno na operáciu, zvyšok pôjde do klasickej ambulancie.

Obe ambulancie budú zhodné. Každá z nich bude mať čakáreň. V SIMUL8 budeme musieť použiť prvok Queue pre každú z nich, v našej aplikácii je queue súčasťou Processing Centra. Ďalej si chceme zadefinovať silné a slabé hodiny v ordináciách, počas ktorých bude prítomných viac, alebo menej lekárov. Od polnoci do 06:00 hod. budú vo vnútri

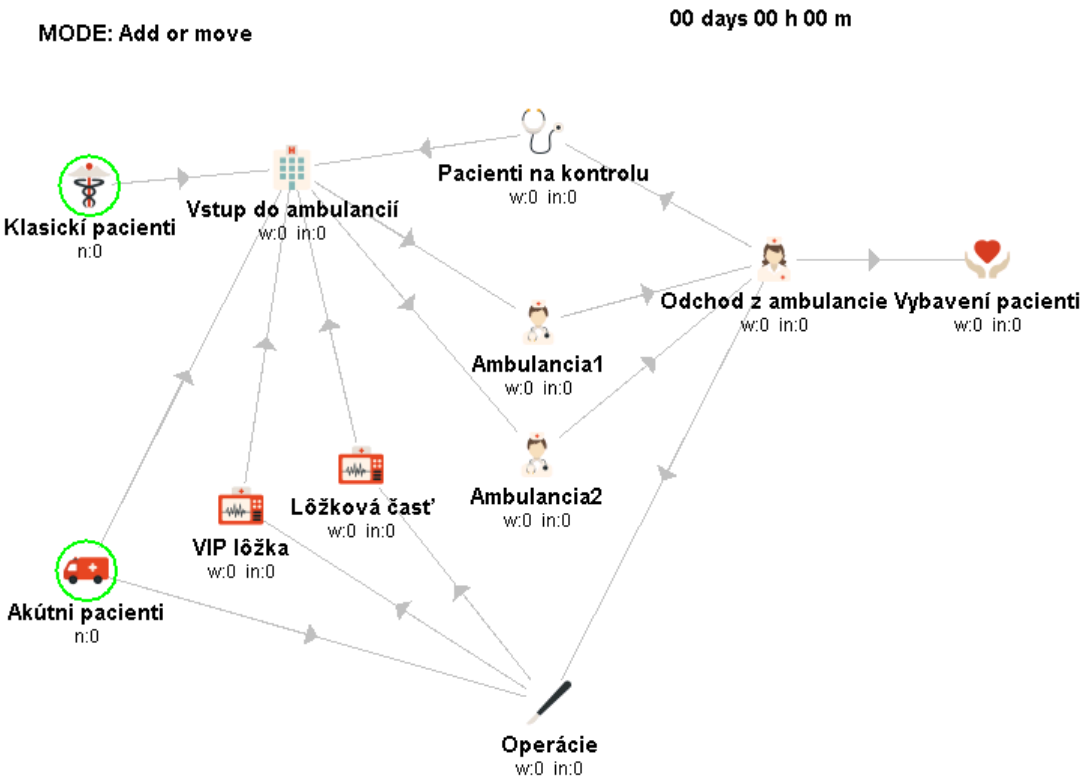
prítomní po dvaja lekári, od 06:00 do 12:00 hod. bude vo vnútri 5 lekárov, od 12:00 do 12:30 hod. bude obedná pauza, počas ktorej budú zas vo vnútri prítomní dvaja lekári. Od 12:30 do záverečnej o 14:00 hod. budú zas k dispozícii piati lekári pre každú ordináciu a od 14:00 hod. zas len dvaja lekári. Každý pacient strávi vnútri od 10 minút do jednej hodiny. V našom simulátore si ich zadefinujeme jednoducho ako capacity pomocou timeDependera a čas, ako základné vlastnosti Processing Centra. V SIMUL8, to budeme mať o čosi zložitejšie. Pre každú z ordinácii využijeme 3 Activity. Prvá nám bude rozdeľovať pacientov podľa času do nasledujúcich dvoch. Jedna z nich bude smerovať do Queue s kapacitou 2, druhá tiež a okrem toho aj do ďalšieho, ktorý bude mať kapacitu pre 3 aktérov s Routing Out nastaveným na Shortest Queue.

Nasledujúcim bodom bude odchod z ordinácií. V našom simulátore na to využijeme Processing Center s nulovým zdržaním, v SIMUL8 Activity taktiež s nulovým zdržaním. Z tohto bodu budú aktéri pokračovať do troch bodov. Najčastejšie, so 70% pravdepodobnosťou, to bude do výstupného bodu simulácie, 20% pacientov bude poslaných domov na PN, kde strávia týždeň. V našom simulátore to bude Processing Centra s týždňovým zdržaním, v SIMUL8 to bude Queue s týždňovým zdržaním. Odtiaľ sa zas vrátia na kontrolu do ordinácie. Posledných 10% aktérov pôjde na operáciu.

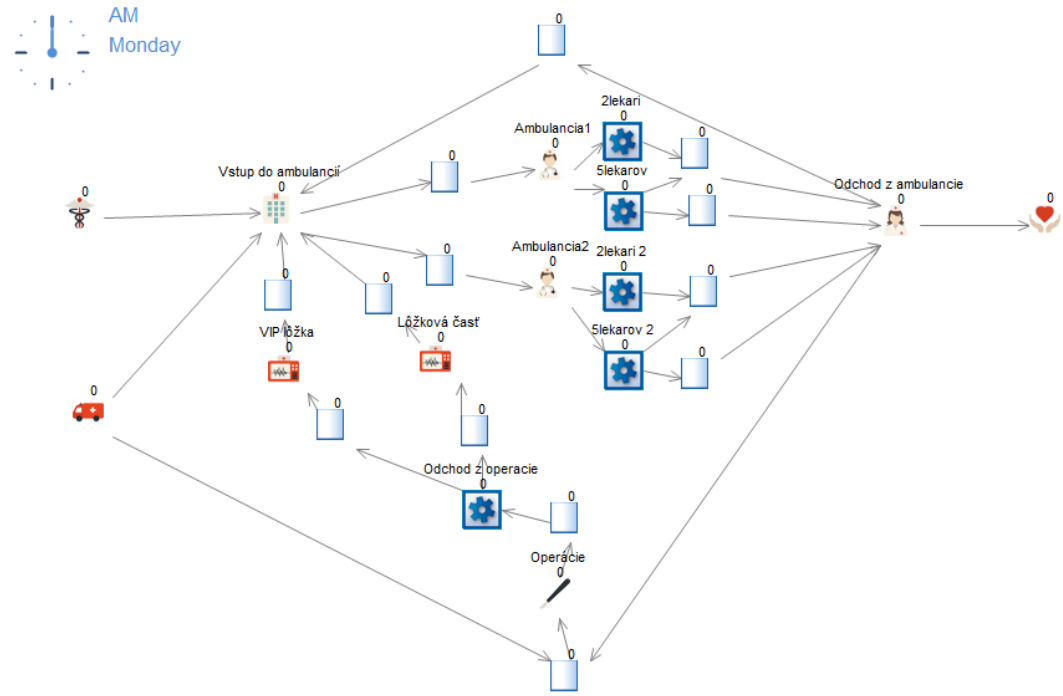
V operačnej časti bude tiež čakáreň a operovaní budú môcť byť maximálne dvaja pacienti súčasne. Každá operácia bude trvať od 10 do 60 minút. V našom simulátore to dosiahneme nakonfigurovaním Processing Centra, v SIMUL8 na to použijeme Queue pre čakáreň, potom Activity pre vstup na operáciu, za ktorou bude ďalšie Queue s kapacitou 2 a zdržaním od 10 do 60 minút. V SIMUL8 použijeme ešte jednu Activity na odchod z operácie a na následné smerovanie aktérov do ďalších častí.

Z operácie budú pacienti pokračovať do lôžkovej časti, pričom 20% pôjde do VIP časti a zvyšok do štandardnej lôžkovej časti. VIP bude mať kapacitu 20 a klasická lôžková časť bude mať kapacitu 100. Aktéri sa tu zdržia od 1440 do 14400 minút. V našom simulátore na to využijeme dve Processing Centrá, v SIMUL8 na to využijeme Activity v kombinácii s Queue so zdržaním. Z lôžkovej časti budú pacienti pokračovať ďalej do lekárskych ordinácií.

Takto sme si vytvorili dva rovnako fungujúce modely v našom simulačnom programe SimulApp a v programe SIMUL8. Tieto 2 simulácie budú k dispozícii aj v prílohe.



Obrázok 24: Simulácia v SimulApp. Obrázky použité v simulácii dostupné na internete: http://www.flaticon.com/authors/freepik



Obrázok 25: Simulácia v SIMUL8. Obrázky použité v simulácii dostupné na internete: http://www.flaticon.com/authors/freepik

3.2.2 Porovnanie výsledkov simulácií

Pri porovnávaní výsledkov simulácií sa zameriame na porovnanie počtov generovaných aktérov a počtov aktérov, ktorí prešli jednotlivými časťami simulácie. Nakoniec porovnáme, koľko aktérov je aktuálne pri skončení simulácie v daných častiach modelu. Tieto časti simulácie je najjednoduchšie porovnať v rámci nášho simulačného programu a SIMUL8. Ostatné merané jednotky sú v našom programe algoritmicky riešené veľmi podobne, tým pádom ich nebude treba overovať zvlášť. Vybrali sme si dvojtýždňový časový úsek, na ktorom uskutočníme 10 meraní.

Merať budeme nasledujúce časti modelu:

- klasický pacienti – počet generovaných aktérov;
- akútni pacienti – počet generovaných aktérov;
- vyliečení pacienti – počet aktérov, ktorí tu opustili simuláciu;
- pacienti, ktorí čakajú na kontrolu – počet, koľko ich tu je po skončení simulácie;
- VIP lôžka – počet, koľko ich je využívaných na konci simulácie;
- klasické lôžka – počet, koľko ich je využívaných na konci simulácie;
- ambulancia 1 – počet aktérov, ktorí v nej boli na vyšetrenie od začiatku simulácie;
- ambulancia 2 – počet aktérov, ktorí v nej boli na vyšetrenie od začiatku simulácie;
- operácie – počet operovaných pacientov od začiatku simulácie;

V nasledujúcich tabuľkách sú výsledky desiatich simulácii, spustených po sebe. V prílohe sú tieto hodnoty aj graficky znázornené. Priemerné výsledky z viacerých simulácií sú si veľmi podobné a tým pádom sme úspešne verifikovali náš simulačný program.

	Klasickí pacienti		Akútni pacienti		Vyliečení pacienti	
1.	422	431	302	300	568	585
2.	423	448	291	301	583	592
3.	427	428	300	301	558	560
4.	428	430	292	295	574	571
5.	431	415	293	303	571	582
6.	409	412	290	288	555	545
7.	427	422	286	289	568	561
8.	424	433	311	306	570	572
9.	439	422	296	296	582	564
10.	433	425	304	297	580	562
Priemer	**426,3**	**426,6**	**296,5**	**297,6**	**570,9**	**569,4**
	SIMUL8	**SimulApp**	**SIMUL8**	**SimulApp**	**SIMUL8**	**SimulApp**

Tabuľka 4: Výsledky simulácií (1)

	Pacienti na kontrolu		Využité VIP lôžka		Využité klasické lôžka	
1.	99	88	9	10	45	45
2.	77	100	12	9	41	47
3.	97	98	13	11	55	53
4.	82	96	13	14	51	41
5.	90	87	7	10	51	37
6.	85	89	18	14	39	44
7.	94	84	8	18	47	48
8.	100	107	18	13	43	43
9.	89	92	10	11	48	50
10.	88	93	10	17	52	46
Priemer	**90,1**	**93,4**	**11,8**	**12,7**	**47,2**	**45,4**
	SIMUL8	**SimulApp**	**SIMUL8**	**SimulApp**	**SIMUL8**	**SimulApp**

Tabuľka 5: Výsledky simulácií (2)

	Prešli ambulanciou 1		Prešli ambulanciou 2		Počet operovaných	
1.	397	428	416	396	140	140
2.	405	447	397	408	132	135
3.	416	384	406	427	145	137
4.	421	423	390	378	145	124
5.	415	414	396	394	140	133
6.	400	379	383	407	140	136
7.	422	423	366	377	129	135
8.	413	393	413	441	136	145
9.	416	421	403	389	147	139
10.	421	418	390	386	129	141
Priemer	**412,6**	**413**	**396**	**400,3**	**138,3**	**136,5**
	SIMUL8	**SimulApp**	**SIMUL8**	**SimulApp**	**SIMUL8**	**SimulApp**

Tabuľka 6: Výsledky simulácií (3)

3.3 Vízia do budúcnosti

V aktuálnej verzii aplikácie sme schopní vytvoriť základný model podnikového procesu. Aplikácia je pripravená na budúce rozšírenie, máme naprogramovaných niekoľko algoritmov, ktoré sa zatiaľ nevyužívajú. S využitím existujúcich objektov a dedičnosti Javy vieme jednoducho vytvoriť nové objekty.

3.3.1 Grafika

Pri vytváraní simulácií v našej aplikácií sme prišli na niekoľko nedostatkov, ktoré by bolo vhodné vylepšiť. Prvým z nich je nemožnosť priblížiť alebo oddialiť si simuláciu. Pri väčších modeloch by bolo vhodné mať možnosť objekty si zmenšiť. Ďalšou možnosťou vylepšenia do budúcnosti je lepšia vizualizácia naplnenia Prociessincg Centier. Bolo by vhodné použiť podobnú vizualizáciu, aká je prítomná v aplikácii SIMUL8 pri Queue.

Pri programovaní sme navrhli grafické objekty, ktorým sa dajú nastavovať rozmery. No zatiaľ len pomocou zdrojového kódu. Bolo by vhodné doprogramovať rozhranie pre nastavenie veľkosti jednotných objektov. Pre samotnú simuláciu nie je táto časť veľmi dôležitá, ale bola by vhodná pre vizuálnu stránku nášho simulátora.

Poslednou grafickou súčasťou, ktorú by bolo v budúcnosti vhodné doprogramovať,

je ukladanie celých obrázkov priradených k objektom simulácie do uložených simulácií. Momentálne je v súbore .sas uložená iba cesta k obrázku. Ak otvoríme túto uloženú simuláciu na počítači, kde sú iné cesty k daným obrázkom, prípadne v danom počítači nie sú uložené dané obrázky, tak sa nám načíta simulácia bez priradených obrázkov. Tým pádom nám už len ostáva ručne spätne nastaviť obrázky jednotlivým objektom.

3.3.2 Simulácia

Pri samotnej simulácii by bolo v budúcnosti vhodné doprogramovať rozhranie pre viac distribúcií náhodných čísel. Niektoré distribúcie už máme naprogramované v balíčku util, no nemáme k nim naprogramované vhodné rozhranie. Momentálne generujeme náhodné čísla pomocou uniformného rozdelenia.

Každé z Procesing Centrum v našom simulátore obsahuje aj rad pre čakajúcich aktérov. No tento rad je momentálne neobmedzenej dĺžky. Preto by bolo vhodné v budúcnosti implementovať možnosť nastavenia maximálnej kapacity radu. Zatiaľ na to môžeme využiť nastavenie kapacity Processing Centra a tento Processing Center používať na reprezentáciu samotného radu čakajúcich aktérov.

V simulácii zbierame veľké množstvo výsledkov, no nie ku všetkým sa vieme dopracovať cez rozhranie. Napríklad by bolo vhodné implementovať možnosť prejsť si výsledky vhodne zoskupených aktérov podľa toho, ktorými časťami simulácie prešli.

A ako poslednú možnosť na vylepšenie sa pozrieme na dĺžku cesty medzi bodmi simulácie. Bolo by vhodné prepočítavať dĺžku cesty aktérov medzi bodmi simulácie podľa vzdialenosti objektov a nastavenej rýchlosti simulácie, a tým pádom by sme ju mohli zaradiť do výsledkov simulácie. V tomto momente na ňu neprihliadame.

Záver

V záverečnej práci sme sa zamerali na návrh a tvorbu simulačného nástroja v programovacom jazyku. Zaoberali sme sa výberom vhodného vývojového prostredia a návrhom samotnej aplikácie. Aplikáciu sme programovali v jazyku Java s využitím sady nástrojov Swing. Pre programovanie sme si vybrali pokročilé vývojové prostredie IntelliJ IDEA od českej firmy JetBrains a programovali sme na operačnom systéme Linux. Všetky tieto nástroje sú bezplatné, no napriek tomu dostatočne kvalitné pre plnohodnotnú prácu.

Naprogramovali sme aplikáciu, ktorú sme schopní spustiť na akomkoľvek operačnom systéme a v každom systéme sa správa po funkčnej aj vizuálnej stránke rovnako. Aplikáciu sme otestovali pod operačnými systémami Windows, MAC OS a Linux.

Celú aplikáciu sme rozdelili do viacerých logických častí a každú z nich sme podrobne popísali. Takto bude možné aplikáciu veľmi jednoducho dopĺňať o novú funkcionalitu. Do aplikácie sme naprogramovali rôzne knižnice, ktoré ešte naplno nevyužíva, no v budúcnosti nám dokážu uľahčiť prácu pri dopĺňaní funkcionality. Väčšina kódu v aplikácii bola programovaná s dôrazom na budúce opätovné využitie.

Pri programovaní aplikácie sme sa zamerali na to, aby sme mohli simulovať situácie z reálneho života, aby sme v aplikácii mohli vytvárať prvky, ktorých vlastnosti sa budú môcť jednoducho meniť v závislosti od času. Pri tom sme sa snažili dodržať dostatočnú podobnosť tvorby simulácie so simuláciou vytváranou v komerčnom programe SIMUL8, aby sme dokázali vytvoriť zhodné modely a porovnať ich.

Po naprogramovaní aplikácie sme sa zamerali na návrh modelu ekonomického procesu. Na modelovanie sme si vybrali návrh malej kliniky s dvomi druhmi pacientov, lôžkovou časťou a časťou, kde sa budú vykonávať operácie. Kládli sme dôraz na to, aby sme v simulácii využili čo najviac prvkov a nastavení, ktorými naša aplikácia disponuje. Takto sme vytvorením jedného modelu dokázali overiť veľkú časť funkcionality našej aplikácie.

Tento model sme následne vytvorili aj v programe SIMUL8. Toto sa nám podarilo s využitím viacerých prvkov, keďže sú v našej aplikácii niektoré prvky riešené odlišne ako v SIMUL8. Základný prvok nášho simulačného programu si môžeme predstaviť ako spojenie dvoch prvkov programe SIMUL8, no pri vhodnom nakonfigurovaní dokáže

vstupovať ako jeden konkrétny prvok v SIMUL8.

Následne sme si určili, ktoré výstupy pre nás budú pri simulácii dôležité. Tieto výstupy sme si pri viacnásobnom spúšťaní simulácie zaznamenávali. Potom sme ich zapísali do prehľadných tabuliek a následne vizualizovali pomocou grafov. Porovnaním týchto meraní sme zistili, že výsledky oboch simulácií sa k sebe veľmi približujú. Vďaka týmto výsledkom sme zhodnotili, že simulácia v našej aplikácii sa dokáže dostatočne priblížiť simulácii programu SIMUL8. Takto sa nám podarilo úspešne overiť funkčnosť nášho simulačného programu.

Výsledkom našej práce je funkčný simulačný program SimulApp, v ktorom dokážeme simulovať základné ekonomické procesy. Táto práca ukázala, čo všetko je spojené s programovaním vlastnej simulácie priamo v programovacom jazyku a že tvorba podobného softvéru je možná aj bez väčšej investície finančných prostriedkov. Veríme, že táto aplikácia sa v budúcnosti stane základom pre ďalší vývoj.

Zoznam použitej literatúry

Knižné zdroje

DLOUHÝ, Martin, Jan FÁBRY a Martina KUNCOVÁ. Simulace podnikových procesů. Vyd. 1. Brno: Computer Press, 2007. ISBN 978-80-251-1649-4

SCHILDT, Herbert. Java 7, Výukový kurz. Brno: Computer Press, 2012. ISBN 978-80-251-3748-2.

Online zdroje

TESAŘ, Jiří; BARTOŠ, Petr. České Budějovice: Pedagogická fakulta Jihočeské univerzity v Č. Budějovicích, Katedra fyziky, [cit. 2017-03-30]. Dostupné na internete: <http://ds-p.vscht.cz/konference_matlab/MATLAB06/prispevky/tesar_bartos/tesar_bartos.pdf>

NOVOTNÝ, Luděk. Masarykova Univerzita, Fakulta Informatiky, Historie a vývoj jazyka Java, [cit. 2017-04-02]. Dostupné na internete: <http://www.fi.muni.cz/usr/jkucera/pv109/2003p/xnovotn8.htm>

Hradec Králové: Univerzita Hradec Králové, Pedagogická fakulta, Katedra fyziky a informatiky, [cit. 2017-03-30]. Dostupné na internete: <http://www.black-hole.cz/soubory/mc.pdf>

CACI Products, [cit. 2017-03-30]. Dostupné na internete: <http://simprocess.com/about-simprocess/ >

SIMUL8 Corporation, [cit. 2017-03-30]. Dostupné na internete: <https://www.simul8.com/>

ROSETTA CODE, [cit. 2017-04-02]. Dostupné na internete: <https://rosettacode.org/wiki/Language_Comparison_Table>

ORACLE, Java editions, [cit. 2017-03-31]. Dostupné na internete: <http://www.oracle.com/technetwork/java/index.html>

ORACLE, Serializable Objects, [cit. 2017-03-31]. Dostupné na internete: <https://docs.oracle.com/javase/tutorial/jndi/objects/serial.html>

BOLTON, David.Three Java IDEs Compared, [cit. 2017-03-28]. Dostupné na internete: <http://insights.dice.com/2013/10/24/three-java-ides-compared-147/>

VLATKO, Natali. An overview of the top Java IDEs, [cit. 2017-03-28]. Dostupné na internete: <https://jaxenter.com/the-top-java-ides-114599.html>

FOURNOVA SOFTWARE, Learn Version Control with Git, [cit. 2017-03-28]. Dostupné na internete: <https://www.git-tower.com/learn/git/ebook/en/desktop-gui/appendix/from-subversion-to-git>

GEEKFORGEEKS, Graph and its representations, [cit. 2017-03-28]. Dostupné na internete: <http://www.geeksforgeeks.org/graph-and-its-representations/>

Zoznam príloh

Príloha č. 1: Zoznam tabuliek

Príloha č. 2: Grafy porovnania simulácií v SIMUL8 a SimulApp

Príloha č. 3: Zdrojové kódy aplikácie (na CD)

Príloha č. 4: Skompilovaná aplikácia vo formáte .jar (na CD)

Príloha č. 5: Simulácie zo SIMUL8, ambulancia.S8 a SimulApp, ambulancia.sas (na CD)