UNIVERZITA PARDUBICE
Fakulta elektrotechniky a informatiky

Vytvoření distribuovaného simulačního modelu jednoduchého logistického systému

Vojtěch Samotán

Bakalářská práce
2014

Prohlášení autora

Prohlašuji, že jsem tuto práci vypracoval samostatně. Veškeré literární prameny a informace, které jsem v práci využil, jsou uvedeny v seznamu použité literatury.

Byl jsem seznámen s tím, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorský zákon, zejména se skutečností, že Univerzita Pardubice má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona, a s tím, že pokud dojde k užití této práce mnou nebo bude poskytnuta licence o užití jinému subjektu, je Univerzita Pardubice oprávněna ode mne požadovat přiměřený příspěvek na úhradu nákladů, které na vytvoření díla vynaložila, a to podle okolností až do jejich skutečné výše.

Souhlasím s prezenčním zpřístupněním své práce v Univerzitní knihovně.

V Pardubicích dne 25. 4. 2014 Vojtěch Samotán

Poděkování

Rád bych poděkoval panu Ing. Josefu Brožkovi za pomoc a rady a věcné připomínky při vytváření této bakalářské práce.
Poděkování patří i mé rodině a přátelům za morální podporu.

Anotace

Bakalářská práce se zabývá vypracováním reálného simulačního modelu, implementovaného v programovacím jazyku Java s využitím distribuované simulace dle standardu HLA.

Klíčová slova

Modelování a simulace, Java, HLA, distribuovaná simulace.

Title

Creating a distributed simulation model of a simple logistics system

Annotation

The bachelor thesis deals with the design of the real simulation model implemented in the programming language Java while using the distributed simulation according to the HLA standard.

Keywords

Modeling and Simulation, Java, HLA, distributed simulation.

1 Obsah

Seznam zkratek

HLA	High Level Architttecture
RTI	Runtime Infrastucture
OMT	Object Model Templatr
MOM	Management Object Model
JVM	Java Virtual Machine
GUI	Graphical User Interface
JDC	Java Database Connectivity
OOP	Objektově Orientované Programování
UML	Unified Modeling Language
pRTI	Pitch RunTime Infrastructure

Seznam obrázků

Seznam tabulek

Úvod

Tématem této bakalářské práce je vytvořit reálný simulační model vybraného, logistického systému. Jako simulační model byla vybrána budova staré polikliniky ve Dvoře Králové nad Labem, který popisuje běžný provozní chod zdravotnického zařízení.

Práce je rozdělena na teoretickou a praktickou část.

Úvod teoretické části bakalářské práce se zabývá pojmy z modelování a simulace a programovacím jazykem Java. Dále je zde shrnuta historie jazyka a použité objekty v praktické části. Následně je popsán standart HLA, jeho historie, RTI a pravidla HLA pro federace a federáty.

V praktické části se práce věnuje měření dat na poliklinice ve Dvoře Králové nad Labem. Pro sběr dat muselo být vyjednáno povolení od majitele budovy pana Hampla, kterému tímto děkuji. Měření v budově bylo realizováno ve třech dnech, vždy od otevření budovy v 07:30 do 11:30. Sběr dat proběhl ve spolupráci s kolegou Lukášem Bašem, který model zpracoval v prostředí Arena. Kvůli rozlehlosti budovy bylo potřeba pro sběr dat čtyř lidí. V budově byly měřeny jednotlivé časy výstupu osob do pater, doba chůze po chodbách, intervaly mezi příchody osob, doby strávené v ordinacích, čas jízdy výtahem, doba čekání na výtah, atd. Dále se práce zabývá zpracováním těchto naměřených dat na histogramy, ze kterých je sestaveno časové rozdělení pro model. Práce také obsahuje vytvoření simulačního modelu v jazyce Java a popis metod použitých v modelu. Na závěr je popsáno vytvoření distribuované simulace s využitím frameworku HLA-VA.

2 Modelování a simulace

Chceme-li vytvořit simulační model, musíme napřed definovat několik termínů. Ve své práci vycházím z pojetí, ke kterému se přiklání literatura [10].

Modelování je proces tvorby modelu, tedy náhrada zkoumaného systému jeho modelem. Simulace je provádění experimentů s cílem lepšího pochopení chování simulovaného systému. Modelování a simulace využívá numerických metod, které využívají číslicových počítačů.

Důvod k tvorbě simulačního modelu je ten, že provádění experimentů s reálným objektem, např. automobilem, může být velmi časově náročné a nebezpečné. Při vytvoření simulačního modelu, můžeme s modelem experimentovat dle vlastního uvážení a velmi bezpečně (nehrozí zničení reálného simulovaného objektu), nehledě na časovou nenáročnost při provádění experimentů.

2.1 Základní pojmy

- **Systém -** je uskupení (soubor) prvků nebo objektů, které tvoří jednotný celek a mají mezi sebou určité vazby, které se navzájem mohou ovlivňovat. Rozlišujeme systémy:
  a) reálné systémy
  b) nereálné systémy (fiktivní systémy)

- **Model** - slovo model má v běžné řeči význam předlohy. Model je tedy abstrakce simulovaného reálného objektu, využívá zjednodušení složité reality modelovaného objektu.

- **Modelování** - je proces tvorby modelu, tedy náhrada originálu modelem, kde každému prvku originálu je přiřazen prvek modelu a každému atributu prvku originálu je přiřazen atribut prvku modelu.

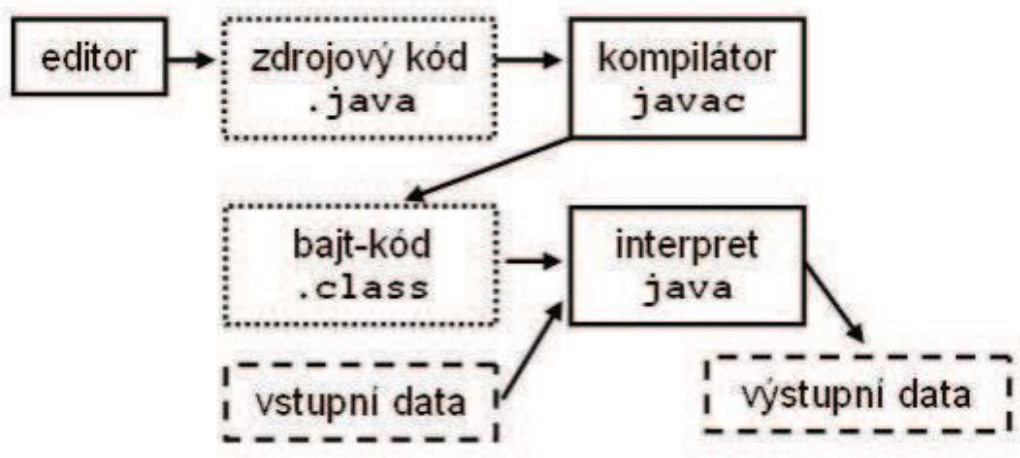
- **Simulace** - je technika, jejíž podstatou je náhrada zkoumaného systému jeho simulátorem s cílem získat informace o originálu.

- **Validace modelu** - je porovnání modelu s realitou, jestli známé výsledky odpovídají realitě a pro známé scénáře model funguje správně.

- **Verifikace modelu**- ověření správnosti modelu, zda prvky modelu odpovídají prvkům modelovaného objektu.

3 Programovací jazyk Java

Pro vytvoření simulačního modelu byl vybrán programovací jazyk Java. V této kapitole bude popsán programovací Java a použité objekty v projektu. Java je objektově orientovaný jazyk vyšší úrovně a je jedním z nejpoužívanějších jazyků na světě, zejména díky své přenositelnosti a nezávislosti na architektuře. Java totiž bude stejně fungovat na jakémkoliv operačním systému, aplikace, která je vytvořená v Javě, se spouští na Java virtuálním stroji (JVM). Další z klíčových vlastností Javy je distribuovanost, která podporuje aplikace v síti (práce se vzdálenými soubory umožňuje vytvářet distribuované aplikace - vestavěná podpora TCP/IP). Dalšími vlastnostmi jsou robustnost a bezpečnost. Java je jazykem hybridním, tj. jazykem interpretovaným a zároveň i kompilovaným. Program se nejprve přeloží bajtkódu a pak až následně je interpretován JVM.



Obrázek 1: Překlad a interpretace v jazyce Java Zdroj[1]

3.1 Historie

Programovací jazyk Java byl vyvinut na začátku 90. let společností Sun Microsystems, která si dala za úkol vytvořit snadný, ale efektivní a objektově orientovaný programovací jazyk, vytvořený modifikací jazyků C a C++. Původním názvem tohoto programovacího jazyka byl Oak (anglicky dub), podle stromu, který rostl před okny hlavního vývojáře Jamese Goslinga. Když později firma zjistila, že tento název je již zabrán pro jiný programovací jazyk, přejmenovali svůj jazyk na Java, tj. americký slangový výraz pro kávu, která se stala i logem jazyku. V roce 1995 byla Java oficiálně představena na konferenci ve verzi 1.0, která obsahovala pouhých 211 veřejných tříd. V roce 1995 začal také masově pronikat Internet k běžným lidem a Java se tak stala záhy velmi populární, zejména díky Java appletům.

Verze 1.1 se objevila 2 roky od oficiálního uvedení. Java 1.1 přinesla řadu vylepšení, a to vnitřní a vnořené třídy a upravené knihovny. Počet veřejných tříd se rozrost na 477.

Další evolucí byla verze 1.2 s kódovým jménem Playground, která se objevila v roce 1998. Tato nová verze přinesla přepracovanou knihovnu a přidání řady dalších knihoven. Počet veřejných tříd vzrost na 1524. Tato nová verze se začala označovat kvůli značné přepracovanosti jako Java 2.

Ve verzi 1.3 s kódovým označením Kestrel, která se objevila v roce 2000, autoři pokračovali ve vývoji a verze přinesla další rozšíření knihovny. Počet veřejných tříd opět stoupl, a to na hodnotu 1840.

Verze jazyka 1.4 s označením Mertlin přidala například podporu internetového protokolu IPv6 a klíčové slovo assert, které se používalo k ladění programu. Počet veřejných tříd po opětovném rozšíření knihovny vzrostl na 2730.

V roce 2004 byla vydána další verze, a to verze 1.5 s kódovým jménem Tiger, ve které se opět rozšířila knihovna. Počet veřejných tříd stoupl na 3270. Dalším vylepším je například import statických objektu, rozšíření příkazu for a rozšíření znakové sady Unicode. Tato verze je ale častěji v publikacích označována jako Java 5.0.

Další vývoj Javy byl v roce 2006 k verzi 1.6 (Java 6) s kódovým jménem Mustang. Tato verze přinesla například podporu Java Database Connectivity (JDC) ve verzi 4.0, vylepšení grafického uživatelského prostřední (GUI). Java 6 obdržela celkem 71 aktualizací, které opravují především chyby v bezpečnosti. Tyto opravy jsou vydávány až do roku 2014.

V roce 2009 byla firma Sun Microsystems odkoupena firmou Oracle, a tak verze Java 1.7 (Java 7) s kódovým jménem Dolphin, je již vydaná samotnou firmou Oracle v roce 2011. Nová verze přináší například možnost využití datového typu String v příkazu switch, binární celočíselné literály a vylepšení odchytávání výjimek.

Zatím poslední verzí je Java 1.8 (Java 8) s kódovým označením Spider, která vyšla v březnu roku 2014. Vydání další verze Javy je plánována na rok 2016.

3.2 Objektově orientované programování

První objektově orientovaný jazyk se objevil v 60. letech se jménem Simula 67. Tento programovací jazyk se využíval k diskrétním simulacím. Dalším jazykem, který stavěl na myšlence jazyka Simula, byl jazyk SmalTalk. Díky jazyku SmalTalk se OOP rozšířilo na akademickou půdu a k vědcům. Ovšem skutečný rozmach OOP nastal až v Bellových laboratořích, kde Bjarne Stroustrup modifikoval jazyk C a nazval ho jazyk C with classes (C s třídami). Tento programovací jazyk byl v roce 1983 přejmenován na C++.

Programovací jazyk Java je objektově orientovaný a přenositelný na jiné platformy, tedy pracuje s objekty. Objekt je abstrakcí z reálného světa, kde každý objekt má svoje vlastnosti a metody. Podíváme-li se kolem sebe, uvidíme spoustu objektů z reálného světa.

Například objekt televizor, který má pouze dvě proměnné, a to zapnuto a vypnuto, analogicky jsou jeho metody zapnout a vypnout.

3.3 Třídy

Ve větších projektech se objevují tisíce až desetitisíce objektů, proto nastává potřeba je nějak roztřídit do skupin. Mohou existovat objekty se společnými vlastnostmi. Například může existovat tisíce jízdních kol, která jsou jako jeden model od stejného výrobce, vyrobeny stejně. Každé kolo je vyrobeno podle stejného scénáře a mají totožné součásti. V objektově orientované terminologii můžeme říct, že právě jedno jízdní kolo je instancí jedné třídy objektů, známé jako jízdní kola. V třídách jsou definovány společné vlastnosti jejich instancí a také nástroje pro jejich vytváření. Budeme-li mít již zmíněnou třídu Kola, tak jakékoliv kolo bude instancí třídy Kola. Třída Kola bude popisovat jejich vlastnosti např. počet míst, rychlost, převody, apod.

```
class Kola {

 int prevod = 0;
 int rychlost = 0;
 int pocetMist = 1;

 void zmenaPrevod(int novaHodnota) {
      cadence = novaHodnota;
 }

 void zmenaPocetMist (int novaHodnota) {
      gear = novaHodnota;
 }

 void zrychli(int rychlost) {
      speed += rychlost;
 }

 void zabrzdi(int zpomaleni) {
      speed -= zpomaleni;
 }

 void printStates() {
      System.out.println("prevod:"+prevod+" rychlost:"+rychlost+
          "       pocetMist:"+pocetMist);
 }
  }
```

Příklad třídy Kola, která popisuje pouze vlastnosti *prevod, rychlost, pocetMist*. Dále třída obsahuje metody na změny těchto vlastností. Změna zařazeného převodu metodou *zmenaPrevodu*, která má jako vstupní argument novou celočíselnou hodnotu zařazeného stupně, přepisuje starou hodnotu. Metodu na zrychlení *zrychli*, která o vložený argument inkrementuje rychlost a úplnou analogii, je metoda na zpomalení *zabrzdi*, která o určitý argument zabrzdí.

3.3.1 Rozhraní

Rozhraní (Interface) definuje množinu metod, která může být implementovaná celou třídou. Interface pouze metody popisuje, samotná implementace se provádí v rámci tříd.

V Javě neexistuje vícenásobná dědičnost, to znamená, že třída může dědit vždy maximálně od jednoho objektu. Avšak člověk se často ocitne v případě, že potřebuje využít vícenásobnou dědičnost. Pro tento případ třída může implementovat libovolný počet rozhraní, které nám definuje množinu metod. To nám do jisté míry nahrazuje více násobnou dědičnost.

```
19 public interface ICasPrichodu<E>extends Comparable<E> {
20
21 public Calendar dejCas();
22
23
24 }
```

Ukázka rozhraní vytvořeného v praktické části.

3.3.2 Návrhový vzor Singleton

Singleton, neboli česky jedináček, je návrhový vzor, který umožňuje přístup k instanci třídy. Singleton se využívá v případě, že potřebujeme, aby v programu existovala jen jedna instance dané třídy.

```
24 public class Generator {
25
26 private Random generator = null;
27 private static Generator g = null;
28
29 private Generator() {
30 this.generator = new Random(50000);
31     }
32
33 public static Generator getInstance() {
34 if (g == null) {
35             g = new Generator();
36         }
37 return g;
38     }
```

Příklad Singletonu využité v třídě Generátor.

To, že v programu bude pouze jedna instance, nám zajistí metoda *getInstance()*. Metoda zajistí v případě, že generátor ještě nebyl vytvořen, zavolání konstruktoru, vytvoření generátoru a poté metoda vrací nově vytvořený generátor. V případě, že již jednou byl generátor vytvořen, metoda vrací již dříve vytvořený generátor.

3.4 Prioritní fronta

Dále v projektu je využita datová struktura Prioritní fronta, konkrétně její implementace PriorityQueu, která je standardní součástí balíčku java.util.PriorityQueue. Prioritní fronta je v podstatě seznam položek, kterým je přiřazena priorita. Často potřebujeme, aby přístup k informacím byl seřazen podle priority. Tato datová struktura se chová podobně jako zásobník a fronta tedy odebere prvek s nejnižší či s nevyšší prioritou. Vkládat lze prvky dle libosti.

```
private static PriorityQueue<ICasPrichodu> fronta= new
PriorityQueue<ICasPrichodu>();
```

Příklad vytvoření prioritní fronty využité v projektu.

Abstraktní datová struktura PriorityQueue obsahuje metody:

- fronta.add(ICasPrichodu os) - vkládání prvků do fronty
- fronta.remove() - odebrání prvku s nejvyšší prioritou
- fronta.isEmpty() - zjištění, jestli je fronta prázdná.

3.5 Datový typ Calendar

K vytvoření simulačního modelu potřebujeme, aby nám model pracoval s časem. S datem a časem se v Javě dříve pracovalo pomocí třídy Date, ta je však již zastaralá a nahradila ji třída Calendar. Calendar je abstraktní třída, která se nachází v balíčku java.util.Calendar. Calendar uchovává v paměti nastavený datum a čas. K vytvoření instance Calendar slouží statická metoda *getInstance()*.

```
Calendar kalendar = Calendar.getInstance();
```

Nově vytvořená instance obsahuje kalendář, ve kterém je nastavený aktuální datum a čas. K výpisu kalendáře se používá formátovaný výpis, který nám zajišťuje třída DateFormat.

```
DateFormat formatData = new SimpleDateFormat("d.MMMM yyyy H:mm");
```

Tento formát nám zajistí, že kalendář bude vypisovaný ve tvaru den. měsíc. rok. hodina:minuta.

```
System.out.println(formatData.format(kalendar.getTime()));
```

Samotný výpis přes třídu DateFormat.

```
Opustil ordinaci s id263
V čase 28.červen 2014 10:57:40
```

Obrázek 2: Příklad výpisu kalendáře

Jelikož při tvorbě simulačního modelu se potřebujeme pohybovat v čase, je na třídě Calendar využita metoda *add*, jejímž vstupním argumentem je datový typ int, udávající počet vteřin, o který má být čas inkrementován.

4 High-Level Architecture

High-Level Architecture (HLA) je výkonná technologie pro vytváření distribuovaných simulací. Distribuovaná simulace dle standardu HLA je simulace, která běží na více strojích nezávisle na platformě. Stroje jsou spojeny do jednoho simulačního celku. Spolupráce mezi jednotlivými simulacemi, nazývaných federace, je řízena pomoci Run-Time Infrastructure (RTI). Standart HLA vznikl původně pro vojenské účely na Úřadu pro modelování a simulace Ministerstva obrany USA. V roce 1997 uvolnila normu armáda pro veřejnost.

Terminologie HLA:

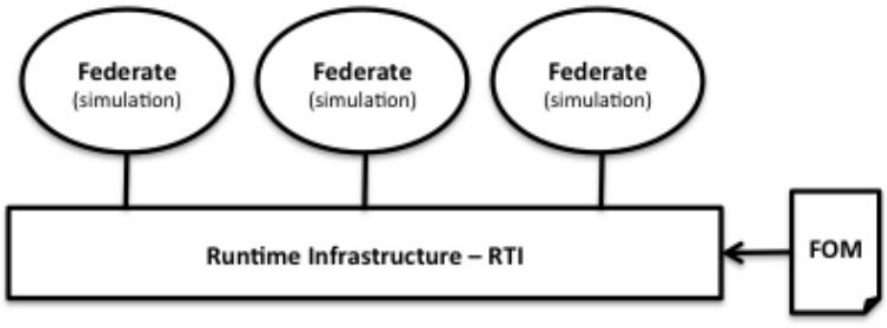
- Federate - neboli člen federace, tj. simulace kompatabilní s HLA, který se účastní na tvorbě federace.

- Federation - neboli federace, je simulační systém složený ze dvou a více federatu, které společně spolupracují přes RTI.

- Federation Object Model (FOM) - soubor, který popisuje, jaká data budou vyměněna v rámci federace.

- Simulation Object Model (SOM) - objektový model, který definuje údaje, které sdílí člen federace v rámci federace.

- Interakce - veškerá komunikace probíhá ve formě interakcí, kde každá interakce může mít sadu parametrů.



Obrázek 3: Topologie HLA Zdroj [2]

4.1 Run-Time Infrastructure (RTI)

Jednotlivý členové federace jsou propojeni přes RTI, které umožňuje jednotlivým členům sdílet informace. Členi mohou sdílet jejich objekty, atributy. RTI poskytuje členům federace služby, které jsou rozděleny do šesti skupin:

- Federation Management - správa federace umožňuje vytvoření federace, připojení k federaci, odhlášení z federace a zrušení federace.
- Declaration Management - správa deklarací umožňuje určit druhy dat, které budou předávány mezi členy federací.
- Ownership Management- správa vlastnictví.
- Object Management - zajišťuje výměnu dat mezi členy federací.
- Time Management - umožňuje členům federace koordinovat logický čas s ostatními členy federace a řídí doručování událostí.
- Data distribution Management (DDM) - správa distribuce.

4.1.1 Pitch RTI

Pitch RTI je komerčním produktem švédské firmy Pitch Technologies AB. Implementace tohoto RTI existuje ve více variantách, a to od pRTI 1.3, kde se používá starší standart HLA 1.3, včetně novější pRTI 1516 verze až po nejnovější verzi, která implementuje HLA Evolved. Všechny varianty splňují certifikáty DMSO, které potvrzují splnění specifikace rozhraní HLA.

Obě varianty pRTI jsou napsány v jazyce Java a pro členy federací poskytují rozhraní v jazycích Java a C++. Pitch RTI existuje i ve verzi Trial, která ovšem použití limituje na dva federáty.

4.2 Pravila HLA

HLA obsahuje deset pravidel pro členy federací v dokumentu norem "HLA rules". Pravidla jsou rozdělena do dvou skupin, a to pravidla pro federace a pravidla pro členy federace. Zde je zjednodušená verze těchto pravidel.

4.2.1 Pravidla pro Federace

1. Všechny federace musí mít FOM.

2. Všechny instance objektů se musí nacházet v členech federace, nikoliv v RTI.

3. Všechny informace popsané ve FOM se vyměňují pouze přes RTI.

4. Členové federace mohou komunikovat s RTI pouze v souladu se specifikací rozhraní HLA.

5. Každý atribut objektu jakékoliv instance objektu může být vlastněn jen jedním členem federace.

4.2.2 Pravidla pro členy Federace

1. Každý člen federace musí mít SOM.

2. Každý člen federace musí odesílat a přijímat data v závislosti na jejich SOM.

3. Každý člen federace musí být schopný převést vlastnictví v souladu s jejich SOM.

4. Každý člen federace musí být schopný měnit podmínky, podle kterých jsou zajišťovány aktualizace atributů, v souladu s jejich SOM.

5. Každý člen federace musí být schopný řídit simulační čas takovým způsobem, aby bylo možné provést aktualizace určitého atributu.

4.3 Framework HLA-VA

„Hlavní motivací pro vznik frameworku „virtual assistant“ je usnadnění přístupu k HLA. I když využijeme i zcela nejjednodušší aplikace, která je vyvíjena tak, aby odpovídala doporučením HLA, a to pro provoz na nejjednodušším RTI je nutné provést tyto kroky:

1. Vytvoření OMT pomocí specializovaného nástroje, nebo pomocí textového editoru. Vytváření probíhá v XML.
2. Připojení všech nutných knihoven k programu.
3. Vytvoření objektů, které budou implementovat všechny metody obecné definice Federate (resp. NullFederate).
4. Vytvoření metod pro připojení do federace, identifikace federátů a dalších obslužných metod.
5. Vytvoření metod pro obsluhu každé jednotlivé interakce.
6. Vytvoření metod pro obsluhu časové synchronizace.
7. Stanovení pravidel pro synchronizace a řešení samotné synchronizace v rámci konkrétního uzlu.
8. Programování aplikace konkrétního modelu (resp. logického procesu).

Pokud se však rozhodneme použít knihovny HLA-VA, celý tento proces se velice zkrátí a to konkrétně na tyto následující kroky:

1. Vytvoření OMT pomocí specializovaného nástroje. Není tak nutné znát specifikaci, ani XML.
2. Připojení knihovny HLA-VA k aplikaci.
3. Využívání základních funkcí (definováno 6 generických funkcí).

Největší výhodou celého řešení je právě usnadnění přístupu k HLA, kdy je možné vytvářet modely velmi rychle.

Navíc je díky tomu, že knihovna RTI je pouze připojena k celé knihovně a není v ní fixně zkompilována, je možné používat (téměř) libovolné RTI v součinnosti s touto knihovnou." (Zdroj [9])

5 Statistika

5.1 Pravděpodobnost

Statistická pravděpodobnost náhodného jevu je číslo, které nám určuje míru očekávatelnosti výskytu jevu. Náhodným jevem se rozumí opakovaná činnost za stejných podmínek, kde výsledek činnosti je nejistý a závisí na náhodě (např. hod mincí, losování loterie). Pravděpodobnost se reprezentuje reálným číslem od 0 do 1, kde pravděpodobnost s hodnotou 0 je jev, který nemůže nastat a opačně jev s hodnotou 1 je jev jistý.

$$P(A) = \frac{m}{n}$$

Kde:

- m - počet příznivých výsledků,
- n - počet všechno možných výsledků.

Pro pravděpodobnost libovolného jevu A platí:

$$0 \leq P(A) \leq 1$$

5.2 Histogram

Histogram graficky znázorňuje distribuci dat pomocí sloupcového grafu, kde jednotlivé sloupce (třídy) jsou stejné šířky. Výška sloupců udává četnost jevů v dané třídě. Je důležité zvolit správný počet tříd tak, aby nebyl žádný interval prázdný.

Počet tříd je dán Sturgesovým pravidlem:

$$K = 1 + 3{,}3 \log(n)$$

Kde:

- K… je počet tříd,
- n …je počet pozorování.

Velmi důležité je též správné zvolení intervalů tříd. Nesprávné zvolení tříd by mělo za následek špatnou informační hodnotu plynoucí z grafu histogramu.

Interval tříd je dán vztahem:

$$interval\ tříd = \frac{X_{max} - X_{min}}{K}$$

Kde:

- X_{max}…je maximální hodnota z pozorování,
- X_{min}…je minimální hodnota z pozorování,
- K…… je počet tříd dán Sturgesovým pravidlem.

5.2.1 Příklad zpracování dat pro histogram

Jako vstupní data pro histogram použijeme náhodně vygenerovaná čísla.

Data	1	2	3	4	5	6
1	8	6,7	6,5	7,7	6,3	6,8
2	5,5	6	5,8	7,2	6	6,2
3	7	7,9	5,3	7,4	5,8	6,9
4	7,9	5,9	5,5	7,5	5,4	5
5	6,1	7,2	5,1	7,2	7,7	5,2
6	6	6,6	6,9	7,3	7,4	6,9

Tabulka 1: Náhodná data

Nejprve je nutné určit počet tříd Sturgesovým pravidlem, počet vzorků určíme z tabulky.

$K = 1 + 3{,}3\log(36)$

$K = 6{,}13$ zaokrouhleno na 6.

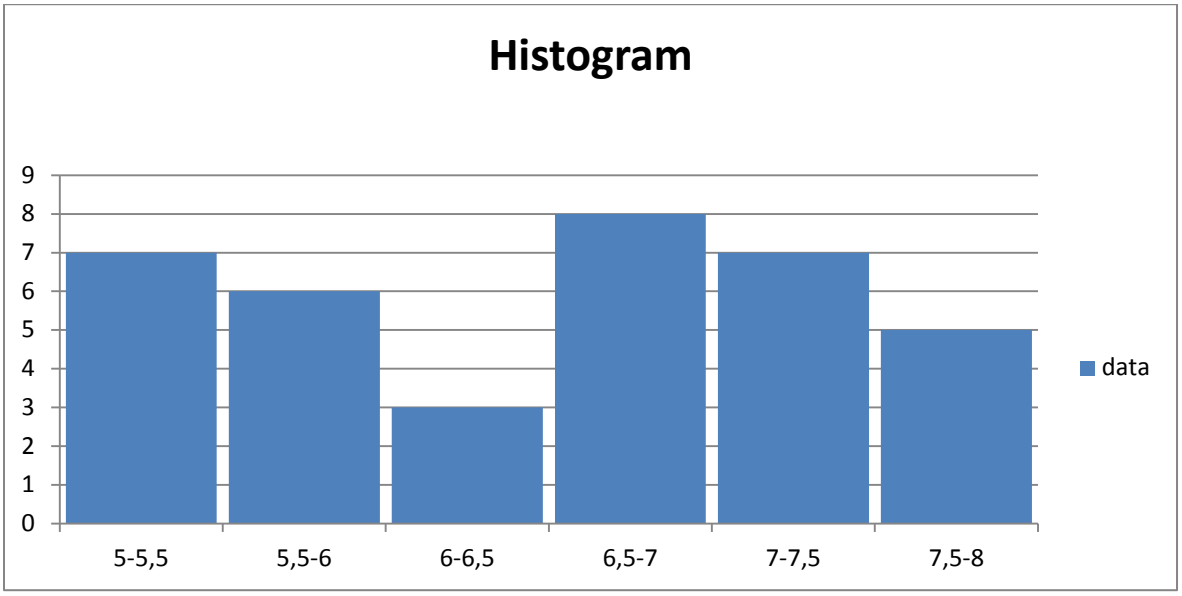
Dále je nutné určit minimum a maximum z dat, ze kterých se poté spočítá interval tříd.

$X_{min}=5$ $X_{max}=8$

$$interval\ tříd = \frac{8 - 5}{6}$$

$$interval\ tříd = 0{,}5$$

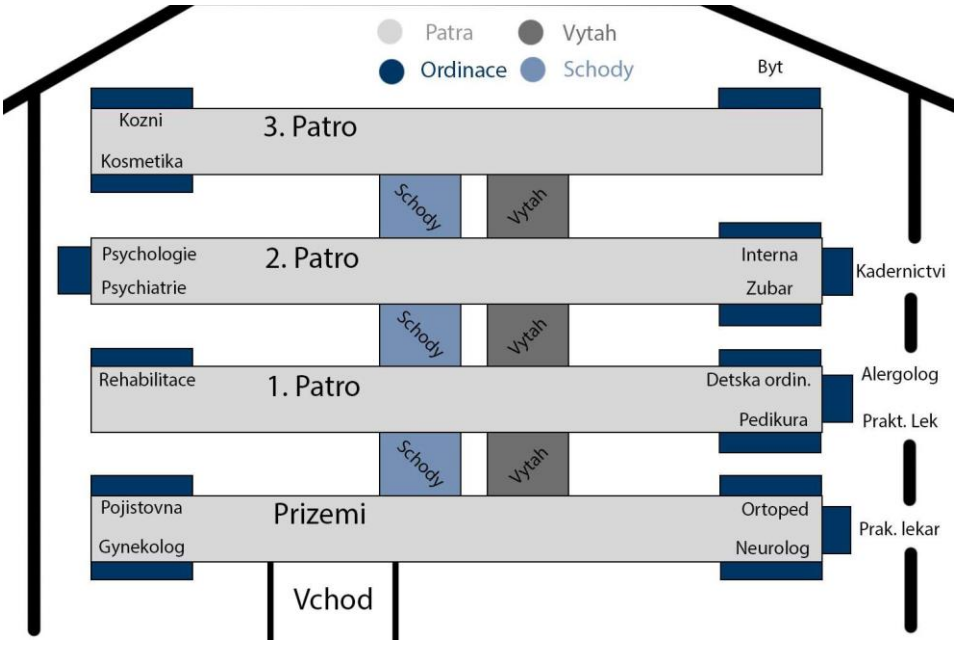
Jednoduchými výpočty se spočítalo, že počet tříd histogramu je roven po zaokrouhlení 6 a velikost intervalu je 0,5. Teď zbývá už jen data seřadit podle intervalů.



Obrázek 4: Histogram příklad

6 Sběr a analýza dat

Sběr dat pro simulační model proběl ve třech dnech. Měření bylo realizováno na staré poliklinice ve Dvoře Králové nad Labem. Budova je třípatrová, každé patro spojuje schodiště a v přízemí je umístěn výtah.



Obrázek 5: Model budovy

Níže uvedená nasbíraná data jsou jen vybraným příkladem. Nasbíraná data jsou velmi rozsáhlá a ke každému měření určité činnosti byly zpracovány dva grafy histogramů. Veškerá zpracovaná data z měření jsou obsažená na přiloženém CD.

V rámci budovy byly měřeny časy:

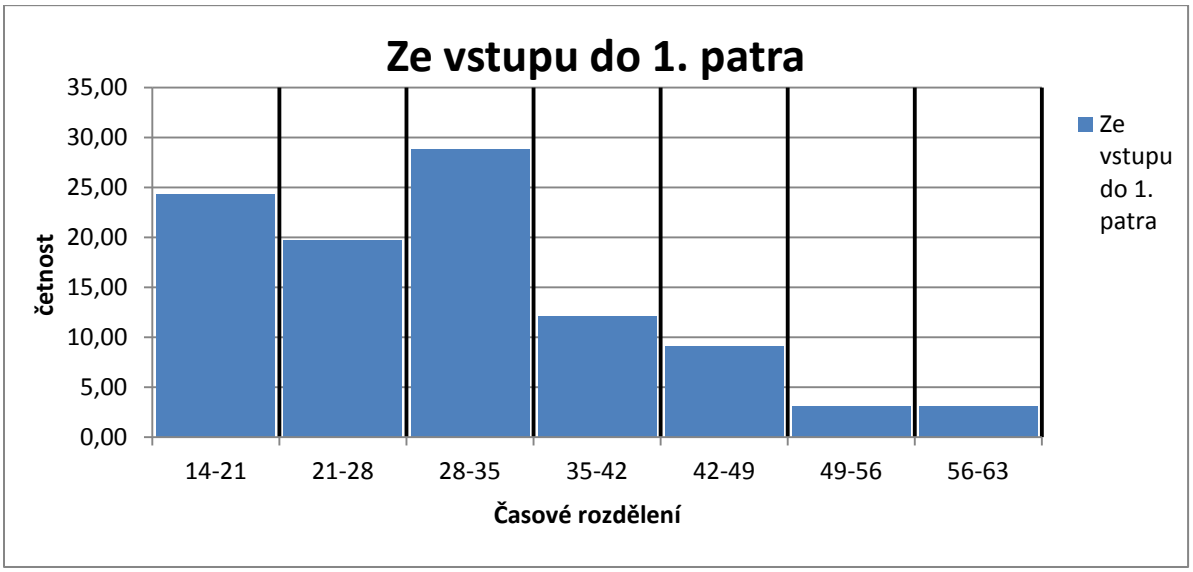
- Časy výstupů do jednotlivých pater - čas od otevření vchodových dveří, výstupu osoby po schodech až po vstup na poslední schod.
- Doby chůze po chodbách - od vstupu osoby na poslední schod na schodišti se začal počítat čas chůze po chodbě až do doby vstupu do čekárny.
- Čekání na výtah - doba, kterou osoba čeká na výtah.
- Doby jízdy výtahem do jednotlivých pater.
- Intervaly mezi příchodem lidí do budovy.

6.1 Časy výstupů do jednotlivých pater

Časy se měřily od vstupu lidí do budovy, tj. od otevření dveří až po výstupu osoby do patra.

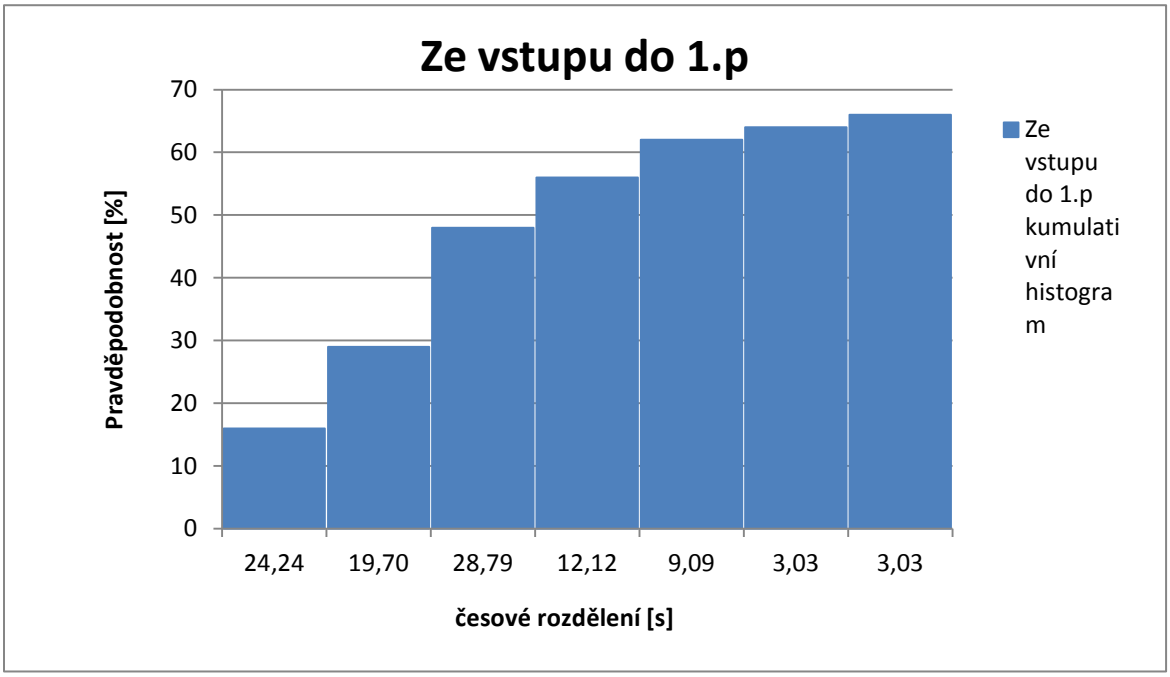
6.1.1 1. patro

Z naměřených dat byly zpracovány histogramy dle pravidel, které jsou popsána v kapitole Histogramy.



Obrázek 6: Histogram ze vstupu do 1p.

Pro určení pravděpodobností byl využit kumulativní histogram, ze kterého je lépe čitelné rozdělení.



Obrázek 7: Histogram ze vstupu do 1p. kumulativní

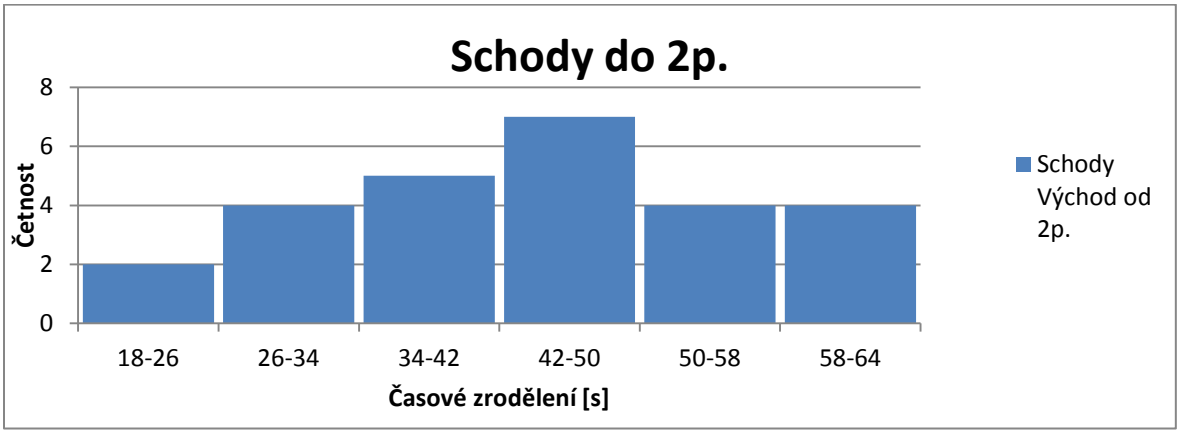
Zjištěné rozdělení pravděpodobnosti z kumulativního histogramu:

Intervaly [s]	Pravděpodobnost [%]
14-21	36,73
21-28	29,84
28-35	43,62
35-42	18,37
42-49	13,77
49-56	4,59
56-63	4,59

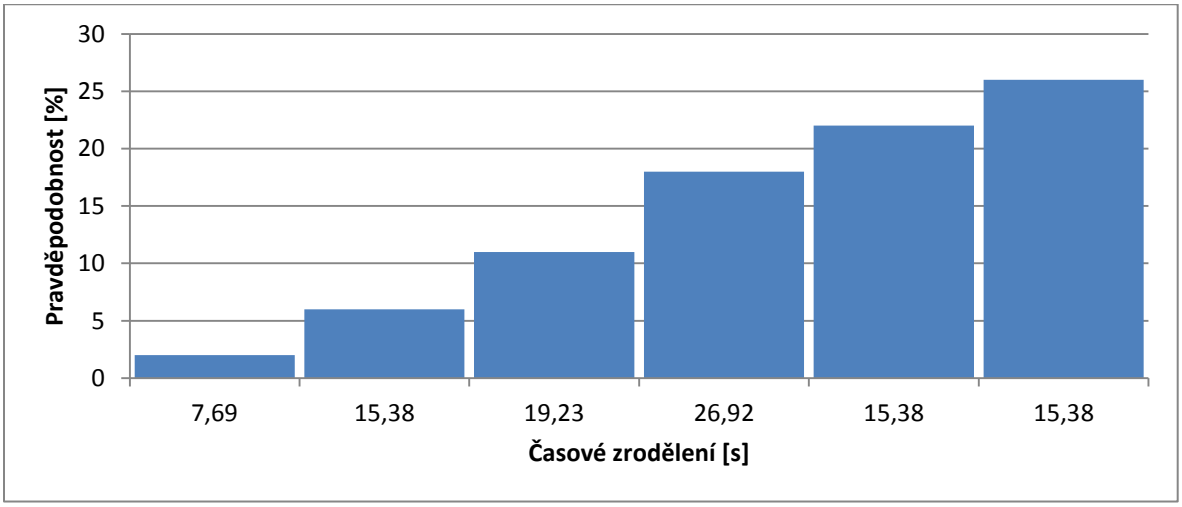
Tabulka 2:Pravděpodobnost z kumul. histogramu

6.1.2 2. patro

Naměřené a zpracované hodnoty pro dobu výstupu osob po schodech do druhého patra od vstupu budovy.



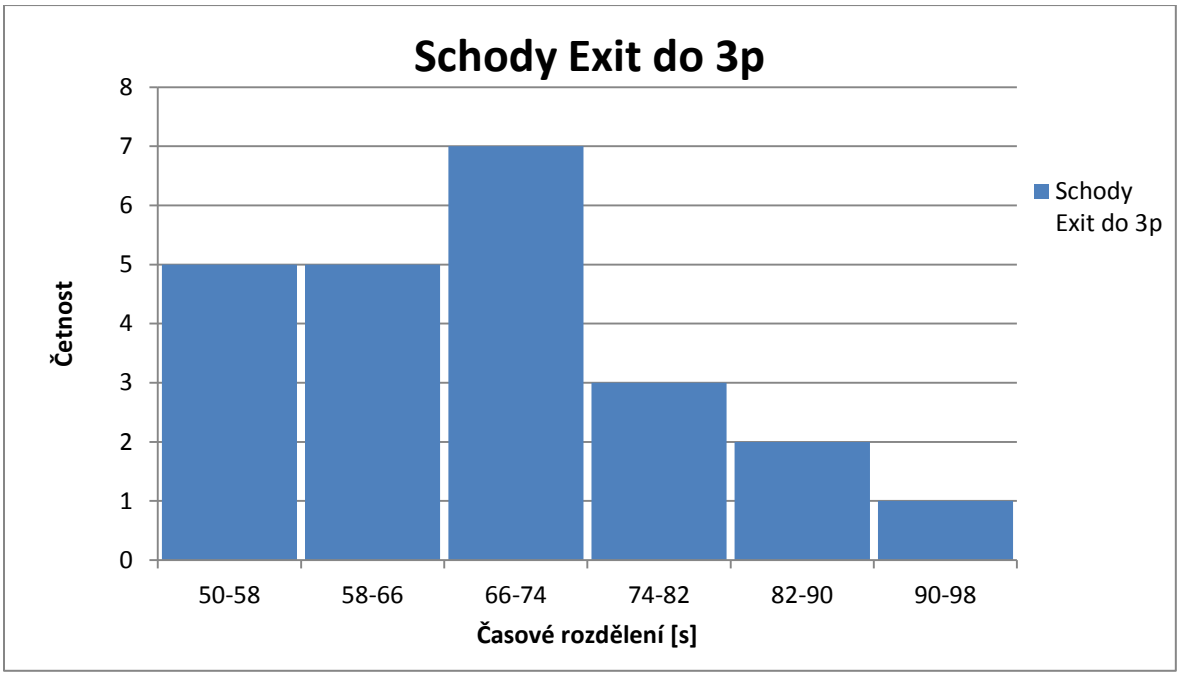
Obrázek 8: Schody do 2p.



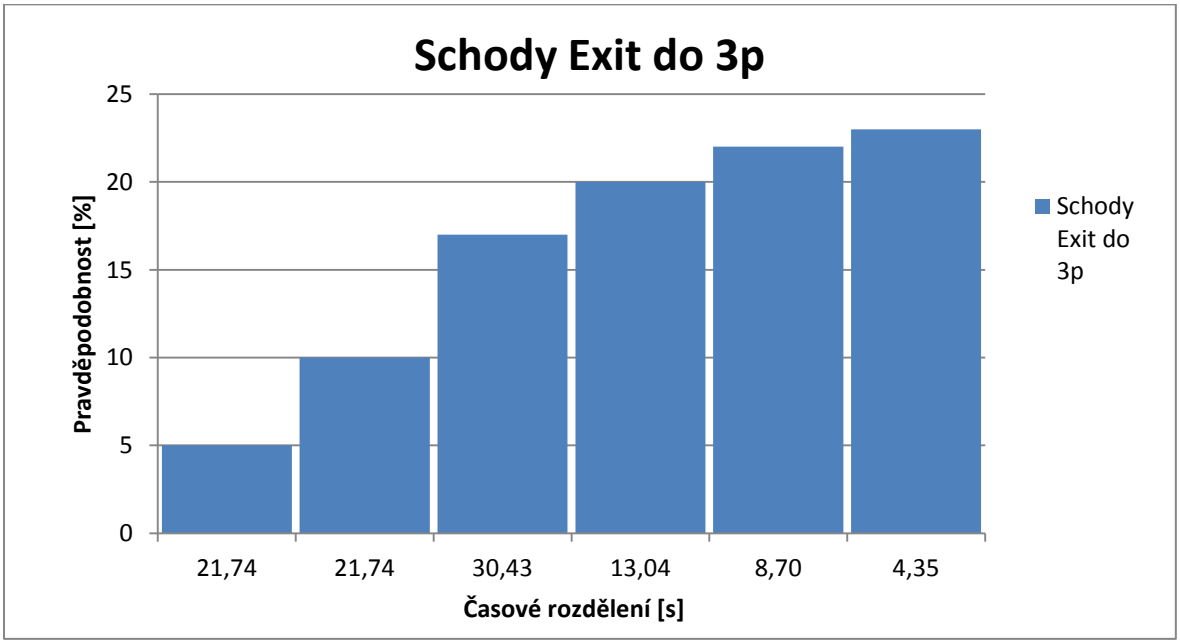
Obrázek 9: Schody do 2p. kumulativní

6.1.3 3. patro

Naměřené a zpracované hodnoty pro dobu chůze osob do třetího patra od vstupu budovy.



Obrázek 10: Histogram do 3. patra

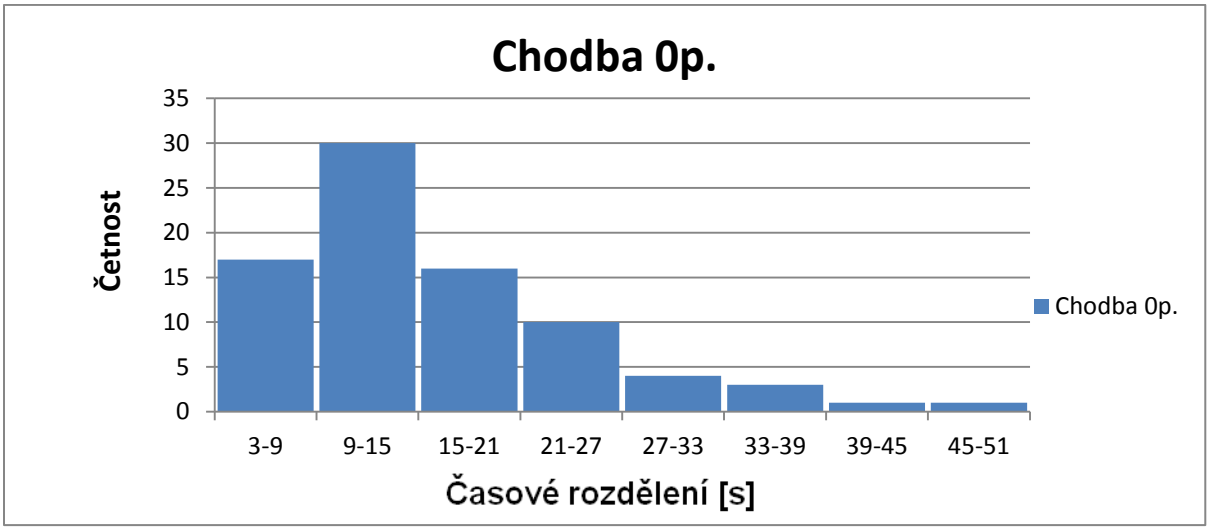


Obrázek 11: Histogram do 3. patra kumulativní

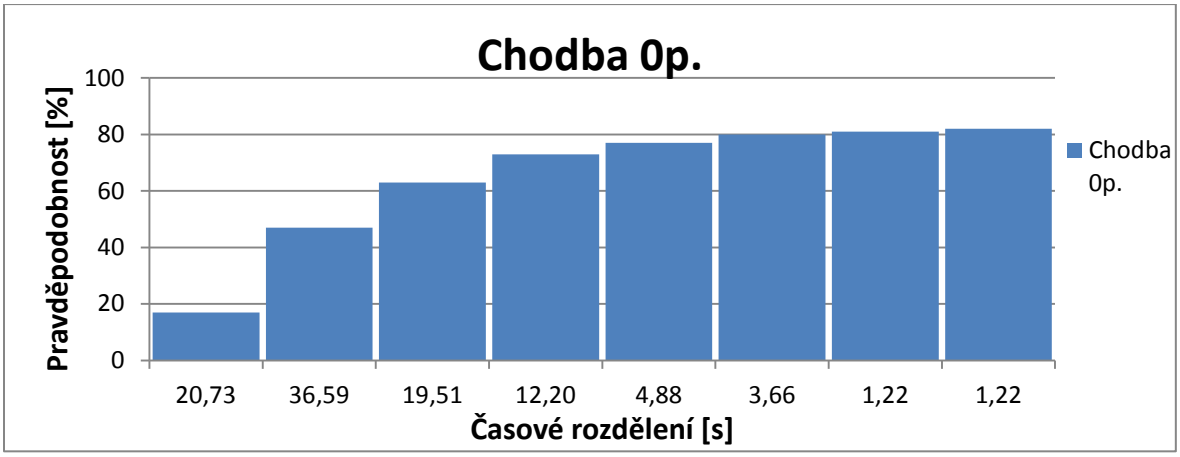
6.1.4 Chodby

6.1.4.1 Přízemí

Naměřené a zpracované doby chůze po chodbě v přízemí.



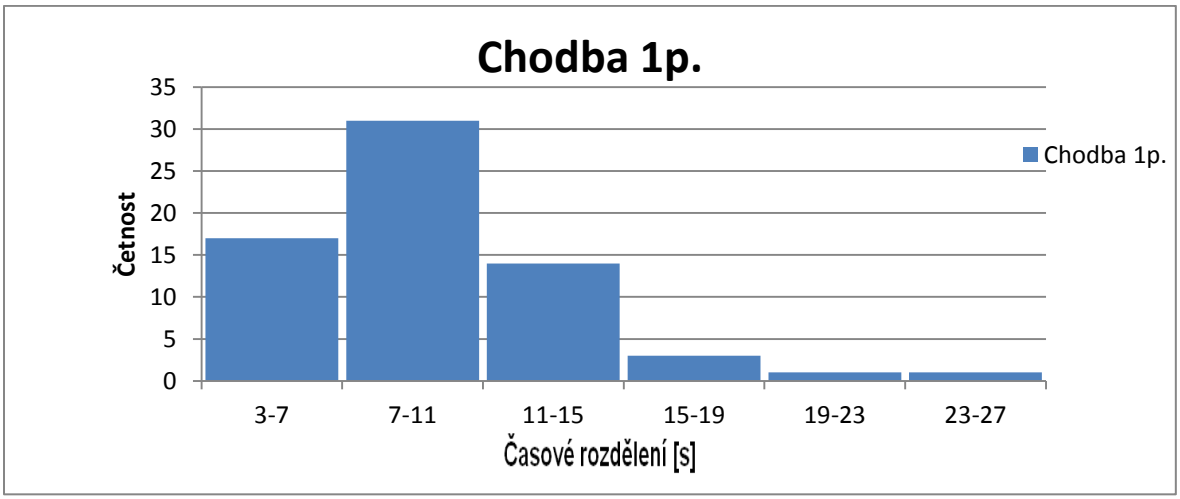
Obrázek 12: Histogram chodba 0p.



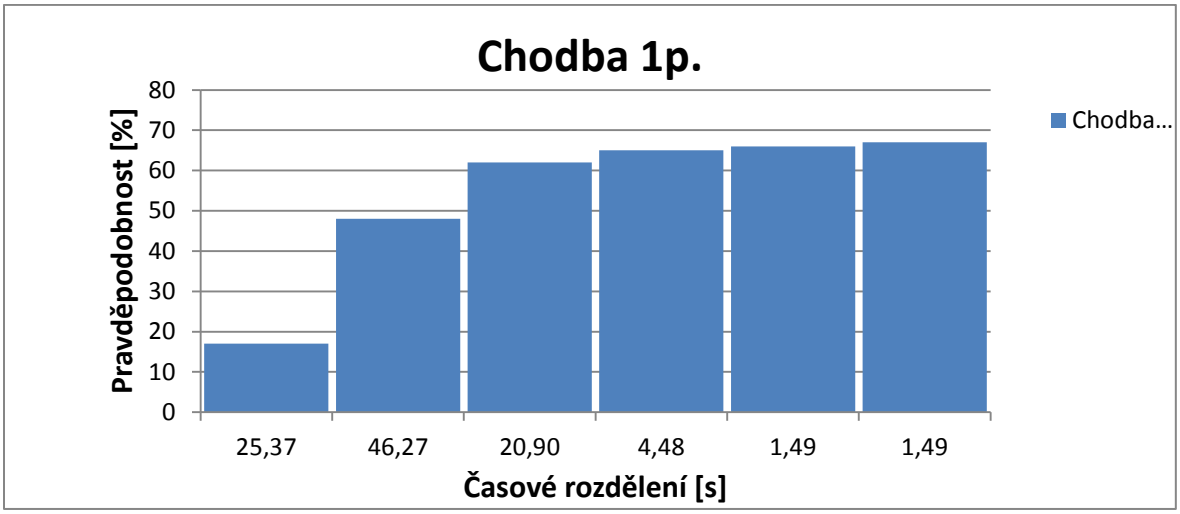
Obrázek 13: Histogram chodba 0p. kumulativní

6.1.4.2 První patro

Naměřené a zpracované doby chůze po chodbě v prvním patře.



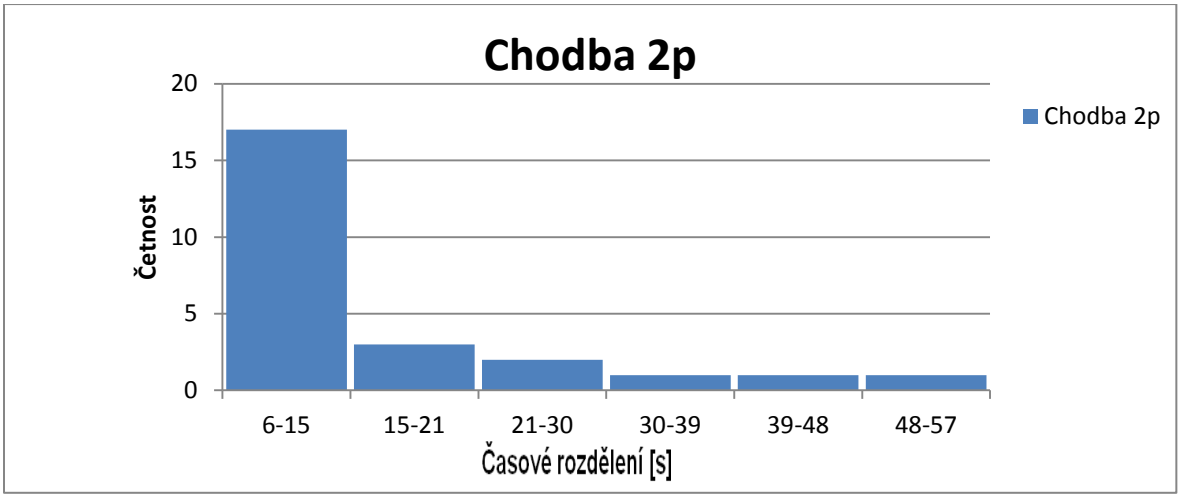
Obrázek 14: Histogram chodba 1p.



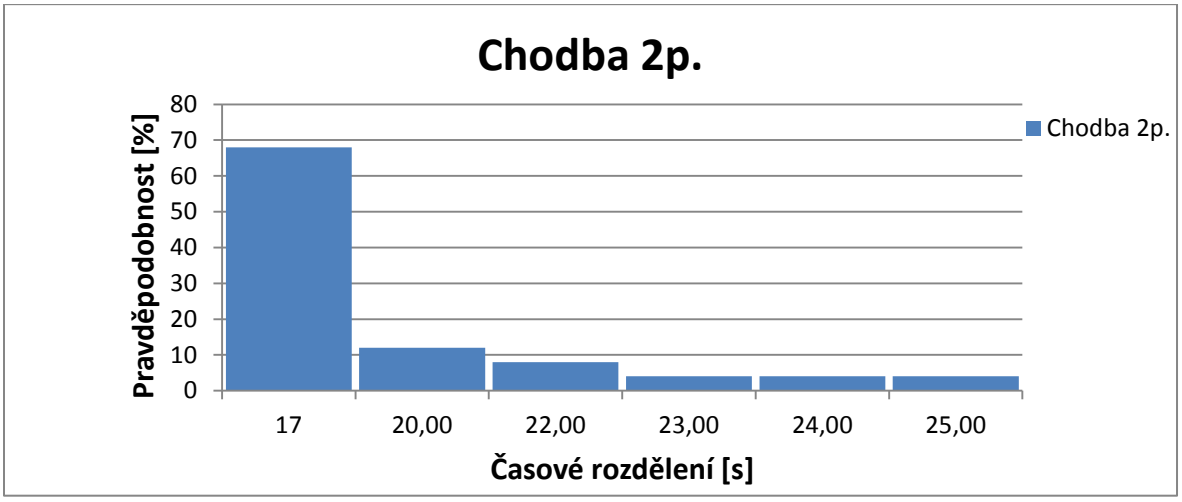
Obrázek 15: Histogram chodba 1p. kumulativní.

6.1.4.3 Druhé patro

Naměřené a zpracované doby chůze po chodbě v druhém patře.



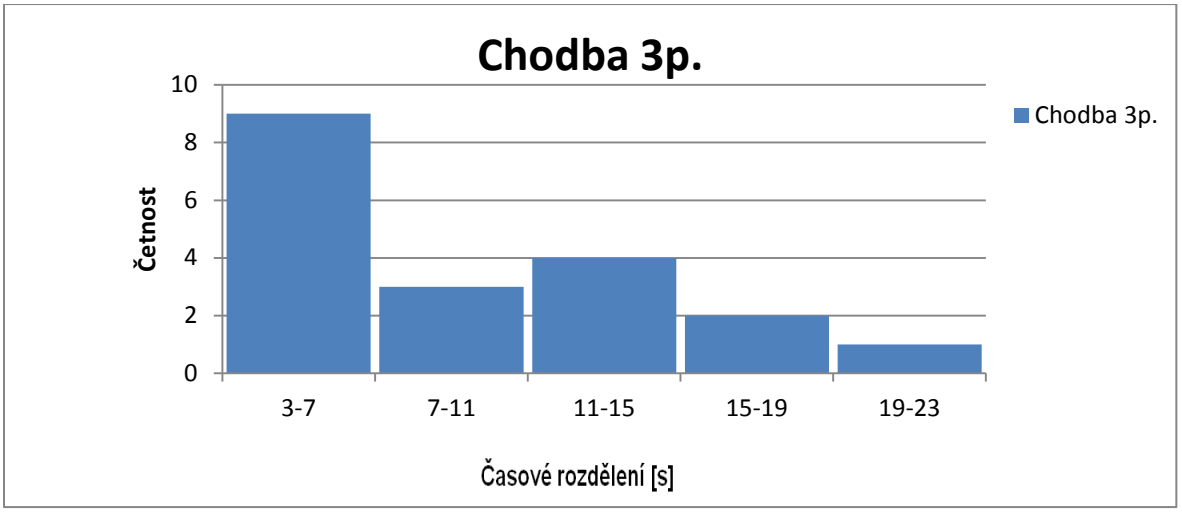
Obrázek 16: Histogram chodba 2p.



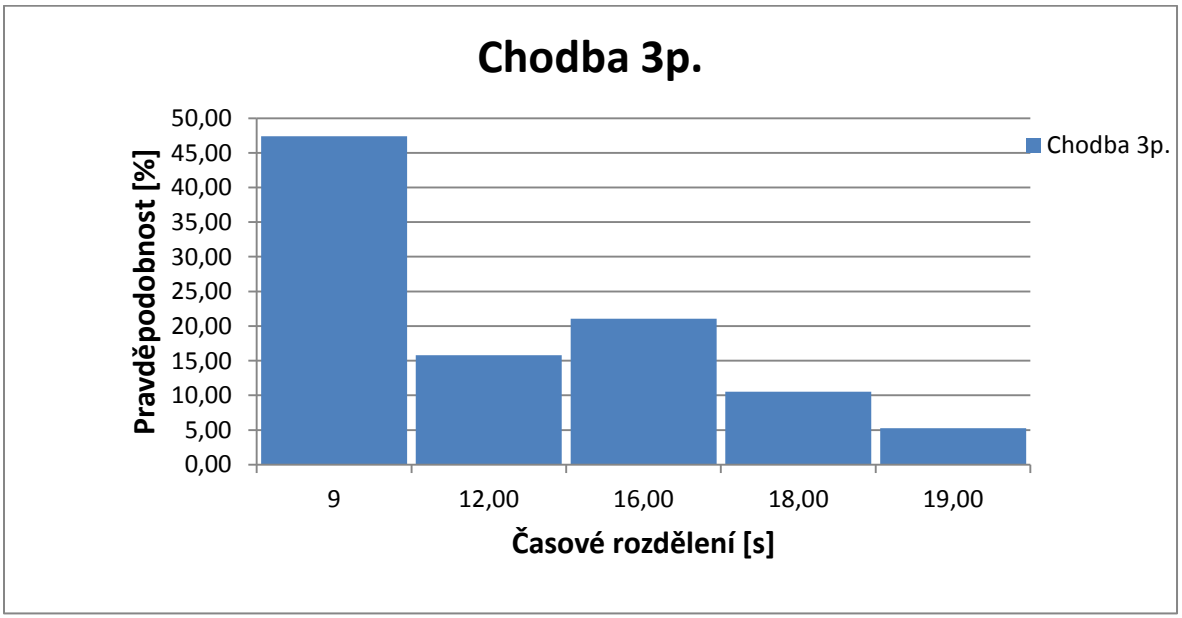
Obrázek 17: Histogram chodba 2p. kumulativní.

6.1.4.4 Třetí patro

Naměřené a zpracované doby chůze po chodbě v třetím patře.



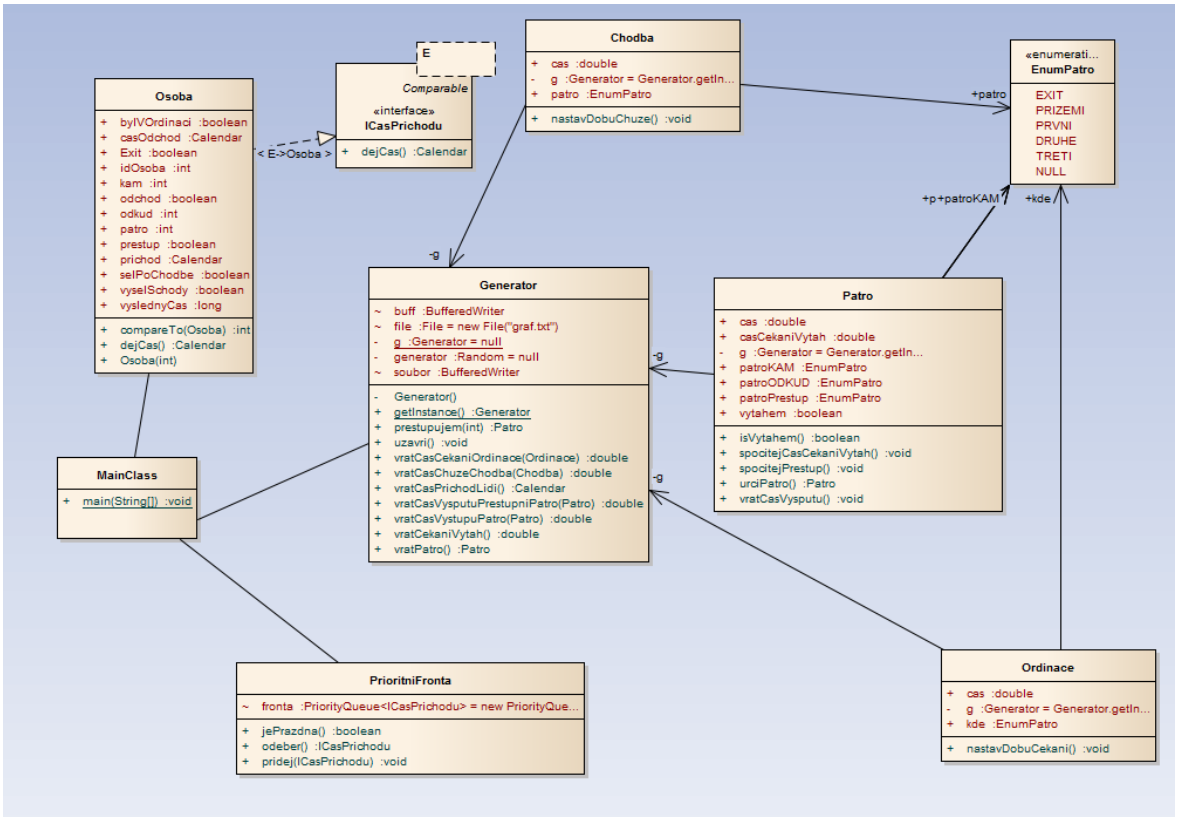
Obrázek 18: Histogram chodba 3p.



Obrázek 19: Histogram chodba 3p. kumulativní.

7 Tvorba simulačního modelu v jazyce Java

Cílem vytvoření simulačního modelu je vytvořit model, který odpovídá své předloze, tedy budově v Dvoře Králové nad Labem. Pro tvorbu modelu byl využit programovací jazyk Java. Pro vytvoření simulačního programu bylo nutné navrhnout všechny třídy, které obstarávají veškerou logiku modelu.



Obrázek 20: UML diagram

7.1 Třída Osoba

Třída Osoba implementuje rozhraní ICasPrichodu<Osoba>, které nám zajišťuje vkládání do prioritní fronty podle proměnné *prichod*, který je typu *Calendar*.

Osoba obsahuje pět boolean proměnných (*vyselChody*, *odchod*, *selPochodbe*, *bylVOrdinaci*, *Exit*), které nám slouží k orientaci vykonaných činnosti osoby v rámci modelu. Osoba dále obsahuje proměnné typu Calendar (*prichod*, *casOdchodu*), které nám zajišťují časový přehled příchodu a odchodu osoby do budovy a z budovy. Nakonec Osoba ještě obsahuje celočíselnou proměnou *idOsoba*, která slouží k identifikaci osoby v rámci modelu.

```
18 public class Osoba implements ICasPrichodu<Osoba> {
19
20 public int idOsoba;
21 public long vyslednyCas;
22 public boolean vyselSchody;
23 public boolean odchod;
24 public boolean selPoChodbe;
25 public boolean bylVOrdinaci;
26 public boolean Exit;
27 public int patro;
28 public boolean prestup;
29 public int kam;
30 public int odkud;
31 public boolean vytahem;
32 public Calendar prichod;
33 public Calendar casOdchod;
34
35 public Osoba(int idOsoba) {
36
37 this.idOsoba = idOsoba;
38         vyslednyCas=0;
39         vyselSchody=false;
40         selPoChodbe=false;
41         bylVOrdinaci=false;
42         Exit=false;
43         odchod=false;
44
45     }
```

Ukázka části kódu třídy Osoba.

Třída osoba obsahuje následující metody:

- *public Osoba(int id)* - parametrický konstuktor, který má jako vstupní argument *idOsoba*. Kontruktor nastaví všechny proměnné boolean na false. Konstruktor inicializuje proměnnou *prichod*, metodou Generatoru *vratCasPrichodLidi()*.

- *public Calendar dejCas()* - metoda vrací čas příchodu osoby. Tato metoda se používá při třídění osoby v prioritní frontě.

- *public int compareTo(Osoba o)* - metoda *compareTo* porovnává dvě proměnné *prichod*. Metoda *compareTo* vrací 0, pokud jsou oba časy příchodu stejné. Záporné číslo vrací, pokud je porovnávaný čas příchodu menší než předaný parametrem a kladné číslo vrací, pokud je větší.

- *public String toString()* - metoda, která zajistí převedení všech proměnných do řetězce String, kde jsou jednotlivé parametry odděleny středníkem.

7.2 Třída EnumPatro

Třída EnumPatro je výčtovým typem, který obsahuje seznam všech pater. Třída navíc obsahuje hodnotu *NULL*, která se používá v případě, že osoba nepřestupuje. Potom se do instanční proměnné *patroPrestup* ze třídy Patro inicializuje hodnota NULL.

```
12public enum EnumPatro{
13     EXIT,
14     PRIZEMI,
15     PRVNI,
16     DRUHE,
17     TRETI,
18     NULL
19 }
```

Ukázka vytvořené třídy EnumPatro.

7.3 Třída Chodba

Třída Chodba obsahuje pouze instanci třídy Generátor pro generování rozdělení, pro proměnnou *cas* a instanci *EnumPatro*, která obsahuje informace o tom, v kterém patře se bude rozdělení počítat. Třída má pouze jedinou metodu *nastavDobuChuze()*, která volá metodu *vratCasChuzeDoba()* z generátoru. Tato metoda jako vstupní argument dostává instanci Chodby.

```
17publicclass Chodba  {
18private Generator g = Generator.getInstance();
19public EnumPatro patro;
20publicdouble cas;
21
22publicvoid nastavDobuChuze(){
23  cas=g.vratCasChuzeChodba(this);
24  }
25 }
```

7.4 Třída Ordinace

Třída Ordinace obsahuje instanci třídy Generátor, která slouží k rozdělení času metodou generátoru *vratCekaniOrdinace()* do proměnné typu double *cas*. Třída dále obsahuje proměnnou EnumPatro *kde*, která obsahuje patro, ve kterém se ordinace nalézá.

```
16 public class Ordinace {
17 private Generator g = Generator.getInstance();
18 public EnumPatro kde;
19 public int ordinace;
20 public double cas;
21
22 publicvoid nastavDobuCekani(){
23      cas=g.vratCasCekaniOrdinace(this);
24  }
25 }
```

Třída obsahuje i proměnnou *ordinace*, která je celočíselného typu int. Proměnná nám udává, do které ordinace má osoba namířeno. Proměnná nabývá hodnot v závislosti na proměnné *kde*.

Pro hodnotu PRIZEMI z proměnné *kde* :

- Jednička je přiřazena gynekologii,
- dvojka je přiřazena neurologii,
- trojka je přiřazena ortopedii,
- čtyřka pro praktického lékaře.
- a pětka pro pobočku pojišťovny.

Pro hodnotu PRVNI z proměnné *kde*:

- Jednička pro rehabilitace,
- dvojka pro dětského lékaře,
- trojka pro pedikúru,
- čtyřka pro alergologii,
- a pětka pro praktického lékaře.

Pro hodnotu DRUHE z proměnné *kde*:

- Jednička pro psychologii,
- dvojka pro psychiatrii,
- trojka pro interní ambulanci,
- čtyřka pro zubaře,
- a pětka pro kadeřnictví.

Pro hodnotu TRETI z proměnné *kde*:

- Jednička pro kožní,
- dvojka pro kosmetiku.

7.5 Třída Patro

Třída Patro obsahuje informace, kam, odkud a zda daná osoba přestupuje, v proměnných *EnumPatro*. Další informace o tom, zda osoba jede výtahem, či nikoliv, v proměnné boolean *vytahem*. Následují dvě proměnné typu double. Jedna pro čas výstupu do jednotlivých pater *cas* a druhá pro dobu čekání na výtah *casCekaniVytah*. Ve třídě je instance generátoru pro zpracování všech potřebných údajů.

```
16 public class Patro {
17 private Generator g = Generator.getInstance();
18 public EnumPatro patroKAM;
19 public EnumPatro patroODKUD;
20 public EnumPatro patroPrestup;
21 public double cas;
22 public boolean vytahem;
23 public double casCekaniVytah;
24
25 public Patro urciPatro() {
26         Patro p = g.vratPatro();
27         p.cas = g.vratCasVystupuPatro(p);
28 return p;
29     }
30 public void spocitejCasCekaniVytah() {
31         casCekaniVytah = g.vratCekaniVytah();
32     }
```

Třída obsahuje neparametrický konstruktor, ve kterém se volá metoda generátoru *vratPatro()*. Dále má tři další metody, ve kterých se volají metody na spočítání času čekání na výtah, spočítání času pro osoby přestupující a vrácení času výstupu v případě, že osoba budovu opouští a je potřeba znovu spočítat čas sestupu po chodech. Poslední je metoda *isVytahem()*, která vrací proměnou boolean *vytahem*.

7.6 Třída PrioritniFronta

Třída PrioritniFronta je simulačním jádrem projektu postaveném na plánování událostí do budoucnosti. Při použití metody plánování událostí je známá doba trvání události již před jejím začátkem. Ve třídě je použita abstraktní datová struktura *PriorityQueue*, která přejímá *ICasPrichodu*. Třída obsahuje následující metody:

- *void pridej(ICasPrichodu os)* - metoda přidává do prioritní fronty osoby. Pořadí osob ve frontě záleží na instanční proměnné *prichod* typu Calendar, která obsahuje čas příchodu. Osoby jsou uspořádány od nejnižšího času po nejvyšší.

- *ICasPrichodu odeber()* - metoda odebírá a vrací osobu s nejnižším časem příchodu.

- *boolean jePrazdna()* - testuje zda prioritní třída obsahuje ještě nějaké osoby.

7.7 Třída Generátor

Třída generátor je implementována dle návrhového vzoru Singleton. Konstruktor třídy lze tedy zavolat pouze jednou, pak se předává již vytvořená instance. Třída Generátor poskytuje veškeré statistické rozdělení pro simulační model.

Ke generování náhodných čísel v této metodě je použita instance třídy Random. Na její instanci se volá metoda *nextDouble()*, která vrací čísla v intervalu <0,1>.

7.7.1 Metoda vratPatro()

Metoda *vratPatro()* nám zajišťuje rozdělení osob do příslušných pater, možnosti přestupu, jízdu výtahem. Návratovým typem metody je Patro.

Zavoláním metody *vratPatro()* inicializujeme Patro, které je deklarováno v každé instanci třídy Patro. Metoda si vytvoří 3 proměnné typu double (*pravdepodobnostPatro*, *pravdepodobnostVytah*, *prestup*), do kterých se vygeneruje náhodné číslo, které je vynásobeno 100 pro jednoduchost práce s proměnnými.

Pravděpodobnosti pro výstup lidi do jednotlivých pater včetně možnosti využití výtahu:

Patro	Pravděpodobnosti[%]
Přízemí	37,78
1. patro	29,30
2. patro	15,48
3. patro	17,44
∑	100,00

Tabulka 3: Pravděpodobnost patra

Metoda otestuje proměnnou *pradepodobnostPatro*, jestli je v rozsahu:

- 0 až 15,18 - Metoda přiřadí druhé patro.
- 15,48 až 32,92 - Metoda přiřadí třetí patro.
- 32,92 až 62,22 - Metoda přiřadí první patro.
- a více než 62,22 - Metoda přiřadí přízemí.

Jelikož je pravděpodobnost uvedená včetně možnosti jízdy výtahem, je nutné, po testu rozsahu a příslušného přiřazení patra, otestovat, zda nově vytvořená osoba půjde po schodech či využije výtah, zda půjde pouze do jednoho patra, nebo bude přestupovat.

Z naměřených hodnot bylo spočítáno, že pro první patro využije výtah 12,77% lidí. Pro druhé patro již 35,13% a pro třetí patro 48,9% lidí. Program využívá pro tuto pravděpodobnost proměnou *prestup*, která je také naplněna v rozsahu 0 až 1 a pro zjednodušení práce následně vynásobená stem.

Úplnou analogií je v metodě prováděna možnost přestupu lidí mezi patry s následující pravděpodobností. Osoby jdoucí do prvního patra přestupují ve 33,91% případu do druhého patra. Osoby jdoucí do druhého patra přestupují do prvního patra ve 16,07% případu a do třetího patra v 20,51% případů. Přestup z třetího patra byl v rámci měření zaznamenán pouze do patra druhého, a to v 23,06% případů.

Ukázka části kódu z metody *vratPatro()*, zobrazená část kódu se týká třetího patra.

```
673 public Patro vratPatro() {
674         Patro patro = new Patro();
675 double pravdepodobnostPatro = generator.nextDouble() * 100;
676 double pravdepodobnostVytah = generator.nextDouble() * 100;
677 double prestup = generator.nextDouble() * 100;
678 
679 if (pravdepodobnostPatro >= 15.48 & pravdepodobnostPatro < 32.92) {
680 
681 if (pravdepodobnostVytah <= 48.92) {
682                 patro.vytahem = true;
683             } else {
684                 patro.vytahem = false;
685             }
686 if (prestup <= 23.06) {
687 patro.patroPrestup = EnumPatro.DRUHE;
688                 patro.patroKAM = EnumPatro.TRETI;
689                 patro.patroODKUD = EnumPatro.EXIT;
690 return patro;
```

7.7.2 Metoda vratCasVystupuPatro

Metoda, jejímž návratovým typem je double, se stará o rozdělení času výstupů osob do jednotlivých pater bez časů do přestupních pater. Vstupní argumentem metody je instance třídy Patro, které je již naplněno metodou *vratPatro()*. Metoda opět využívá generátor náhodných čísel z třídy Random do proměnné *pravdepodobnost*.

Zprvu metoda musí zjistit kam má daná osoba namířeno a zda jede výtahem. K tomu slouží vstupní argument Patro, ve kterém jsou uloženy příslušné informace.

Metoda otestuje boolean proměnnou *vytahem* metodou *isVytahem()* obsaženou v třídě Patro. Když metoda vrátí hodnotu false, je jasné, že osoba jde po schodech. Dále je nutné zjistit, odkud a kam osoba jde. K tomu slouží switch, který řeší příslušnou logiku pater a inicializuje jednotlivé časy výstupu do patra podle pravděpodobnosti.

Ukázka části kódu metody:

```
320    if (!patro.isVytahem()) {
321 switch (patro.patroODKUD) {
322 case EXIT:
323 switch (patro.patroKAM) {
324 case PRVNI:
325 pravdepodobnost =generator.nextDouble() * 100;
326 if (pravdepodobnost < 24.24) {
327
328              cas=(14 + (21 - 14) *generator.nextDouble());
329
330  } else if (pravdepodobnost >= 24.24 & pravdepodobnost < 43.94) {
331
332                    cas=(21 + (28 - 21) *generator.nextDouble());
333
334  } else if (pravdepodobnost >= 43.94 & pravdepodobnost < 72.73) {
335
336                 cas= (28 + (35 - 28) * generator.nextDouble());
337
338       } else if (pravdepodobnost >= 72.73 & pravdepodobnost < 84.85){
339
340       cas= (35 + (42 - 35) * generator.nextDouble());
341
342  } else if (pravdepodobnost >= 84.85 & pravdepodobnost < 93.94) {
343
344    cas= (42 + (42 - 49) * generator.nextDouble());
345
346    } else if (pravdepodobnost >= 93.94 & pravdepodobnost < 96.97) {
347
348        cas= (49 + (49 - 56) * generator.nextDouble());
349
350            } else if (pravdepodobnost >= 96.97) {
351
352              cas= (56 + (63 - 56) * generator.nextDouble());
353                                              }
354  break;
```

7.7.2.1 Simulační čas pro první patro

Na základě vytvořených histogramů byly zpracovány pravděpodobnosti časů pro případ výstupu do prvního patra ze vstupu budovy. Pravděpodobnosti jsou následující:

Intervaly [s]	hodnoty pravděpodobnosti	Pravděpodobnost [%]
14-21	0-24,24	24,24
21-28	24,24-43,94	19,70
28-35	49,94-72,73	28,79
35-42	72,73-84,85	12,12
42-49	84,85-93,94	9,09
49-56	93,94-96,97	3,03
56-63	více jak 96,97	3,03
∑		100,00

Tabulka 4: Pravděpodobnost první patro

Příklad výpočtu simulačního času pro hodnoty v intervalu 14-21 s, kde simulační čas je vyjádřen jako:

$$\check{C}as = Interval_{min} + (Interval_{max} - Interval_{min}) * \ r < 0; 1 >$$

$$\check{C}as = 14 + (21 - 14) * \ 0{,}489$$

$$\check{C}as = 17{,}423\text{s}$$

Kde

- r je rovnoměrné rozdělení z intervalu 0 až 1.

7.7.2.2 Simulační čas pro druhé patro

Hodnoty vytvořené pro čas výstupu do druhé patro od vstupu do budovy:

Intervaly [s]	hodnoty pravděpodobnosti	Pravděpodobnost [%]
18-26	0-7,69	7,69
26-34	7,69-23,08	15,38
34-42	23,08-42,31	19,23
42-50	42,31-69,23	26,92
50-58	69,23-84,62	15,38
58-64	více než 84,62	15,38
∑		100,00

Tabulka 5: Pravděpodobnost druhé patro.

Příklad výpočtu simulačního času pro hodnoty v intervalu 42-50 s, kde simulační čas je vyjádřen jako:

$$\check{C}as = Interval_{min} + (Interval_{max} - Interval_{min}) * \ r < 0; 1 >$$

$$\check{C}as = 42 + (50 - 42) * \ 0{,}489$$

$$\check{C}as = 45{,}912\text{s}$$

7.7.2.3 Simulační čas pro třetí patro

Hodnoty vytvořené pro čas výstupu do třetího patra od vstupu do budovy:

Intervaly [s]	hodnoty pravděpodobnosti	Pravděpodobnost [%]
50-58	0-21,74	21,74
58-66	21,74-43,48	21,74
66-74	43,48-73,91	30,43
74-80	73,91-86,96	13,04
80-88	86,96-95,65	8,70
88-96	více než 95,65	4,35
∑		100,00

Tabulka 6: Pravděpodobnost třetí patro.

7.7.2.4 Simulační čas pro přízemí

Intervaly [s]	hodnoty pravděpodobnosti	Pravděpodobnost [%]
3-9	0-20,73	20,73
9-15	20,73-57,32	36,59
15-21	57,32-76,81	19,51
21-27	76,83-89,02	12,20
27-33	89,02-93,90	4,88
33-39	93,90-97,56	3,66
39-45	97,56-98,78	1,22
45-51	více jak 98,78	1,22
∑		100,00

Tabulka 7: Pravděpodobnost přízemí.

7.7.3 Metoda vratCasVystupuPrestupniPatro()

Metoda *vratCasVystupuPrestupniPatro()* se stará o osoby, které nemají za cíl pouze jedno patro, ale v rámci modelu přestupují do více pater. Metoda, jejímž návratovým typem je double, vrací čas potřebný na přestup z jednoho patra do druhého. Vstupním argumentem metody je instance třídy Patro, ze kterého metoda zjistí, odkud a kam daná osoba přestupuje.

V rámci měření bylo zaznamenáno přecházení z prvního do druhého patra, z druhého do prvního a z třetího do druhého. Ostatní jevy, jako např. přecházení z třetího patra do prvního apod., nebyly zaznamenány.

Přechody	Pravděpodobnost [%]
1p. Do 2p.	33,92
2p. Do 3p.	20,52
3p. Do 2p.	23,07
2p. Do 1p.	16,08

Tabulka 8: Přechody pater pravděpodobnost

Metoda zprvu zkontroluje, zda osoba přestupuje, a to proměnnou ze vstupního argumentu typu EnumPatro *patroPrestup*, v případě, že tato proměnná obsahuje z tohoto enumu hodnotu *NULL*, přestup se nekoná a metoda vrací 0. V opačném případě metoda zjistí přes další proměnné ze vstupního argumentu konkrétně *patroODKUD* a *patroKAM*, kam jednotlivé osoby mají namířeno a vrátí příslušný čas. Ukázka kódu ukazuje přestup z prvního patra do druhého, včetně časových rozdělení, které se řídí podle vzorce uvedeného v podkapitole (6.4.2.1).

```
switch (patro.patroODKUD) {
    case PRVNI:
        switch (patro.patroKAM) {
            case DRUHE:
                if (pravdepodobnost < 30.77) {

                    cas = (16 + (19 - 16) * generator.nextDouble());

                } else if (pravdepodobnost >= 30.77 & pravdepodobnost < 61.54) {

                    cas = (long) (19 + (22 - 19) * generator.nextDouble());

                } else if (pravdepodobnost >= 61.54 & pravdepodobnost < 80.77) {

                    cas = (22 + (25 - 22) * generator.nextDouble());

                } else if (pravdepodobnost >= 80.77 & pravdepodobnost < 88.46) {

                    cas = (25 + (28 - 25) * generator.nextDouble());

                } else if (pravdepodobnost >= 88.46 & pravdepodobnost < 96.15) {

                    cas = (28 + (31 - 27) * generator.nextDouble());

                } else if (pravdepodobnost >= 96.15) {

                    cas = (31 + (34 - 31) * generator.nextDouble());
                }

        }
        break;
```

Obrázek 21: metoda vratCasPrestupuLidi().

7.7.4 Metoda vratCasChuzeChodba()

Metoda *vratCasChuzeChodba()* se stará, jak již název napovídá, o rozdělení času v jednotlivých chodbách v rámci celé budovy. Návratovým typem metody je double, který vrací příslušné rozdělení. Vstupním argumentem metody je instance třídy Chodba, ve které je v instanční proměnné typu EnumPatro *patro* uvedeno, pro které patro se rozdělení požaduje. V ukázce kódu je zobrazeno rozdělení pro dobu chůze v prvním patře.

```
public double vratCasChuzeChodba(Chodba o){
    double cas=0.0;
    double pravdepodobnost= generator.nextDouble()*100;
    switch(o.patro){
        case PRVNI:
                                if (pravdepodobnost < 24.57) {

                                   cas =  (3 + (7 - 3) * generator.nextDouble());

                                } else if (pravdepodobnost >= 24.57 & pravdepodobnost < 71.64) {

                                    cas =  (7 + (11 - 7) * generator.nextDouble());

                                } else if (pravdepodobnost >= 71.64 & pravdepodobnost < 92.54) {

                                    cas =  (11 + (15 - 11) * generator.nextDouble());

                                } else if (pravdepodobnost >= 92.54 & pravdepodobnost < 97.01) {

                                   cas = (15 + (19 - 15) * generator.nextDouble());

                                } else if (pravdepodobnost >= 97.01 & pravdepodobnost < 98.51) {

                                    cas =  (19 + (23 - 19) * generator.nextDouble());

                                } else if (pravdepodobnost >= 98.51) {

                                    cas =  (23 + (27 - 23) * generator.nextDouble());

                                }
                                return cas;
```

Obrázek 22: metoda vratCasChuzeChodba().

7.7.5 Metoda vratCekaniVytah()

Tato metoda řeší čekání na výtah podle naměřených hodnot. Uvnitř metody se generuje pravděpodobnost z generátoru typu Random, který je pak testován podle rozsahu z následující tabulky a rozdělení je realizováno vzorcem v kapitole 6.4.2.1.

Intervaly [s]	hodnoty pravděpodobnosti	Pravděpodobnost [%]
0-13	0-54,55	54,55
13-26	54,55-63,64	9,09
26-39	63,64-77,27	13,64
39-52	77,27-86,36	9,09
52-65	86,36-90,91	4,55
65-78	90,91-95,45	4,55
78-91	a více než 95,45	4,55
∑		100,00

Tabulka 8: Pravděpodobnost čekání na výtah.

7.7.6 Metoda vratCasPrichodLidi()

Poslední metoda, která je v třídě Generátor napsána, je metoda *vratCasPrichodLidi()*. Metoda řeší rozložení příchodu lidí do budovy od jejího otevření v 07:30 hodin.

Časové rozdělení[s]	Pravděpodobnost [%]
0-1700	26,92
1700-3400	19,23
3400-5100	15,38
5100-6800	11,54
6800-8500	7,69
8500-10200	7,69
10200-11900	5,77
11900-13600	5,77

Tabulka 9: Příchod lidí do budovy

Z tabulky je patrné, že budova je nejvíce vytížená v ranních hodinách a s postupem času příchod lidé slábne.

```
if (pravdepodobnost < 28.57) {
    cas = (0 + (1700 - 0) * generator.nextDouble());
} else if (pravdepodobnost >= 28.57 & pravdepodobnost < 48.98) {

    cas = (1700 + (3400 - 1700) * generator.nextDouble());

} else if (pravdepodobnost >= 48.98 & pravdepodobnost < 65.31) {

     cas = (3400 + (5100 - 3400) * generator.nextDouble());

} else if (pravdepodobnost >= 65.31 & pravdepodobnost < 77.55) {

     cas = (5100 + (6800 - 5100) * generator.nextDouble());

}  else if (pravdepodobnost >= 77.55 & pravdepodobnost < 85.71) {

   cas = (6800 + (8500 - 6800) * generator.nextDouble());

} else if (pravdepodobnost >= 85.71 & pravdepodobnost < 93.88) {

   cas = (8500 + (10200 - 8500) * generator.nextDouble());

}else{
     cas = (10200 + (11900 - 10200) * generator.nextDouble());
}
```

Obrázek 23: metoda vratCasPrichodLidi()

7.8 Verifikace

7.8.1 Verifikace modelu

Verifikace modelu vychází především z ověření správnosti modelu v porovnání s modelovanou budovou. Model nám obsahuje veškeré náležitosti, jako jsou ordinace, výtah, chodby, schodiště. Model byl taktéž porovnán s modelem vytvořeným Lukášem Bašem v prostředí Arena, kde oba modely prokázaly vysokou míru shody.

7.8.2 Verifikace třídy Generátor

Pro ověření správnosti jednotlivých rozdělení z třídy Generátor, byla vytvořena jednoduchá statistická aplikace. V aplikaci bylo vytvořeno 100 osob. Při počtu replikací rovné deseti a pomocí instance Generátoru jim byly přiřazeny cíle v bodově, tj. cílová patra, ordinace, využití schodů, využití výtahu, možnosti přestupu. V rámci aplikace byla data shromažďována, podrobena zkoumání a srovnána s měřením.

Schody	Model[%]	Měření[%]
1.patro	30,13	29,30
2.patro	16,63	15,48
3.patro	17,50	17,44
Přízemí	35,75	37,78

Tabulka 10: Verifikace schody.

Tabulka zachycuje rozdělení osob do jednotlivých pater budovy. Z tabulky je jasné, že model odráží skutečnost zjištěnou měřením.

Výtah	Model[%]	Měření[%]
1.patro	3,5	3,82
2.patro	6,6	5,41
3.patro	9,1	8,58

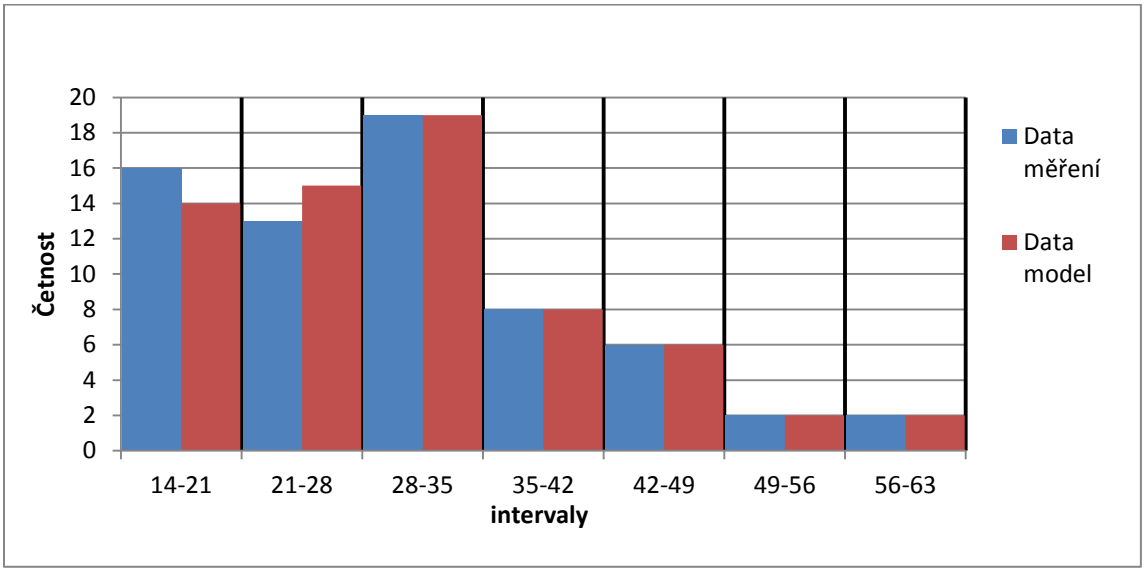
Tabulka 11: Verifikace výtah.

I z tabulky verifikace výtahů je patrné, že model odráží skutečnost.

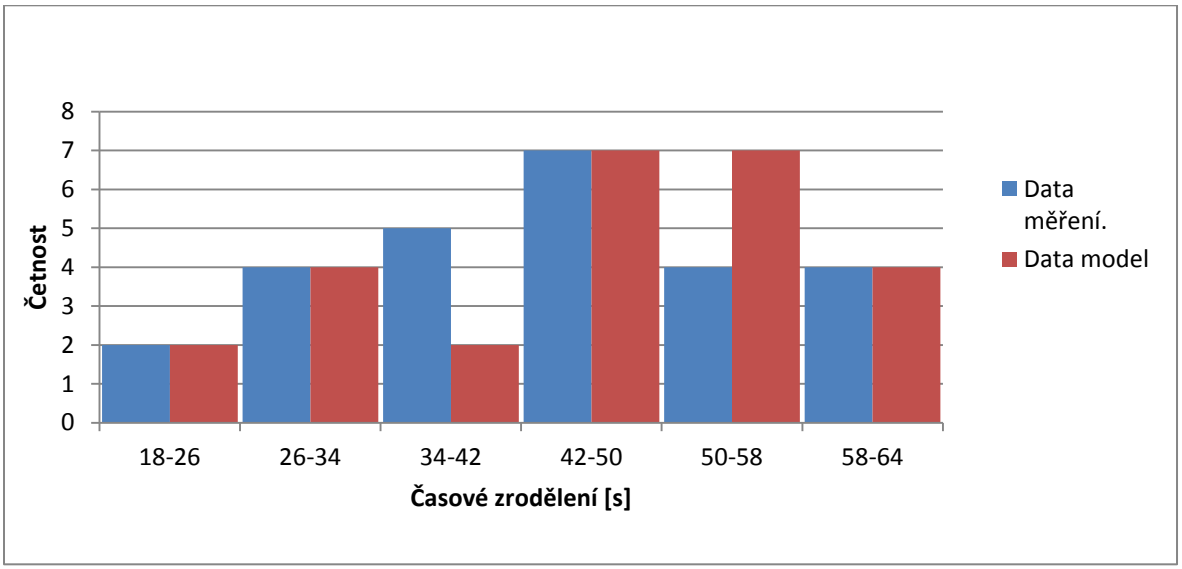
Přestup	Model[%]	Měření[%]
1p. Do 2p.	10,7	11,04
2p. Do 3p.	3	2,98
3p. Do 2p.	3,9	3,54
2p. Do 1p.	2,5	1,90

Tabulka 12: Verifikace přestupy.

Z výše uvedených tabulek je zřejmé, že zvolené rozdělení pro jednotlivé případy je správné a vytvořený simulační model odráží realitu simulovaného objektu. Další možností verifikace dat, je zkontrolování časových rozdělení pro jednotlivé metody. Pro tento účel byla vybrána metoda ze třídy Generátor *vratCasVysputuPatro()*, která řeší případ výstupu osob do prvního a druhého patra ze vstupu budovy. Data byla z projektu zapsána do textového souboru a porovnána s údaji z měření.



Obrázek 24: Verifikace metoda vratCasVysputuPatro() 1p.



Obrázek 25: Verifikace metoda vratCasVysputuPatro() 2p.

8 Tvorba distribuované simulace HLA

Pro potřeby tvory distribuované simulace jsou vytvořeny dva projekty. První projekt se stará o vstup lidí do budovy a také obstarává dvě patra, přízemí a první patro. Druhý projekt má za úkol simulovat třetí a druhé patro. Pro tvorbu HLA simulace je využit framework HLA-VA.

8.1 Vytvoření Ambassadora a připojení k Federaci

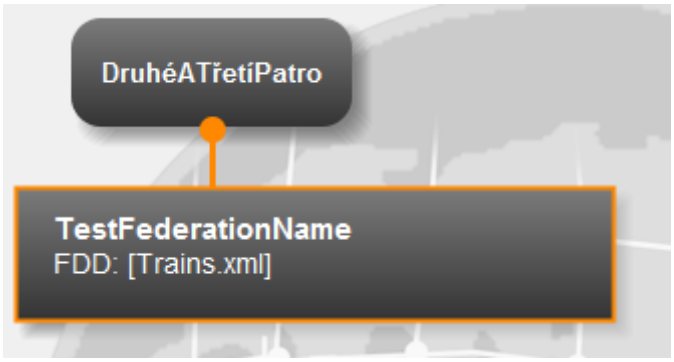
Ambassador je umístěn v každém běžícím federátu. Ambassador nám umožňuje volat funkce RTI a provádět i zpětná volání. Každý federát obsahuje dva Ambassadory, a to RTI Ambassador a Federete ambassador. V API pro jazyk Java mají rozhraní RTIambassador i FederateAmbassador podobu rozhraní tohoto jazyka. Za implementaci rozhraní RTIambassador je zodpovědný dodavatel software RTI.

```
RtiFactory rtiFactory = RtiFactoryFactory.getRtiFactory();
_rtiAmbassador = rtiFactory.getRtiAmbassador();
_encoderFactory = rtiFactory.getEncoderFactory();

_rtiAmbassador.joinFederationExecution(USERNAME, FEDERATETYPE, FEDERATION_NAME);
```

Obrázek 26: Ambassador.

Příklad připojení k spuštěné federaci, která má vždy svůj jedinečný název, jelikož může existovat více běžících federací. V případě, že bylo vytvoření Ambassadora a připojení k běžící federaci úspěšné, v programu pRTI se objeví název federace a připojený federát.



Obrázek 27: pRTI

8.2 Odesílání zpráv federátům

Pro odesílání a příjímání zpráv je ve frameworku v třídě Federate napsána metoda *sendMessage*, jejímž vstupním argumentem je řetězec String, který nese samotnou zprávu. Metoda *sendMassage* je volána ze třídy HLAKomunikace.

```
ParameterHandleValueMap parameters = _rtiAmbassador.getParameterHandleValueMapFactory().create(1);
HLAunicodeString messageEncoder = _encoderFactory.createHLAunicodeString();
messageEncoder.setValue(message);
parameters.put(_parameterIdText, messageEncoder.toByteArray());
parameters.put(_parameterIdSender, messageEncoder.toByteArray());
_rtiAmbassador.sendInteraction(_messageId, parameters, null);
```

Obrázek 28: Úryvek metody *sendMessage*

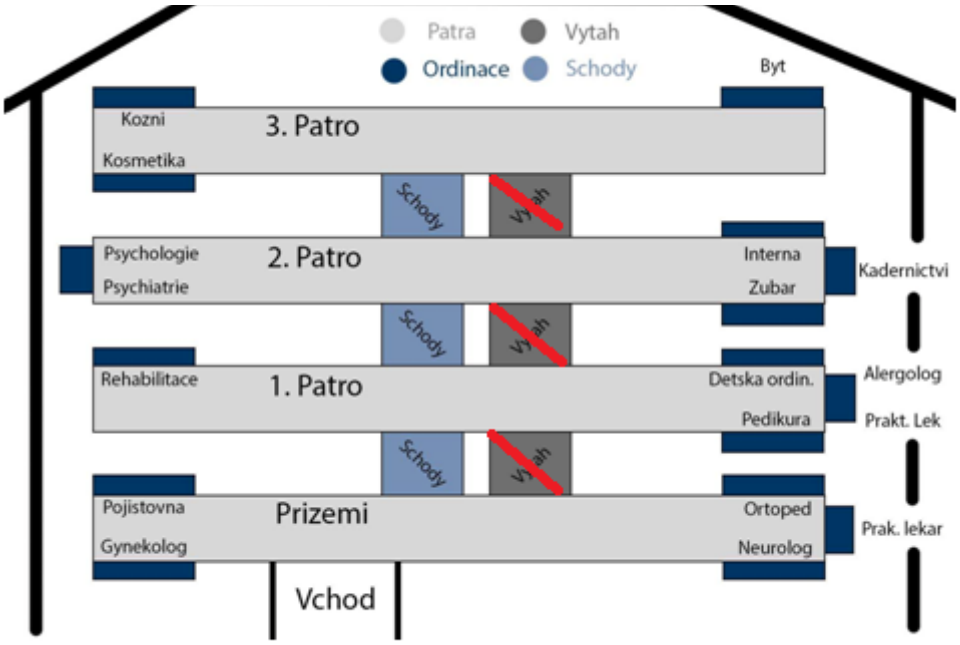
Pro příjímání zpráv je ve frameworku metoda *receiveInteraction*, která po přijetí zprávy zavolá metodu *incomingMessageCallBack* ze třídy HLAKomunikace, kde se vypíše obsah zprávy a jméno odesílatele. Následně je pak zavolána metoda *prijmy* ze třídy Program, která má implementovánu veškerou logiku simulace. Metoda *prijmy* má jako vstupní argument řetězec typu String, se kterým se dále pracuje.

Příklad komunikace:

Po připojení hlavního projektu, který má na starost příchod lidí, přízemí a první patro, se čeká na připojení druhého projektu, který řeší druhé a třetí patro. Po připojení druhý projekt odešle String s obsahem "start". První projekt zatím neaktivně čeká (využití metody *wait ()). Po* přijmutí první zprávy projektem je vlákno probuzeno a program začíná svoji činnost. Po skončení simulace první projekt odesílá zprávu "stop" a po přijmutí druhým projektem se program ukončí.

9 Alternativní scénáře

Po vytvoření distribuované simulace podle standardu HLA, se nám naskýtá možnost nasimulovat méně pravděpodobné jevy v budově. Například porucha výtahu, uzavření jednoho patra, či zrušení určité ordinace. Pro náš případ vybereme možnost nefunkčnosti výtahu.



Obrázek 29: Model budovy bez výtahů

Při tomto scénáři se zvýší využití schodů o následující přírůstek:

Schody	%- Model bez výtahu	% - Model s výtahem
1. patro	29,47	32,73
2. patro	17,15	53,57
3. patro	17,57	25,47

Tabulka 13: Schodiště včetně výtahů

Dalším možným alternativním scénářem je přestěhování, či zrušení ordinací v rámci pater, kde by se změnila pravděpodobnost přístupů do dané ordinace, respektive do daného patra. Jako příklad vezmeme zrušení ordinace kožního oddělení ve třetím patře. Do této ordinace osoby přicházely v 45 % případů. Pravděpodobnost příchodu do třetího patra by klesla na hodnotu 4,61%. Dalším ovlivněním by byla změna přestupů z třetího a do třetího patra.

Závěr

Cílem této bakalářské práce bylo vytvořit distribuovaný simulační program v jazyce Java dle standardu High Level Architecture (HLA). Teoretická část práce popisuje programovací jazyk Java a hlavní principy HLA.

Navrhované řešení modelu polikliniky je poměrně rozsáhlé a skládá se ze dvou částí. První část se zabývá tvorbou simulačního modelu, kde bylo potřeba shromáždit dostatečné množství dat pro tvorbu simulačního programu. Samostatné měření proběhlo ve třech dnech v budově polikliniky v Dvoře Králové nad Labem. Dále byla naměřená data zpracována a bylo sestaveno množství histogramů, ze kterých se zpracovalo časové rozdělení pro příslušné případy.

Druhá část se zabývá tvorbou simulačního modelu v jazyce Java a následnou implementací HLA s využitím frameworku HLA-VA. Při tvorbě simulačního modelu bylo potřeba navrhnout a naprogramovat třídy pro správný chod simulace, verifikovat a validovat časová rozdělení pro zajištění správnosti simulace. Posledním krokem bylo vytvoření distribuované simulace, kde se simulační model rozdělil na dva projekty a vypracování alternativních scénářů.

Veškeré zdrojové kódy jsou dostupné na přiložném CD, včetně souborů se zpracovanými histogramy a naměřenými hodnotami.

Použitá literatura

[1] HEROUT, Pavel. *Java* [online]. 2009 [cit. 2014-04-05]. Dostupné z WWW: <http://www.pit-plzen.cz/sites/default/files/java.pdf>.

[2] THE HLA TUTORIAL[online]. 2014 [cit. 2014-04-26]
Dostupné z WWW:
<http://www.pitch.se/images/files/tutorial/TheHLAtutorial.pdf>.

[3] KOŽEŠINA, Stanislav. Distrbuovaná simulace podle standartu HLA pro použití v smulační knihovně J-SIM, 2005. 86 s. Dimplomová práce. Fakulta aplikovaných věd Univerzita Plzeň.

[4] SLAVÍČEK, Pavel. Distribuované simulační prostředí. [s.l.], 2006. 90 s. Dizertační práce. Vysoké učení technické v Brně.

[5] SCHILDT, Herbert. *Java 7: výukový kurz*. 1. vyd. Brno: Computer Press, 2012, 664 s. ISBN 978-80-251-3748-2.

[6] ECKEL, Bruce. *Thinking in Java*. 2nd ed. Upper Saddle River, NJ: Prentice Hall, 2000, 1127 p. ISBN 01-302-7363-5.

[7] Wikipedia [online]. 2014 [cit. 2014-04-26]. High level architecture (simulation). Dostupné z WWW:
<http://en.wikipedia.org/wiki/Java_version_history#J2SE_1.4_.28February_6.2C_2002.29>

[8] BROŽEK, Josef, Antonín KÁVIČKA a ONGGO. *High level architecture virtual assisant framework*. Pardubice, 2014.

[9] KŘIVÝ , Ivan, KINDLER, Evžen. Simulace a modelování. 1. vyd. Ostrava: Ostravská univerzita, 2001. 146 s. ISBN 80-7042-809-0