ABAcore Java

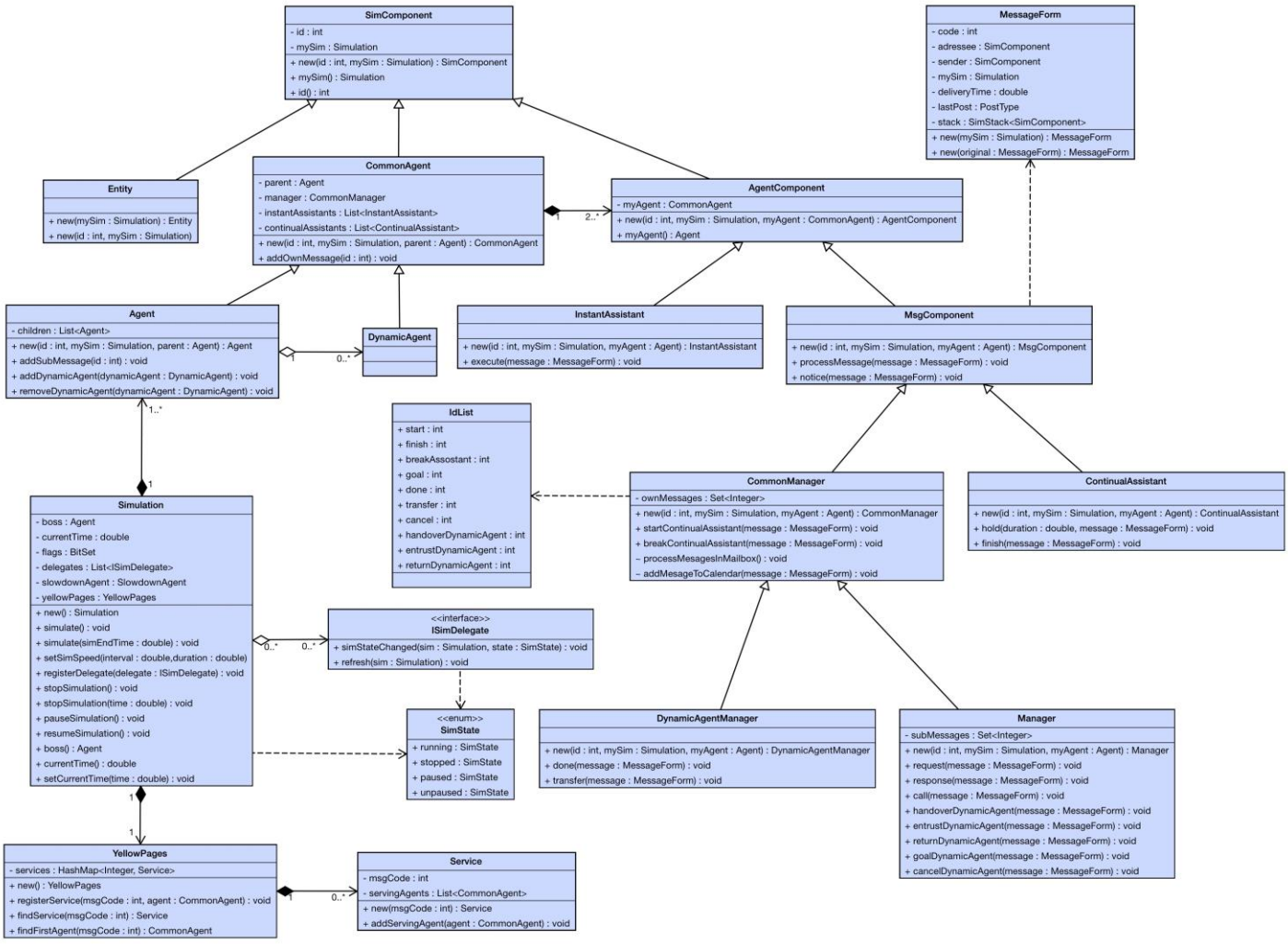
Metodická príručka

Návrh simulačného modelu

Najskôr je potrebné identifikovať agentov, z ktorých sa bude skladať simulačný model a správy, ktoré si agenti medzi sebou posielajú. Agenti tvoria hierarchickú štruktúru. Na vrchole hierarchie je agent boss, ktorý je zodpovedný za celý simulačný model. Každý agent má definované za čo je zodpovedný, pričom môže určitú časť delegovať svojim podriadeným agentom. Každý agent môže mať ľubovolný počet podriadených agentov a s výnimkou agenta modelu na vrchole hierarchie má práve jedného rodiča. Každý agent musí mať jedného manažéra a môže mať ľubovolný počet asistentov.

Správa poslaná medzi dvoma agentami môže byť niekoľkých typov: notifikácia (notice), na ktorú sa neočakáva odpoveď, alebo požiadavka (request), na ktorú je požadovaná odpoveď (response).

Diagram tried simulačného jadra



Implementácia simulačného modelu

Pre jednotlivých agentov a komponenty sú vytvorené triedy, ktoré sú potomkami tried balíčka OSPABA a sú v nich prekryté príslušné abstraktné metódy. Nasleduje popis toho, ako je možné tieto triedy implementovať:

Simulácia

Trieda reprezentuje agentovo orientovanú simuláciu. Je odvodená od triedy Simulation balíčka OSPABA.

```
import OSPABA.Simulation;
public class MySimulation extends Simulation
{ ... }
```

Pre každého agenta je vytvorený atribút a je inicializovaný v konštruktore. Z agentov je práve jeden agent boss - agent na vrchole hierarchie. Boss je určený tým, že parameter konštruktora 'parent' má hodnotu null.

```
private MyAgent1 _agent1;
public MySimulation()
{
  _agent1 = new MyAgent1(Id.agent1, this, null);
  _agent2 = new MyAgent2(Id.agent2, this, _agent1);
}
```

Trieda Simulation obsahuje metódy prepareSimulation(), prepareReplication(), replicationDidFinish() a simulationDidFinish() ktoré môžu byť prekryté a doplnené o kód ktorý je vykonaný v konkrétnych bodoch životného cyklu simulácie.

```
@Override public void prepareSimulation() {
   super.prepareSimulation();
   // Kód vykonaný pri spustení simulácie
}
@Override public void prepareReplication() {
   super.prepareReplication();
   // Kód vykonaný na začiatku každej replikácie
}
@Override public void replicationFinished() {
   // Kód vykonaný na konci každej replikácie
   super.replicationDidFinish();
}
@Override public void simulationFinished() {
   // Kód vykonaný po ukončení simulácie
   super.simulationDidFinish();
}
```

V prekrytých metódach môže byť realizovaný zber štatistýk a aktualizácia UI. Okrem prekrytia metód je možné dodať kód ktorý má byť vykonaný pri týcho udalostiach aj prostredníctvom lambda funkcii.

```
MySimulation sim = new MySimulation();
sim.onPrepareSimulation(s -> {
    // Kód vykonaný pri spustení simulácie
  });

sim.onPrepareReplication(s -> {
    // Kód vykonaný pri spustení simulácie
  });
```

Simulácia je spustená volaním metódy simulate(int replicationCount) alebo simulate(int replicationCount, double simEndTime). Je vykonaných replicationCount replikácií a v druhom prípade je naplánované ukončenie simulácie na čas simEndTime.

```
int replicationCount = 50, replicationLength = 10000;
MySimulation sim = new MySimulation();
sim.simulate(replicationCount, replicationLength);
```

Správa

Agenti spolu komunikujú posielaním správ. Všetky správy sú potomkami triedy OSPABA.MessageForm. Pred odoslaním správy je potrebné nastaviť parametre, ktoré špecifikujú akým spôsobom bude správa doručená. Z hľadiska používateľa knižnice sú najdôležitejšími parametre code a addressee. Je niekoľko spôsobov, akými sa posielajú správy, pričom niektoré z nich nastavujú tieto parametre automaticky.

Parameter addressee definuje adresáta správy - komponent, ktorému bude správa doručená. Adresátom môže byť agent alebo asistent agenta. Asistenti agenta môžu komunikovať jedine s manažérom agenta, preto pri posielaní správy manažérovi nie je potrebné nastavovať adresáta správy.

Atribút code, definuje kód správy, podľa ktorého adresát správy vie, akým spôsobom ju má spracovať.

```
MyMessage message = new MyMessage(sim);
message.setCode(Mc.codeA);
message.setAddressee(Id.agentA);
```

Je vytvorený potomok triedy OSPABA.MessageForm a je potrebné implementovať abstraktnú metódu createCopy(). Dôvodom, prečo nestačí kopírovací konštruktor je to, že simulačné jadro vytvára kópie správ, ktorých trieda je definovaná mimo knižnice.

```
import OSPABA.MessageForm;
```

```
public class MyMessage extends MessageForm {
  private SomeEntity _value;
  public MyMessage(Simulation mySim) {
    super(mySim);
    _value = defaultValue;
  }
  public MyMessage(MyMessage original) {
    super(original);
    _value = original._value;
  }
  @Override public MessageForm createCopy()
  { return new MyMessage(this); }
  public SomeEntity value()
  { return _value; }
  public void setValue(SomeEntity value)
  { _value = value; }
}
```

Identifikátory správ a komponentov simulácie

Každá správa a každý komponent (agent, manažér, asistent) musí mať jedinečný identifikátor. Bolo by vhodné, aby boli prístupné odkiaľkoľvek, čo je možné dosiahnuť niekoľkými spôsobmi. Odporúčaným riešením je vytvoriť podtriedu triedy OSPABA.IdList a identifikátory definovať ako jej statické atribúty. Vytvára sa trieda, nie enum, pretože v jazykoch java a C# nie je možné pre enum vytvoriť potomka. OSPABA.IdList používa simulačné jadro a sú v nej už definované niektoré identifikátory, napríklad start a finish, ktoré používajú kontinuálni asistenti. Je teda možné pracovať rovnakým spôsobom so systémovými a používateľskými identifikátormi. Je vhodné definovať dve triedy, jednu pre kódy správ a jednu pre identifikátory komponentov (agentov, manažérov a asistentov).

```
public class MC extends IdList {
  public static final int messageA = 0;
  public static final int messageB = 1;
}
public class Id extends IdList {
  public static final int agentXXX = 0;
  public static final int managerXXX = 1;
  public static final int assistentXXX = 2;
}
```

K identifikátorom týchto tried potom môžeme pristupovať nasledovným spôsobom:

```
Id.agentXXX  // používateľom definovaný identifikátor agenta
Mc.messageA  // používateľom definovaný kód správy
Mc.start     // identifikátor poskytnutý simulačným jadrom
```

Riadiaci agent

Trieda predstavuje základný stavebný prvok agentovo orientovanej simulácie - agenta. Agenti sú usporiadaní v hierarchickej štruktúre na vrchole ktorej stojí agent nazývaný "Boss". Každý agent sa skladá s komponentov, ktoré môžeme rozdeliť do 3 skupín: manažér, kontinuálni asistenti a okamžití asistenti. Každá skupina má presne vymedzené kompetencie a právomoci. V konštruktore sú inicializovaní asistenti a manažér agenta. Každému je prvým parametrom konštruktora priradený jedinečný identifikátor. Komponenty agenta sú po vytvorení automaticky uložené v nadtriede agenta, preto ich nie je potrebné uložiť ako atribút vo vytvorenej podtriede. V prípade, že je potrebné ku komponentu priamo pristúpiť, je ho možné vyhľadať metódou agenta findAssistant(int), alebo sa preň vytvorí atribút. Okrem toho je potrebné zaregistrovať všetky správy, ktoré je agent schopný spracovať.

Atribúty ktoré je potrebné inicializovať na začiatku každej replikácie by mali byť inicializované v metóde prepareReplication(), ktorá je automaticky spustená pred začiatkom každej replikácie, pričom je potrebné túto metódu zavolať aj na nadtriede. Metódu prepareReplication je možné prekryť aj na komponentoch agenta (manažér, asistenti).

```
import OSPABA.Agent;
public class MyAgent extends Agent
{
  private Stat _stat;
  private AssistantYYY _assistantYYY;
  public MyAgent(int id, Simulation m'Sim, Agent parent)
  {
    super(id, mySim, parent);
    new MyManager(Id.managerXXX, mySim, this);
    new AssistantXXX(Id.assistantXXX, mySim, this);
    _assistantYYY = new AssistantYYY(Id.assistantYYY, mySim, this);
    addOwnMessage(Mc.messageB);
  }
  @Override public void prepareReplication() {
    super.prepareReplication();
    _stat = new Stat();
  }
  public AssistantYYY assistantYYY()
  { return _assistantYYY; }
}
```

Manažér

Manažér je komponentom riadiaceho agenta, ktorý zabezpečuje komunikáciu s inými agentami a riadenie správania agenta pričom používa asistentov agenta. Manažér agenta je odvodený od triedy Manager balíčka OSPABA.

Je potrebné prekryť abstraktnú metódu processMessage(MessageForm). Táto metóda spracuje všetky správy poslané agentovi. Je teda potrebné rozlíšiť o akú správu sa jedná podľa jej kódu získaného metódou code() parametra message a spracovať ju adekvátnym spôsobom.

Agent môže prijať správu s totožným kódom od viacerých odosielateľov. V takom prípade môže byť nutné zistiť, ktorý komponent túto správu poslal, napríklad ako message.sender().id().

```
import OSPABA.Manager;
public class MyManager extends Manager {
  public MyManager(int id, Simulation mySim, Agent myAgent) {
    super(id, mySim, myAgent);
  }
  @Override public void processMessage(MessageForm message) {
    switch(message.code()) {
    case Mc.messageA:
      // spracovanie správy s kódom Mc.messageA
    break;
    case Mc.finish:
      switch (message.sender().id()) {
      case Id.assistantA:
        // spracovanie správy s kódom Mc.finish v prípade,
        // že odosielateľom je asistent A
      break;
      case Id.assistantB:
        // spracovanie správy s kódom Mc.finish v prípade,
        // že odosielateľom je asistent B
      break;
      }
    break;
    }
  }
}
```

Pri spracovaní správy môže manažér posielať správy iným agentom. Ak posiela len jednu správu, môže preposlať práve spracovávanú správu, ktorej predtým nastaví požadované parametre (adresát, kód, …). Pre vyhľadanie agenta, ktorý bude nastavený ako adresát správy, je možné použiť metódu findAgent() objektu simulácie. Na poslanie správy použije jednu z metód notice, request alebo response. V prípade použitia response, nie je potrebné špecifikovať adresáta správy, ten je automaticky nastavený na agenta, ktorý správu poslal metódou request. Request je možné urobiť viacnásobne.

```
message.setCode(Mc.messageA);
message.setAddressee(mySim().findAgent(Id.agentXXX));
notice(message);
```

V prípade, že je posielaných správ niekoľko, vytvoria sa kópie spracovávanej správy kopírovacím konštruktorom alebo metódou createCopy().

```
original.setCode(Mc.messageB);
original.setAddressee(mySim().findAgent(Id.agentXXX));
```

```
MessageForm copy = original.createCopy();
copy.setCode(Mc.messageC);
request(original);
response(copy);
```

Manažér môže pri spracovaní správy spustiť prácu kontinuálneho asistenta, alebo nechať vykonať činnosť promptného asistenta.

```
messageD.setAddressee(myAgent().findAssistant(Id.continualXXX));
startContinualAssistant(messageD);

messageE.setAddressee(myAgent().findAssistant(Id.instantXXX));
execute(messageE);
```

Spracovanie správy väčšinou metód (notice, request, response, startContinualAssistant, assistantFinished, hold, …) funguje tak, že je správa doručená do poštovej schránky agenta (alebo je zaradená do kalendára správ v prípade, že sa jedná o správu s oneskorením) a príjemca ju spracuje neskôr. To znamená, že po návrate metódy správa ešte nebola spracovaná. Výnimkou sú metódy execute a call, ktoré správu spracujú okamžite.

Asistenti agenta

Manažér pre svoju činnosť využíva asistenciu ďalších interných komponentov agenta, asistentov. Asistentov môžeme rozdeliť podľa viacerých hľadísk.

Z hľadiska funkcie asistentov:

1. Senzory (dotaz a monitor) - zabezpečujú sprístupnenie informácií o stave prostredia,
2. Riešitelia (poradca a plánovač) - podporujú rozhodovanie manažéra tým, že mu poskytujú návrhy riešenia problémov,
3. Efektory (akcia a proces) - vykonávajú rozhodnutia manažéra o zmene stavu systému.

Z hľadiska trvania činnosti asistentov:

1. Okamžití asistenti (dotaz, poradca, akcia) - vykonajú svoju činnosť okamžite bez zmeny simulačného času,
2. Kontinuálni (monitor, plánovač, proces) - výkon ich činnosti trvá nenulový simulačný čas.

Okamžitý asistent

Je potomkom jednej z tried OSPABA.Action, OSPABA.Adviser a OSPABA.Query. Musí prekryť abstraktnú metódu execute(MessageForm), v ktorej vykoná činnosť, za ktorú je daný promptný asistent zodpovedný.

```
import OSPABA.Action
public MyAction extends Action {
  public MyAction(int id, Simulation mySim, CommonAgent myAgent) {
    super(id, mySim, myAgent);
```

```
  }
  @Override public void execute(MessageForm message) {
    // činnosť asistenta
  }
}
```

Kontinuálny asistent

Kontinuálny asistent je potomkom jednej z tried OSPABA.Process, OSPABA.Scheduler alebo OSPABA.Monitor. Musí prekryť abstraktnú metódu processMessage(MessageForm). Spôsob spracovania správ je podobný spracovaniu správ u manažéra agenta s tým rozdielom, že sa nepoužívajú správy typu request a response, ale správy typu notice, finish a hold. Kontinuálny asistent je spustený manažérom metódou startContinualAssistant(MessageForm), ktorá automaticky nastaví kód správy na Mc.start. Manažér tiež môže ukončiť činnosť kontinuálneho asistenta metódou breakContinualAssistant(MessageForm), ktorá asistentovi doručí správu s kódom Mc.breakCA. Po ukončení svojej činnosti o tom asistent notifikuje svojho agenta poslaním správy metódou assistantFinished, ktorá automaticky nastaví kód správy na Mc.finish.

```
import OSPABA.Process;
public class MyProcess extends MyProcess {
  public MyProcess(int id, Simulation mySim, CommonAgent myAgent) {
    super(id, mySim, myAgent);
  }
  @Override public void processMessage(MessageForm message) {
    switch (message.code()) {
    case Mc.start:
      message.setCode(Mc.mesageA)
      hold(duration, message);
    break;
    case Mc.mesageA:
      // automaticky nastaví kód správy na Mc.finish a
      // adresáta na myAgent
      assistantFinished(message);
    break;
    case Mc.breakCA:
      // ...
    break;
    }
  }
}
```

Metóda hold má na rozdiel od ostatných metód určených pre posielanie správ dodatočný parameter: duration - trvanie činnosti. Je to jediný spôsob, akým je možné posunúť simulačný čas. Adresátom správy je vždy komponent, ktorý túto správu odoslal. Trvanie činnosti môže byť dané rozdelením pravdepodobnosti. Pre generovanie týchto čísel je možné použiť napríklad balíček OSPRNG. V takom prípade bude objekt generátora rozdelenia

pravdepodobnosti atribútom kontinuálneho asistenta a pri použití metódy hold ako parameter trvania činnosti dosadíme vzorku vygenerovanú generátorom.

```
import OSPRNG.ExponentialRNG;
// konštruktor:
ExponentialRNG _exp = new ExponentialRNG(EX); // EX = stredná hodnota
// processMessage:
hold(_exp.sample(), message);
```

Po dosiahnutí simulačného času posunutého o trvanie činnosti od zaslania správy typu hold je asistentovi táto správa doručená. Spôsob spracovania správy môže byť rôzny. Pri type proces (Process), asistent notifikuje agenta o jeho ukončení. Pri type plánovač (Scheduler), asistent notifikuje agenta o výskyte naplánovanej udalosti a súčasne naplánuje novú udalosť.

Príklad: Proces - notifikuje manažéra o ukončení procesu

```
case Mc.messageA:
  // automaticky vyplní addressee na myAgent() a code na Mc.finish
  assistantFinished(message);
break;
```

Príklad: Plánovač - notifikuje manažéra o výskyte udalosti a naplánuje ďalšiu (napríklad príchod zákazníkov do systému z okolia)

```
case Mc.messageA:
  // naplánovanie ďalšej udalosti
  MessageForm copy = message.createCopy();
  hold(_exp.sample(), copy);

  // notifikácia manažéra
  message.setCode(Mc.messageB);
  message.setAddressee(myAgent());
  notice(message);
break;
```

Po ukončení činnosti asistenta (metóda assistantFinished) je jeho agentovi doručená správa s kódom Mc.finish. Ak má manažér niekoľko kontinuálnych asistentov rozlišuje, ktorý z nich ukončil svoju činnosť na základe odosielateľa správy (message.sender().id()).

Dynamickí agenti

Na rozdiel od riadiacich agentov, ktorí tvoria pevnú štruktúru definovanú pri spustení simulácie, dynamickí agenti môžu vznikať a zanikať počas behu simulácie. Dynamický agent vždy patrí pod správu jedného riadiaceho agenta, ktorý mu zadáva ciele, ktoré je dynamický agent povinný plniť, pričom môže sledovať aj vlastné ciele, pokiaľ nie sú v rozpore s cieľom zadaným riadiacim agentom.

Dynamickí agenti modelujú inteligentné entity systému. Nie sú na konci replikácie zmazaní, preto v prípade, že dynamickí agenti vznikajú a zanikajú počas behu simulácie (t.j. ich počet nie je stanovený na začiatku simulácie a teda sú vytvorení v konštruktore riadiaceho agenta pri spustení simulácie), je potrebné pred spustením replikácie odstrániť dynamických agentov z predchádzajúcej replikácie. Dynamických agentov je môžné zmazať volaním metódy clearDynamicAgents() riadiaceho agenta.

```
public void prepareReplication() {
  clearDynamicAgents();
  super.prepareReplication();
}
```

V príklade je clearDynamicAgents() zavolané pred super.prepareReplication(), pretože to zavolá prepareReplication() na existujúcich dynamických agentoch, čo síce nie je v tomto prípade nesprávne, ale je to zbytočné.

Notifikácie o zmene stavu simulácie

Sledovanie zmien stavu simulačného jadra je možné jedným s dvoch spôsobov.

Prvou možnosťou je priame vloženie kódu, ktorý sa má vykonať pri zmene stavu do simulačného jadra prostredníctvom lambda funkcií.

```
Simulation simulation = new MySimulation();
simulation.onRun(sim -> { ... });
simulation.onStop(sim -> { ... });
simulation.onPause(sim -> { ... });
simulation.onResume(sim -> { ... });
simulation.onRefreshUI(sim -> { ... });
```

Druhou možnosťou je registrácia delegáta simulácie - návrhový vzor observer. Simulácia môže mať ľubovolný počet delegátov. Táto trieda je notifikovaná o zmenách stavu simulácie a v prípade, že je nastavená rýchlosť simulácie je tiež notifikovaná o potrebe prekreslenia grafického rozhrania. Trieda implementuje rozhranie ISimDelegate a zaregistruje sa metódou registerDelegate(ISimDelegete) objektu simulácie.

```
import OSPABA.ISimDelegate;
public class MyDelegate implements ISimDelegate {
        public void simStateChanged(Simulation sim, SimState state) {
                switch(state) {
                case running:
                        // ...
                break;
                case stopped:
```

```
                    // ...
                break;
                case paused:
                    // ...
                break;
                }
        }
        public void refresh(Simulation sim) {
                // aktualizácia grafického rozhrania
        }
}
```

Spomalenie simulácie je možné nastaviť pomocou metódy setSimSpeed(interval, duration) objektu simulácie. Každých ‘interval’ jednotiek simulačného času simulácia zastane na ‘duration’ sekúnd a delegáti simulácie sú o tom notifikovaní zavolaním metódy refresh(Simulation) a je vykonaný kód onRefreshUI. Volaním metódy setMaxSimSpeed() je vypnuté spomalenie simulácie.

Synchronizácia simulácie

Pri aktualizácii UI je potrebné predísť súčasnému prístupu k údajom vo vlákne simulácie a GUI vlákne. Počas vykonávania metód popisovaných v predchádzajúcej sekcii (refresh a onRefreshUI) je vykonávanie simulácie zastavené. V prípade potreby prístupu k údajom simulácie z asynchrónneho vlákna poskytuje simulačné jadro metódy invokeSync() a invokeAsync(). Metóda invokeSync() blokuje aktuálne vlákno pokým nie sú spracované všetky správy v aktuálnom simulačnom čase.

```
int value;
sim.invokeSync(() ->{
    value = sim.agent009().value();
  });
// use value

int value = sim.invokeSync(() ->{
    return sim.agent009().value();
  });
// use value


sim.invokeAsync(() ->{
    int value = sim.agent009().value();
    // use value
  });
```

Generátory náhodných čísel

Balíček OSPRNG poskytuje niekoľko generátorov náhodných čísel rôznych rozdelení. Všetky generátory sú potomkami triedy RNG. Pre každé rozdelenie je vytvorený objekt a metódou sample() je vygenerované pseudonáhodné číslo z daného rozdelenia.

Príklad použitia generátorov:

1. Importovanie tried balíčka OSPRNG

```
import OSPRNG.*;
```

2. Vytvorenie inštancie rozdelenia (v príklade exponenciálneho zo strednou hodnotou 11 a empirického rozdelenia). Generátory majú niekoľko konštruktorov. Okrem konštruktora, ktorého parametrami sú len parametre rozdelenia môžu naviac preberať ako parameter generátor násad typu java.util.Random a offset ktorý posunie hodnoty generované generátorom o zadanú hodnotu.

```
// exponenciálne rozdelenie so strednou hodnotou 11
ExponentialRNG _exp = new ExponentialRNG(11d);

// erlangovo rozdelenie so strednou hodnotou 0.5, disperziou 0.01
// a vlastným generátorom násad
ErlangRNG _erlang = new ErlangRNG(0.5, 0.01, new Random(5367029));

// normálne rozdelenie so strednou hodnotou 2, smerodajnou
// odchylkou 0.5 a offsetom 1.5. Generuje hodnoty 1.5 + N(2, 0.5)
NormalRNG _normal = new NormalRNG(2.0, 0.5, 1.5);

EmpricRNG _empiric = new EmpricRNG(
            new EmpiricPair(new UniformDiscreteRNG(100, 120), 0.2),
            new EmpiricPair(new UniformDiscreteRNG(121, 175), 0.6),
            new EmpiricPair(new UniformDiscreteRNG(176, 200), 0.2)
  );
```

3. Pre samotné generovanie čísel je použitá metóda sample()

```
_exp.sample()
_erlang.sample()
_normal.sample()
_empiric.sample()
```

Zber štatistík

Pre zber štatistík poskytuje balíček OSPStat triedy Stat a WStat. Triedu Stat je možné použiť na zber štatistík, pričom predpokladá rovnakú váhu vzoriek (váha rovná 1). Jednotlivé údaje sú pridávané metódou addSample(double). Výsledné štatistiky poskytuje volaním príslušných metód (mean(), stdev(), confidenceInterval_95).

Trieda WStat by mala byť použitá pri výpočte priemeru, kde sú váhy vzoriek dané simulačným časom (napríklad štatistika dĺžky frontu). Použitie triedy WStat je identické s použitím triedy Stat, s tým rozdielom, že pri jej vytvorení je parametrom konštruktora inštancia simulácie, od ktorej trieda získava aktuálny čas.

Trieda SimQueue balíčka OSPDataStruct je implementáciou frontu, ktorý automaticky zbiera štatistiky o dĺžke frontu. Poskytuje metódy enqueue() a dequeue() pre pridanie a odobranie prvkov. Ako parameter konštruktora preberá inštanciu typu OSPStat.WStat, ktorá je po ukončení práce s frontom prístupná metódou lengthStatistic().