Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# Simulované prostredie pre učiacich sa agentov

2011 Bc. Ladislav Benc

Univerzita Komenského v Bratislave

Fakulta matematiky, fyziky a informatiky

# Simulované prostredie pre učiacich sa agentov

Diplomová práca

Študijný program : Aplikovaná informatika
Študijný odbor: 9. 2. 9. Aplikovaná informatika
Školiace pracovisko: Katedra aplikovanej informatiky
Školiteľ: Mgr. Michal Čertický
Evidenčné číslo: ec91d679-a0b6-4f88-96a2-9dc90075b0d7

Bratislava, 2011 Bc. Ladislav Benc


Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

# ZADANIE ZÁVEREČNEJ PRÁCE

**Meno a priezvisko študenta:** Bc. Ladislav Benc
**Študijný program:** aplikovaná informatika (Jednoodborové štúdium, magisterský II. st., denná forma)
**Študijný odbor:** 9.2.9. aplikovaná informatika
**Typ záverečnej práce:** diplomová
**Jazyk záverečnej práce:** slovenský

**Názov:** Simulované prostredie pre učiacich sa agentov

**Cieľ:** Návrh a realizácia simulovaného prostredia pre trénovanie učiacich sa agentov. Cieľom je implementácia lesného sveta, o ktorého vlastnostiach môže agent usudzovať, ako modulu do simulačného servera. Server bude s klientmi komunikovať po sieti. Jeho dizajn umožní nie len flexibilnú úpravu jednotlivých prvkov simulácie, ale aj jednoduchú definíciu celých nových prostredí. Očakávané využitie je vo výskume alebo výučbe agentovo-orientovaného programovania.

**Vedúci:** Mgr. Michal Čertický

**Dátum zadania:** 03.11.2009

**Dátum schválenia:** 05.05.2011

doc. RNDr. Roman Ďurikovič, PhD.
garant študijného programu

.....
študent

.....
vedúci

## Poďakovanie

Svojmu školiteľovi Michalovi Čertickému chcem poďakovať za rady, ústretovosť a rýchle odpovede. Rád by som poďakoval aj Mišovi Antoničovi za názory a diskusie ohľadom tejto práce a programovania všeobecne. V neposlednom rade ďakujem všetkým, ktorí ma počas jej písania podporili a všetkým ľuďom, ktorí na mňa nezanevreli, keď som ich kvôli práci zanedbával.

## Abstrakt

V práci realizujeme simulovaný svet, ktorý môže inteligentný agent skúmať, usudzovať o jeho vlastnostiach, kategorizovať objekty, na ktoré naráža a pozorovať výsledky svojich akcií. Simulácia je realizovaná ako modul do TerraSim – nášho sieťového servera, na ktorý sa pripájajú klienti aby inštruovali agentov. Podporované sú jedno- aj multiagentové scenáre. Dbáme na jednoduché použitie z užívateľského aj z programátorského hľadiska: modulárna architektúra umožňuje jednoduché definovanie nových senzorov, aktuátorov alebo celých svetov, preto jeho využitie nie je obmedzené iba na jeden účel. Uvádzame príklady použitia a prikladáme aplikáciu na manuálne ovládanie ľubovoľného druhu simulácie a vizualizáciu vnemov agenta.



**Kľúčové slová**: agentovo orientované programovanie, simulácia

## Abstract

In this thesis we propose a simulated world in which an intelligent agent can solve various tasks. Our goal was to design an environment the agent can explore, reason about, categorize various objects it encounters and observe results of its actions. The simulation itself is implemented as a module for TerraSim – our solution working as a network server, allowing remote control of agents. Single- as well as multi-agent scenarios are supported. The modular architecture of TerraSim enables us to easily define multiple different scenarios with new sets of sensors, actuators or even whole worlds, making its potential uses quite broad. We also provide usage examples, an application for controlling the agent in any kind of simulation and a visualization application.

**Keywords**: agent-oriented programming, simulation

# Obsah

# Zoznam obrázkov

# Zoznam úryvkov kódu

# 1 Úvod

Agentovo orientované riešenie problémov umelej inteligencie nie je novinkou. Bežný prístup k jeho realizácií však so sebou prináša niektoré obmedzenia. Agent býva dizajnovaný na riešenie pevne danej množiny problémov. Dokáže sa pohybovať iba v prostredí, ktorého charakteristiky sú dopredu známe a boli zohľadnené pri jeho implementácii.

Alternatívny prístup je tvorba agentov, ktorí sú schopní sami sa naučiť kauzálne vzťahy, resp. dôsledky ich akcií v danom svete. V poslednej dobe vznikli rôzne metódy pre takéto učenie sa. Môžeme z nich spomenúť napríklad heuristické učenie akcií založené na greedy prehľadávaní [Zettlemoyer, a iní, 2003], perceptrónový algoritmus z [Mourao, a iní, 2010], učenie sa pomocou inferencie nad logickými programami [Balduccini, 2007], konverzia učenia akcií na problém splniteľnosti [Yang, a iní, 2007] alebo rôzne prístupy k učeniu sa v zašumených a čiastočne pozorovateľných prostrediach [Čertický, 2009], [Čertický, 2011].

Primárnou úlohou tejto práce je vytvoriť simulovaný svet, fungujúci ako sieťový server, obsahujúci prostredie v ktorom sa môžu učiaci sa agenti pohybovať. Lesné prostredie, pomenované jednoducho ForestWorld, bolo navrhnuté tak, aby agent mohol využiť svoje kognitívne schopnosti na odhaľovanie súvislostí v ňom.

Sekundárnym cieľom je navrhnúť architektúru servera, ktorý sme nazvali TerraSim, modulárne. Takto získanú flexibilitu potom možno využiť na budúce rozširovanie funkcionality a hlavne na definíciu nových svetov. Vo výslednej aplikácii nie je problém navrhnúť a zadefinovať úplne nový svet s novou logikou a vlastnou množinou agentov, senzorov a aktuátorov. (Príklad uvádzame v kapitole 4 Príklad použitia – implementácia sveta)

Prikladáme všetky zdrojové kódy, dokumenty a materiály ktoré vznikli v priebehu tvorby práce. Tiež sú voľne dostupné na Internete vrátane prípadných ďalších verzií a úprav na adrese TerraSimu http://terrasim.codeplex.com/.

## 1.1 Prehľad pojmov

V tejto práci používame prevažne termíny, s ktorými sa človek zo základnou znalosťou programovania a agentovo orientovaného riešenia problémov pravdepodobne už stretol. Pre úplnosť základné z nich uvedieme skôr než ich začneme používať.

**Agent** je „niečo čo vníma a koná“ [Russell, a iní, 1995]. Racionálny agent má nejaké presvedčenia a na ich základe sa snaží dosiahnuť ciele. Vníma pomocou **senzorov** a koná pomocou zariadení zvaných **aktuátory**. V kontexte TerraSim budeme **lokálnym agentom** nazývať agenta, ktorého logika je implementovaná na strane servera. V kontraste s ním **vzdialený (užívateľský) agent** je ovládaný cez sieťové pripojenie a vykonáva príkazy, ktoré server dostane od klienta (používateľa).

Pri popise prostredí používame terminológiu podľa knihy Artificial Intelligence: A Modern Approach [Russell, a iní, 1995]. Prostredie je:

- Plne pozorovateľné, ak ho senzory agenta vždy zachytávajú celé; inak je čiastočne pozorovateľné.
- Epizodické, ak sú agentove interakcie rozdelené do diskrétnych epizód. V každej z nich agent vníma a vykoná akciu. Nikdy sa nepotrebuje rozhodovať na základe predošlých epizód ani plánovať dopredu. V opačnom prípade je prostredie neepizodické (sekvenčné).
- Deterministické, ak nasledujúci stav určený iba predošlým stavom a akciou agenta; inak je stochastické.
- Dynamické, ak sa môže zmeniť kým agent premýšľa. Ak nemôže, je statické. Ak sa s časom nemôže zmeniť prostredie, ale hodnotenie výkonu agenta áno, hovoríme že prostredie je semidynamické.
- Diskrétne, ak existuje obmedzený počet vnemov a akcií; inak je spojité.

Používame bežnú terminológiu objektovo orientovaného programovania (OOP) a predpokladáme, že čitateľ je s ňou oboznámený. V prípade použitia iných termínov sú tieto v súlade s bežným významom v prostredí jazyka C#. Za špecifickú zmienku stojí **zapečatená** trieda (**sealed** class) – trieda, z ktorej nemožno dediť. Jej použitie má rôzne výhody – návrh je jednoduchší, pretože nemusíme zvažovať následky jej rozširovania

dedením. Zároveň prípadné odstránenie zapečatenia nikdy nespôsobí chybu v existujúcom kóde, takže v prípade potreby je možné hierarchiu bez dodatočných komplikácií rozšíriť.

**Single (dynamic) dispatch** je proces dynamického viazania (**binding**) správ ku konkrétnej metóde objektu. Používa sa na volanie takých metód objektov, kde správna verzia metódy nemôže byť určená počas kompilácie. Je nosnou konštrukciou polymorfizmu v OOP, kde sa používa na viazanie metódy podľa aktuálneho typu objektu na ktorom je zavolaná.

**Multiple dispatch** je metóda preťažovania (overloading) funkcii, ktorá na rozdiel od single dispatch nerozhoduje o verzii metódy ktorá bude zavolaná iba na základne typu prvého, ale všetkých objektov. Takejto metóde hovoríme **multimetóda.**

**Marshalling** je proces transformácie pamäťovej reprezentácie objektu do iného formátu, vhodného na prenos alebo uchovávanie.

**Trasa** (**trace**) je záznam informácií o priebehu vykonávania programu – ide o druh logovania. Správy zaznamenané pri trasovaní sú rozdelené do viacerých úrovní podľa ich závažnosti. Používa sa nie len na debugovanie, ale aj na monitorovanie behu aplikácie.

## 1.2 Existujúce riešenia

*Human-level AI's Killer Application* [Laird, a iní, 2000] uvádza, že výbornou pôdou pre výskum v umelej inteligencii porovnateľnej s človekom sú simulácie. Zameriava sa najmä na videohry, odôvodňujúc to tvrdením, že ich vývoj rýchlo napreduje a k dispozícií je veľké množstvo hotových riešení, ktoré sa dajú jednoducho využiť pri návrhu a testovaní algoritmov umelej inteligencie. Vývojári viacerých herných aplikácií dávajú k dispozícii nástroje na tvorbu prostredí a svetov. To umožňuje výskumníkom využiť existujúcu sadu komplexných pravidiel, simuláciu fyziky a iné mechanizmy a sústrediť sa výhradne na prácu na UI. Výhodou tohto prístupu je tiež veľká popularita videohier – pri testovaní úspešnosti a efektivity agentov nie je na rozdiel od iných druhov úloh v iných prostrediach problém najímať doménových expertov. Skúsených hráčov je medzi bežnými ľuďmi dostatok a práve oni často najlepšie dokážu ohodnotiť správanie sa svojho súpera, prípadne ohodnotiť, či a do akej miery sa jeho správanie ponáša na správanie človeka [Laird, a iní, 2000]. Dôraz je kladený aj na adaptabilitu – agent musí byť schopný reagovať flexibilne v čo najväčšom množstve situácií.

Takýto prístup sa stal v posledných rokoch pomerne populárnym, preto sú medzi existujúcimi riešeniami uvedené buď hry, ktoré boli aj komerčne distribuované, alebo simulácie navrhnuté za účelom výskumu.

Pogamut je založený na videohre Unreal Tournament 2004. Jeho engine umožňuje ovládať herné charaktery externými kontrolnými mechanizmami, pripojenými po sieti, podobne ako je to v TerraSime. Používa jazyk UnrealScript a Java platformu NetBeans a poskytuje tak kompletné prostredie na vývoj inteligentných agentov [Gemrot, a iní, 2009]. Agent sa vo svete UT2004 pohybuje v reálnom čase v trojrozmernom prostredí. Sleduje ho z pohľadu vlastných očí (ide o tzv. FPS hru) a k dispozícii je vizualizácia vo vysokej grafickej kvalite. Môže plniť rôzne ciele v štýle „eliminuj protivníkov“, „získaj cieľový objekt“ (známy ako „capture the flag“ mód) a iné. Prostredie je dynamické, neepizodické, čiastočne pozorovateľné, stochastické a spojité.

Cieľom Pogamutu je zjednodušiť vytváranie agentov a akcií. Aj tie zložité, ako pathfinding alebo zbieranie informácií z vnemov sú pripravené na použitie, umožňujúc užívateľovi sústrediť sa na zaujímavejšie časti [Gemrot, 2011]. K dispozícii je pripojenie pre externý reaktívny plánovač pre agenta. Obsahuje napríklad interface na jednoduché prepojenie s POSH plánovačom [Bryson, 2008].

Pogamut využívajú viaceré projekty a súťaže s rôznymi zameraniami. Za všetky menujeme BotPrize, súťaž vo vytváraní umelej inteligencie (UI) do hier a GeneticBots, projekt zameraný na evolúciu správania agentov použitím genetických algoritmov.

Simuláciou s podobným zameraním je Jazzbot, založený na open source FPS hre Nexuiz. Je postavený vo frameworku jazyka Jazzyk, dizajnovanom na využitie heterogénnych reprezentácií informácií v rámci jediného agenta riadeného množinou (potenciálne vnorených) pravidiel v tvare *„ak podmienka tak akcia“* [Novák, 2009]. Agenti riešia ciele ako hľadanie predmetov alebo mapovanie. Takisto ako Pogamut, prostredie je dynamické, sekvenčné, čiastočne pozorovateľné, stochastické a spojité.

ORTS je prostredie pre štúdium real-time UI problémov ako je pathfinding, práca s neúplnými informáciami a plánovanie v doméne strategických hier v reálnom čase (RTS). Motiváciou výskumu je popularita týchto hier a „zlý stav umelej inteligencie v nich“ [Buro, 2010]. Charakteristiky prostredia sú opäť rovnaké ako v predošlých dvoch prípadoch.

Práve inšpirácia RTS hrami spôsobila, že ORTS sa nášmu riešeniu prístupom ku svetu podobá viac než predošlé dve prostredia.

TerraSim je podobný simuláciám popísaným vyššie vo viacerých aspektoch. Simulácia beží na serveri, na ktorý za pripájajú klienti a riadia správanie svojich zástupcov vo virtuálnom svete. Viaceré charakteristiky prostredia sú tiež totožné (pozri nižšie). Zatiaľ čo ostatné projekty pokrývajú veľkú časť výskumných záujmov, nie sú dostačujúce pre všetky účely. Kladieme väčší dôraz na flexibilitu agenta a jeho nezávislosť vzhľadom na prostredie. Na jeho testovanie preto potrebujeme svet, ktorý bude obsahovať informácie o objektoch a súvislostiach medzi nimi, ktoré nebudú zjavné ihneď. Agent potom pozorovaním javov vo svojom okolí a sledovaním následkov svojich akcií získa potrebné informácie; použitím svojho kognitívneho centra môže vyvodzovať dostatočne všeobecné hypotézy o fungovaní sveta.

## 1.3 Prehľad navrhovaného riešenia

Agent sa pohybuje v prostredí prírody. Jeho úlohou je skúmať flóru zvieratami obývaných lesov a lúk. Zadané mu môžu byť ciele ako „zisti, aké druhy rastlín sa v okolí vyskytujú," „zisti, či je tu vhodné prostredie pre rast húb" alebo „vytvor mapu rastlinstva."

So všetkými entitami v prostredí môže agent interagovať. Môže zbierať a polievať rastliny, vykonávať ich analýzu, plašiť zver alebo ju kŕmiť. Práve vo vlastnostiach týchto objektov sú skryté informácie o prostredí, ktoré môže skúmajúci agent odhaliť. Ak vidí dubák, muchotrávku, snežienku a papraď, môže zistiť, že prvé dve nemajú chlorofyl a teda patria do odlišnej kategórie. Môže ich skúsiť poliať vodou a pozorovať, či to ovplyvní ich rast, alebo nimi môže nakŕmiť zver a pozorovať úhyn po konzumácii muchotrávky. Ako bolo spomenuté vyššie, tieto vzťahy medzi objektmi prostredia boli modelované kvôli trénovaniu a testovaniu agentov učiacich sa kauzálne vzťahy a efekty ich akcií.

Simulovaný agent je napájaný elektricky, solárnymi kolektormi a akumulátorom, čo obmedzuje jeho pohyblivosť a vnáša do rozhodovania a plánovania element správy zdrojov. V prípade slnečného počasia je agent schopný vykonávať všetky akcie prakticky bez obmedzenia. V prípade zlého počasia alebo počas noci môže pracovať na napájanie akumulátora, ktorého kapacita je však obmedzená a po krátkej dobe používania je nutné ho solárnym panelom opäť nabiť.

Simulačný server podporuje aj multiagentové scenáre. Každý agent môže ľubovoľnému inému poslať ľubovoľné textové dáta. Tieto server nijakým spôsobom nekontroluje ani do nich inak nezasahuje. Po prijatí ich ihneď pošle recipientovi.

Prostredie sa správa dynamicky, sekvenčne a stochasticky. Aj ak užívateľský agent nevykoná žiadnu akciu, čas v simulácii naďalej beží a prostredie sa mení (počasie, akcie iných agentov). V prostredí sú prítomní aj agenti – zvieratá, ktorých správanie je do značnej miery náhodné a rozhodujú sa nezávisle od vzdialeného agenta. Ak je prítomných naraz viacero užívateľský agentov, v každom časovom kroku sa vykonajú akcie prijaté od tých, ktorí sa nejaké rozhodli vykonať. Ostatní agenti sú ignorovaní a ostávajú neaktívni. Prostredie je vo svojej podstate diskrétne. Pre všetky účely, vrátane vnemov, nám stačia malé rozsahy celočíselných hodnôt alebo reťazcov.

## 1.4 Štruktúra dokumentu

V tejto práci popisujeme proces vzniku simulovaného sveta TerraSim, od počiatočného konceptu až po výstupné pomocné aplikácie. V kapitole *Návrh* sa budeme venovať detailom návrhu a zdôvodňovať rozhodnutia a kompromisy učinené počas jeho tvorby. Popíšeme architektúru servera, koncepty ktoré za ním stoja. Predstavíme mechanizmy riadiace svet ForestWorld, implementovaný ako simulačný modul pre server TerraSim. V kapitole *Implementácia* sa budeme zaoberať výberom nástrojov použitých na realizáciu a relevantnými implementačnými detailmi. Kapitola *Príklad použitia – implementácia sveta* ukazuje spôsob, akým sa dá od základov vytvoriť a spustiť nový svet pre TerraSim. Na záver načrtneme možnosti prípadného rozširovania projektu pre ďalšie účely.

# 2 Návrh

V tejto kapitole dizajnujeme všetky aspekty TerraSimu. Budeme sa zaoberať architektúrou serveru – sieťovej, komunikačnej i simulačnej časti. Navrhneme aj logiku riadiacu simulovaný svet a udalosti v ňom, ako počasie, striedanie dňa a noci alebo mechanizmus šírenia správ medzi agentmi. Popíšeme všetky druhy entít, ktoré do sveta môžu vstúpiť. Načrtneme spôsob, akým môže agent pozorovať prostredie v ktorom sa nachádza a ovplyvňovať ho, potom predstavíme lesný svet ForestWorld a jeho stavebné prvky; terén, rastliny, zvieratá. Ukážeme, ako funguje načrtnutý spôsob interakcie s prostredím v tomto svete a zadefinujeme konkrétne senzory a aktuátory, ktoré vzdialení užívatelia budú mať k dispozícii. Úlohou tejto kapitoly nie je rozoberať implementačné detaily alebo programovacie techniky. Informácie tohto charakteru sú uvedené v kapitole 3.

Dôležitým prvkom dizajnu TerraSimu je flexibilita. Jednotlivé súčasti, na ktoré sa dá nazerať ako na samostatné funkčné celky, boli pri dizajne oddelené do samostatných modulov. Ich vzájomná závislosť a množstvo komunikácie medzi nimi bolo minimalizované. Takýto prístup umožňuje sústrediť sa pri implementácii na jeden kompaktný celok, uľahčuje údržbu kódu a testovanie. Zjednodušené je tak aj rozširovanie (pridanie nového modulu) a potenciálna budúca úprava, keď v prípade zmeny požiadaviek stačí vymeniť jeden separátny modul. Pre jednoduchšiu orientáciu v báze kódu sme pre každú významnú vrstvu aplikácie vytvorili vlastný namespace, obsahujúci všetky pridružené triedy.

Všetky veličiny sveta sú kvôli jednoduchosti spracovania reprezentované diskrétnymi hodnotami. Premenné nadobúdajú ako hodnoty buď prirodzené čísla alebo textové reťazce. V druhom prípade ide často o úzko vymedzenú množinu hodnôt predstavujúcu vymenovaný typ, ako napríklad logické hodnoty („true“, „false“) alebo intenzitu (od „low“ cez „average“ po „full“).

Každá entita v simulovanom svete má typ a meno. Meno je reťazec, jej jednoznačný identifikátor. Určuje sa vždy pri konštrukcii objektu a jeho pridelenie nie je zodpovednosťou servera. Pretože je dovolené definovať a simulovať rôzne svety, prideľovanie identifikátorov je zodpovednosťou zabezpečovateľa obsahu. (Viac o tejto téme je v sekcii 2.9.)

Typ entity je tiež reťazec. Nie je nutné aby bol určený pre každú entitu a slúži prevažne ako doplňujúca informácia pre agenta. Hoci forma nie je TerraSimom vynucovaná, meno aj typ sa držia istých konvencií. Typ je vždy názov druhu entity, napísaný malými písmenami bez medzier. Meno je písané rovnakým spôsobom a skladá sa z typu a čísla, ktoré je inkrementované po pomenovaní každej entity. Tak je z mena vždy zrejmý druh objektu a jednoznačný číselný identifikátor (každé číslo je pri pomenovaní použité najviac raz). Napríklad pomenovanie agenta a troch húb by mohlo vyzerať nasledovne:

| **Typ** | **Meno** |
|---|---|
| sampinon | sampinon_1 |
| agent | agent_2 |
| muchotravka | muchotravka_3 |
| sampinon | sampinon_4 |

V nasledujúcich sekciách sa budeme venovať najprv návrhu modelu agenta, potom svetu ForestWorld a nakoniec dizajnu servera.

## 2.1 Agenti, senzory a aktuátory

Ako sám názov napovedá, centrálnou ideou agentovo orientovaného riešenia problémov sú *agenti*. Ich hlavnými črtami je schopnosť vnímať prostredie v ktorom sa nachádzajú, interagovať s ním, meniť ho. Na to využívajú dve skupiny zariadení. Prvou sú *senzory*. Sú agentovými zmyslami, zabezpečujú generovanie vnemov zo sveta. Druhou sú *aktuátory* (niekedy v literatúre označované *efektory*), zaisťujúce agentovu schopnosť ovplyvňovať objekty okolo seba. Agent potrebuje možnosť sledovať dôsledky svojich akcií. Každá z nich má určité trvanie, ktoré agent vníma a po ktorého ukončení sa prejavia jej následky.

Okrem kolekcie senzorov a aktuátorov potrebuje agent nejaký druh kognitívneho centra, inteligentnej jednotky, ktorá bude riadiť jeho správanie. V TerraSime rozlišujeme dva druhy agentov. Prvým sú *lokálni* (definovaní triedou `Agent` v mennom priestore `TerraSim.Simulation`, rovnako ako všetky ostatné triedy v tejto sekcii). Ich správanie je určené kognitívnou jednotkou, ktorá je súčasťou simulácie. Vo svete ForestWorld sú takýmito agentmi zvieratá.

Druhým typom sú *vzdialení* alebo *užívateľskí* agenti (trieda `UserAgent`). Po tom, ako sieťový server simulácie nadviaže spojenie s klientom, vytvorí a pridelí mu takéhoto agenta. Na strane servera sú zadefinované jeho senzory a aktuátory, nie však jeho inteligencia. TerraSim bude po sieti posielať klientovi vnemy a bude od neho očakávať inštrukcie pre jeho aktuátory. Užívateľskí agenti sú teda vstupným bodom pre použitie simulácie – všetka interakcia so svetom prebieha prostredníctvom nich.

Model agenta musí byť dostatočne flexibilný, aby splnil požiadavky na modularitu. Pridanie či odobranie senzora (trieda `Sensor`) alebo aktuátora (trieda `Actuator`) do dizajnu sveta nesmie vyžadovať žiadnu modifikáciu ani na strane vzdialeného klienta, ani na strane servera. To znamená, že po pridaní daného komponentu ku agentovi ten musí byť hneď schopný ho používať bez zásahu do ktorejkoľvek časti vnútornej logiky.

Docieliť tento efekt pri senzoroch je jednoduché – stačí, aby začali agentovi posielať vnemy. Abstraktná trieda `Sensor` preto obsahuje hlavne metódu `Sense` na snímanie prostredia, ktorej implementáciu musí zabezpečiť každý senzor. Pri aktuátoroch sa agent potrebuje aktívne dozvedieť, aké možnosti ich využitia má k dispozícii. Preto abstraktná trieda `Actuator` po dediacich triedach vyžaduje nie len implementáciu metód na zadanie príkazu (`Perform`) a update (`Update`), ale aj metódu ktorá vráti názvy všetkých akcií, ktoré je aktuátor schopný vykonať.

## 2.2 Svet lesa - ForestWorld

Ako sme popísali v úvode, prostredie lesa bolo navrhnuté tak, aby umožňovalo usudzovanie o svojich zákonitostiach. V tejto sekcii ho stručne popíšeme – začneme celkovým dizajnom, prejdeme ku neživým entitám, lokálnym agentom a nakoniec popíšeme senzory a aktuátory, ktoré má k dispozícii vzdialený agent.

### 2.2.1 Prostredie

Vo svete sa nachádza 6 druhov terénu - tráva, krovie, štrk, skaly, voda a piesok. Ovplyvňujú agenta a entity, ktoré sa na nich nachádzajú. Napríklad na agenta, stojaceho na lúke (tráve) dopadá viac svetla ako na agenta stojaceho v kroví. Tiež pri pohybe po štrku a piesku sa víri prach a zacláňa agentov slnečný kolektor, znižujúc jeho efektivitu. Skaly sú úplne nepriechodné. Faktory ovplyvňujúce množstvo svetla sa kumulujú, preto agent stojaci v zlom počasí večer v lese bude mať oveľa horšie osvetlenie a tým pádom aj menej energie, než agent stojaci za slnečného dňa na lúke alebo pri vode.

Terén hrá úlohu aj pri niektorých akciách. Agent nemôže čerpať vodu do svojej nádrže ak nie je v blízkosti vody (pozri popis aktuátorov nižšie) a huby sa nemôžu množiť ak nie je v ich okolí pôda dostatočne vlhká.

Pretože agent má mať možnosť skúmať dôsledky svojich akcií vo svete a kategorizovať objekty, ktoré vidí, uvedieme entity vyskytujúce sa vo svete a vzťahy medzi nimi.

Flóra sveta sa rozdeľuje do dvoch základných skupín – huby a rastliny. Všetky triedy, ktoré ju reprezentujú, sa nachádzajú v `TerraSim.ForestWorld.Entities` a dedia triedu `Plant`. Sú objektom agentovho skúmania, môžu byť agentom pozorované (pozri sekciu Senzory nižšie), analyzované (sekcia Aktuátory) a manipulované.

Základný rozdiel medzi rastlinami a hubami je, že huby neobsahujú chlorofyl. Zástupcovia oboch skupín môžu, ale nemusia byť jedovatí alebo hydrofilní (preferujúci vlhké prostredie). K týmto informáciám agent nemá prístup kým nevykoná analýzu. Ich spoločnou charakteristikou je, že pri vytvorení majú určenú najmenšiu a najväčšiu veľkosť, akú môžu dosiahnuť. V priaznivých podmienkach rastú rýchlejšie a po dosiahnutí dospelosti, teda určitej minimálnej veľkosti (líšiacej sa v závislosti od druhu) sa môžu rozmnožiť. Keď rastlina dosiahne požadovanú veľkosť, má *šancu* rozmnožiť sa – to znamená vytvoriť novú rastlinu rovnakého druhu a minimálnej veľkosti na niektorom zo susedných polí. Element náhody zavádzame preto, aby sme predišli ich príliš rýchlemu množeniu. Ak by sa rastlina mala rozšíriť v každom kroku simulácie, onedlho by aj jej potomkovia boli dospelí a produkovali ďalších potomkov. Nastal by exponenciálny rast rastlín a zbytočné zahltenie prostredia. Každý kus flóry, ktorý agent nájde, môže zodvihnúť. Spôsobí tak jej smrť a zabráni akémukoľvek ďalšiemu rozmnožovaniu.

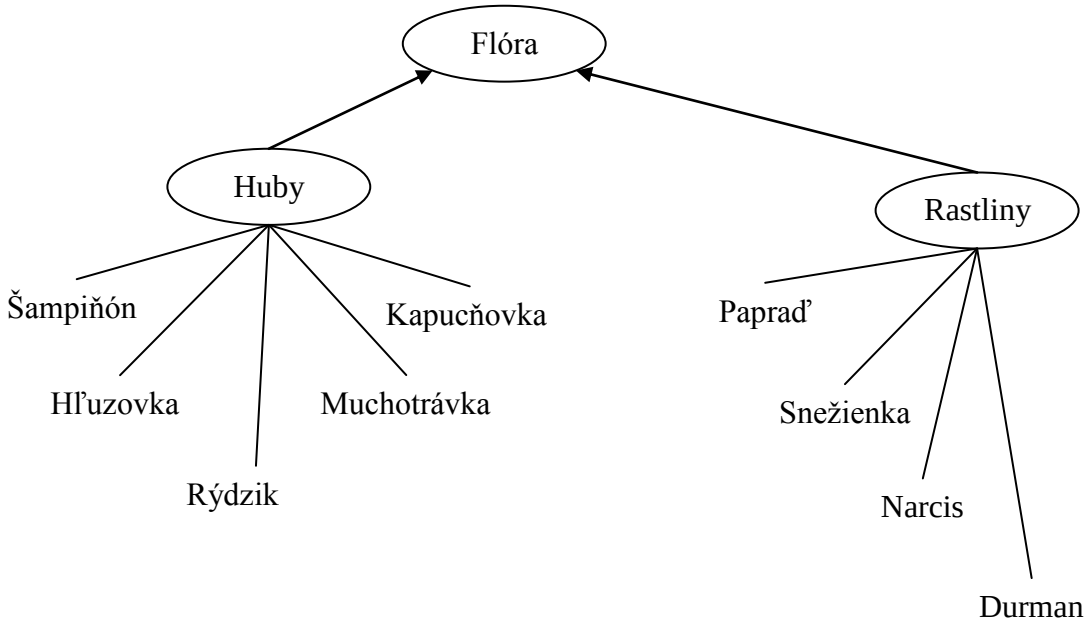
Všetky huby (implementované triedou `Fungus`) sú hydrofilné. To v kontexte simulácie znamená, že sa im najlepšie darí ak sa vyskytujú vo vlhkom prostredí. Rast môže vyvolať aj agent ich polievaním. Ak prší, dôjde k nemu iba s určitou pravdepodobnosťou z dôvodov podobných tým zmieneným vyššie. Rozlišujeme tieto druhy (so skrátenými názvami použitými v simulácii):

- Šampiňón (White mushroom) – najmenšia z húb.
- Hľuzovka (Truffle), Rýdzik (RedPine) – väčšie huby.

- Muchotrávka (Amanita) – menšia jedovatá huba.
- Kapucňovka (Galerina) – väčšia jedovatá huba.

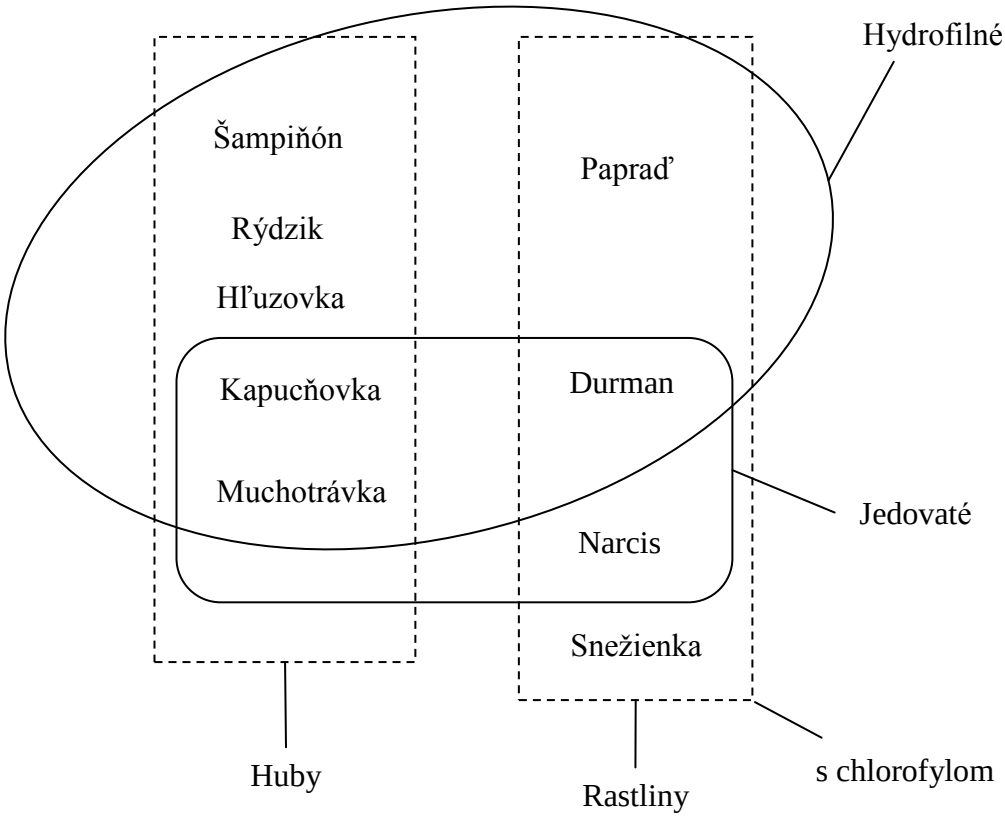
Rastliny (implementované triedou `GreenPlant`) obsahujú chlorofyl. Ak sa nachádzajú na dobre osvetlenom mieste, môže to urýchliť ich rast. Čím viac svetla na ne dopadá, tým väčšia je šanca rastu. Môže ho vyvolať aj dážď alebo polievanie. U hydrofilných rastlín je efekt zaliatia markantnejší. Rastliny môžu dosahovať podstatne väčšiu veľkosť než huby a dospelosť dosahujú neskôr. Pri rozmnožovaní môže nový jedinec vzniknúť aj o niečo ďalej než len na susediacom políčku. Jednotlivé druhy sú:

- Papraď (Fern) – veľká rastlina, je hydrofilná.
- Snežienka (Snowdrop) – najmenšia rastlina.
- Narcis (Narcissus) – jedovatá.
- Durman (Datura) – najväčšia rastlina, hydrofilná a jedovatá.




**Obrázok 1 Spôsob reprezentácie flóry**

Obrázok 1 znázorňuje spôsob, akým je flóra v simulácii reprezentovaná. Huby a rastliny sú zastúpené triedami, jednotlivé druhy ale sú iba ich inštanciami, pretože sa medzi sebou líšia len parametrami. Inými slovami, ak rozdiel medzi dvoma druhmi rastliny spočíva v dátach, nie v logike, je zbytočné rozširovať hierarchiu tried.




**Obrázok 2 Rozdelenie rastlín do kategórií**

Rastliny je možné rozdeliť do rôznych prekrývajúcich sa kategórii. Môže ísť o úlohu zadanú agentovi.


**Obrázok 3 Rastliny vzostupne zoradené (smerom zhora dolu) podľa maximálnej veľkosti, ktorú môžu dosiahnuť**

K objektom z lesného sveta patrí aj kŕmidlo pre zver (trieda `AnimalFeeder`). Agent môže do neho umiestňovať (alebo z neho vyberať) predmety. Kŕmia sa z neho zvieratá, lokálni agenti simulácie (trieda `Animal`, obe v mennom priestore `TerraSim.ForestWorld.Entities`). Zvieratá sa v bežnom prípade užívateľskému agentovi ani nevyhýbajú, ani ho nevyhľadávajú. Pohybujú sa po okolí a raz za určitú dobu vyhladnú. Vtedy ak majú v dohľade nejaké kŕmidlo, vydajú sa k najbližšiemu a najedia sa z neho. Sýte sa potom vrátia k pôvodnej činnosti. Ak sa vyskytnú viaceré zvieratá vo vzájomnej blízkosti, môžu sa rozmnožiť. Inteligencia zvieraťa je implementovaná ako aktuátor „rozhodovacie centrum zvieraťa“ (trieda `AnimalDecisionCenter` v namespaci `TerraSim.ForestWorld.Actuators`).

Agent môže do ich kolobehu života zasahovať. Môže položiť do kŕmidla rastlinu alebo hubu a sledovať reakciu zvieraťa. Ak niektorý z uložených predmetov bol jedovatý, zviera po jedle uhynie. Môže použiť svoju zbraň a zviera zastreliť. Hluk výstrelu zaženie všetky okolité živočíchy na útek.

Popísali sme prostredie, jeho princípy a zákonitosti. Ďalšou dôležitou kapitolou je užívateľský agent a jeho schopnosti interakcie s prostredím. V nasledujúcich odsekoch popíšeme najprv jeho senzory a potom aktuátory – ako boli navrhnuté a aké akcie nimi môže vykonávať.

### 2.2.2 Senzory

Senzory sú agentovými zmyslami vo svete, v ktorom sa nachádza. Ich implementáciu ako súhrnnej funkcie, ktorá zo stavu sveta na vstupe vygeneruje všetky pozorované informácie sme zavrhli v prospech samostatných senzorických modulov. Pre každý z nich vytvárame vlastnú triedu. Takto môžeme jednoducho prispôsobiť scenár podľa aktuálnych požiadaviek.

So senzormi agent nijako aktívne nekomunikuje, iba od nich prijíma dáta. S výnimkou akumulátora nevstupujú ani do žiadnej inej interakcie – iba zo stavu sveta na vstupe extrahujú relevantné dáta. Úlohou akumulátora nie je len informovať o stave batérie, ale ju aj pri dostatočnom osvetlení nabíjať. Tiež umožňuje ostatným aktuátorom čerpať z neho energiu na vykonanie akcií. Všetky triedy reprezentujúce senzory sa nachádzajú v mennom priestore `TerraSim.ForestWorld.Sensors`.

| | |
|---|---|
| Akumulátor | Trieda `Accumulator`<br>Akumulátor udržuje agentovu zásobu energie a informuje o nej. Ak sa vyčerpá, nebude možné vykonať žiadnu akciu.<br>V každom kroku simulácie sa v obmedzenej miere dobije solárnym panelom – čím viac svetla na agenta dopadá, tým väčší je prírastok energie. |
| Barometer | Trieda `Barometer`<br>Ako názov naznačuje, barometer poskytuje agentovi informáciu o aktuálnom atmosférickom tlaku, od ktorého sa odvíja počasie. (Pozri sekciu 2.4.) |
| Senzor jasu | Trieda `BrightnessSensor`<br>Informuje agenta o množstve svetla, ktoré na neho dopadá. Od neho závisia viaceré javy, najmä ale obnova energie v agentovom akumulátore a rýchlosť rastu zelených rastlín. |
| Kamera | Trieda `Camera`<br>Poskytuje agentovi informácie o jeho okolí. Zaznamená okolitý terén, objekty, ich vlastnosti a stav počasia. |
| Hodiny | Trieda `Clock`<br>Tento jednoduchý senzor informuje agenta o tom, aký je momentálne čas v simulácii. (Viac o čase je v sekcii 2.3.) |
| Senzor vlhkosti | Trieda `HumiditySensor`<br>Informuje agenta o vlhkosti pôdy na jeho aktuálnej pozícii. Vlhkosť súvisí s rýchlosťou rastu rastlín. |

### 2.2.3 Aktuátory

Aktuátory sú jedinými nástrojmi, ktorými agent môže ovplyvňovať prostredie okolo seba. Každý z nich poskytuje niekoľko akcií, ktoré je schopný vykonať. Po tom, ako agent dostane príkaz, ten je zaslaný konkrétnemu aktuátoru, ktorý ho môže vykonať. Aktuátor

skontroluje, či dostal potrebné parametre v korektnom tvare, overí či sú splnené podmienky pre vykonanie akcie a či na ňu agent má dosť energie. Ak všetko prebehne bez problému, zaháji jej vykonávanie. Vo svete ForestWorld platí, že aktuátory fungujú nezávisle od seba, paralelne a každý môže vykonávať súčasne len jednu akciu. Väčšina z nich uchováva štatistiky o ich použití, ktoré na požiadanie agentovi zašle.

Dôvodov, prečo akcia môže zlyhať môže byť viacero a sú intuitívne zrejmé. Napríklad ak sa agent pokúsi vyliať vodu, keď už žiadnu nemá, alebo ju nabrať z miesta, kde žiadna nie je, nič sa nevykoná. Podobný prípad nastáva, ak sa pokúsi použiť analyzátor rastlín na objekt, ktorý nie je rastlinou. Každý aktuátor môže zlyhať, ak má agent vybitý akumulátor.

Všetky triedy reprezentujúce aktuátory sa nachádzajú v mennom priestore `TerraSim.ForestWorld.Actuators`. Rovnako ako pri senzoroch, rozdelenie schopností agenta do logických celkov umožní odobrať alebo pridať časti funkcionality podľa potreby. Konkrétny formát príkazov (ich mená a parametre) je popísaný v prílohe.

| | |
|---|---|
| Ruka | Trieda `Arm`<br>Umožňuje agentovi uchopiť jeden predmet. Potom ho môže položiť buď do iného objektu (do kŕmidla), alebo na zem (na určenú pozíciu v blízkosti agenta). |
| Vodná nádrž | Trieda `WaterTank`<br>Drží zásobu vody. Agent ju môže využiť na polievanie rastlín. Po piatich polievaniach sa voda minie a musí byť dočerpaná z nejakej vodnej plochy. |
| Analyzátor rastlín | Trieda `PlantAnalyzer`<br>Vykoná analýzu rastliny, na ktorú je použitý. Tým agent získa prístup ku dovtedy skrytým údajom – obsah chlorofilu, jedovatosť a hydrofília. |

Kolesá

Trieda `Wheels`

Zabezpečuje pohyb agenta. Obsahuje metódy na otáčanie vľavo a vpravo, a na pohyb vpred a vzad. Agent sa vždy nachádza v jednom zo šiestich stupňov natočenia – každé korešponduje s jedným susedným šesťuholníkom.

Zbraň

Trieda `Gun`

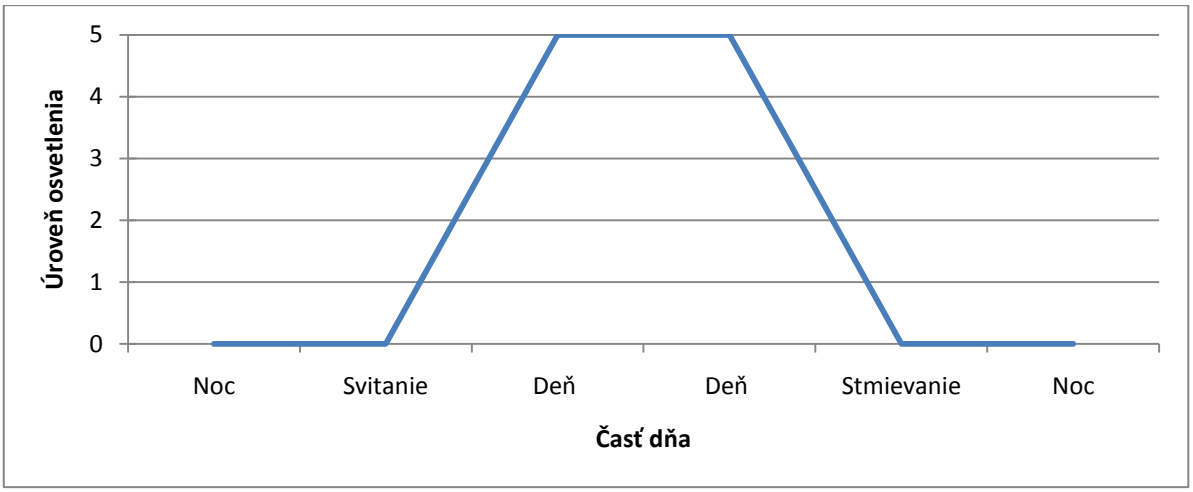
Agent ju používa na streľbu. Výstrelom môže zabiť zacieleného agenta alebo zviera. Zásobník zbrane má obmedzenú kapacitu (pri štandardných nastaveniach 3 náboje). Po jeho vyprázdnení ho musí agent opäť naplniť. Môže to vykonať ľubovoľne veľa krát (teda celkový počet nábojov nie je obmedzený).

Každý výstrel spôsobí hlasný zvuk, po počutí ktorého sa zver pokúsi ujsť.

## 2.3 Plynutie času

Tok času hrá dôležitú úlohu. Riadi sa ním mnoho elementov simulácie, ako napríklad dĺžka vykonávania akcií, zmeny dňa a noci alebo zmeny počasia. V prostredí simulovaného sveta je čas oddelený od reálneho času. Každý virtuálny deň je rozdelený na definovaný počet častí[1] rovnakej dĺžky. Tieto sú najmenšou jednotkou času v simulácií. V každom časovom kroku prebehne výpočet jej ďalšieho stavu. Zahŕňa vykonanie akcií užívateľských agentov, agentov prostredia, aktualizáciu počasia a stavu sveta. (Viac o týchto udalostiach je v sekciách 2.1, 2.4, a 2.8). Dĺžku časovej jednotky je tiež možné nastaviť v konfigurácií serveru.

[1] Tento počet je možné zmeniť v konfigurácií servera. Prednastavená hodnota je 100.




**Obrázok 4Úroveň osvetlenia v priebehu dňa**

V priebehu dňa sa úroveň osvetlenia sveta mení. Začína najväčšou tmou (nocou), po čase sa postupne zmení na najjasnejšiu a potom sa opäť zotmie. Čas svitania, stmievania (určené v jednotkách od začiatku dňa) a dĺžku ich trvania možno nastaviť, prípadne striedanie dňa a noci úplne vypnúť.

Na intenzite svetla priamo závisí aj správanie užívateľského agenta. Každá jeho akcia spotrebováva napätie z akumulátora, ktorý sa dobíja slnečným kolektorom - tým rýchlejšie, čím viac svetla naň dopadá. Preto ak by sa agent ocitol v situácii kde by mal vybitý akumulátor a žiaden prístup ku svetlu, ostal by permanentne vyradený z prevádzky. Vzniku tejto situácie zabránime tak, že v každom mieste vždy bude prítomná aspoň minimálna úroveň osvetlenia (teda nepripustíme úplnú tmu).

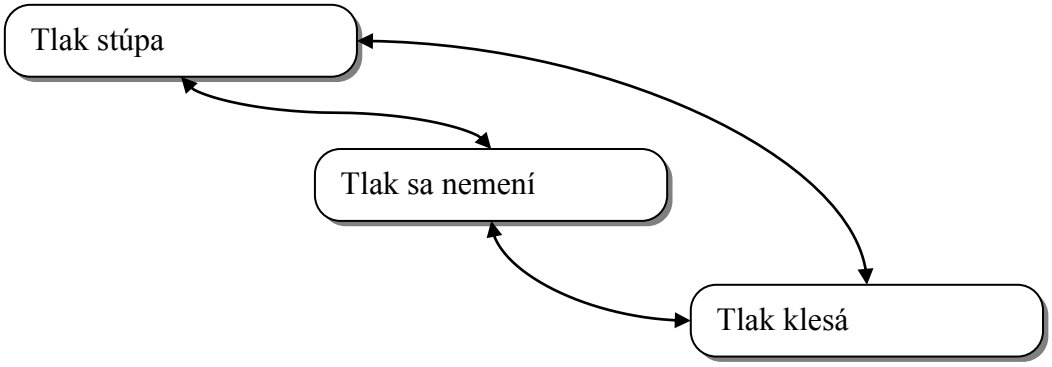
## 2.4 Počasie

Počasie sa môže nachádzať v jednom z troch stavov – slnečno, zamračené alebo dážď. Pri daždi sú zhoršené svetelné podmienky a pôda naberá vlhkosť[2]. Keď je zamračené, úroveň svetla je o niečo vyššie. Pri slnečnom počasí je svetlo na maximálnej úrovni a pôda vysychá.

Model počasia je založený na atmosférickom tlaku [Addison, 2003]. Svet vo svojom stave udržuje jeho hodnotu a na jej základe určuje momentálny stav počasia. V prípade že tlak klesne pod určenú hranicu, prší. Ak prekročí hornú hranicu, je slnečno. Ak sa nachádza medzi týmito dvoma hranicami, je zamračené. Tlak sa mení podľa

[2] Vlhkosť pôdy má vplyv primárne na rast rastlín.

Markovovho modelu na obr. 5. Prirodzene toto riešenie je len aproximáciou, počasie v prírode sa správa oveľa komplexnejšie a menej predvídateľne.




**Obrázok 5 Markovov model zmeny atmosférického tlaku**

V každom kroku simulácie sa vypočíta nový stav Markovovho modelu. Môže ním byť stúpanie, zachovanie alebo pokles atmosférického tlaku. Podľa získaného výsledku sa následne upraví tlak (s istou náhodnou odchýlkou) a určí sa nový stav počasia. V prípade vygenerovania stavu „zachovaj tlak“ je ten upravený o malú kladnú alebo zápornú konštantu – to znamená, že povoľujeme mierne výkyvy. V prípade, že sa tlak pohybuje na hranici medzi dvoma stavmi (napríklad medzi „zamračené“ a „dážď“), získame premenlivé počasie.

Markovov model je riadený maticou pravdepodobností prechodu medzi jednotlivými stavmi. Tie patria ku definícii sveta (teda nie sú fixne určené v kóde aplikácie, ale v externom súbore). Tak je umožnené ich nastavenie rýchlo meniť podľa požiadaviek. Viac informácii o konkrétnom odporúčanom nastavení je v prílohe. (Sekcia 7.1)

## 2.5 Bezpečnosť

Simulácia je navrhovaná pre použitie v akademickom prostredí a predpokladá sa jej využitie primárne pri výučbe alebo výskume v oblasti inteligencie agentov. Preto nekladieme žiadne zvýšené požiadavky na bezpečnosť. Vzhľadom na modulárny dizajn projektu a úplné oddelenie komponentov komunikujúcich cez sieť je možné v prípade dodatočného vzniku takýchto požiadaviek modifikovať triedu implementujúcu sieťové jadro tak, aby realizovala kryptograficky zabezpečenú autentifikáciu a šifrovala komunikáciu. Žiadne ďalšie úpravy nebudú na strane servera nutné.

## 2.6 Kapacitné požiadavky

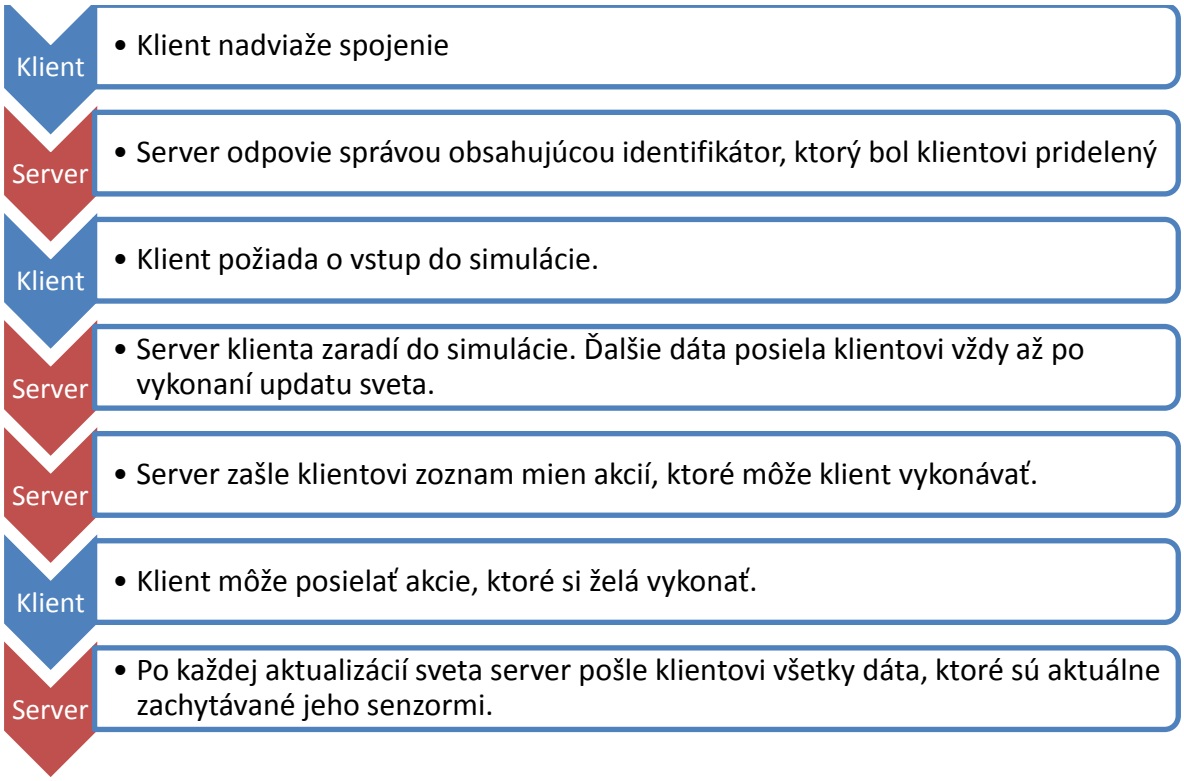
Simulátor bol navrhovaný primárne na využitie v prípadoch s jedným agentom. Podporované sú aj multiagentové scenáre. Väčšina testov prebiehala s jedným alebo najviac tromi súčasne prítomnými agentmi. Predpokladali sme, že ich v žiadnom momente nebude v simulácií prítomných viac ako päť. V takýchto prípadoch server pracuje korektne, no nebolo vykonané žiadne rozsiahlejšie testovanie. Pre viacero informácií o tejto problematike pozri sekciu *5.1 Možnosti rozšírenia*.

## 2.7 Komunikačný protokol a sieť

Komunikácia medzi TerraSim serverom a klientmi používa centralizovaný klient - server model. Po začatí simulácie sa spustí sieťový server, ktorý obsluhuje pripájajúcich sa klientov a relevantné udalosti či správy posúva simulačnému jadru (pozri kapitolu 2.10). Očakáva sa, že počet klientov súčasne prítomných v simulácii bude malý, v bežnom prípade iba jeden. Server beží v oddelenom vlákne (threade) a pre každého nového vzdialeného klienta vytvorí nové vlákno, ktoré ho bude obsluhovať. V tejto sekcii zadefinujeme protokol, ktorým sa riadi komunikácia, zadefinujeme jej formát a popíšeme realizáciu sieťového servera. Všetky triedy používané v kontexte siete sú zadefinované v mennom priestore `TerraSim.Network`. Implementáciou sa bližšie zaoberáme v sekcii 3.2.

### 2.7.1 Protokol

Komunikácia je iniciovaná zo strany klienta. Ak je simulovaný svet plný, server v tomto bode pripojenie odmietne. V opačnom prípade je klient pripojený a na strane servera je vytvorený agent (inštancia triedy `UserAgent`) ktorý mu je pridelený. Potom môže posielať príkazy aktuátorom tohto agenta. Príkazy sú na strane servera zhromažďované a vykonané pri vyhodnocovaní ďalšieho kroku simulácie. Po každom kroku sú agentovi zaslané vnemy z jeho senzorov bez ohľadu na to, či sa pokúsil vykonať akciu.




**Obrázok 6 Protokol - priebeh komunikácie klienta a servera**

### 2.7.2 Formát správ

Formátom správ sa myslí dátová reprezentácia, použitá na výmenu informácií medzi komunikujúcimi. TerraSim používa obyčajný (plain) text. Jeho výhodou je čitateľnosť a ľahkosť použitia. Potenciálnou nevýhodou je vysoká redundancia a zvýšená náročnosť na sieťový prenos, v praxi bol však vždy únosný a žiaden problém sa neprejavil.

Vynímajúc režijné správy, väčšina komunikácie smerom zo servera ku klientovi bude obsahovať informácie vnímané agentovými senzormi – či už o vlastnostiach objektov alebo o prebiehajúcich akciách. Opačným smerom, od klienta serveru to budú informácie o agentom vykonávaných akciách. Všetky dáta sú vyjadrené predikátmi. Informácie o vlastnostiach sú reprezentované predikátom *má atribút (has attribute)* vo formáte

*has_attribute(cieľ, názov, hodnota)*

kde *cieľ* je objekt, o ktorého vlastnosti predikát hovorí. Je určený reťazcom – unikátnym menom objektu v simulácii. *Názov* určuje názov atribútu objektu a *hodnota* jeho hodnotu. Napríklad vnem hovoriaci že objekt menom *drevo_4* je hnedý vyzerá nasledovne: *has_attribute(drevo_4, farba, hnedá)*.

Informácie o prebiehajúcich akciách sú vyjadrené predikátom *vykonáva akciu (performs action).* Jeho štruktúra je

*performs_action(aktér, akcia, parameter_1, parameter_2)*

Hodnota *aktér* je nositeľ deja, entita, ktorá akciu vykonáva. Je určený reťazcom – unikátnym menom. *Akcia* je meno akcie a ostatné dve hodnoty sú jej argumentmi. Teda zápis vnemu že agent menom *agent_2* ukladá objekt, ktorý práve drží na pozíciu určenú súradnicami (5, 9) má podobu: *performs_action(agent_2, polož, 5, 9).* Vzdialený agent posiela správy s jeho akciami podobným spôsobom. Príklad (výpis) priebehu takejto komunikácie je uvedený v prílohe.

Zvažovali sme použitie dvoch rôznych reprezentácií údajov s vyššie uvedenou štruktúrou. Zjavnou voľbou by bol zápis v predikátoch tak, ako bol popísaný v predošlých odsekoch. Generovanie aj parsovanie takéhoto zápisu by bolo jednoduché[3], ale opakovanie názvov predikátov by bolo veľmi redundantné.

Nakoniec sme zvolili formát JSON [Crockford, 2006]. Vychádza z objektovej notácie JavaScriptu, má jednoduchú a čitateľnú syntax a je značne rozšírený. Jeho spracovanie (parsovanie) je jednoduché a je ľahké sa mu vyhnúť. Existuje mnoho voľne dostupných knižníc, ktoré ho spoľahlivo implementujú a poskytujú aj pokročilejšie služby, ako napríklad automatická serializácia/deserializácia. (Viac o tejto tematike je v kapitole Implementácia, v sekcii 3.2.) Súbor vnemov je potom reprezentovaný dvoma poľami – prvé, zvané `has_attribute` obsahuje usporiadané trojice reprezentujúce jednotlivé vnemy o vlastnostiach, druhé, zvané `performs_action` obsahuje štvorice reprezentujúce vnemy o vykonávaných akciách.

Alternatívou ku JSON bol formát XML. Mechanizmus DTD (definícií typov dokumentov) a uvádzanie mien všetkých objektov (tagov) poskytujú väčšiu robustnosť a spoľahlivosť. JSON je ale stručnejší a jednoduchšie spracovateľný a v prípadoch ako je sieťová komunikácia v TerraSim[4] nie je potrebná dodatočná kontrola správnosti. Sieťové

[3] Vzhľadom na to, že nechceme klásť obmedzenia na platformu a jazyk, v ktorom bude realizovaný vzdialený agent pripájajúci sa na server, musíme brať do úvahy fakt, že nebude použitá naša implementácia protokolu a parsovania správ. Preto je v našom záujme, aby bol formát správy čo najprístupnejší a najjednoduchšie spracovateľný.

[4] Zatiaľ čo správy môžu byť značne rozsiahle, ich štruktúra je jednoduchá a neobsahujú serializované zložité objekty ani hierarchie.

jadro preto používa JSON ale je navrhnuté tak, aby bolo možné jednoducho pridať podporu ďalších formátov.

### 2.7.3 Komunikácia medzi agentmi

V multiagentových scenároch potrebujeme umožniť komunikáciu medzi agentmi. Zabezpečuje ju akcia *tell*, ktorú má každý agent k dispozícii. Očakáva dva argumenty – prvým je meno cieľového agenta, druhým samotný text správy. Do obsahu komunikácie server žiadnym spôsobom nezasahuje. Po prijatí správy *tell* iba nájde cieľového agenta a posunie mu ju bez akejkoľvek kontroly.

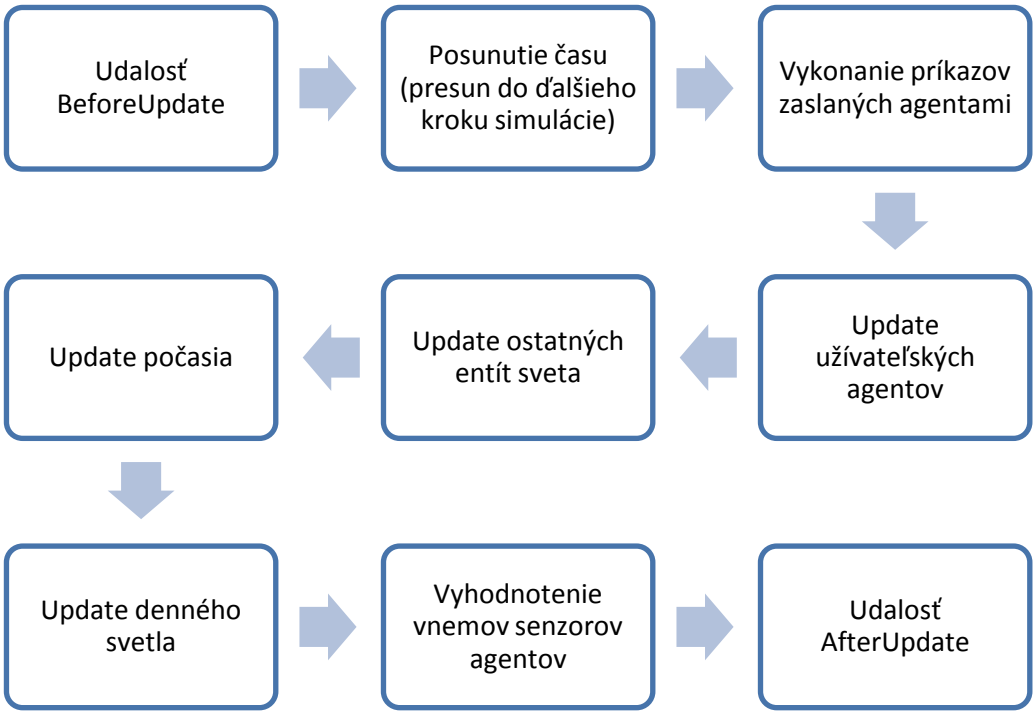
## 2.8 Svet

Simulovaný svet je v programe reprezentovaný inštanciou triedy `World` (menný priestor `TerraSim.Simulation`), ktorá zabezpečuje hlavné mechanizmy jeho fungovania a zároveň oddeľuje obsah sveta od zvyšku aplikácie. (Pozri sekciu 2.9 Modularita.) Práve tu sú udržované informácie o všetkých objektoch, od terénu, cez agentov prostredia až po užívateľských agentov, ako aj informácie o prostredí – aktuálny čas, deň, maximálny počet vzdialených klientov, úroveň osvetlenia, tlak či počasie[5]. Tiež obsahuje mechanizmy komunikácie v rámci simulácie. (Pozri sekciu 2.11)

Osobitnú úlohu vo svete majú užívateľskí (vzdialení) agenti, čo vo viacerých ohľadoch ovplyvnilo jeho celkový dizajn. Chceme, aby sa agenti v prostredí mohli čo najúčinnejšie prejaviť – preto ich akcie majú prioritu pred ostatnými akciami.

Každý príkaz, ktorý server prijme od vzdialeného agenta je poslaný svetu na spracovanie. Príjem správ ale beží v reálnom čase (obsluhovaný vláknami sieťového servera, pozri sekciu 3.2), simulácia v diskrétnych časových krokoch. Tento rozdiel preklenieme tak, že po prijatí správy ju svet zaradí do radu príkazov na vykonanie. Tam sú uchované až do najbližšieho updatu, v ktorom sú realizované. Pre lepšiu ilustráciu celého procesu bližšie popíšeme jeho priebeh.

[5] Použitie počasia je voliteľné. V jednoduchších scenároch nemusí byť potrebné, preto je možné jeho použitie pri vytváraní inštancie triedy `World` zakázať. (Pozri triedu `WorldSettings` v tom istom mennom priestore.)




**Obrázok 7 Priebeh aktualizácie sveta**

Svet poskytuje dve udalosti súvisiace s updatom – jednu, vyvolanú pred jeho vykonaním a jednu bezprostredne po ňom. Simulácia vstúpi do ďalšieho kroku. Začína vykonaním príkazov prijatých od agentov, spomínaných vyššie. Potom sa vykoná aktualizácia stavu (update) užívateľských agentov a ostatných entít sveta (lokálnych agentov alebo bežných entít simulácie). Nakoniec sa aktualizuje počasie (v prípade že sa používa) a stav svetla. V tomto bode je update dokončený. Vyhodnotia sa vnemy senzorov vzdialených agentov, čím sú získané dáta pripravené na serializáciu a odoslanie vzdialeným agentom.

### 2.8.1 Mapa

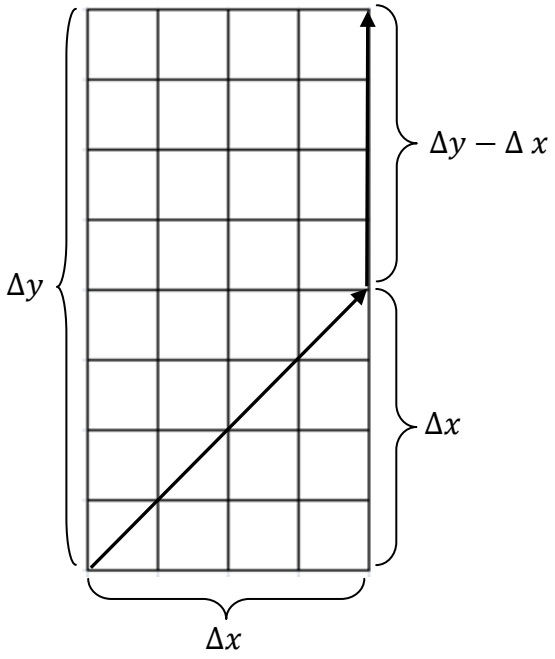
Samotný terén v ktorom sa agenti pohybujú je reprezentovaný mriežkou. (Trieda `Grid`; všetky triedy zmieňované v tejto sekcii sú v mennom priestore `TerraSim.Simulation`.) Tejto mriežke aj s jej obsahom hovoríme mapa. Môže fungovať v jednom z dvoch základných módov – štvorcová alebo šesťuholníková. Mód je dôležitý pri definícii aktuátorov a senzorov – ak sú definované pre fungovanie na štvorcovej mriežke, nemusia fungovať aj na šesťuholníkovej a opačne. Preto sa treba pri dizajne sveta rozhodnúť, ktorý variant bude použitý. ForestWorld, lesný svet ktorý je súčasťou tejto práce, používa šesťuholníkovú mriežku.

Po konštrukcii objektu typu Grid, pri ktorej sa určí typ mriežky, už medzi použitím štvor- a šesťuholníkovej verzie nie je rozdiel. Ku základným metódam, ktoré trieda poskytuje, patrí `Distance` – metóda poskytujúca odhad vzdialenosti medzi dvoma políčkami.

Vzdialenosť na štvorcovej mriežke sme definovali ako maximum z absolútnych hodnôt rozdielov v rámci jednotlivých súradníc. Teda pre body $A = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix}$ a B $= \begin{pmatrix} x_2 \\ y_2 \end{pmatrix}$ je vzdialenosť

$$Distance(A, B) = \max\{|x_1 - x_2|, |y_1 - y_2|\}$$

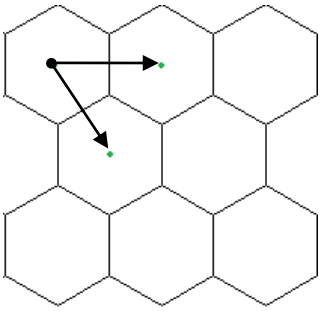
Tento vzťah používame pretože mriežka používa osem-susednosť[6]. Bez ujmy na všeobecnosti predpokladajme, že rozdiel $\Delta x$ v súradnici $x$ je menší než rozdiel $\Delta y$ v súradnici $y$. Potom po najkratšej ceste urobíme $\Delta x$-krát krok šikmo a $\Delta y - \Delta x$ krokov rovno. Spolu sme teda urobili $\Delta y$ krokov, pričom platí $\Delta y = \max\{\Delta x, \Delta y\}$.




**Obrázok 8 Metrika na štvorcovej mriežke**

Pri šesťuholníkoch nie je postup takto priamočiary. Chceme vypočítať vzdialenosť na dvojrozmernej mreži s neortogonálnou bázou.

[6] V prípade štvor-susednosti by sme použili manhattanskú metriku, teda súčet absolútnych rozdielov súradníc.

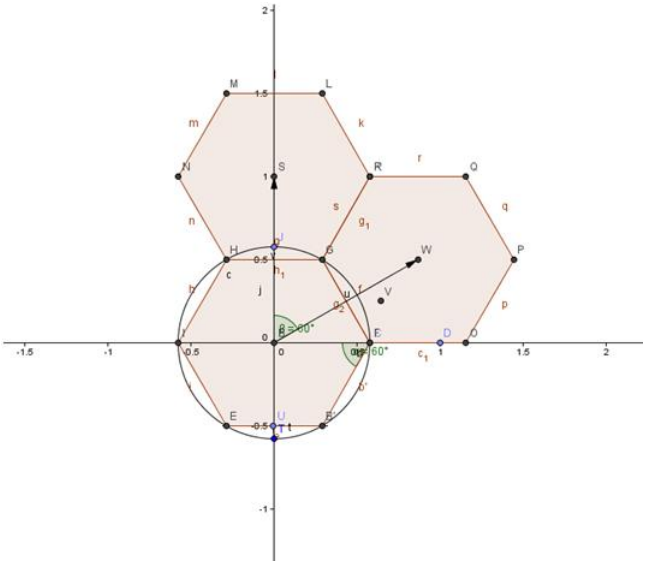


**Obrázok 9 Neortogonálna báza**

Tento problém redukujeme na výpočet vzdialenosti na ortonormálnej báze. Nájdeme zobrazenie z bodov $A, B$ v pôvodnom priestore na body $A', \mathrm{B}'$ v ortonormálnom priestore. Vzájomný vzťah báz týchto priestorov vidíme na obrázku nižšie. Potom platí

$$\begin{pmatrix} \mathrm{A} \\ \mathrm{B} \end{pmatrix} = \mathrm{M} \begin{pmatrix} \mathrm{A}' \\ \mathrm{B}' \end{pmatrix}$$

kde matica M v tvare $\mathrm{M} = \begin{pmatrix} \mathrm{m}_{1,1} & \mathrm{m}_{1,2} \\ \mathrm{m}_{2,1} & \mathrm{m}_{2,2} \end{pmatrix}$, ktorú získame vyriešením vzniknutej sústavy rovníc, je matica transformácie z pôvodného priestoru do nového, ortonormálneho. V ňom potom vypočítame vzdialenosť bežným spôsobom.

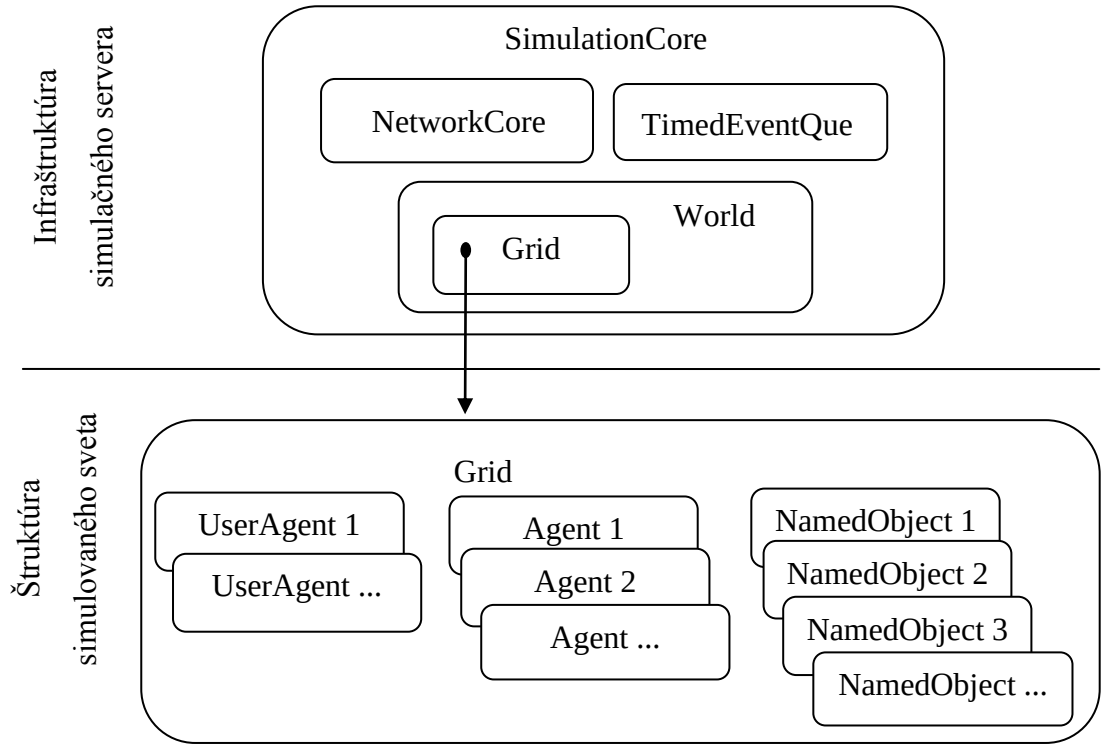


**Obrázok 10 Vzťah štvorcovej a šesťuholníkovej mriežky**

## 2.9 Modularita

Simulátor TerraSim bol navrhnutý tak, aby poskytoval veľa možností budúceho rozširovania. Prvou významnou črtou je možnosť jednoducho definovať nové simulácie. Druhou je existencia rozhrania, ktoré umožní runtime správu a spúšťanie rôznych modulov so simuláciou.

Celá aplikácia je rozdelená do viacerých vrstiev. Najdôležitejšie je jej rozdelenie do dvoch celkov. Prvý zaobstaráva spoločné prvky simulácie. Ku nim patrí správa sieťovej komunikácie, plynutia času (teda iterácií simulácie), simulácia počasia a ostatná časť réžie. Tieto prvky sú pre všetky svety rovnaké, aj keď nie každý z nich ich musí využiť všetky. Pozri Obrázok 11 – časť „Infraštruktúra simulačného servera“.



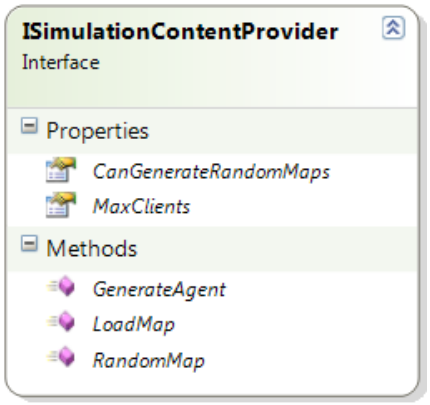

**Obrázok 11 Vzťah pevne danej a užívateľsky definovateľnej časti simulácie**

Druhým celkom je samotný simulovaný svet. Framework TerraSim poskytuje triedy, ktoré slúžia na jeho vytvorenie a manipuláciu s ním. Hlavnou spomedzi nich je `World`, popísaný v sekcii 2.8. Je zapečatená, teda nie je možné ju rozširovať. Zastrešuje simulačnú logiku sveta. Každá inštancia reprezentuje jeden samostatný svet a je populovaná entitami, ktoré sa zúčastňujú simulácie. Tieto entity rozdeľujeme do troch kategórií. Prvou sú základné objekty schopné bežných interakcii. Sú založené na triede `NamedObject` a reprezentujú najmä objekty s jednoduchším správaním (vnútornou logikou); v našom lesnom svete sú to napríklad rastliny alebo kŕmidlo pre zvieratá. Druhou sú všetci zložitejší, interakcie schopní agenti. Vychádzajú z triedy `Agent` a používajú model aktuátorov a senzorov, ako bol popísaný v sekcii 2.1. Poslednú kategóriu tvoria užívateľskí agenti. Ich implementácia (v triede `UserAgent`) je tiež zapečatená (nie je možné ich dediť). Možnosti správania užívateľského agenta sú úplne definované množinou jeho senzorov

a aktuátorov. Táto časť simulácie je flexibilná a dá sa definovať podľa ľubovôle, vytvárajúc rôzne simulácie. Na obrázku 11 s ňou korešponduje časť označená „Štruktúra simulovaného sveta“.

Na tomto spoločnom základe sú stavané simulácie. Pri ich realizácii stačí zadefinovať entity a populovať nimi svet. V bežnom prípade sa očakáva najväčší počet objektov prvej kategórie, menej agentov a podstatne menej užívateľských agentov. (Pozri Obrázok 11.) V kapitole 4 Príklad použitia – implementácia sveta ukazujeme, ako vytvoriť jednoduché prostredie a začať simuláciu.

Majúc na mysli zámer udržať použitie TerraSimu na jednoduchej úrovni, mohli by sme namietať že pre človeka bez prehľadu o celkovej architektúre systému je prvé použitie príliš komplikované, krivka učenia značne strmá. Zavedieme rozhranie zabezpečovateľa obsahu `ISimulationContentProvider`, ktorého implementácia je postačujúca na spustenie simulácie.




**Obrázok 12 ISimulationContentProvider interface**

Jeho použitie je nasledovné: programátorovi, ktorý chce zadefinovať vlastnú simuláciu, stačí vytvoriť triedu, ktorá ho implementuje. Ako vidieť na obrázku 12, tento interface je veľmi jednoduchý. Poskytuje základné meta-informácie o svete (maximálny počet súčasne pripojiteľných agentov a možnosť generovať náhodné svety) a metódy na jeho vytvorenie (načítanie, prípadne generovanie sveta a vytvorenie a umiestnenie agenta vo svete). Existujúca infraštruktúra sa potom postará o spustenie samotného servera a jeho

obsah napopuluje podľa metódy `LoadMap` z tohto interfaceu.[7] (Praktický príklad použitia tohto rozhrania je v kapitole 4.)

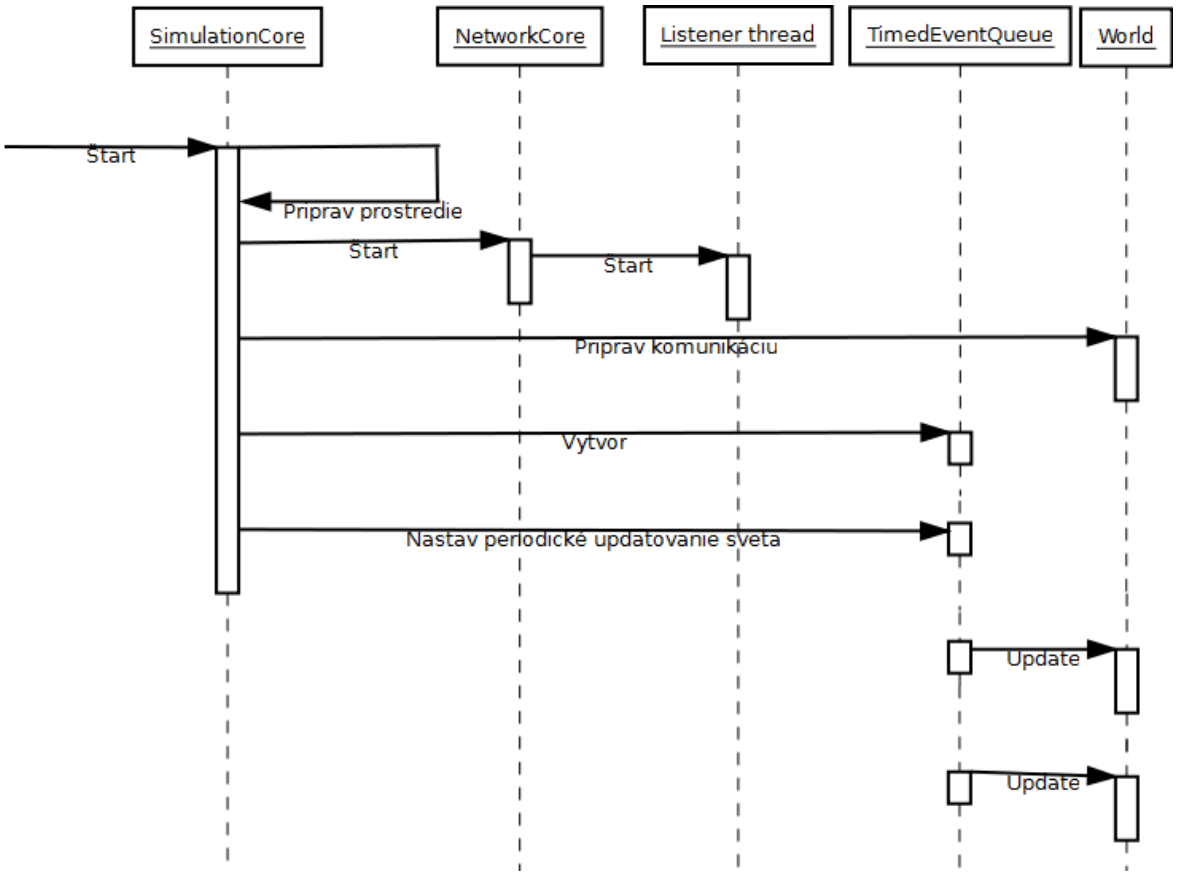
V prípade, že bude potrebné spravovať viaceré simulácie naraz, či už ide o rovnaké alebo rôzne svety, takto navrhnutá architektúra dovoľuje jednoduché pridanie plug-in funkcionality. Je možné zadefinovať dynamické knižnice (DLL), obsahujúce triedu implementujúcu `ISimulationContentProvider`. Tie aplikácia môže detegovať, načítať počas runtime, zobraziť ich metadáta a spustiť simuláciu.

## 2.10 Simulačné jadro

Simulačné jadro (implementované triedou `SimulationCore`) zastrešuje funkcionalitu celej simulácie. Spája správu inštanciácie sveta podľa definovanej logiky, správu nastavení, simuláciu sveta a sieťovú komunikáciu. Všetky tieto časti sú realizované modulmi popísanými vyššie. Taktiež poskytuje funkcionalitu spúšťania udalostí po uplynutí určitého časového úseku (pozri kapitolu 3.4) a zodpovedá za správu behu času v simulácii. V princípe by sme túto triedu mohli prirovnať k realizácií návrhového vzoru „Facade“ zo sady návrhových vzorov známej ako *Gang of Four* [Gamma, a iní, 1994].

Na obrázku nižšie vidíme, ako prebieha inicializácia a spúšťanie simulácie po zadaní pokynu pre štart. Ukladanie detailov do trace záznamu sme z obrázku kvôli prehľadnosti vynechali.

[7] Ak konkrétny SimulationContentProvider neposkytuje funkcionalitu generovania náhodných máp, štandardným správaním príslušnej metódy je zahlásenie výnimky indikujúcej chýbajúcu implementáciu (NotImplemented exception).




**Obrázok 13 Štart simulačného jadra**

Simulačné jadro dostane hotový svet, ktorý bude nositeľom simulácie, o jeho prípravu sa preto nestará. (Zdôvodnenie tohto prístupu pozri v sekcii 2.9.) Načíta konfiguračné informácie, najpodstatnejšou z nich je dĺžka trvania jedného dňa v simulácii. Od toho sa odvíja aj množstvo času, ktoré budú mať agenti v každom kroku k dispozícii na voľbu akcie. Nasleduje príprava sieťovej komunikácie a spustenie updatovania sveta.

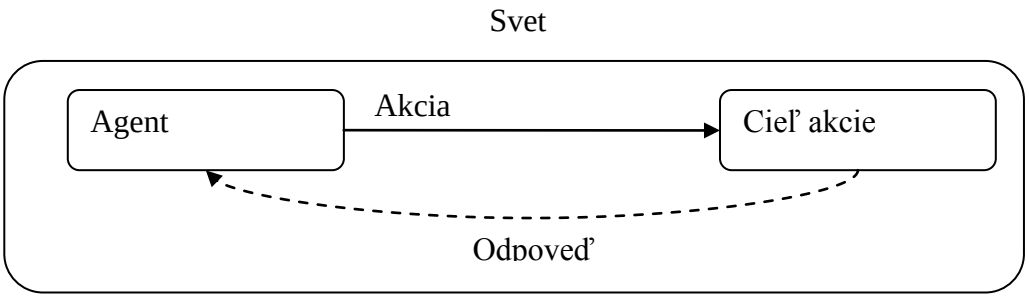
Trieda `SimulationCore` tiež dáva k dispozícii metódy na plánovanie udalostí riadiacich sa reálnym časom (realizované pomocou mechanizmu z 3.4), používaných pri správe behu servera. Je možné využiť ich napríklad na pravidelné zhromažďovanie štatistík o priebehu simulácie alebo jej plánované zastavenie po uplynutí času vyhradeného na riešenie úloh agentmi.

## 2.11 Komunikácia v rámci simulácie (posielanie správ)

V mnohých prípadoch budú musieť agenti komunikovať s ostatnými entitami prostredia. Niekedy pôjde o jednostrannú notifikáciu (napríklad agent notifikuje cieľ akcie, že ho polial vodou), inokedy o obojsmernú interakciu (komunikácia medzi aktuátorom –

motorom a senzorom – batériou, ohľadom možnosti vykonania pohybu) alebo vysielanie správ do okolia (výstrel začujú všetci v blízkosti).

V nasledujúcich sekciách sa budeme venovať návrhu týchto mechanizmov. Ich realizácia nie je úplne priamočiara. Z modulárnej architektúry simulácie vyplýva, že agent nemusí (a nebude) mať znalosti o objektoch, na ktorých svoje akcie vyvoláva. Preto nemôže byť použité priame volanie metód a musíme nájsť spôsob, ako komunikáciu zabezpečiť bez znalostí o architektúre recipienta správy.




**Obrázok 14 Priame volanie metódy**

Dodatočnou požiadavkou je, že ak príde ku pokusu o volanie neexistujúcej metódy (teda vykonanie akcie, na ktorú cieľový objekt nebol stavaný a nevie, ako má reagovať), nesmie prísť ku chybe v aplikácii. Realistický predpoklad je, že väčšina objektov nebude vedieť (ani mať dôvod) reagovať na väčšinu akcií. Využijúc vyššie uvedený príklad: huba môže po zaliatí vodou narásť, ale vodná plocha alebo kŕmidlo pre zver nebudú ovplyvnené. Ďalej je do simulovaného sveta flexibilne možné pridávať nových agentov s novými akciami. Z týchto dôvodov by bolo nepraktické a nemožné implementovať každú akciu v každej entite. Očakávané správanie pri neznámej akcii teda bude jej ignorovanie. Inšpiráciou pre takéto riešenie bol [Benc, 2009].

### 2.11.1 Odozva na udalosti

Na úvod jeden príklad použitia. Agent v simulácii má aktuátor *zbraň* (gun), ktorým môže strieľať po objektoch v okolí. Pri každom výstrele musí aktuátor overiť, či má k dispozícii dostatok munície, či vidí cieľ (tj. je v dostrele) a podobne. Tieto kontroly (a následný update stavu aktuátora) všetky prebehnú „vo vnútri“ aktuátora, preto nepredstavujú žiaden problém. Ten nastáva v komunikácií s príjemcom správy. Ak by agent (resp. jeho aktuátor) poznal implementáciu cieľového objektu, vedel by, či má metódu obsluhujúcu postrelenie a v prípade potreby by ju zavolal. Túto znalosť ale nemá.

Najjednoduchším riešením by bolo zavedenie spoločnej metódy pre všetky objekty, ktorá by akceptovala meno a parametre želanej akcie. Každá entita by ju implementovala podľa svojich požiadaviek, spracovala správy ktoré potrebuje a ostatné ignorovala. Tým by sme splnili požiadavky uvedené v predošlej sekcii.

Nevýhodou takéhoto prístupu by bola neprehľadná a prácna implementácia. Každá entita by musela okrem poskytovania potrebnej logiky aj parsovať správy, čo so sebou prináša zvýšené riziko chyby, nutnosť definovať (a následne udržiavať) reťazcové literály pomenúvajúce akcie a podobne. Zároveň sa snažíme udržať hierarchiu objektov čo najjednoduchšiu a tým pádom aj čo najplytšiu.

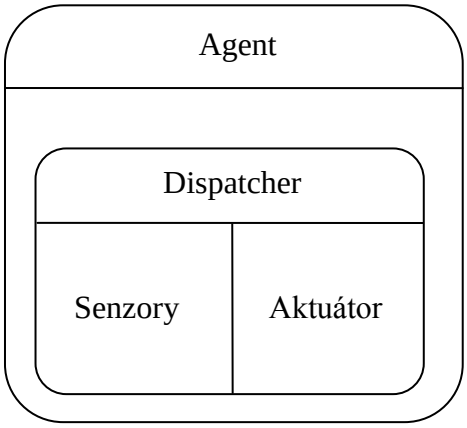
Jedným z možných riešení by mohlo byť využitie multimetód. Od bežných virtuálnych metód sa líšia tým, že ich rezolúcia počas runtime sa neodvíja len od typu jedného, ale viacerých argumentov. V bežnom prípade virtuálneho volania (single dispatch) volaná metóda prislúcha konkrétnej inštancii a po jej zavolaní sa pomocou virtuálnej tabuľky metód vyberie jej správna implementácia. V multiple dispatch scenári sa vyberá metóda podľa runtime typu *všetkých* argumentov. Napríklad ak chceme reprezentovať akciu streľby, definovali by sme metódy ako streľ(agent1, zviera) a streľ(agent2, strom) a výber správnej metódy ponecháme na rezolučný algoritmus.

Druhým riešením je dynamické volanie metód na cieli akcie. Agent sa môže na recipientovi pokúsiť dynamicky zavolať metódu, obsluhujúcu príslušnú udalosť. V prípade že metóda je v danom objekte k dispozícii, je zavolaná s určenými parametrami; v opačnom prípade pokus o jej volanie zlyhá a skončí chybou.

Oba vyššie uvedené princípy (a ich realizácie) sú úzko späté s použitým programovacím jazykom. Podrobnosti implementácie budeme rozoberať v kapitole 3. Otázkou naďalej zostáva, ktorá časť entity v simulácii má zabezpečovať túto funkcionalitu. Naivným riešením by bolo rozšíriť (dedením) agenta a obohatiť ho o požadované správanie. To je ale v rozpore s celkovou dizajnovou filozofiou projektu. Snažíme sa zbytočne nerozširovať existujúce základné triedy agentov a udržať ich hierarchiu čo najplytšiu.[8]

[8] Viac o návrhoch hierarchií je v [Benc, 2009]

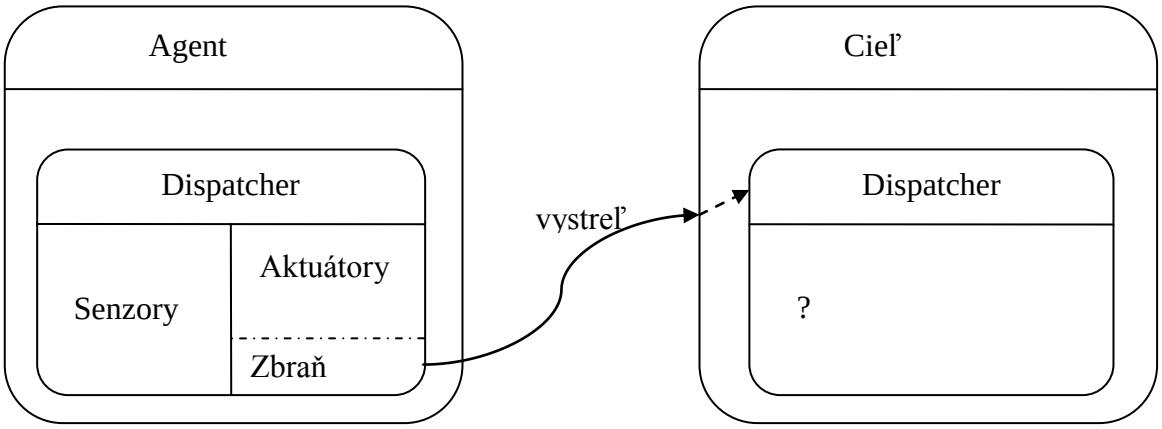
Zavedieme preto nový objekt, vlastnený agentom. Pri volaní ľubovoľnej metódy prislúchajúcej k udalosti bude táto volaná na tomto objekte. Tým oddelíme implementáciu správania od triedy agenta. Pri použití stačí definovať takýto objekt, ktorý budeme nazývať *sprostredkovateľom* (*dispatch mediator* alebo *dispatcher*) a priradiť ho ku agentovi.




**Obrázok 15 Agent so sprostredkovateľom správ**

Akákoľvek dynamicky volaná metóda potom bude smerovaná na tohto sprostredkovateľa. V prípade že je k dispozícií, bude vykonaná; v prípade že nie je, nespôsobí žiadne vedľajšie efekty. Takýto spôsob rozširovania funkcionality parametrizáciou, bez nutnosti modifikovať alebo dediť existujúce triedy, je bežne používaný okrem iného napríklad v Objective-C Cocoa frameworku [Apple Inc., 2010].

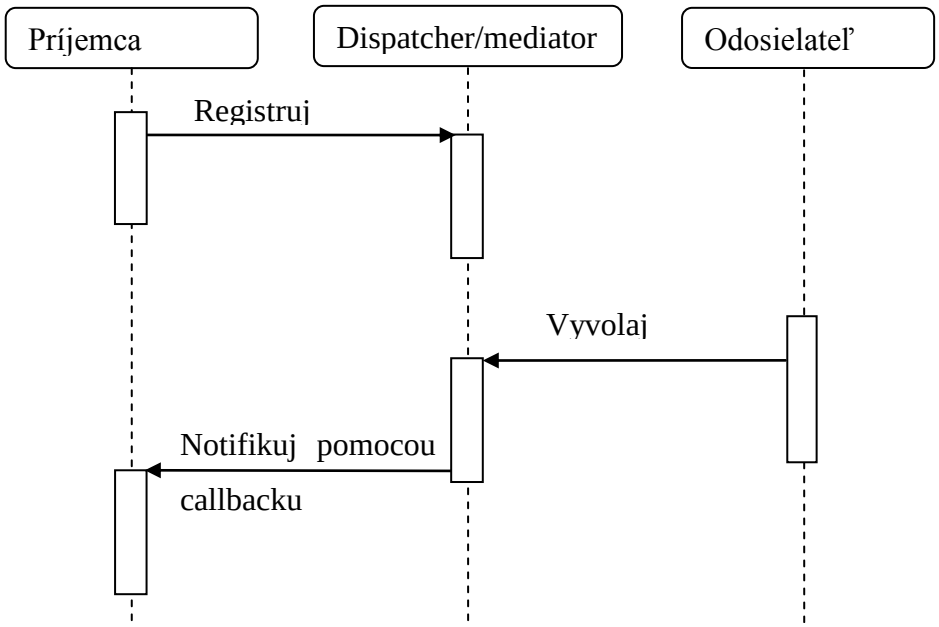
Pre objasnenie konceptu uvedieme príklad použitia. Uvažujme situáciu, v ktorej Agent chce strieľať na Cieľ. Vydá pokyn svojmu aktuátoru Zbraň, aby použil akciu *vystreľ*. Agent ale nemá žiadnu znalosť o štruktúre Cieľa. Môže sa spoľahnúť iba na fakt, že ako každý objekt v simulácií, Cieľ má definovaný Dispatcher, starajúci sa o vykonávanie správ. Aktuátor Zbraň sa teda pokúsi zavolať akciu *vystreľ* na dispatcheri prislúchajúcemu cieľu. To sa buď podarí a akcia sa vykoná, alebo správanie v tejto situácií nebolo pre cieľ definované a nevykoná sa žiadna akcia.




**Obrázok 16 Použitie dispatchera pri vykonávaní akcií**

### 2.11.2 Vysielané udalosti

Ak chceme vysielať správu o nejakej udalosti viacerým recipientom bez nutnosti poznať ich vopred, mechanizmus navrhnutý v predošlej kapitole nestačí. Zavádzame preto systém, ktorý umožní agentom zaregistrovať sa u centrálneho dispatchera. Určia meno správy prislúchajúcej k udalosti, o ktorej chcú byť notifikovaní a metódu, ktorá má byť zavolaná pri jej výskyte. Zaregistrovať sa, ale aj vyvolať ktorúkoľvek udalosť môže ľubovoľný objekt simulácie. Na každú udalosť môže byť zaregistrovaných ľubovoľne veľa odberateľov.




**Obrázok 17 Registrácia a volanie callbacku**

Na obrázku Príjemca je agent, ktorý chce byť notifikovaný o výskyte určitej udalosti U. Preto sa zaregistruje u dispatchera, uvedie názov U a metódu, ktorá má byť volaná keď U nastane. Od tohto bodu keď ktorýkoľvek agent spôsobí výskyt U, dispatcher zavolá všetky callback metódy asociované s touto udalosťou a do každej z nich pošle obdržané argumenty. Podrobnejšie sa tomuto mechanizmu venujeme v kapitole Implementácia, sekcia 3.3.

# 3 Implementácia

V predošlej kapitole sme popísali všetky aspekty simulovaného prostredia TerraSim. Teraz sa budeme venovať implementácii. Popíšeme použité technológie, postupy, programovacie praktiky a rozčlenenie bázy kódu do jednotlivých projektov.

Programovací jazyk a prostredie sme volili tak, aby umožnili veľkú flexibilitu a zároveň poskytli funkcie na dostatočne vysokej úrovni. Chceli sme čo najviac času venovať dizajnu konceptov a funkcionality a nemusieť sa zaoberať low-level implementačnými detailmi. Preto sme zvažovali jazyky Python alebo C#.

Oba z nich umožňujú dynamickú rezolúciu volaných metód a poskytujú prostriedky na pohodlnú prácu s kolekciami dát inšpirované funkcionálnym programovaním [9]. V Pythone sa používa list comprehension, C# má k dispozícii podstatne bohatší mechanizmus, LINQ (Language INtegrated Query). Oba tiež umožňujú použitie multimetód. V C# sa dajú implementovať pomocou `dynamic` [Burchall, 2009], v Pythone rôznymi spôsobmi použitím dekorátorov [Rossum, 2005] alebo použitím existujúcich implementácií (modul multimethods alebo PEAK). Ich využitie sme však nakoniec obišli.

Finálna voľba padla na jazyk C# a framework .NET vo verzii 4. Oproti Pythonu poskytuje okrem iného aj statickú analýzu, kontrolu typov a praktické vývojové prostredie. Použili sme Visual Studio 2010 Professional. Okrem bežných IDE funkcií ako kompilátor a debugger poskytuje aj nástroje na tvorbu a spúšťanie unit testov, refaktorizáciu a generovanie kódu podľa UML Class diagramov (a opačne, vizuálnej reprezentácie vzťahov tried definovaných v kóde). Na správu zdrojového kódu sme použili source control systém Subversion. Repozitár s finálnou verziou a prípadnými budúcimi updatami sa nachádza na CodePlex stránke TerraSimu [Benc, 2011].

## 3.1 Programovacie praktiky

Pri programovaní kladieme dôraz na čitateľnosť a prehľadnosť kódu. Snažíme sa o konzistentný štýl písania a pomenovávania tried, rozhraní, premenných a iných druhov objektov. Všetky texty a názvy v celej code base sú v anglickom jazyku. Systém je dizajnovaný modulárne a coupling (vzájomnú prepojenosť) medzi subsystémami sa snažíme udržať na minimálnej úrovni.

[9] Poskytujú samozrejme oveľa bohatšiu funkcionalitu. Zmieňujeme iba pre nás najpodstatnejšie časti.

Držíme sa paradigmy objektovo orientovaného programovania, no pri tvorbe hierarchie tried dodržujeme dodatočné pravidlá. Programujeme na rozhranie, nie na implementáciu – teda abstrahujeme od realizačných detailov jednotlivých tried. Dedenie, ako jeden z najpevnejších vzťahov do akého môžu triedy vstúpiť, používame v malom množstve a v prípade že je to možné, nahrádzame ho kompozíciou [Benc, 2009]. Takto sa snažíme zvýšiť čitateľnosť a zjednodušiť údržbu kódu.

Aby bolo možné počas nasadenia aplikácie sledovať jej správanie, dôležité moduly (ako simulačné alebo sieťové jadro) používajú funkcie trasovania (trace) na záznam udalostí. Riešenie trasovania v .NET umožňuje pridať sledovanie trace záznamu, nastavenie jeho parametrov a ukladanie do súboru aj bez rekompilácie, iba upravením konfiguračného súboru. Záznamy sú rozdelené do kategórií podľa úrovne dôležitosti: informácie, varovania a chyby. Odberateľ môže určiť, o ktorý stupeň závažnosti má záujem. Niektoré aplikácie (3.5 Server s užívateľským rozhraním, 3.6 Manuálne ovládaný klient) priamo vypisujú text trace vo svojom užívateľskom rozhraní. Jednoduchý spôsob nastavenia odoberania je ukázaný v sekcii 4.3 Príprava simulácie.

Zdrojové kódy sú do projektov rozdelené nasledovne:

- TerraSim
    - Typ projektu: .NET 4 DLL knižnica
    - Hlavná časť projektu. Obsahuje všetky komponenty aplikácie súvisiace so simuláciou, definície sveta s vysávačom (VacuumBot) a lesného sveta (ForestWorld) a jadro manuálneho klienta (pozri sekciu 3.6).
    - Obsahuje niekoľko dokumentov s doplňujúcimi informáciami, diagramy tried alebo pomocné skripty.
    - Implementovaný ako DLL knižnica, aby bolo možné začleniť ho do ďalších aplikácií (pozri napríklad projekt VacuumBot nižšie)
- Tests
    - Typ projektu: Test project
    - Pre niektoré dátové štruktúry a algoritmy sme vytvorili unit testy. Tento súbor testov umožňuje kontrolovať, či sa jednotlivé komponenty správajú

podľa očakávaní a pri zmenách alebo refaktorizácii kódu skontrolujú, či v niektorom z nich nevznikla chyba.

- ServerUI (sekcia 3.5)
    - Typ projektu: WPF aplikácia
    - Vizuálne užívateľské rozhranie pre ForestWorld simuláciu. Umožňuje načítanie sveta, nastavenie parametrov a spustenie simulácie.
- ManualClientConsole (sekcia 3.6)
    - Typ projektu: WPF aplikácia
    - Umožňuje sa pripojiť do ľubovoľnej TerraSim simulácie. Využije vstavané mechanizmy na zistenie schopností agenta, preto ho dokáže ovládať v ľubovoľnej simulácii (bez ohľadu na použité senzory a aktuátory).
- VisualForestWorld (sekcia 3.7)
    - Typ projektu: WPF aplikácia
    - Vychádza z manuálneho klienta. Umožňuje ovládať agenta vo svete ForestWorld. Poskytuje vizualizáciu – grafickú reprezentáciu agentových aktuálnych vnemov.
- VacuumBot
    - Typ projektu: Windows Forms
    - Ako demonštrácia možnosti vytvoriť iné svety pre TerraSim vznikol svet s vysávajúcim agentom, ktorého cieľom je upratať miestnosť v ktorej sa nachádza. V kapitole 4 na ňom predvádzame postup definície sveta a spustenia simulácie.
    - Aplikácia obsahuje server aj vizualizáciu celého sveta. Na jej ovládanie použijeme manuálneho klienta spomínaného vyššie (alebo vlastného klienta).

## 3.2 Komunikačný protokol a sieťové jadro

V sekcii 2.7 sme popísali dizajn sieťového servera a komunikačného protokolu. V tejto sekcii ho spresníme a doplníme o technické údaje.

Server realizujúca trieda sa volá `NetworkServer` a (ako všetky so sieťou súvisiace komponenty) nachádza sa v mennom priestore `TerraSim.Network`. Zabezpečuje komunikáciu cez TCP/IP pomocou Windows Sockets (Winsock), resp. ich riadenej implementácie v .NET. Poskytuje na odber udalosti (events) notifikujúce o pripojení či odpojení klienta a o prijatí správy.

Po štarte serveru sa vytvorí thread počúvajúci na pridelenom porte. Ak nie je určené inak, použije sa port 8183. Tento thread pri každom novom pokuse vzdialeného klienta o nadviazanie komunikácie vytvorí nový thread, ktorého úlohou bude zabezpečovať komunikáciu iba s týmto jedným klientom.

Dátovým formátom komunikácie je JSON. Využívame jeho existujúcu implementáciu tretej strany, Json.NET [Newton-King, 2011]. Táto knižnica obsahuje bohatú funkcionalitu vrátane serializácie/deserializácie a podpory pre LINQ. Umožňuje jednoduchý, deklaratívny prístup ku parsovaniu súborov. Ak určíme logickú štruktúru súboru, stačí nám na úrovni programu zadefinovať objekt s touto štruktúrou zloženou z dostupných primitív a použiť Json.NET na deserializáciu. Tým získame správne typovanú pamäťovú reprezentáciu tohto súboru bez akejkoľvek nutnosti spracovania textu.

Na reprezentáciu kolekcie dát používame triedu `DataCollection`. Do nej ukladáme primárne vnemy o akciách a vlastnostiach, zachytené senzormi agenta. Je nezávislá od reprezentácie, do ktorej bude marshallovaná pri posielaní sieťou. Konverziu má na starosti statická trieda `Marshallers`. Poskytuje metódy na marshalling dát do predikátovej alebo JSON podoby. Jej rozšírením môžeme pridať nové formáty komunikácie.

Okrem funkcionality na strane servera poskytujeme aj implementáciu klienta (trieda `NetworkClient`). Je možné ju použiť z ľubovoľného programovacieho jazyka z rodiny CLR. Zabezpečuje všetku funkcionalitu potrebnú na komunikáciu so simulačným serverom. Pomocou eventov notifikuje o pripojení alebo odpojení zo servera, prijatí správy a jej spracovaní. Tiež umožňuje posielanie správ vo viacerých formátoch.

## 3.3 Dispatcher správ

V sekcii 2.11 sme navrhli model šírenia správ slúžiaci na obsluhu udalostí. Pozrieme sa bližšie na spôsob jeho realizácie.

Multimetódy sú metódy, ktoré môžu byť definované ako preťažené. Z rôznych variantov sa potom za behu programu vyberá najvhodnejší kandidát podľa typov argumentov. Na rozdiel od štandardného, single dispatch mechanizmu sa metóda nevyberá podľa typu prvého, ale všetkých argumentov. Takáto funkcionalita je dostupná napríklad v Common Lispe [Keene, 1988], ale nebýva základnou výbavou bežných moderných jazykov. V staticky typovaných jazykoch (Java, C#) väčšinou chýba. V prípade dynamických jazykov je možné ju pridať. V C# od verzie 4.0, v ktorej bola zlepšená podpora dynamického volania metód sú realizovateľné tiež – buď priamo, využitím typu dynamic [Burchall, 2009], alebo hotovej knižnice tretej strany, akou je napríklad multimethod-sharp [Jones, 2010].

Multimetódy možno jednoducho využiť na realizáciu interakcií medzi objektmi napríklad nasledovne:

```
1  class Agent { ... }
2  class Plant { ... }
3  class Animal { ... }
4  function eat(Agent, Plant) { ... }
5  function eat(Agent, Animal) { ... }
6  function eat(Animal, Plant) { ... }
7  u = new Agent
8  p = new Plant
9  a = new Animal
10 eat(u, p)
11 eat(u, a)
12 eat(a, p)
```

**Kód 1 Použitie multimetód**

V kóde vyššie uvažujeme triedy `Agent`, `Plant` a `Animal` a funkciu `Eat`. Single dispatch jazyk by nedokázal rozlíšiť ktorú verziu metódy `Eat` má zavolať; použitím

multimetód tento problém odstránime a vybraná bude metóda, ktorej signatúra sa najviac podobá použitým argumentom.

Napriek zjavným výhodám, multimetódy narážajú na rôzne úskalia. Priamo ich podporuje iba málo jazykov a v prípade použitia nadstavby, ktorá ich funkcionalitu pridá do iného jazyka môže vzniknúť nežiadaný výkonnostný overhead. Problémom je aj samotný fakt, že výber metódy sa deje podľa *typu* objektov. To by znamenalo mať špecializovaný objekt pre obsluhu správy. V kapitole 2 uvádzame, že nechceme vytvárať nutnosť dediť z aktorov. Následkom toho by s konkrétnou správou boli previazané triedy, nie konkrétne určená metóda[10]. Bolo by možné volať tieto metódy na komponent určený na sprostredkovanie správ. V tom prípade ale vieme nájsť aj riešenie bez použitia multimetód.

Ďalším úskalím je všeobecná syntaktická nekonzistencia medzi volaním multimetód a volaním inštančných metód. Pri multimetódach ide už o spomínané `Eat(a, b)`. Celý program je ale písaný v objektovej paradigme, takže bežné volanie vyzerá ako `a.Eat(b)`. Pri takomto zápise ale nie je intuitívne zrejmé, že by mal výber správneho preťaženia metódy prebiehať podľa oboch parametrov, keď vo väčšine volaní to tak nie je.

Väčšina prípadov bude vyzerať ako na obrázku 14 (Priame volanie metódy) – bude v nich figurovať práve jeden volajúci a cieľ. Schému navrhnutú v 2.11.1 aj 2.11.2 potom môžeme implementovať pomerne priamočiaro. Na implementáciu použijeme jazyk C# a prostredie .NET vo verzii 4.0. Od tejto verzie sú v nich k dispozícii dynamické objekty. Držíme sa princípu „je jednoduchšie žiadať odpustenie než povolenie," používaného v kontexte niektorých dynamických jazykov [Python Software Foundation, 2011]. Štandardný postup volania metódy `Action` s parametrom `param`, aktorom `A` na cieli `Target` preto bude vyzerať nasledovne:

```
1  try {
2     Target.DispatchMediator.Action(param)
3  }
4  catch (MethodNotFoundException)
5  { /* žiadna akcia */ }
```

[10] Tým by bolo znemožnené aj použitie lambda výrazov na mieste callback funkcií.

**Kód 2 Dynamické volanie metódy**

Pokúsime sa zavolať želanú metódu, predpokladajúc že je k dispozícii. V prípade že nebola nájdená, runtime zahlási výnimku, ktorú zachytíme a ignorujeme. Tak sme okrem vykonania správnej akcie zabezpečili zachovanie nezmeneného stavu v prípade zlyhania volania.

### 3.3.1 Vysielanie udalostí, posielanie parametrov a spätná väzba

Problém zachytávania výskytu udalosti je v praxi bežný, preto moderné programovacie jazyky poskytujú funkcie na jednoduchú prácu s nimi. Aj v jazyku C# sú k dispozícií pod menom udalosti (events). V našom prípade namiesto nich použijeme riešenie s nízkymi režijnými nákladmi – v slovníku budeme pre každú správu (určenú menom) uchovávať kolekciu akcií, ktoré sa majú vykonať.

Keď entita chce vyvolať nejakú udalosť, určí jej argument a môže určiť aj dodatočnú callback metódu. Táto je poslaná do metód registrovaných ku tejto správe ako ďalší argument. Môže (ale nemusí) byť touto metódou využitá na poslanie informácií späť volajúcej entite. (Pozri príklad nižšie.) Veľmi dôležité je uvedomiť si, že callback metóda bude poslaná do *každej* metódy registrovanej pri tejto správe. Implikuje to viaceré významné následky. Pretože agent nevie, koľko odberateľov je zaregistrovaných, nemal by callback byť príliš výpočtovo náročný – jeho mnohonásobné opakovanie by spôsobilo výkonnostný problém. Podstatný je tiež fakt, že ak callback nejakým spôsobom modifikuje stav iného objektu (napríklad vlastníka), bude tento stav modifikovaný viackrát. V extrémnom prípade by mohla vzniknúť vnútorná nekonzistencia dát.

V nasledujúcom príklade ilustrujeme použitie týchto konceptov. Uvažujme správu menom „compute“, ktorá predpokladá na vstupe vstup číslo a callback metódu. Na jej odoberanie zaregistrujeme dve metódy, ktoré vykonajú rôzne výpočty. Potom udalosť compute vyvoláme a budeme pozorovať, ako sa systém bude správať. Kvôli prehľadnosti boli viaceré detaily z ukážky vynechané, prezentujeme pseudo-kód.

```
1  MessageBroadcaster b;
2  function sucet(a, callback):
3  callback(a + 2)
4  function sucin(a, callback):
5  callback(a * 2)
6  b.Subscribe("compute", sucet)
7  b.Subscribe("compute", sucin)
8  vysledok = 0
9  callback = (i) => { vysledok = i }
10 b.Broadcast("compute", 3, callback)
```

**Kód 3 Príklad na použitie systému udalostí**

Na 2. riadku je zadeklarovaná funkcia, ktorá ku vstupnému argumentu pripočíta 2 a s výsledkom zavolá callback funkciu. Na riadku 4 je podobná funkcia, ktorá vstupný argument vynásobí dvomi. Na riadku 5 a 6 ich zaregistrujeme na správu `compute`. Na 8. zadefinujeme premennú `vysledok`, v ktorej budeme chcieť mať hodnotu vypočítanú funkciou. Na 9. riadku zadeklarujeme a do premennej `callback` priradíme anonymnú funkciu, ktorá iba svoj jediný argument priradí do premennej `vysledok`. Následne vyvoláme udalosť `compute` s parametrom 3 a funkciou callback. Aká bude po jej vykonaní hodnota premennej `vysledok`?

Je dôležité si uvedomiť, že na riadku 9 sa vytvorí lexikálny uzáver nad premennou `vysledok`. Funkcia callback preto môže manipulovať s jej hodnotou napriek tomu, že je umiestnená v nadradenom kontexte. V každom prípade sa vykonajú obe funkcie `sucet` aj `sucin` a nie sú žiadne garancie na ich poradie. Bez ujmy na ďalšej diskusii predpokladajme, že sa vykonajú v poradí v akom sú napísané vyššie.

Najprv je zavolaná funkcia `sucet`. Tá zavolá callback, ktorá do premennej `vysledok` priradí hodnotu 5 (3 + 2). Následne funkcia `sucin` opäť zavolá `callback`, ktorá do `vysledok` priradí 6 (3×2). Finálna hodnota teda bude 6 a závisí od poradia, v ktorom boli funkcie volané. Preto nie je vhodné umiestňovať náročný kód do callback funkcií, robiť lexikálne uzávery nad premennými, ku ktorým pristupuje viacero vláken

(threadov) súčasne[11] alebo nad properties, ktorých set metódy spôsobujú vedľajšie účinky alebo sú výpočtovo náročné.

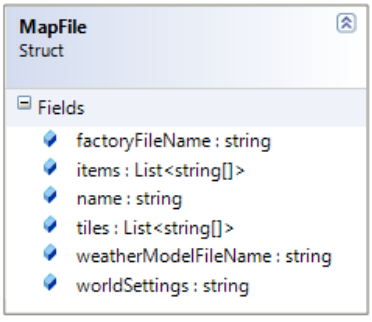
## 3.4 Časový rad udalostí

Server niektoré úlohy potrebuje vykonávať s oneskorením, meraným nie v čase simulácie, ale v reálnom čase. Využitie spočíva napríklad v možnosti obmedziť dĺžku behu serveru, vynútiť beh garbage collectora po uplynutí dlhšej doby, logovanie, notifikácie a iné. Za týmto účelom zavádzame pomocnú štruktúru `TimedEventQueue` (z menného priestoru `TerraSim.Simulation`). Po vytvorení je zastavená a nevykonáva udalosti až do jej explicitného spustenia.

Poskytuje metódy na naplánovanie vykonania akcie. Parametrom sa určuje časový interval po ktorom sa má vykonať a či sa má vykonať opakovane. Akcia je vždy reprezentovaná bezparametrovou funkciou (`Action`). Ak volajúci potrebuje použiť konkrétnu funkciu a ako argumenty jej poslať dodatočné dáta, vytvorí lexikálny uzáver nad týmito parametrami tak, aby získal bezparametrovú funkciu. V C# je to možné realizovať napríklad pomocou anonymných funkcií (resp. lambda výrazov).

## 3.5 Server s užívateľským rozhraním

Pre simulovaný svet s lesom („Forest world“) poskytujeme pripravený server s užívateľským rozhraním, ktoré umožňuje meniť nastavenia a načítať rôzne mapy. Každá mapa má nasledovnú štruktúru:

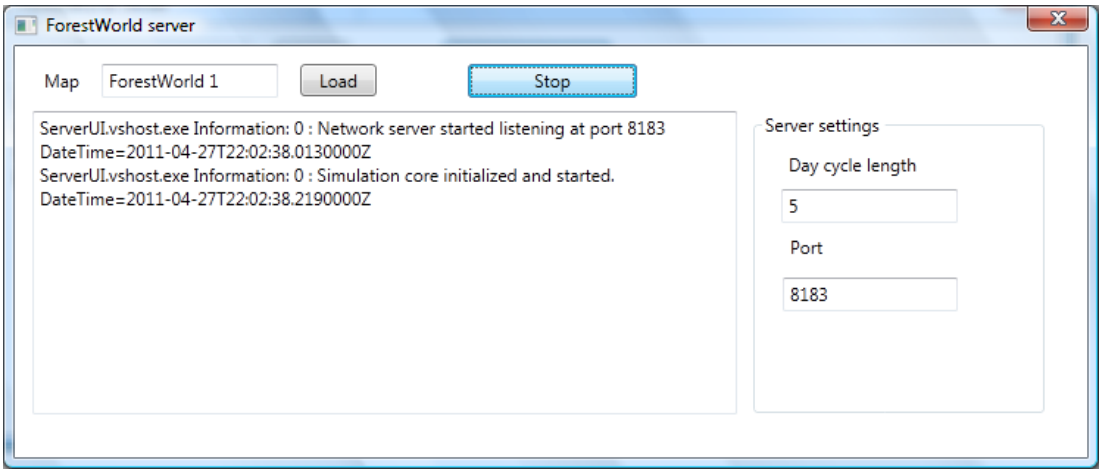



**Obrázok 18 Definícia štruktúry súboru s mapou**

Ako vidno na obrázku, súbor obsahuje názov mapy, popis políčok (terénu) a rozmiestnenia predmetov a ostatných entít. Okrem toho obsahuje názvy súborov,

[11] Ak by iný thread prečítal hodnotu premennej vysledok po zapísaní čísla 5, ale pred zapísaním 6, mohlo by prísť ku vnútornej nekonzistencii.

v ktorých sa nachádzajú nastavenia sveta a továreň na tvorbu políčok (pozri sekciu 2.8) a model počasia (sekcia 2.4).

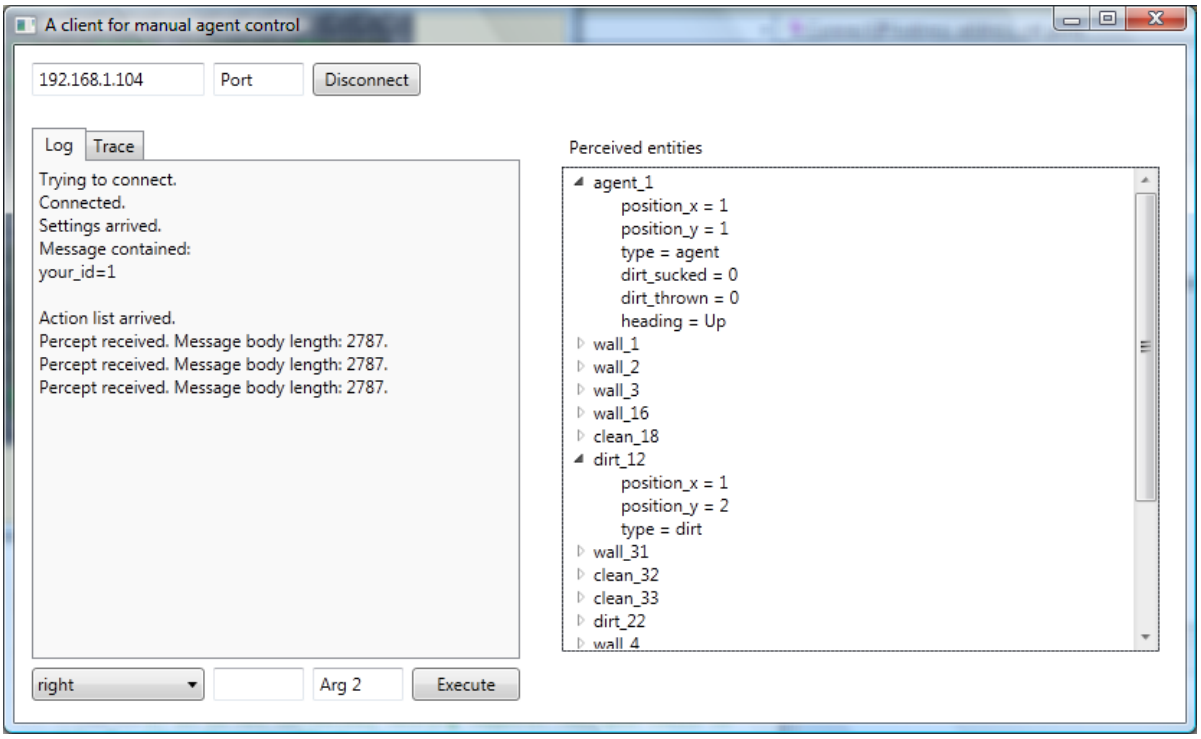



**Obrázok 19 Bežiaci server s užívateľským rozhraním**

## 3.6 Manuálne ovládaný klient

Pre uľahčenie testovania funkčnosti servera sme implementovali oddelenú aplikáciu. Poskytuje možnosť pripojiť sa na ľubovoľný server a riadiť agenta manuálne zadávanými príkazmi. Po zadaní IP adresy (prípadne aj portu, ak sa líši od predvoleného) sa vytvorí spojenie so serverom. Následne je na strane servera vytvorený agent. Klient (okrem iného) obdrží zoznam akcií, ktoré je schopný daný agent vykonávať. Tie sú potom sprístupnené v užívateľskom rozhraní. Takto je zabezpečená použiteľnosť aplikácie so všetkými simulovanými svetmi, bez ohľadu na ich sadu aktuátorov.

Aplikácia priebežne vypisuje záznamy o svojej činnosti a priebehu komunikácie so simulátorom. Po prijatí dát o vneme agenta aktualizuje zoznam entít, ktoré boli agentom pozorované a zobrazí ho. Užívateľ môže kedykoľvek vybrať nejakú z dostupných akcií, zadať jej parametre a zadať ju na vykonanie.



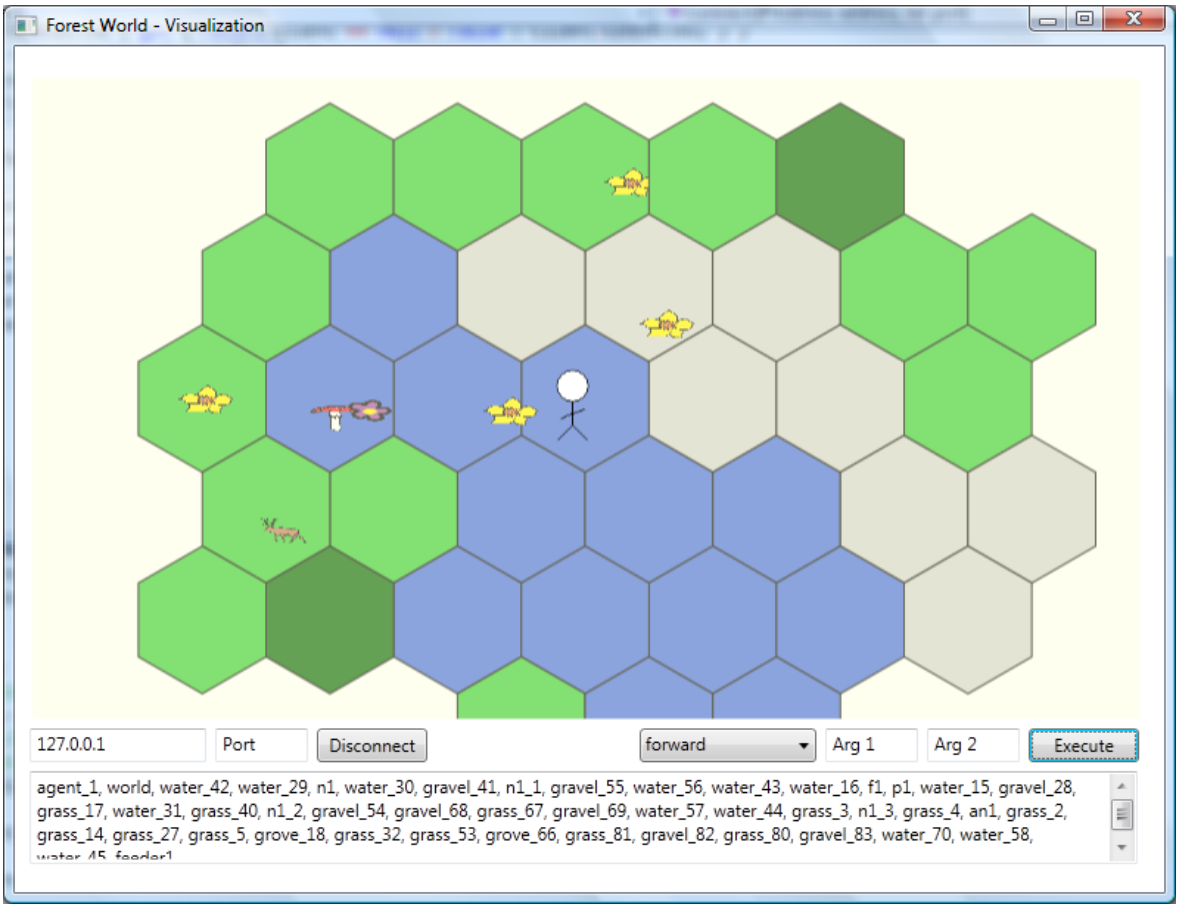

**Obrázok 20 Klient manuálneho ovládania agenta.**

## 3.7 ForestWorld – vizualizácia

Aplikácia VisualForestWorld sa podobá manuálnemu klientovi – umožňuje ručné ovládanie agenta. Rozdiel spočíva v pridanej jednoduchej vizualizácii agentových vnemov, čo môže uľahčiť orientáciu pri ovládaní. Nevýhodou je použiteľnosť iba v prípade sveta ForestWorld.

Rozloženie ovládania je podobné klientovi v predošlej sekcii. Po zadaní adresy a portu (ak sa líši od predvoleného) sa môžeme pripojiť na server. Klient získa zoznam akcií a menný zoznam entít, ktoré vníma a môže chcieť použiť ako parameter akcie.

Vo vizualizácii sú farebne odlíšené druhy terénu a grafickými ikonami zaznačené rôzne druhy predmetov. Zobrazenie bledne a tmavne podľa aktuálnej dennej doby (množstva svetla vo svete). Ovládaný agent sa nachádza vždy v strede obrazu. V každej iterácii sa zobrazuje iba obsah posledných vnemov.




**Obrázok 21 Vizualizácia lesného sveta**

# 4 Príklad použitia – implementácia sveta

V tejto kapitole predvedieme, ako napísať serverovú aplikáciu použitím nášho frameworku. Ako bolo spomenuté v 2.9, framework je navrhnutý modulárne. To umožňuje jednoducho implementovať simuláciu s vlastnou logikou, ale využívajúcu spoločné režijné mechanizmy. Predvedieme to na simulácii založenej na svete obsahujúcom robota s vysávačom, používaným vo výučbe umelej inteligencie na FMFI UK. Funkčná verzia simulácie je obsiahnutá v zdrojových kódoch priložených ku tomuto textu. (Logika sveta je v hlavnom projekte, v mennom priestore `TerraSim.VacuumBot`. Implementácia servera s vizualizáciou je v projekte `VacuumBot` s rovnomenným menným priestorom.)

Pri štarte simulácie je vytvorená uzavretá miestnosť obdĺžnikového tvaru. Jej vnútro, reprezentované štvorcovou sieťou, obsahuje buď priechodné políčka – podlahu, alebo nepriechodné – steny. Niektoré priechodné políčka sú zasypané špinou. Úlohou agenta umiestneného do tejto miestnosti je vysať všetku špinu.

Prostredie má tieto charakteristiky: Miera úspešnosti je počet akcií vykonaných agentom. Počítajú sa všetky akcie – otáčanie, posun i vysávanie. Aktuátory má k dispozícii dva – motor (implementovaný triedou `Motor`) a vysávač (trieda `Vacuum`). Motor zabezpečuje pohyb a dokáže vykonať otočenie vľavo, vpravo a chod rovno, vysávač dokáže vysať špinu. Agentovým senzorom je kamera (trieda `Camera`), ktorá mu v každom kroku sprostredkuje informácie o jeho okolí. Vnemy zahŕňajú 25 okolitých políčok – štvorec veľkosti 5×5, so stredom na pozícii agenta.

Agent sa nachádza v obdĺžnikovej miestnosti ľubovoľnej veľkosti, no vníma z nej iba štvorcový výsek. Celý svet sa skladá zo štvorcových políčok ktoré obsahujú samostatné objekty. Je teda čiastočne pozorovateľné (neprístupné) a diskrétne. Jedinou entitou, schopnou spôsobiť zmenu v prostredí je agent – ostatné objekty sú nepohyblivé a neaktívne. Prostredie teda je statické a deterministické. Agent svojimi akciami v ňom zanecháva dôsledky – preto je sekvenčné. Z dizajnu frameworku, ktorý bude hostiť simuláciu nám vyplýva možnosť určiť, či chceme aby šlo o jedno- alebo multiagentové prostredie.

## 4.1 Senzory a aktuátory

Začneme implementáciou senzoru. Kamera (trieda `Camera` dediaca z abstraktnej triedy `Sensor`) má vrátiť informácie o všetkom, čo sa nachádza v okolí agenta. Trieda `World` poskytuje funkcionalitu na enumeráciu susedných polí v mape sveta, preto je kód priamočiary.

```
1  public override void Sense(WorldUpdateEventArgs args)
2  {
3     LastInput.Clear();
4     foreach (var square in
5           args.World.Map.EnumerateNeighbours(Owner.PositionX,
6           Owner.PositionY, 2))
7     {
8           foreach (var obj in square)
9           {
10                obj.Marshal(LastInput);
11          }
12    }
13    foreach (var obj in args.World.Map[Owner.PositionX,
14          Owner.PositionY])
15    {
16          if (obj != this.Owner)
17          {
18                obj.Marshal(LastInput);
19          }
20    }
21 }
```

**Kód 4 Senzor - kamera**

Na riadku 3 vymažeme vnemy zaznamenané v predošlom kroku (kolekcia `LastInput`), aby sa nemiešali s novými informáciami. V cykle na riadku 4 enumerujeme cez všetky políčka v okolí pozície agenta. Veľkosť okolia je určená polomerom 2. Tým získame štvorec veľkosti 5x5 (2 do každého smeru), vynímajúc pole v strede (to je počiatočné, preto nepatrí medzi susedov).

V cykle na riadku 8 preberieme všetky objekty na tom ktorom políčku a pridáme ich dáta to kolekcie LastInput. Cyklus na riadku 13 už len zabezpečuje pridanie dát o všetkých entitách na poli, na ktorom stojí agent (a ktoré bolo vynechané z enumerácie susedov).

Aktuátory sú o niečo zložitejšie. Z ich kódu v tejto kapitole vynecháme časti nepodstatné pre celkovú funkcionalitu, ako je napríklad zhromažďovanie štatistík o používaní akcií. Vysávač, implementovaný triedou `Vacuum` dediacou z abstraktnej triedy `Actuator`, si nepotrebuje pamätať žiaden dodatočný stav. Ak dostane príkaz a agent stojí na políčku so špinou, odstráni ju. Ak dostane iný príkaz, neurobí nič.

Najprv vyznačíme zoznam akcií, ktoré agent dokáže vykonávať. Kód je priamočiary – vytvoríme novú implementáciu virtuálnej metódy `ActionList`, ktorá vráti zoznam obsahujúci názov jedinej akcie – `suck` (vysaj).

```
1. public override string[] ActionList()
2. {
3.     return new string[] { "suck" };
4. }
```

**Kód 5 Vytváranie zoznamu akcií pre vysávač**

Metódy `Marshal` a `Update` môžeme pre naše účely nechať prázdne, preto ich neuvádzame. Obe sú to abstraktné metódy v rodičovskej triede, preto im treba zadefinovať aspoň prázdne telo. Ostáva metóda `Perform`, ktorá dostane ako parameter názov akcie, jej argumenty a informácie o aktuálnom stave sveta. Práve v nej sa rozhoduje, ako sa aktuátor bude správať.

```
1. public override void Perform(WorldUpdateEventArgs args,
2.    string action, string argument1, string argument2)
3. {
4.     var objects = args.World.Map[Owner.PositionX,
5.         Owner.PositionY].Objects;
6.     var obj = from o in objects
7.       where (o.Type == "dirt")
8.       select o;
9.     var e = obj.GetEnumerator();
```

```
10.    if (e.MoveNext())
11.   {
12.         objects.Remove(e.Current);
13.   }
```

**Kód 6 Hlavná metóda vysávača**

Postup je opäť jednoduchý. Na riadku 4 vyberieme zoznam objektov na políčku, na ktorom sa agent práve nachádza. Na riadku 6 vykonáme dopyt, ktorý vráti enumerátor všetkých objektov z tohto zoznamu, ktorých typ je „dirt“[12]. Ak nejaký bol nájdený, odstránime ho (riadok 12).

Motor je zložitejší. Na rozdiel od vysávača, potrebuje si udržovať dodatočnú informáciu o aktuálnom natočení agenta (Kód 7 - riadky 1 a 5). Statické pole `movementOffsets` obsahuje vymenované súradnice polí, ktoré budeme považovať za susedné. Sú relatívne ku aktuálnej pozícii agenta.

```
1  private enum Orientations { Up = 0, Right = 1, Down = 2, Left =
   3 };
2  private static int[][] movementOffsets = new int[][] {
3     new int[] {0, -1}, new int[] {1, 0}, new int[] {0, 1},
4     new int[] {-1, 0}};
5  private Orientations orientation = Orientations.Up;
```

**Kód 7 Stav motora**

```
1  public override string[] ActionList()
2  {
3     return new string[] { "left", "right", "forward" };
4  }
5  public override void Marshal(DataCollection data)
6  {
7     data.AddAttribute(Owner.Name, "heading",
   orientation.ToString());
8  }
9  private void Turn(int direction)
```

[12] V skutočnosti sa neprehľadáva celý zoznam – výsledkom vyhľadávacieho LINQ výrazu je výrazový strom (expression tree) a jednotlivé výsledky sú vyhodnocované lenivo, až pri enumerácii.

```
10 {
11    int o = (int)orientation;
12    o = (o + 4 + direction) % 4;
13    orientation = (Orientations)o;
14 }
```

**Kód 8 Pomocné metódy motora**

Implementujeme niekoľko pomocných metód. `ActionList` vracia zoznam vykonateľných akcií. V tomto prípade sú tri: „left“, „right“ a „forward“ pre otočenie vľavo, vpravo a krok dopredu. Metóda `Marshal` ukladá do dátovej kolekcie agenta dáta, ktoré sa mu majú zaslať. Jediným vlastným atribútom motora je orientácia, preto ho pridáme pod názvom „heading“ (riadok 7). Metóda `Turn` otočí agenta v smere `direction`. Kladná hodnota znamená otočenie vpravo, záporná vľavo.

Ostáva implementovať poslednú metódu, ktorej úlohou je samotné rozhodovanie, ktorú akciu vykonať. Bude sa rozhodovať na základe reťazca, ktorý dostala ako argument od sveta, v ktorom sa nachádza. (Svet ho dostal od simulačného servera a ten zas priamo zo sieťového jadra.) V prípadoch otočenia vľavo a vpravo (riadky 17 – 23) je postup jednoduchý – stačí zavolať metódu `Turn` so správnym parametrom.

```
1  public override void Perform(WorldUpdateEventArgs args, string
   action, string argument1, string argument2)
2  {
3     switch (action)
4     {
5           case ("right"):
6                 Turn(1);
7                 break;
8           case ("left"):
9                 Turn(-1);
10                break;
11          case ("forward"):
12                var newX = Owner.PositionX +
13                      movementOffsets[(int)orientation][0];
14                var newY = Owner.PositionY +
```

```
15                      movementOffsets[(int)orientation][1];
16                if (!args.World.Map[newX, newY].
17                  Any(o => (o.Type == "wall") || (o.Type ==
   "agent")))
18                {
19                      args.World.MoveObject(Owner, newX, newY);
20                }
21                break;
22          }
23 }
```

**Kód 9 Logika motora**

V prípade pohybu vpred vieme podľa aktuálneho natočenia agenta určiť (pomocou vyššie zmieneného poľa ofsetov) súradnice, na ktoré sa má posunúť (riadky 12 – 15). Následne je nutné overiť, či cieľové políčko neobsahuje iného agenta alebo stenu (riadok 16). V tom prípade naň agent nemôže vstúpiť. Ak je pole voľné, príkazom `MoveObject` (r. 19) agenta presunieme.

Tým sme získali celú sadu aktuátorov a komponentov, ktorú v simulácii budeme potrebovať. V nasledujúcej sekcii predvedieme, ako ich pripojiť ku agentovi a vytvoriť svet pripravený na začatie simulácie.

## 4.2 Užívateľský agent a svet

V kapitole 2 sme popisovali riešenie modularity. Podstatnú úlohu hrá interface `ISimulationContentProvider` – poskytovateľ obsahu. Jeho úlohou je uľahčiť tvorbu simulácií vymedzením častí logiky, ktoré musia byť zabezpečené. Vytvorením triedy, ktorá ho implementuje, získame posledné chýbajúce prvky novej simulácie. Na ukladanie dát použijeme vždy formát JSON a využijeme možnosti jednoduchého parsovania súborov a deserializáciu objektov uložených v nich.

Každý objekt, implementujúci `ISimulationContentProvider`, o sebe poskytuje základné metadáta. Tie obsahujú informácie o tom, najviac koľko klientov môže byť súčasne prítomných v simulácii, a či tento poskytovateľ obsahu dokáže náhodne generovať mapy. Ako maximálny počet klientov určíme 2 a generovať náhodné mapy nebudeme. Tieto vlastnosti sa počas existencie objektu nebudú meniť. Implementujeme ich

pomocou vlastností (zadefinovaných použitým interfacom) vracajúcich konštantnú hodnotu. Novú triedu nazveme `VacuumBotContentProvider`.

```
1  public sealed class VacuumBotContentProvider :
2     ISimulationContentProvider
3  {
4     public int MaxClients { get { return 2; } }
5     public bool CanGenerateRandomMaps { get { return false; } }
6  }
```

**Kód 10 Metadáta o svete s vysávačom**

Aby sme mohli začať tvoriť mapu sveta, budeme potrebovať políčka, ktoré ako dlaždice (tiles) vyplnia priestor. Opäť použijeme pripravený mechanizmus – továreň na dlaždice (trieda `GroundTileFactory`). Stačí otvoriť dátový prúd (stream) s definíciami polí, zvyšnú časť práce urobí logika konštruktoru. Vzniknutú továreň uložíme do stavovej premennej triedy VacuumBotContentProvider.

```
1  GroundTileFactory factory = null;
2  public VacuumBotContentProvider()
3  {
4      using (Stream src = new FileStream("tiles.json",
5           FileMode.Open, FileAccess.Read))
6     {
7           factory = new GroundTileFactory(src);
8     }
9  }
```

**Kód 11 Konštrukcia továrne**

Metóda GenerateAgent na požiadanie vytvorí nového agenta. Referencia na ňu bude odoslaná vytváranému svetu (pozri Kód 13, riadok 5), ktorý ju zavolá vždy pri príchode nového agenta do sveta. Na vstupe očakáva celé číslo, ktoré sa stane jeho identifikátorom. Vracia usporiadanú trojicu hodnôt *(A, X, Y)* kde *A* je vytvorený agent (inštancia triedy `UserAgent`) so špecifikovaným identifikátorom. *X* a *Y* sú súradnice vo svete, na ktoré má byť umiestnený. Takýmto spôsobom môže poskytovateľ obsahu kontrolovať, ako sa budú agenti vo svete rozmiestňovať.

```
1  public Tuple<UserAgent, int, int> GenerateAgent(int agentId)
2  {
3      var agent = new UserAgent(string.Format("agent_{0}", agentId),
4          agentId);
5      agent.AddActuator(new VacuumBot.Vacuum(agent));
6      agent.AddActuator(new VacuumBot.Motor(agent));
7      agent.AddSensor(new VacuumBot.Camera(agent));
8      return new Tuple<UserAgent, int, int>(agent, 1, 1);
9  }
```

**Kód 12 Generovanie agentov**

Na riadku 3 je vytvorený nový agent. Na riadkoch 5 – 7 sú vytvorené a ku agentovi pridané aktuátory a senzor.

Posledným kúskom skladačky ostáva metóda `LoadMap`, ktorej úlohou je načítať mapu a vrátiť inštanciu sveta, pripravenú na štart simulácie. Opäť uvádzame zjednodušenú verziu kódu. Inštanciu triedy `Grid`, reprezentujúcu štvorcovú mriežku ktorú predstavuje mapu sveta, vytvoríme existujúcou metódou. Tá na vstupe očakáva stream obsahujúci popis mapy vo formáte JSON a továreň schopnú vytvárať políčka, používané v tomto popise. Stream je súčasťou vstupných argumentov a továreň už máme pripravenú (pozri Kód 11). Po skonštruovaní mriežky (riadok 4) môžeme vytvoriť svet[13] (riadok 5).

```
1  public World LoadMap(Stream mapInputStream, WorldSettings settings,
2     MarkovModel weatherModel)
3  {
4      Grid grid = Grid.FromStream(mapInputStream, factory);
5      return new World(grid, settings, GenerateAgent, weatherModel);
6  }
```

**Kód 13 Načítanie mapy a vytvorenie sveta**

[13] V tomto kroku môžeme svet ešte pred začatím simulácie rôzne modifikovať. Najjednoduchšou z možností je rozšírenie množstva informácií obsiahnutých vo vstupnom streame.

## 4.3 Príprava simulácie

Všetky časti logiky, potrebné na začatie simulácie, máme pripravené. Posledným krokom je vytvorenie samotného servera a spustenie sieťovej komunikácie. Tiež chceme byť informovaní, či všetko prebieha podľa očakávaní. Na to využijeme trasovacie informácie, poskytované rôznymi komponentmi servera a nastavíme textový výstup tak, aby ich vypisoval do konzoly programu.

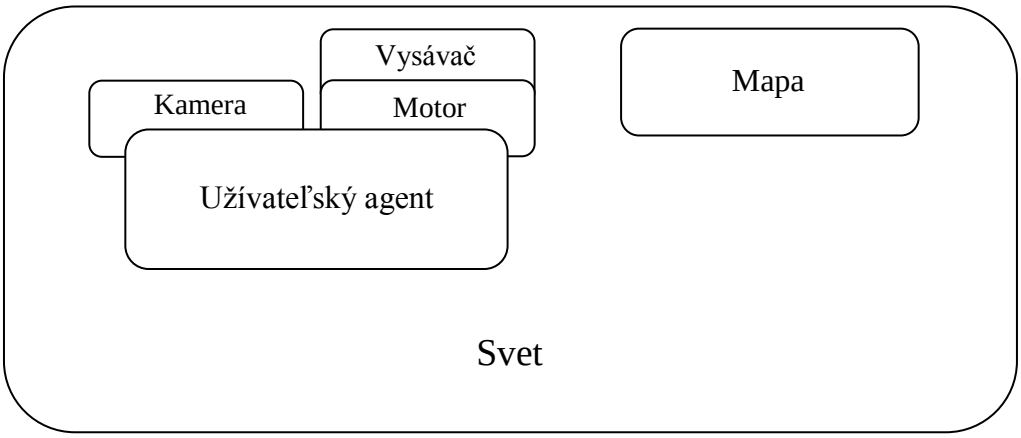
```
1  TraceListener listener = new TextWriterTraceListener(Console.Out);
2  listener.TraceOutputOptions = TraceOptions.DateTime;
3  Trace.Listeners.Add(listener);
4  SimulationCore core = new SimulationCore();
5  ISimulationContentProvider contentProvider =
6          new VacuumBotContentProvider();
7  WorldSettings settings = WorldSettings.Default;
8  settings.WeatherEnabled = false;
9  using (var map = new FileStream("vacuum_map.json",
10    FileMode.Open, FileAccess.Read))
11 {
12     world = contentProvider.LoadMap(map, settings, null);
13 }
14 core.Start(ServerSettings.Default, world);
```

**Kód 14 Nastavenie a štart servera**

Na riadkoch 1 – 6 nastavíme odchytávanie trasovacích informácií spolu s časom ich výskytu a vytvoríme nové inštancie simulačného jadra (trieda `SimulationCore`) a poskytovateľa obsahu (trieda `VacuumBotContentProvider`). Nastavenia sveta ponecháme na pôvodných hodnotách s jednou výnimkou – zakážeme simuláciu počasia, ktorú momentálne nepotrebujeme (r. 7 a 8). Potom vytvoríme svet podľa mapy nahranej zo súboru (r. 12) a odštartujeme simuláciu.


**Obrázok 22 Vizualizácia sveta pre vysávajúceho agenta**

V tejto kapitole sme demonštrovali použiteľnosť projektu pri definovaní nových simulácií. Navrhli sme jednoduchý scenár – svet s robotom, ktorého úlohou je vysávanie. Navrhli a naprogramovali sme sadu aktuátorov a senzorov, ktoré bude používať. Potom sme ukázali, ako jednoduchým spôsobom zabezpečiť všetku aplikačnú logiku, potrebnú na spustenie simulácie. Výsledný projekt možno priamo začať používať – napríklad pomocou manuálne ovládaného klienta, ktorý je tiež súčasťou tejto práce a hovoríme o ňom v sekcii 3.6 Manuálne ovládaný klient.




**Obrázok 23 Entity vo vytvorenom svete**

# 5 Záver

V tejto práci sme si stanovili za cieľ implementovať simulované prostredie pre učiacich sa agentov. Splnili sme ho tak, že sme implementovali flexibilný server schopný spúšťať rôzne užívateľom definované simulácie. Potom sme zadefinovali simuláciu ForestWorld – lesného sveta, postaveného podľa požiadaviek agenta usudzujúceho o svojom prostredí. Implementovali sme tiež jednoduchší, ale plne funkčný svet s vysávajúcim robotom a ukázali na ňom postup jeho tvorby, od navrhovania a realizácie senzorov a aktuátorov až po spustenie simulácie.

Výhodou tohto riešenia je veľká flexibilita. Modifikácia simulovaného sveta je priamočiarou záležitosťou a pridanie či odobratie komponentu je jednoduché. Nenáročné vytváranie úplne nových svetov otvára možnosti využitia TerraSimu na rôzne ciele na poli výskumu alebo výučby umelej inteligencie. Logika a štruktúra simulácie môžu byť úplne zmenené a pri tom môže stále využívať nosné prvky ako model počasia alebo denného cyklu.

Na uľahčenie použitia projektu dávame k dispozícii viaceré podporné aplikácie, menovite server s vyššie spomenutým vysávajúcim agentom a vizualizáciou, slúžiaci ako ukážka a úvod do použitia TerraSimu z programátorského hľadiska. Ďalej poskytujeme grafické rozhranie ku ForestWorld serveru, vizualizáciu s možnosťou ovládať klienta vo svete TerraSim a aplikáciu na manuálne ovládanie ľubovoľnej simulácie.

Prikladáme všetky materiály ktoré vznikli v priebehu jej tvorby. Tiež sú voľne dostupné na Internete vrátane prípadných ďalších verzií a úprav na adrese TerraSimu http://terrasim.codeplex.com/

## 5.1 Možnosti rozšírenia

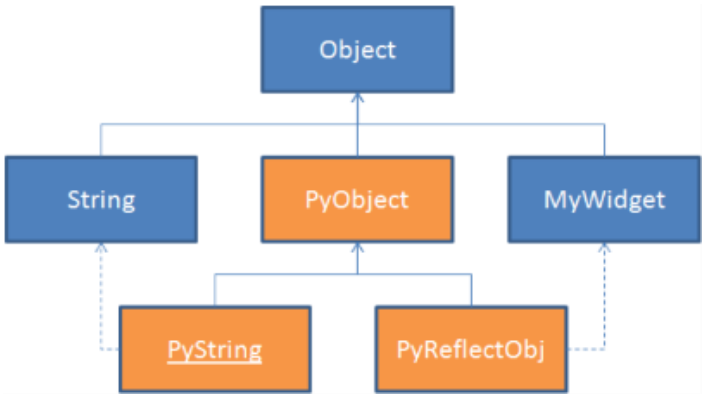
Implementovaný software spĺňa všetky vytýčené požiadavky. Napriek tomu môžu vzniknúť situácie, v ktorých by jeho úplné nasadenie mohlo vyžadovať zmeny v jeho realizácii. V tejto kapitole načrtneme niektoré takéto scenáre, spolu s možnosťami ich riešenia.[14]

[14] Možné rozšírenie bezpečnosti sme zhrnuli v kapitole 2.5 Bezpečnosť, viac sa mu venovať v tejto kapitole nebudeme.

### 5.1.1 Konfigurácia a skriptovanie

Ak by sa ukázalo byť praktické mať možnosť meniť ešte viac parametrov simulácie alebo pridávať nové komponenty do simulácie bez nutnosti rekompilácie programu, je možné využiť Python ako skriptovací jazyk. Jeho existujúca implementácia pre .NET, IronPython [15], je jednoducho prepojiteľná s existujúcou aplikáciou. V porovnaní s bežnými implementáciami iných dynamických či skriptovacích jazykov pre staticky typované jazyky má viaceré výhody.

Štandardným prístupom pri implementácii skriptovacieho jazyka je zavedenie typového systému tohto jazyka v hostiteľskom jazyku, čím sa prakticky stáva podmnožinou typového systému hostiteľského jazyka. Prostredie hosteného a hostiaceho jazyka potom spolu komunikujú a vykonávajú konverzie medzi ekvivalentnými typmi. Príkladom je Jython, implementácia jazyka Python pre Javu.



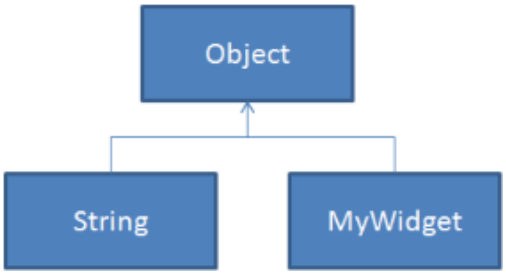

**Obrázok 24 Vzťah objektov Jythonu a Javy. Zdroj: [Chiles, a iní, 2009]**

Na obrázku vidíme tmavo objekty Javy a bledo ekvivalentné objekty Jythonu. Ak chceme poslať reťazec z prostredia Pythonu do Javy, musíme vykonať marshalling z Jython wrapperu PyString na natívnu Java triedu String (a opačne).

Výhodou použitia jazyka C# a IronPythonu (alebo iného DLR jazyka) je, okrem jednoduchej integrácie, objektový model runtime dynamických jazykov (DLR) pre .NET. Jeho konštrukcia umožňuje priamo použiť existujúcu infraštruktúru tried CLI (Common Language Infrastructure) [Chiles, a iní, 2009]. Praktickým dôsledkom je úspora na režijných nákladoch (nie je nutné konštruovať wrappery a robiť marshalling medzi rôznymi reprezentáciami funkčne a logicky ekvivalentných objektov) a zjednodušenie

[15] Aj so zdrojovými kódmi dostupný na http://ironpython.codeplex.com/

interoperability. Všetky existujúce triedy, vrátane .NET frameworku, aj tých implementovaných v tejto práci sú použiteľné bez ďalšieho medzikroku. Vďaka týmto faktorom nie je principiálny problém aplikáciu rozšíriť alebo používať jej komponenty v iných jazykoch než na implementáciu použitý C#.




**Obrázok 25 Triedy v CLI sú spoločné pre statické (C#) aj dynamické (IronPython, IronRuby) jazyky. Zdroj: [Chiles, a iní, 2009]**

### 5.1.2 Multiagentové scenáre

V prípade nasadzovania nášho servera na simuláciu rozsiahlych svetov s veľkým počtom agentov (rádovo desiatky až stovky) by mohli nastať komplikácie, neprejavujúce sa v prípadoch bežného použitia. Adaptácia na takéto scenáre by mohla vyžiadať rozsiahle zmeny vo viacerých častiach aplikácie.

Hlavným kandidátom na bottleneck je implementácia sieťového jadra. (Jeho dizajn a bližší popis je v kapitole 2.7 Komunikačný protokol a sieť.) Server počúva na určenom porte a pri príchode nového klienta vytvorí nové vlákno (thread,) ktoré ďalej zabezpečuje komunikáciu s týmto klientom. Pri nízkom počte klientov je overhead nových vláken zanedbateľný, s ich zvyšujúcim sa počtom sa môže stať relevantným faktorom. Nutným by sa mohlo stať kompletné prepísanie celého sieťového jadra.

Vhodnou technológiou na takýto účel je WebSocket, obojsmerný, plne duplexný komunikačný kanál realizovaný cez jedno TCP spojenie. Primárne je určený pre webové technológie, využitie ale má v ľubovoľnej aplikácii s klient-server architektúrou. Vo World Wide Web Consortium (W3C) prebieha štandardizácia WebSocket API, protokol štandardizuje Internet Engineering Task Force (IETF). Implementácia protokolu môže byť realizovaná knižnicou tretej strany, napríklad Alchemy Websockets [Olivine Labs, LLC, 2011]. Dostupná pod Lesser General Public License (LGPL) licenciou, táto knižnica je schopná udržovať veľké množstvá súčasných spojení a využíva thread pool na zachovanie vysokej runtime efektivity.

Takto náročná aplikácia by si tiež vyžiadala mnoho hodín dodatočného testovania za cieľom zistiť, či server je schopný poskytnúť dostatočný uptime a či je dostatočne stabilný. V oblasti pamäťových nárokov nie sú žiadne predpokladané prekážky, pretože pamäťový odtlačok jedného agenta a jeho komponentov je pomerne malý.

### 5.1.3 Dynamické správy

V sekcii 2.11 sme navrhli systém posielania správ, zabezpečujúci komunikáciu komponentov v rámci simulácie. Tento prístup je výhodný svojou flexibilitou a robustnosťou. Zároveň udržuje coupling medzi komponentmi na nízkej úrovni. Jeho implementácia je v prostredí C#.NET 4 priamočiara, pretože táto verzia frameworku priamo podporuje dynamické volania na inak staticky typovanom kóde. Naše riešenie zahŕňa dispatcher objekt, na ktorý boli smerované všetky dynamické volania a ktorý sa staral o ich spracovanie.

Tento prístup je možné rozšíriť a zovšeobecniť, aby ho natívne podporovali všetky objekty simulácie upravením rozhrania `TerraSim.Simulation.ISimulationObject` tak, aby dedilo rozhranie `IDynamicMetaObjectProvider` (namespace `System.Dynamic`). To poskytuje metódu vracajúcu objekt typu `DynamicMetaObject` (v rovnakom namespaci) zabezpečujúci runtime viazanie (binding) operácií. Ten by mohol sám zaobstarať buď vykonanie správnej akcie, alebo jej delegáciu na existujúci dispatcher.

Zjednodušilo by sa tak volanie dynamických metód odstránením nutnosti ich volania na dispatcheri a vznikla by možnosť volať ich na viacerých objektoch, na ktorých to momentálne nie je možné, napríklad aktuátory a senzory. V niektorých prípadoch by to malo za následok odstránenie nutnosti použitia zložitejších komunikačných mechanizmov na posielanie správ, popísaných v sekcii 2.11.2.

# 6 Bibliografia

**Addison, Matthew. 2003.** Meteorology - barometric pressure and rain. *All Experts.* [Online] 9 11, 2003. [Cited: 12 1, 2010.] http://en.allexperts.com/q/Meteorology-Weather-668/barometric-pressure-rain.htm.

**Apple Inc. 2010.** Objective-C Runtime Programming Guide. [Online] 2010. [Dátum: 30. 4 2011.] http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/ObjCRuntimeGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40008048-CH1-SW1.

**Balduccini, M. 2007.** Learning Action Descriptions with A-Prolog: Action Language C. *AAAI Spring Symposium.* 2007.

**Behrens, Tristan, a iní. 2010.** Multi-Agent Programming Contest Scenario Proposal (2011 Edition). *Multi-Agent Programming Contest.* [Online] 20. 11 2010. [Dátum: 31. 1 2011.] http://www.multiagentcontest.org/.

**Benc, Ladislav. 2009.** *Architektúry objektových hierarchií.* Bratislava : Univerzita Komenského, 2009.

**—. 2011.** TerraSim - A Simulated Environment. [Online] 2011. http://terrasim.codeplex.com/.

**Bryson, Joanna. 2008.** POSH Action Selection. [Online] 23. 3 2008. [Dátum: 7. 2 2011.] http://www.cs.bath.ac.uk/~jjb/web/posh.html.

**Burchall, Laurion. 2009.** Multimethods in C# 4.0 with 'dynamic'. [Online] 13. 8 2009. [Dátum: 21. 3 2011.] http://blogs.msdn.com/b/laurionb/archive/2009/08/13/multimethods-in-c-4-0-with-dynamic.aspx.

**Buro, M. 2010.** ORTS - A Free Software RTS Game Engine. [Online] 15. 7 2010. [Dátum: 30. 4 2011.] http://skatgame.net/mburo/orts/.

**Crockford, D.** Introducing JSON. [Online] [Dátum: 27. 4 2011.] http://www.json.org/.

**—. 2006.** The application/json Media Type for JavaScript Object Notation (JSON). [Online] 7 2006. [Dátum: 27. 4 2011.] http://tools.ietf.org/html/rfc4627.

**Čertický, M. 2011.** Online Action Learning Techniques for Noisy and Partially Observable Domains. 2011.

**Čertický, Michal. 2009.** *An Architecture for Universal Knowledge-based Agent.* Bratislava : Comenius University, 2009.

**Gamma, Erich, Helm, Richard a Johnson, Ralph. 1994.** *Design Patterns: Elements of Reusable Object-Oriented Software.* s.l. : Addison-Wesley Professional, 1994. ISBN 9780321700742.

**Gemrot, a iní. 2009.** Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. *Agents for Games and Simulations, LNCS 5920.* s.l. : Springer, 2009.

**Gemrot, Jakub. 2011.** Pogamut - Virtual characters made easy. [Online] 2011. [Dátum: 30. 4 2011.] http://diana.ms.mff.cuni.cz/main/tiki-index.php.

**Chiles, Bill a Turner, Alex. 2009.** *Dynamic Language Runtime.* 14. 5 2009.

**Jones, Jes. 2010.** multimethod-sharp. [Online] 2010. [Dátum: 21. 3 2011.] http://code.google.com/p/multimethod-sharp/.

**Jonker, Catholijn M. and Treur, Jan. 1998.** *Agent-based Simulation of Animal Behaviour.* Amsterdam : Centrum voor Wiskunde en Informatica, 1998. ISSN 1386-369X.

**Keene, Sonya. 1988.** *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* s.l. : Addison-Wesley, 1988. ISBN 0-201-17589-4.

**Laird, John E. a Lent, Michael van. 2000.** *Human-level AI's Killer Application: Interactive Computer Games.* s.l. : National Conference on Artificial Intelligence, 2000.

**Mourao, K., Petrick, R. P. A. a Steedman, M. 2010.** Learning action effects in partially observable domains. *ECAI 2010.* 2010.

**Newton-King, James. 2011.** Json.NET. [Online] 21. 4 2011. [Dátum: 27. 4 2011.] http://james.newtonking.com/pages/json-net.aspx.

**Novák, Peter. 2009.** Jazzbot. [Online] 24. 2 2009. [Dátum: 30. 4 2011.] http://jazzyk.sourceforge.net/projects/jazzbot.html.

**Olivine Labs, LLC. 2011.** Alchemy Websockets. [Online] 2011. [Dátum: 27. 3 2011.] http://www.olivinelabs.com/index.php/projects/71-alchemy-websockets.

**Python Software Foundation. 2011.** Python v2.7.1 documentation. [Online] 23. 3 2011. [Dátum: 23. 3 2011.] http://docs.python.org/.

**Reynolds, Craig W. 1987.** *Flocks, Herds, and Schools: A Distributed Behavioral Model.* Anaheim, California : Computer Graphics, 1987. pp. 25-34.

**Rossum, Guido van. 2005.** Five-minute Multimethods in Python. [Online] 29. 3 2005. [Dátum: 29. 4 2011.] http://www.artima.com/weblogs/viewpost.jsp?thread=101605.

**Russell, Stuart J. a Norvig, Peter. 1995.** *Artificial Intelligence: A Modern Approach.* s.l. : Prentice-Hall, Inc., 1995. ISBN 0-13-103805-2.

**Yang, Q., Wu, K. a Jiang, Y. 2007.** Learning action models from plan examples using weighted MAX-SAT. *Artificial Intelligence.* 2007.

**Zettlemoyer, L. S., Pasula, H. M. a Kaelbling, L.P. 2003.** Learning probabilistic relational planning rules. *MIT TR.* 2003.

# 7 Prílohy

## 7.1 Markovov model počasia

Model počasia sme opísali v kapitole 2.4. Naším cieľom bolo vytvoriť model zmien v atmosférickom tlaku, ktorý bude tvoriť dostatočne dlhé obdobia bezo zmeny tlaku, ale tiež obdobia poklesu alebo nárastu tlaku dosť dlhé na to, aby došlo ku zmene počasia. Preto šanca zostania v už existujúcom stave je vysoká, s malými možnosťami vychýlenia. Šanca stabilizácie počas klesania (alebo stúpania) je tiež pomerne veľká, takmer 50%. Konkrétne hodnoty prechodných pravdepodobností sme doladili na základe empirických testov. Uvádzame ich hodnoty pre stúpanie (*rise*), klesanie (*fall*) a zachovanie (*stay*) hodnoty tlaku v ich JSON zápise. Pretože ide o percentuálne pravdepodobnosti, každý riadok matice sa musí sčítať na 1.

```
[
  {
   "TransitionProbabilities": [0.35, 0.45, 0.2],
   "Name": "RISE"
  },
  {
   "TransitionProbabilities": [0.1, 0.8, 0.1],
   "Name": "STAY"
  },
  {
   "TransitionProbabilities": [0.2, 0.45, 0.35],
   "Name": "FALL"
  },
]
```

## 7.2 Priebeh sieťovej komunikácie

Pre ilustráciu v chronologickom poradí uvedieme správy, aké agent dostáva keď sa pripája na server. Hneď po nadviazaní a prijatí spojenia (teda server nie je plný a neodmietol ho), klient dostane správu s identifikátorom, ktorý mu bol pridelený.

```
1  your_id=1
```

Následne server odošle správu obsahujúcu zoznam akcií, ktoré môže agent vykonať.

```
2  ["gun_stats","shoot","reload","motor_stats","left","right","forwar
   d","backward","watertank_stats","water","refill","analyze","take",
   "put_on_position","put_in_item"]
```

Potom nasleduje bežná výmena správ; agent dostane vnemy zo senzorov. Kvôli obsiahlosti sme ich text skrátili.

```
3  {
4   "has_attribute":
5     [["agent_1","position_x","1"],["agent_1","position_y","1"],
6     ["agent_1","type","user_agent"],["agent_1","bullets_left","3"],
7  (…)
8  "performs_action":[]
9  }
```

## 7.3 Akcie vo ForestWorld

Pripájame prehľad akcií, ktoré dokážu vykonať aktuátory agenta vo ForestWorld. Pri každej uvádzame jej meno a parametre, ak nejaké potrebuje. Akcie:

- forward, backward, left a right hýbu a otáčajú agenta
- shoot strieľa. Ako prvý argument očakáva meno cieľovej entity
- reload naplní zásobník zbrane muníciou
- water poleje vodou objekt, ktorý dostane ako prvý argument
- refill doplní vodu do nádrže agenta, ak ako argument dostane názov dostatočne blízkeho políčka s vodou
- analyze analyzuje rastlinu určenú menom (v prípade že sa nachádza v dosahu agenta)
- take zodvihne predmet, určený menom
- put_on_position položí držaný predmet na políčko určené súradnicami x a y.
- put_in_item položí držaný predmet do iného predmetu určeného menom (napr. do kŕmidla)