Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Tímový projekt
Robocup
Dokumentácia robotického hráča Jim
Inžinierske dielo

Bc. Peter Filípek
Bc. Metod Rybár
Bc. Michal Segeč
Bc. Juraj Šimek
Bc. Martin Vrabec
Bc. Miroslav Wolf

Tím č. 8:	Infinity
Vedúci projektu:	Ing. Ivan Kapustík
Predmet:	Tímový projekt I
Ročník:	2014/2015
Akademický rok:	1.
Mailový kontakt:	team8@centrum.sk

Log zmien v dokumentácií

Dátum zmeny	Názov sekcie	Autor	Verzia
12. október 2014	Ako čítať a písať túto dokumentáciu	Metod Rybár	0.1
12. október 2014	Popis základných častí	Metod Rybár	0.1
12. október 2014	Monitorovanie zmien v dokumentácií	Metod Rybár	0.1
15. október 2014	Úvod 1	Metod Rybár	0.1
15. október 2014	Globálne ciele pre ZS/LS	Metod Rybár	0.1
15. október 2014	Analýza 4	Metod Rybár	0.1
15. október 2014	Návrh	Metod Rybár	0.1
15. október 2014	Implementácia	Metod Rybár	0.1
15. október 2014	Testovanie	Metod Rybár	0.1
15. október 2014	Príloha A: Inštalačná príručka 31	Metod Rybár	0.1
15. október 2014	Príloha B: Štandard pre kód 31	Metod Rybár	0.1
19. október 2014	Globálne ciele pre ZS/LS	Metod Rybár	0.2
19. október 2014	Globálne ciele pre zimný semester 2	Metod Rybár	0.2
19. október 2014	Príloha B: Štandard pre kód 31	Juraj Šimek	0.3
19. október 2014	Titulná strana	Metod Rybár	0.3
23. október 2014	Analýza hráčov 4	Metod Rybár	0.4
23. október 2014	Tomas Boleček 4.1	Martin Vrabec	0.5
23. október 2014	Jaroslav Grega 4.2	Martin Vrabec	0.5
23. október 2014	Ján Hudec 4.3	Metod Rybár	0.5
23. október 2014	Pavol Mešťaník 4.4	Peter Filípek	0.5
23. október 2014	Peter Paššák 4.5	Metod Rybár	0.5

23. október 2014	Martin Košický 4.6	Michal Segeč	0.5
23. október 2014	NaoTH 4.7	Peter Filípek	0.5
23. október 2014	NUBots 4.8	Miroslav Wolf	0.5
23. október 2014	rUNSWift 4.9	Miroslav Wolf	0.5
23. október 2014	Apollo3D 4.10	Michal Segeč	0.5
23. október 2014	Austin Villa 4.11	Michal Segeč	0.5
23. október 2014	RoboCanes 4.12	Michal Segeč	0.5
23. október 2014	BahiaRT 4.13	Juraj Šimek	0.5
27. október 2014	Analýza tímu BahiaRT 4.13	Juraj Šimek	0.5
1. november 2014	Úprava štandardu pre písanie kódu 31	Juraj Šimek	0.5
2. november 2014	Odstránenie prázdnych kapitol	Metod Rybár	0.5
3. november 2014	Oddelenie návodu dokumentácie	Metod Rybár	0.5
4. november 2014	Úprava logovania 5	Juraj Šimek	0.6
4. november 2014	Logger 5.1	Juraj Šimek	0.6
4. november 2014	Príklady logovacích typov 5.1	Juraj Šimek	0.6
4. november 2014	Peter Paššák 4.5	Metod Rybár	0.7
4. november 2014	Ján Hudec 4.3	Metod Rybár	0.7
16. november 2014	Pridanie citácií	Metod Rybár	0.8
16. november 2014	Odstránenie zakomentovaných častí kódu v projekte Jim 6	Miroslav Wolf	0.8
17. november 2014	Odstránenie balíku sk.fiit.jim.garbage 7	Juraj Šimek	0.9
17. november 2014	Úprava logovania pomocou nového loggera 5.2	Juraj Šimek	0.8

17. november 2014	Globálne ciele pre ZS/LS	Metod Rybár	1.0
17. november 2014	Globálne ciele pre zimný semester 2	Metod Rybár	1.0
18. november 2014	NUBots 4.8	Miroslav Wolf	1.1
18. november 2014	Tomas Boleček 4.1	Martin Vrabec	1.1
19. november 2014	Úprava pôvodného zdrojového kódu podľa konvencií písania zdrojového kódu 8	Juraj Šimek	1.2
19. november 2014	Úprava konvencií logovania 31	Juraj Šimek	1.2
19. november 2014	RoboCanes 4.12	Michal Segeč	1.2
19. november 2014	Úvod 1	Metod Rybár	1.3
3. december 2014	Odstránenie nepoužívaných highskillov 9	Juraj Šimek	1.5
9. december 2014	Úprava konfigurácie nastavení agenta 10	Juraj Šimek	1.5
10. december 2014	Sprístupnenie MediaWiki 11	Peter Filípek	1.6
10. december 2014	Oddelenie testov od zdrojových kódov 12	Peter Filípek	1.6
10. december 2014	Odstraňovanie warning hlásení v projekte Jim 13	Miroslav Wolf	1.6
10. december 2014	Implementácia Zero moment point 14	Metod Rybár	1.6
10. december 2014	Pridanie plánov pre semstre z dokumentácie k riadeniu	Metod Rybár	1.7
28. február 2015	Zámena triedy Vector3 za Vector3D 15	Juraj Šimek	1.8
8. marec 2015	Pridávanie hráčov do testovacieho frameworku 16	Juraj Šimek	1.9

12. marec 2015	Refaktorovanie projektu RoboCupLibrary 17	Juraj Šimek	2.0
19. marec 2015	Grafické rozhranie pre kontrolu loggera 18	Juraj Šimek	2.1
4. apríl 2015	Ukladanie konfigurácie GUI loggera 19.1	Juraj Šimek	2.2
7. apríl 2015	Príprava taktík pre RoboCup turnaj 20.1	Juraj Šimek	2.3
11. apríl 2015	Príprava taktík pre RoboCup turnaj 20.1: doplnenie dokumentácie k taktike Turn180	Juraj Šimek	2.3
12. apríl 2015	Analýza chybných JUnit testov 21	Peter Filípek	2.4
12. apríl 2015	Detekcia pádu robota s využitím akcelerometra 22	Peter Filípek	2.4
12. apríl 2015	Implementácia pohybu lopty v TestFrameworku 23	Miroslav Wolf	2.4
12. apríl 2015	Analýza TestFrameworku 24	Miroslav Wolf	2.4
12. apríl 2015	Detekcia pádu pomocou force receptorov na nohách agenta 25	Miroslav Wolf	2.5
12. apríl 2015	Analýza planera HighSkills 26	Metod Rybár	2.5
12. apríl 2015	Tvorba HighSkills a zakomponovanie Zero moment point 27	Metod Rybár	2.5
18. apríl 2015	Otáčanie hráča 28	Juraj Šimek	2.6

18. apríl 2015	Príprava taktík pre RoboCup turnaj 20.2: doplnenie dokumentácie k taktike Turn180	Juraj Šimek	2.6
23. apríl 2015	Nefunkčnosť anotovania hráčov 29	Martin Vrabec	2.7
23. apríl 2015	Príloha A: Inštalačná príručka 31	Miroslav Wolf	2.7
24. apríl 2015	Príloha A: Inštalačná príručka 31 : Pridanie časti pre Linux	Metod Rybár	2.8
24. apríl 2015	Pridanie vynúteného prerušenia HighSkills 30	Metod Rybár	2.9
24. apríl 2015	Tvorba HighSkills a zakomponovanie Zero moment point 27 : Pridanie popisu isOnGround	Metod Rybár	2.9
26. apríl 2015	Príprava taktík pre RoboCup turnaj 20.2: doplnenie dokumentácie k taktikám KickDistance, StabilityWalkTactic, FastWalkTactic	Juraj Šimek	3.0
2. máj 2015	Hlbšia analýza anotácií v TestFrameworku 24.1	Martin Vrabec	3.1
2. máj 2015	Návrh na ďalšiu prácu na hráčovi Jim 31	Miroslav Wolf	3.1
16. máj 2015	Návrh na ďalšiu prácu na hráčovi Jim 31	Miroslav Wolf	3.2

Obsah

1 Úvod

Posledná úprava	Metod Rybár
Platné od	19. november 2014
Poznámky	

RoboCup je medzinárodná súťaž autonómnych robotov, ktorá je určená na podporu výskumu a výučby robotiky a umelej inteligencie.

Každý tím sa v rámci obmedzení, určených pravidlami hry futbal a špecifikami simulačného prostredia, snaží vytvoriť čo najlepšieho hráča. Mužstvo, vytvorené z takýchto hráčov, by malo vyhrať nad mužstvom súpera. O súťaži a doterajšej činnosti je možné dozvedieť sa na stránke STU turnaj v simulovanom robotickom futbale (www.fiit.stuba.sk/robocup).

Simulácia futbalu pôvodne prebiehala iba v dvoch rozmeroch. Pre zvýšenie reálnosti simulácie bolo vytvorené 3D simulačné prostredie, ktoré rozširuje možnosti hry. 3D simulačné prostredie sa pomerne výrazne líši od doposiaľ používaného 2D prostredia, a to jednak spôsobom simulácie, ale hlavne možnosťami ktoré poskytuje hráčom. Na rozdiel od 2D simulácie ide o simuláciu jednotlivých pohybov humanoidného robota.

Na fakulte informatiky a informačných technológií sa vývinu robota venuje tím zložený z jej študentov v rámci predmetu Tímový projekt už od roku 2000. Do roku 2006 vývoj robota prebihal v 2D prostredí, od roku 2006 prebieha vývoj v 3D prostredí.

Hlavným cieľom projektu je vylepšiť doterajšieho hráča pre 3D simuláciu, ktorý dokáže plnohodnotne využívať možnosti poskytované simulačným prostredím.

Tneto dokument slúuži ako dokumentácia k technickým častiam robota Jim. Obsahuje analýzu iných robotických hráčov a návrhy na zlepšenie hráča Jim. Zároveň obsahuje globálne ciele pre jednotlivé semestre.

V tejto dokumentácií sú podrobnejšie rozpísané zmeny vykonané na hráčovi a jeho aktuálny stav.

2 Plán pre zimný semester

Posledná úprava	Miroslav Wolf
Platné od	9. december 2014
Poznámky	

V tomto semestri sa chceme venovať najmä úprave kódov do konvencií a zlepšeniu prehľadnosti kódu. Keďže na projekte pracovalo veľa tímov, ktoré sa nedržali jednotnej konvencie písania kódu, kód sa tak stal neprehľadným. Mnoho funkcií ani nie je opísaných, preto často nie je jasné, na čo daná časť kódu slúži a či je vôbec potrebná. V projekte sa tiež nachádza mnoho duplicít, warningov a nepoužívaných častí. Preto by sme sa chceli v tomto prejsť celý projekt a zamerať sa najmä na zrefaktorovanie, odstránenie zbytočných častí, okomentovanie jednotlivých funkcií a ďalšie úlohy. V rámci týchto úprav sa aj s projektom bližšie zoznámime a lepšie sa v ňom zorientujeme.

Úlohy zapísané do backlogu:

- Zistiť čo je so serverom pre stránku tímu
- Dokončiť konvenciu pre dokumentáciu a vytvoriť šablónu
- Upraviť a zjednotiť logovanie
- Rozbehať Jiru - vyriešiť problém s prístupom
- Zoznámiť sa s gitom a importovať si projekt
- Vyriešiť problém s vyberaním pohybu
- Naštudovať a vytvoriť finalizáciu pohybov
- Otestovať pohyb hlavy
- Identifikácia a vymazanie nepotrebných častí kódu
- Preštudovať a poznačiť si ďalšie diplomové prace a zahraničné tímy
- Zlepšiť rozhodovanie hráča
- Upraviť kód podľa konvencie

- Správa wiki
- Vytvorenie dokumentácie
- Vylepšovanie pohybov
- Otestovanie hráča v predklone
- Zakomponovanie obchádzania
- Vytvoriť stabilizované pohyby
- Vytvoriť dobehnutie pre rýchly pohyb
- Oboznámiť sa s testovacím frameworkom
- Vylepšiť testovací framework
- Popresúvať funkčnosť do knižnice
- Odstrániť warningy
- Prerobenie vectora3 do vectora3d
- Zaviesť definíciu kedy je úloha hotová
- Codereview
- Doplnenie testov/unitestov
- Pripraviť si otázky pre gitmanov
- Presunúť testy do jedného balíčka alebo do iného projektu
- Preštudovať aj naše tímy
- Analyzovať projekt
- Rozbehať roboviz
- Predpovedanie polôh/pozície - upraviť alebo zakomponovať
- Upraviť zisťovanie pádu hráča
- Naštudovať správy - manuál rfc serveru

- Upraviť výpočet Suitability
- Preskúmať funkčnosť anotácií, pripadne ich odstrániť
- Identifikovať a odstrániť duplicitné časti kódu
- Prerobiť TestFramework do Javy
- Preskúmať funkčnosť brankára

2.1 Zhrnutie zimného semestra

Posledná úprava	Miroslav Wolf
Platné od	9. december 2014
Poznámky	

Zimný semester pozostáva z 5 šprintov. Počas týchto šprintov sme sa učili najmä pracovať ako tím a vytvárali sme rôzne metodiky pre riadenie tímu. Náš tím sa riadil metodikou SCRUM a svoj postup v práci zaznamenával do nástroja Jira. Okrem tohto nástroja sme počas semestra začali využívať aj mnohé ďalšie ako napríklad CodeReview, pre značkovanie kódu, Bamboo pre automatické testovanie a Bitbucket, na vytváranie branchov a commitovanie zmien. Pre jednotlivé nástroje boli vytvorené aj metodiky pre ich používanie.

Na začiatku semestra bol pre zimný semester vytvorený plán. Rozhodli sme sa, že sa zameriame najmä na refaktoring - konkrétne úpravu kódu a komentárov do konvencie, odstránenie warning hlásení, presunutie testov do jedného balíčka, pridanie nových komentárov a celkové sprehľadnenie kódu. Pre nájdenie chýb a nedostatkov v kóde sme si nainštalovali plug-in pre eclipse - Unnecessary Code Detector. Zbytočný kód, nevyužívané alebo duplicitné triedy a funkcie tak boli vymazané, prípadne vhodne okomentované a ponechané. Projekt obsahoval aj veľa zakomentovaného kódu. Tento kód sme počas semestra zanalyzovali a otestovali a následne sme väčšinu z tohto kódu vymazali. Okrem týchto úprav sa nám počas semestra podarilo zapracovať aj nové časti. Bol vytvorený nový logger, keďže projekt doteraz obsahoval dva, náš nový logger spojil ich funkciu a staré loggery tak boli nahradené a zjednotené. Ďalej sa nám podarilo vytvoriť ZMP - Zero moment point, ktorý sme implementovali podľa diplomovej práce jedného zo študentov. Vďaka

ZMP sa nám podarilo vytvoriť stabilnejší pohyb hráča, no zatiaľ sú s ním drobné problémy, ktoré sa pokúsime odstrániť v ďalšom semestri.

Počas semestra sme v tíme narazili aj na niekoľko problémov a to najmä na začiatku semestra, kedy sme sa ešte len učili s jednotlivými nástrojmi pracovať. Mali sme problémy s nástrojom Jira, kde sa nám zle vykresľoval burndown-chart, ako aj rôzne iné drobné problémy s týmto nástrojom. Časom sme sa s ním však naučili pracovať. Ďalším problémom bolo samotné vytváranie úloh, kde sme v niektorých prípadoch zle odhadli časy, prípadne sme úlohu málo dekomponovali na menšie časti. Zlé odhady vznikali aj preto, že sme si často nie úplne úlohy premysleli a často sme zabudli, že k úlohám majú byť vytvorené aj dokumentácie a tak sme ich občas museli robiť dodatočne. Okrem toho niekedy úlohy neboli vhodne opísané a pomenované. Každý člen tímu si tiež mal do nástroja značiť svoju vykonanú prácu, čo tiež zrejme v prvých šprintoch nebolo dodržiavané, no neskôr sme si už prácu riadne značili. Občas vznikli aj drobné problémy v komunikácii, no momentálne sa tím prostredníctvom nástroja Jiry a prípadne sociálnych sietí, snaží komunikovať čo najviac, aby každý člen tímu mal prehľad, čo práve ktorý člen tímu robí. V prvých šprintoch sme tiež mali problém v manažmente verzií, kde nám pri spájaní vznikali rôzne problémy, no to bolo spôsobené tým, že každý robil úpravy vo vlastnej vetve. V ďalších šprintoch sme tento problém už odstránili a v rámci možností sme zmeny robili v jednej vetve.

S ďalšími vážnejšími problémami sme sa počas semestra nestretli a mnohé zo spomenutých problémov sa nám počas semestra podarilo odstrániť.

3 Plán pre letný semester

Posledná úprava	Miroslav Wolf
Platné od	9. december 2014
Poznámky	

V letnom semestri najskôr chceme dokončiť rozpracované úlohy, ako napríklad implementovanie ZMP - kde sa nám podarilo vytvoriť stabilnejšiu chôdzu, no ešte nie je úplne zapracovaná do projektu, keďže po páde robota pri tejto chôdzi sa robot nevie postaviť. V zimnom semestri sme sa venovali najmä úpravou kódu a refaktoringu projektu Jim a preto by sme sa v letnom semestri chceli posunúť a prejsť k úpravám projektu TestFramework. Tento projekt obsahuje mnoho chýb a preto by bolo vhodné odstrániť problémy a niektoré časti možno aj prerobiť. Okrem toho by bolo dobré projekt rovnako prejsť a upraviť do konvencií, odstrániť warningy a ďalšie úpravy, podobne ako boli v zimnom semestri vykonané v projekte Jim.

S tímom sme teda predbežne navrhli niektoré úlohy, ktoré by bolo vhodné vrámci letného semestra vypracovať. Medzi tieto úlohy patrí:

- Rozbehanie turnaja v TestFrameworku
- Opravenie problémov s TestFrameworkom
- Vyhodenie nepotrebných častí TestFrameworku
- Pridanie alebo integrovanie novej časti - prípadne prerobiť časti z ruby.
- Vytvorenie rôznych situácií pre jedného alebo viacerých hráčov
- Vytvorenie simulačného prostredia pre konkrétne situácie
- Vylepšiť určovanie pozícií

Okrem týchto úloh nám v backlogu ostalo aj zopár úloh zo zimného semestra a tak by bolo vhodné vybrať aj niektoré z týchto úloh. Medzi tieto úlohy patria:

- Vyriešenie problému s vyberaním pohybu
- Otestovanie pohybu hlavy

- Zlepšenie rozhodovania hráča
- Vytvorenie dobehnutia pre rýchly pohyb
- Prerobenie vector3 na vector3d
- Doplnenie unitestov
- Preskúmanie funkčnosti anotácií
- Preskúmanie duplicitných častí kódu
- a ďalšie...

Okrem spomenutých úloh budeme v letnom semestri aktualizovať a dopĺňať informácie na stránku wiki projektu, ktorú sa nám na konci zimného semestra podarilo rozbehať. V letnom semestri sa tiež chceme zúčastniť Študentskej vedeckej konferencie IIT.SRC 2015.

4 Analýza hráčov a herných postupov v RoboCupe

Posledná úprava	Metod Rybár
Platné od	23. október 2014
Poznámky	Zmena nadpisu

4.1 Tomáš Boleček - Rozšírenie hráča Jim o strategickú vrstvu

Posledná úprava	Martin Vrabec
Platné od	18. november 2014
Poznámky	

Rozšíreniu fakultného hráča Jim sa venoval vo svojej diplomovej práci Tomáš Boleček. [1]

DP je zameraná na rozšírenie existujúceho hráča Jima o strategickú vrstvu. Daný model by mal obsahovať nasledujúce vrstvy: stratégie, taktiky, formácie, sub-taktiky, role, overenie Navrhnutý model musí umožňovať vytváranie správania pre celý tím, ale zároveň aj možnosť vytvárania správania pre konkrétnych jednotlivcov, prípadne pre skupinu jednotlivcov na rovnakej roli alebo s rovnakou charakteristikou.

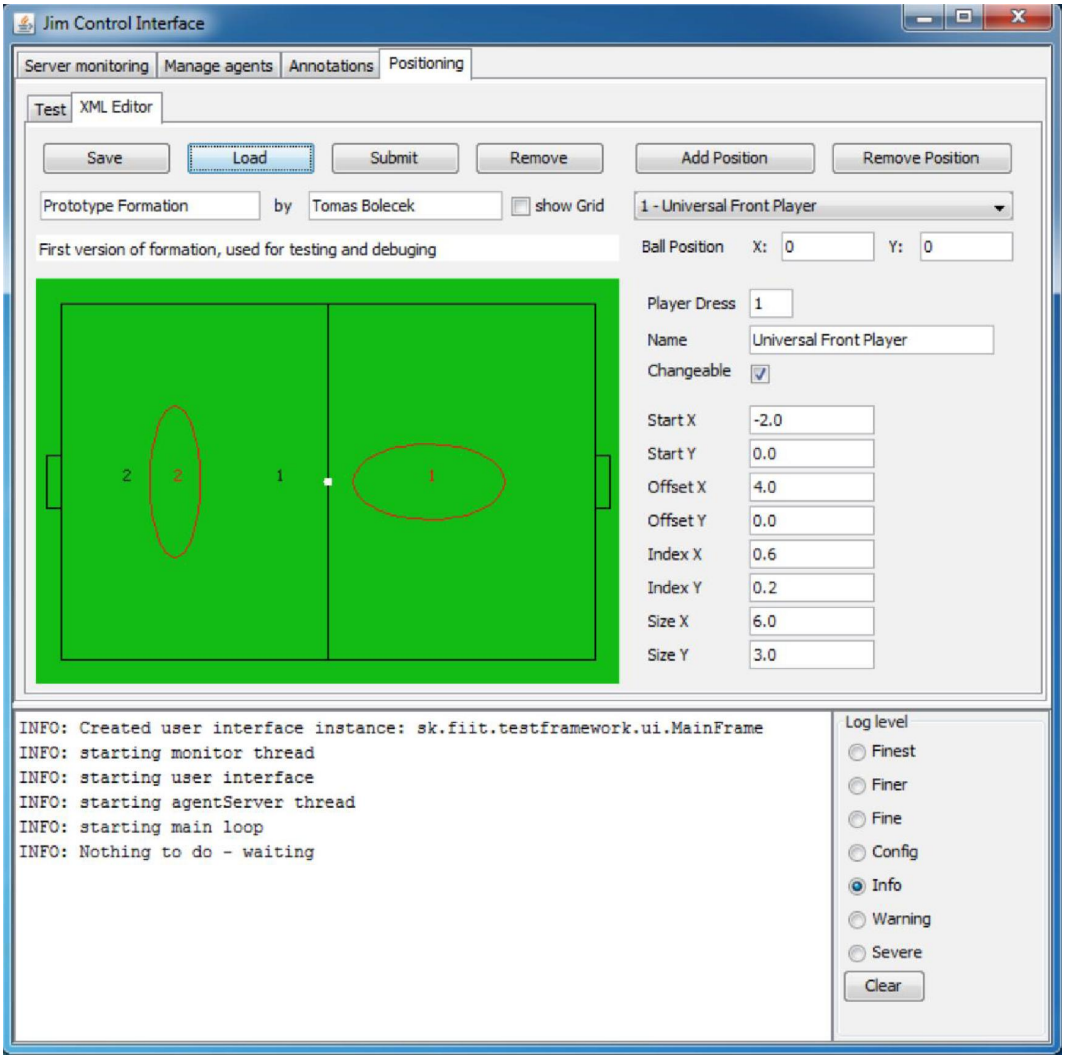
Na vytváranie formácií bolo vytvorené grafické používateľské rozhranie podľa návrhu zobrazenom na obrázku 1.

XML obsahuje dve hlavné časti označené ako <formations> a <data>

Novo zavedené dátové štruktúry

sk.fiit.jim.ai.formation.FormationPosition

Daná dátová štruktúra zapisuje konkrétne pozície pre jednotlivých hráčov. Obsahuje informáciu o polohe, akú má mať hráč v závislosti od polohy lopty, počiatočnú polohu hráča, informáciu o pridelenej roli hráčovi, možnosť hráča dynamického prideľovania a zmeny rolí a informáciu o tom, ktorý hráč pravé hra s loptou (onBall).



Obr. 1: Návrh používateľského rozhrania

sk.fiit.jim.ai.formation.formationData
Obsahuje dáta o konkrétnej formácii. Obsahuje množinu dát polôh hráčov vo formácií, informácie o počte hráčov vo formácii a informácie o autorovi a popis formácie.

sk.fiit.jim.ai.formation.positionPlayerPair
Dátová štruktúra opisujúca pozíciu v danom momente pre konkrétneho hráča

s priradenou polohou vo formácii. Dátová štruktúra je vytvorená zvlášť pre každého hráča vo formácii. Množina týchto dátových štruktúr nám reprezentuje aktuálnu formáciu v hre.

Použité rozhrania

sk.fiit.testframework.ui.interfaces.gameViewerProviderImpl
Dané rozhranie je nami navrhnuté nové rozhranie pre triedu customizableGameView na zobrazovanie stavu hry z rôznych zdrojov. Existujú dve triedy, ktoré implementujú toto rozhranie - onClickProvider a gameStateProvider.

OnClickProvider je využívaný v záložke testovacieho frameworku XML Editor, pričom dané rozhranie nám umožňuje nastaviť pozíciu objektov pomocou kurzora myši alebo pomocou numerických textových poli opisujúcich túto pozíciu. GameStateProvider umožňuje získavanie herného stavu sveta od servera.

Lokalizácia hráča

Lokalizácia bola aktualizovaná na základe už existujúceho hráča. Riešenie bolo implementované v rámci existujúcej triedy AgentPositionCalculator. Spočívalo v ohodnotení nového videnia na základe počtu orientačných bodov, ktoré vidí. Okrem hodnotenia podľa orientačných bodov je nové videnie hodnotené aj podľa odchýlky od posledného videnia.

Komunikácia

Riešenie komunikácie je implementované v triede SpeechManager. Pred odoslaním správy je vytiahnutá aktuálna pozícia lopty a hráča a tieto informácie sú odoslané všetkým hráčom.

Model sveta

Model sveta si vytvára každý hráč samostatne. Bol implementovaný v triede SpeechManager. Je reprezentovaný vektormi pozícií hráčov a lopty v dvoch hash mapách. Každá informácia o polohe lopty je hodnotená na základe vzdialenosti od lopty (podľa váhy). Overenie bolo vykonané tak, že na základe týchto dát bol vypočítaný medián, priemer, minimálna a maximálna

hodnota.

Dynamické prideľovanie rolí

Dynamické prideľovanie rolí bolo implementované v triede PositionPlaceCalculator, ktorá v pravidelných intervaloch prerátava rolu hráča a výsledok ukladá v rámci agentInfo. Prepočet rolí nastáva každé 3 sekundy. Pri prvom spustení sú role priradené podľa zadania z XML. Následne počas hry a vypočítaného stavu hráči menia role iba ak cena tejto výmeny je nižšia ako aktuálna rola hráča. Toto prideľovanie funguje tak, že pre každú dvojicu hráča a rolí sa vypočíta vzdialenosť medzi hráčovou pozíciou a určenou pozíciou vo formácii. Rola bude vybraná na základe najlepšieho výsledku.

Testovanie komunikácie

Testovanie komunikácie bolo implementované v FileWritter. Testovanie prebehlo pomocou logov vytvorených jednotlivými hráčmi, kde bolo vykonaných 1200 cyklov v rámci 2 hier s úspešnosťou 99.97%.

Zhodnotenie

V porovnaní s riešením sa dá vychádzať len z videnia sveta, keďže ostatná funkcia bola implementovaná prvý krát. Pre dynamické prideľovanie rolí pri viac ako 50 testoch, 5 rôznych formáciách a počte hráčov 2 až 5 sa v každej situácii podarilo správne priradiť rolu jednotlivým hráčom. Najväčší prínos je v navrhnutí strategickej vrstvy.

4.2 Jaroslav Grega - Zlepšenie rýchlosti a stability chôdze pomocou genetického algoritmu

Posledná úprava	Martin Vrabec
Platné od	23. október 2014
Poznámky	

V práci sa podarilo zvýšiť rýchlosť chôdze na 0.46 m/s s priem. stabilitou 1 pád na 10 m a odklonom 6.86 m. [5]

Na optimalizáciu pohybov hráča využil genetický algoritmus, kde ako vstup využije hodnoty z už existujúcich pohybov - za gén určoval polohu kĺbov agenta, fitnes funkciu tvoril čas prejdenia zvolenej dĺžky ihriska a pri kopoch dĺžka a presnosť kopu. Výstupom fitnes funkcie je čas a penalizácia za každý pád.Najlepší jedinec postúpi do fázy kríženia. Kríženie je realizované každý s každým.

Ďalšou fázou GA algoritmu je selekcia : v starej populácii sa nájde jedinec s najlepšou fitnes, ten sa presunie k jedincom, ktorí vznikli krížením a mutáciou. Z tejto skupiny sa vyberú jedince, ktoré budú tvoriť novu populáciu. Beh sa bude vykonávať podľa počtu zvolených opakovaní.

Stabilizácia agenta metódou ZMP

Autor vybral metódu ZMP pre jej vysokú popularitu u zahraničných tímov a jeho kvalitný výsledok. Hlavnou myšlienkov ZMP je umiestniť bod tak, aby bol v konvexnej polohe chodidiel. Ak ZMP splna podmienku, pokračuje sa vo vykonávaní pohybu.

Ak nie je splnená (agent by pravdepodobne spadol -> začne sa vykonávať stabilizácia pohybu [nastavenie kĺbov tak, aby sa robot dostal do stabilizovanej polohy]).

Kroky stabilizácie

- zo servera sa získajú informácie o aktuálnom natočení kĺbov agenta a o nasledujúcom natočení kĺbov;
- pomocou vzorca pre ZMP vypočítam pozíciu ZMP nasledujúceho natočenia kĺbov;
- ak je mimo konvexného obalu chodidiel, upravím natočenie kĺbov rúk do takej polohy, aby ZMP bolo v konvexnom obale chodidiel, na to vy-

užijem genetické programovanie, pomocou ktorého budem hľadať funkciu pre stabilizáciu agenta;

- nové nasledujúce natočenie kĺbov odošle na server

Do aplikácie pridal triedu BodyPart ktorá reprezentuje jednotlivé časti agenta a potrebné informácie o častiach ako sú váha, kĺb, na ktorý je časť napojená, rodič častí a geometrický tvar častí.

Do triedy AgentModel pridal metódy pre výpočet CoM a následne ZMP. Na vytvorenie týchto rovníc využil genetické programovanie.

Pri testovaní chôdze zistil, že rýchlosť chôdze je 0.2 m/s namiesto 0.4m/s a dôležitou časťou výberu algoritmu je výber jedincov (selekcia). Dôležitým faktorom pri mutácii pohybu je rozsah, v ktorom sa mutujú gény pohybu - Pri vytváraní pohybov s vysokým percentom mutácie génu sa stavalo, že veľa pohybov bolo nefunkčných a agent sa dostával do stavu, kde sa nevedel pohnúť (prirovnanie k 43 ľuďom, akoby dostal epileptický záchvat). Najlepšie pohyby sa vytvárali, ak bola nastavená maximálna mutácia génov do 20

4.3 Ján Hudec - Motorika hráča simulovaného robotického futbalu

Posledná úprava	Metod Rybár
Platné od	4. november 2014
Poznámky	

V práci [6] sa autor venuje zrýchleniu chôdze robota Nao pomocou princípu Zero Moment Point. Vo výslednom riešení sa chôdza Nao robota zrýchlila na hodnotu približne 27 cm/s (0,972km/h). Výsledný produkt JimJet má zabudovanú stabilizáciu pomocou rúk a nôh, čo dopomohlo k zvýšeniu rýchlosti pôvodného hráča o približne 43%.

Zero Moment Point sa zameriava hlavne na určenie ťažiska robota a berie do úvahy aj rôzne sily ktoré naňho pôsobia.

To je dôležité pri samotnom vykonávaní pohybu. Pri pohybe končatín napríklad u ľudí sa činnosť svalu začne vykonávať až potom, ako sa splnia stanovené podmienky. Preto aj u robota treba každý pohyb brať ako komplexnú spojitú množinu.

Z pôvodných 15 centimetrov za sekundu sa podarilo chôdzu zrýchliť na 27 centimetrov za sekundu. Bolo prevedené zakomponovanie nových informácií o modeli ako hmotnosť, kotvenie častí tela a iné.

Bol popísaný problém so senzormi ktoré poskytujú napríklad pri pohybe len informáciu o zmene od posledného stavu bez referencie na globálne súradnice.

Bolo zavedené využitie ForceResistance perceptora ktorý neobsahuje umelo pridaný šum, ktorý zachytáva silu pôsobiacu na robota a nachádza sa v chodidlách.

Autor predpokladá možné zlepšenie vylepšením stabilizačných funkcií či už za presnejšie alebo univerzálnejšie, presnejších smerovacích funkcií alebo pokusom o simulovanie ľudskej chôdze.

Z tejto práce by sa dala využiť najmä práca na Zero Moment Point. To umožňuje u robotov napríklad tvorbu dynamických pohybov. Netreba pri ňom totiž mať ťažisko v takzvanej zóne stability, pretože Zero Moment Point berie do úvahy aj gravitačnú silu, odstredivé silu a iné. Na základe týchto informácií si dokáže vytvoriť mapu kam má stúpiť v ďalšom kroku tak, aby zostal stabilný.

Táto metóda bola v práci pridané do triedy *AgentModel*

```
private void updateZeroMomentPoint() {
        double x, y, z = 0;
        Vector3D center = centerOfMass;
        x = -1 * (center.getX() - (center.getZ()
          * lastAccelerometer.getX())/
                  ( lastMomentum.getZ() ));
        y = -1 * (center.getY() - (center.getZ()
          * lastAccelerometer.getY())/
                  ( lastMomentum.getZ() ));
        zeroMomentPoint = Vector3D.cartesian(x, y, z);
        zmpHistory.add(zeroMomentPoint);
        while (zmpHistory.size() > MAX_HISTORY_SIZE){
                zmpHistory.remove(0);
        }
        if(zmpHistory.size()>= ACCUMULATOR_STEPS){
                Vector3D akum = Vector3D.ZERO_VECTOR;
                for (int i = 0; i <
                        ACCUMULATOR_STEPS; i++)
                {
                        int ri =
                        zmpHistory.size() - 1 - i;
                        akum =
                        akum.add(zmpHistory.get(ri));
                }
                zeroMomentPoint =
                        akum.divide(ACCUMULATOR_STEPS);
        }
}
```

a teda by sa dala pomerne jednoducho zapracovať do nášho agenta.

Ďalej by bolo vhodné využiť ForceResistance perceptor, ktorý autor využíval vo svojej práci na určovanie pozície tela robota. Tento perceptor zakomponoval hlavne v triedach *FixedObject*, *Body Part* a v *AgentModel*, kde sa hodnoty z perceptora využívajú pri výpočte ZeroMomentPoint.

4.4 Pavol Mešťaník - Implementácia dynamického priameho kopu

Posledná úprava	Peter Filípek
Platné od	23. október 2014
Poznámky	

V práci [13] sa podarilo úspešne implementovať metódu pre realizáciu dynamického priameho kopu. Implementáciou tejto metódy hráč získal schopnosť výberu vzdialenosti, na ktorú chce loptu odkopnúť.

Jednotlivé fázy vykonávania dynamického kopu:

- Lokalizácia, priblíženie a mierenie: Tieto fázy sa môžu opakovať a ich cieľom je dostať hráča do takej vzdialenosti od lopty, z ktorej je schopný vykonať kop.
- Určenie nohy ktorou kopať: Ktorou nohou sa bude kop vykonávať je určené na základe polohy lopty a hráča. Ktorá noha bude využitá na kop je jeden zo vstupných parametrov pre dynamický kop.
- Určenie vzdialenosti na kop: Určene tejto vzdialenosti prebieha porovnaním pozície lopty a cieľovej pozície. Táto vzdialenosť je ďalší potrebný vstup pre dynamický kop.
- Vypočítanie parametrov kopu: Parametre pre kop sa vypočítajú na základe vzdialenosti, na ktorú sa má kopať.
- Vytvorenie fáz dynamického kopu: Po vypočítaní dynamických parametrov kopu sa vytvoria fázy kopu. Tieto môžu byť založené na existujúcom pohybe a len upravovať niektoré z fáz tohto pohybu, alebo môžu byť všetky fázy vytvorené nanovo. Po vytvorení fáz sú tieto zaradené medzi ostatné nedynamické pohyby s novým vygenerovaným menom. Od tohto okamihu je vpodstate vytvorený nový nedynamický pohyb podľa zadaných parametrov.
- Vykonanie kopu: Samotné vykonanie kopu prebieha rovnako ako pri bežnom pohybe.

Výpočet parametrov pre kop prehrebol experimentálne s využitím už existujúceho kopu. Pomocou pokusov nad už existujúcim kopom, sa zistili závislosti efektorov hráča na vzdialenosť kopu. Vzhľadom na to, že pohyb obsahoval náhodné chyby, bolo vykonaných viacero meraní, na základe ktorých sa vytvorili priemerné hodnoty. Dynamický kop, implementovaný v rámci riešenia, bol vytvorený vďaka nájdeniu závislosti medzi vzdialenosťou kopu a zmenou efektora RLE3 pri kope kick_right_normal_stand. Alternatívou je výpočet parametrov na základe sily kopu podľa fyzikálnych parametrov simulačného modelu.

Riešenie bolo overené s použitím plánu PlanPmKick, teda bez hýbania hráčom s pevne určenou pozíciou hráča aj lopty, aby sa vylúčili ostatné vplyvy. Hráč je od lopty na začiatku vzdialený 0,2 jednotiek a lopta je nastavená presne pred jednu z nôh. Pri tejto situácii bol dynamický pohyb vykonávaný so 100% úspešnosťou. Pri nastavení inej situácie na začiatku nebola dosiahnutá 100% úspešnosť kopu vzhľadom na nepresnosti iných pohybov.

Implementácia tohto riešenia prebehla nad agentom vytvoreným tímom A55 Kickers, ktorý využíval ešte architektúru, ktorá obsahovala časti v Ruby.

Náš tím pokračuje v práci na agentovi, po predošlom tíme Gitmen. Vzhľadom na to, že diplomová práca Pavla Meštaníka vznikala v rovnakom čase ako agent tímu Gitmen, nie sú v aktuálnej verzii nášho agenta zapracované funkcie z tejto diplomovej práce. Preto som sa rozhodol stručne opísať, ako by sme mali postupovať v prípade, že by sme sa rozhodli funkcie z tejto diplomovej práce implementovať v našom agentovi.

Pavol Meštaník pracoval na systéme pre podporu dynamických pohybov. Základom tohto systému je trieda DynamicSkill ktorá dedí od triedy HighSkill ktorá sa nachádza v balíku sk.fiit.jim.agent.skills. Obsahuje funkcie potrebné na vytváranie dynamických pohybov počas simulácie.

addSkill - Táto funkcia vytvorí nový LowSkill s určeným menom a priradí ho do zoznamu existujúcich LowSkills. Zároveň sa tu nastavuje prvá fáza pohybu. Slúži ako konštruktor pre LowSkill pri vytváraní dynamických pohybov. Takto vytvorený LowSkill je vrátený ako výstup funkcie.

addPhases - Táto funkcia prevedie zoznam vytvorených fáz do podoby zreťazeného zoznamu, kde každá fáza odkazuje na nasledujúcu fázu. Zároveň sú všetky fázy premenované s použitím mena novovytvoreného dynamického pohybu. Každá fáza je po spracovaní priradená do globálneho zoznamu fáz, odkiaľ je ich možné následne vybrať pre vykonávanie.

createPhase - Táto funkcia slúži na vytvorenie novej fázy podľa zadaných parametrov. Vstupom funkcie je trvanie fázy a zoznam efektorov spolu

s hodnotou na akú sa má efektor zmeniť. Slúži ako konštruktor pre fázy. Novovytvorená fáza je vrátená ako výstup funkcie.

getPhasesForSkill - Táto funkcia vráti zoznam fáz pre zadaný LowSkill. Táto funkcionalita umožní načítať existujúci pohyb ako základ pre dynamický pohyb. Namiesto vytvárania všetkých fáz nanovo je tak možné použiť existujúci pohyb a len upraviť niektoré z fáz, alebo len niektoré konkrétne efektory.

Následne bol vytvorený dynamický kop, ktorý využíva funkcie tejto triedy. Dynamický kop je vytvorený v triede DynamicKickStraight ktorá sa nachádza v balíku sk.fiit.jim.agent.skills.dynamic. Vzhľadom na predošlé prepojenie z Ruby ktoré v aktuálnej verzii agenta už neexistuje, bude nutné reimplementovať triedu pre podporu tohto pohybu,aby bolo zaistené, že hráč lokalizuje loptu, priblíži sa k nej a nastaví sa na správnu pozíciu. Vychádzať sa pritom môže z reimplementovanej triedy Kick ktorá sa nachádza v balíku sk.fiit.jim.agent.highskill.

Trieda DynamicKickStraight obsahuje funkcie, ktoré priamo vypočítavajú a nastavujú parametre kopu.

createDynamicKick - Táto funkcia riadi vytvorenie dynamického kopu. Na začiatku sú načítané fázy z predlohy, podľa zadaného názvu kopu. Následne sú vypočítané zmeny efektorov pre určenú vzdialenosť. S takto vypočítanými hodnotami sú upravené vybrané fázy predlohy a vzniká nový kop. Tento nový kop dostane unikátne meno kombináciou vybraného názvu a vygenerovaného čísla. Toto zabráni konfliktom pri pridávaní viacerých dynamických pohybov rovnakého typu.

getBaseSkillPhases - Táto funkcia vyberá základný kop podľa toho, ktorou nohou sa má kopať a následne s použitím funkcie getPhasesForSkill získa zoznam fáz pre daný kop a tento vráti ako výstup funkcie.

calculateDynamicValue - Táto funkcia získava hodnotu pre dynamickú zmenu fáz kopu v závislosti na zadanej vzdialenosti. V implementácii podľa práce sa využíva len zmena jedného parametra kopu a teda návratovou hodnotou je len jedno číslo.

alterKickPhases - Táto funkcia upraví fázy vybraného kopu podľa vypočítaných hodnôt pre danú vzdialenosť. V implementácii podľa práce je upravovaná len fáza 2 kopu (v poradí je to až 4. fáza).

Pre vytváranie nových dynamických pohybov by bolo vhodné naštudovať reverznú kinematiku pre robota. Prípadne postupovať pri tvorbe nových pohybov rovnako ako autor práce, experimentovaním nad už existujúcimi pohybmi a hľadaním závislostí medzi jednotlivými efektormi využívanými pri

pohybe.

Ako inšpirácia môže poslúžiť aj článok Adaptive Motion Control: Dynamic Kick for a Humanoid Robot od autorov Yuan Xu a Heinrich Mellmann, v ktorom popisujú jednotlivé fázy takéhoto kopu a problémy ktoré pri nich nastávajú.

Zdroj: `http://www.naoteamhumboldt.de/wp-content/papercite-data/pdf/ki-xumellmann-10.pdf`

4.5 Peter Paššák - Optimalizovanie pohybov robota pomocou evolučných algoritmov

Posledná úprava	Metod Rybár
Platné od	4. november 2014
Poznámky	

V práci [16] sa Peter Paššák venuje evolučnému algoritmu, ktorý používa na zlepšovanie pohybov robotického hráča. Evolučným algoritmom sa nahradzovalo ručné vylepšovanie pohybov, ktoré býva veľmi pracné a veľmi ťažko sa s ním dá priblížiť k ideálnemu stavu.

Pri každom pohybe sa dajú určiť vlastnosti, na základe ktorých vieme kvalitatívne posúdiť aké sú pohyby dobré. XML súbor s definíciou pohybu sa mapuje na genóm a po prebehnutí optimalizácie sa prevedie naspäť tak, aby sa dal testovať.

Ako jedince sa berú jednotivé pohyby. Použitá štruktúra sa teda skladá zo zoznamu fáz, z ktorých každá obsahuje názov fázy, nasledujúcu fázu, trvanie fázy a zoznam kĺbov a ich natočenia a tiež iné parametre. Táto štruktúra je vhodná na jednoduchú manipuláciu.

Bola vytvorená trieda pre vytváranie symetrických pohybov. Napríklad kop ľavou nohou sa dá automaticky transformovať na symetrický kop ľavou nohou a uložiť ako nový pohyb.

Nevychádza sa z náhodne vygenerovaných pohybov, ale už z definovaných pohybov. Kedže sa je prehľadávací priestor veľmi veľký, je dobré začať z bodu, ktorý ma aspoň prijateľné riešenie. Prvá generácia sa teda vytvorí mutovaním už známych pohybov.

Následne sa ďalšia generácia tvorí z časti mutáciami, z časti krížením a z časti zachovaním elitných jedincov predchádzajúcej populácie. Nemusí to tak však byť a to ako sa tvorí nová populácia sa dá nastaviť. Generácie sa tvoria pokiaľ nedostaneme požadovanú kvalitu pohybu alebo nedosiahneme maximálny zadefinovaný počet generácií. Pri výbere jedinca sa využíva výber turnajom a výber ruletou.

V práci sa uvažuje nad zlepšovaním pohybov aj pridaním alebo odoberaním fáz pohybu alebo využitia horolezeckého algoritmu a uvažuje sa ešte nad optimalizáciou prihrávok a otáčania.

Podarilo sa zvýšiť vzdialenosť strely zo 4,4 na 5,45 metra. Po upravení fitnes funkcie zlepšenie na 6,96 metra, ale za zvýšenia rozptylu. Po ďalšom

vylepšovaní dĺžka kopu 9,16 metra s dobrou presnosťou. Zlepšenie presnosti sa nepodarilo, ale zachovala sa presnosť pôvodného kopu pri viac ako dvojnásobnej dĺžke kopu.

Bola vytvorená rýchlejšia chôdze. Zrýchlenie bolo z 0,59 metra za sekundu na 0,69 pri znížení vychýlenia a zlepšenia úspešnosti a zároveň sa podarilo aj výrazné zrýchlenie na 0,97 metra za sekundu pri inom jedincovi, avšak s mierne horším vychýlením a úspešnosťou.

Pri chôdzi vzad bolo získané zrýchlenie z 0,1 metra za sekundu na 0,54 metra za sekundu ale za cenu takmer zdvojnásobenia vychýlenia a zhoršenia úspešnosti. Pri úkrokoch nastalo zrýchlenie z 0,05 na 0,16 ale s výrazným zvýšením vychýlenia.

V našom projekte by sa dal použiť vytvorený evolučný algoritmus na vylepšovanie existujúcich aj nových pohybov. Je vytvorený pre kráčanie vpred a vzad, úkroky a kopy a teda by sa dal jednoducho napojiť na staré aj nové pohyby, ktoré by sme upravovali.

Najzaujímavejšie môže byť využitie triedy na automatické preklápanie symetrických pohybov.

```
public class Reversion {
        public void reverseMove() {

        }
}
```

Symetrický pohyb je napríklad chôdza vpred. Je definovaný tým, že ďalší krok cyklu je len obrátená verzia predchádzajúceho kroku, napríklad pohyb ľavou a pravou nohou. Vstávanie napríklad k tomuto nepatrí.

Trieda na automatické tvorenie symetrických pohybov sa dá teda použiť na automatické generovanie symetrického pohybu ktorý meníme bez toho, aby sme museli meniť všetky jeho časti ručne.

4.6 Martin Košický - Nižšie schopnosti hráčov

Posledná úprava	Michal Segeč
Platné od	23. október 2014
Poznámky	

Autor sa venoval analýze evolučných algoritmov [11], ktoré by bolo vhodné použiť na zabezpečenie pohybu kvôli optimalizácií, prípadne pri generovaní trajektórie kĺbov.

CMA-ES (Evolučná stratégia adaptácie kovariančnej matice) – je algoritmus, ktorý slúži na optimalizáciu zložitých, nelineárnych, nekonvexných problémov. Algoritmus pracuje s určitým aktuálnym stredom, kovariančnou maticou a polomerom. Každou iteráciou sa vytvorí niekoľko n>0 jedincov a každému sa vypočíta fitness funkcia. Vyberie sa m<n najlepších. Z týchto m jedincov sa vypočíta stred pomocou váženého priemeru.

Q-Learning – je metóda učenia s odmenou a trestom. Cieľom tejto metódy je, aby bola v každom stave vybratá taka akcia, ktorá by maximalizovala celkovú odmenu. Je to iteratívny proces, pričom v každom kroku sa musí aktualizovať Q hodnota akcie a stavu. Q(s,a) je funkcia ktorý ohodnocuje kvalitu, keď sa v stave s vykoná akcia a.

Autor sa venoval skúmaniu doprednej kinematiky, ktorá rieši problém hľadania súradníc kĺbov, keď sú známe iba ich uhly. Zistil, že v simulovanom robo. futbale sa tieto hodnoty dajú vypočítať pomocou prekladacích a rotačných matíc. Taktiež sa venoval inverznej kinematike, ktorá naopak k známym súradniciam kĺbov hladá hodnoty premenných kĺbov, aby bola dosiahnutá žiadaná pozícia. Skúmal 3 metódy riešenia problému inverznej kinematiky a to: alegabraické, geometrické, iteratívné.

Autor preskúmal, že môžu byť 4 druhy chôdze: 1. Fáza dvojitej podpory – obidve nohy sú na zemi. 2. Fáza pred kývaním – päta jednej nohy sa dvíha zo zeme, ale humanoid má stále fázu dvojitej podpory, keďže stojí palcami na zemi. 3. Fáza základnej podpory – jedna noha je na zemi a druhá vo vzduchu smerom vpred. 4. Postkývajúca fáza – prsty prednej nohy smerujú k zemi a agent má opäť fázu dvojitej podpory.

Preskúmal aj tzv. ZMP, ktorý je jeden z najviac používaných pojmov v robotike. Označuje bod, keď dynamické sily pri kontakte noha so zemou neprodukujú žiaden moment v horizontálnom smere – t.j. horizontálny odpor a gravitačná sila su rovné 0. Autor otestoval daný algoritmus v simulovanom

prostredí, kde pozoroval, že ak sa bod ZMP ocitol mimo podpornej plochy, robot začal padať.

Základný koncept chôdze robota, ktorým sa riadil autor je:

- výber cieľových bodov pre aktívnu nohu
- aplikovanie inverznej kinematiky na inverznú nohu
- vykonať akcie na kĺboch, ktoré vedú k stabilite
- výmena nôh

Autor testoval 2 druhy simulácie a to: ovládanie lokálnej stability minimalizáciu vzdialenosti ZPM od podpornej plochy , a v druhom prípade minimalizáciou vzdial. Posunutého ZPM od podpor. plochy. Plocha bola tvorená 2 bodmi a agent sa snaží svoj ZPM dostať na stred tejto priamky.

Autor avšak zistil, že toto riešenie neviedlo k úspešnej a stabilizovanej chôdzi. Pôvodné riešenie chôdzy agenta nahradil novým postupom:

- Ručný výber póz
- Optimalizácia póz pre stabilitu
- Vytvorenie interpolácie medzi pózami
- Inverznou kinematikou dopočítať uhly v kĺboch
- transformácia cieľových uhlov na uhlové rýchlosti

Robot použití pri simulácií je zložený z 23 častí tela a 22 otočných kĺbov. Na optimalizáciu póz pre stabilitu autor využil CMA-ES, genetický algoritmus, Q-Learning Najlepší výsledok sa podaril získať pomocou Q-Learningu, kde sa podarilo získať najlepšie výsledky, čo sa týka rýchlosti aj stability.

Autor testoval rôzne druhy chôdze:

- Pomalá chôdza – agent prešiel pol ihriska za 59 sekúnd, priem. rýchlosť bola 0,61 hm/h a z 50 pokusov agent nespadol ani raz
- Stredne rýchla chôdza – pol ihriska za 45 sekúnd, rýchlosť – 0,8 km/h, z 50 pokusov agent spadol 7 krát.
- Rýchla chôdza – pol ihriska za 35 sekúnd, rýchlosť – 1,02 km/h, z 50 pokusov agent spadol 10 krát.

- pred spustením optimalizácie agent nedokázal ani raz prejsť celé ihrisko

Autor na základe experimentov zhodnotil, že stabilizáciu v reálnom čase nie je možná bez predvypočítanej trasy. Táto metóda posúva pozíciu trupu agenta relatívne k stacionárnej nohe. Táto metóda najlepšie fungovala pri optimalizácií pomocou učenia Q-learning. Metóda bola úspešná pri vytvorení stabilných a rýchlych chôdzi no niekedy aj táto metóda zlyhala. Vždy keď robot spadol, chyba sa objavil na rozhraní dvoj fáz – napr. keď sa menila noha z kývajúcej pózy na stojacu. Preto zhodnotil, že by bolo vhodné sa zamerať na hladký prechod medzi 2 fázami pohybu.

4.7 Zahraničný tím NaoTH

Posledná úprava	Peter Filípek
Platné od	23. október 2014
Poznámky	

Nao Team Humboldt skrátene NaoTH je Nemecký tým z univerzity Humboldt v Berlíne. Venujú sa Standard Platform League a 3D Simulation. Od roku 2008 sa aktívne zúčastňujú svetových turnajov, získali viac ako desiatku ocenení a niekoľko prvých miest v rôznych turnajoch. Na konte majú aj množstvo publikácií, ktoré sa venujú rôznym oblastiam robotiky a zasahujú tak aj do robocupu.

Vytvorili základ agenta v C++, ktorý je možné stiahnuť a pracovať s ním, túto kostru ďalej rozvíjali a obohacovali o ďalšie schopnosti. Vytvoril aj rozšírenie pre SimSpark Simulator tak, aby zodpovedalo prostrediu a pravidlám v Standard Platform League.

Hráč vie dynamicky meniť rolu v tíme podľa vzdialenosti od lopty a situácie na ihrisku. Vie sa napríklad dynamicky prepnúť z podporujúceho hráča na útočiaceho hráča a podobne.

Robot pri dynamickom kope vie reagovať na miernu zmenu pozície lopty vychýlením nohy, ktorou ide kop vykonať. Robot pri kope sleduje loptu a nie cieľ kam ide kopať, preto je schopný reagovať na zmenu pozície lopty.

Robot pri dynamickom kope je schopný ovplyvniť nielen silu kopu, ale aj smer kopu. Vďaka tejto vlastnosti je robot schopný kopnúť nielen priamo pred seba, ale aj do strany. Podľa situácie je schopný pokryť až 180 stupňov (90 stupňov doľava a 90 stupňov doprava). Táto schopnosť výrazne urýchli proces celého kopu vzhľadom na to, že robot sa nemusí pred kopom úplne presne postaviť a vykonávať rôzne úkroky.

Robot vie dynamicky pri chôdzi meniť pozíciu chodidiel, čo je využívané pre driblovanie (vedenie lopty). Robot tesne pred kontaktom s loptou zmení polohu chodidiel, aby boli do tvaru písmena V. Týmto zabezpečí to, že pri jemnom kontakte chodidla s loptou lopta nezmení výrazne smer a pohne sa mierne pred robota.

Robot aby predišiel pádu pri miernych kolíziách disponuje metódou na stabilizáciu samého seba. Robot sa stabilizuje pomocou metódy, ktorá je založená na neurónových sietiach.

Robot sa vie zorientovať v prostredí a následne podľa toho zaujať pozíciu

na ihrisku. Lokalizácia prebieha pomocou časticového filtra (v našom prostredí SimSpark, pravdepodobne to nie je možné vykonať, keďže ihrisko nie je dostatočne označené a ohraničené).

Zdroj: `http://www.naoteamhumboldt.de/`

4.8 Zahraničný tím NUBots

Posledná úprava	Miroslav Wolf
Platné od	23. október 2014
Poznámky	

Austrálsky tím NUbots z Univerzity of Newcastle má rekord úspechu v RoboCup Standard Platform League od ich prvého vstupu v roku 2002. Tím súťažil vo viacerých kategóriách - v lige pre štvor- nohých robotov, v štandardnej lige a v kid-size lige. V roku 2006 a 2008 získali prvé miesto. Ústredným cieľom tímu je byť vysoko výkonne konkurenčný tím v robotickom futbale.

Hardvér a softvér

Tím používa DARwIn-OP robota ktorý má na nohách senzory. Ich hlavný výskum sa zameriava na používanie metód strojového učenia v softvérových systémoch robota na dosiahnutie vyššej výkonnosti a autonómie. Softvér majú navrhnutý pre prácu viacerých robotických platforiem a všetky jednotlivé moduly boli navrhnuté aby mohli byť ľahko použité v iných systémoch. Senzory a akčné časti sú prístupné pomocou štandardného formátu, bez ohľadu na to na akom robotovi softvér beží. Tohto roku bola vykonaná zásadná zmena softvérovej architektúry a to vylepšenie softvérovej modularity na základe posielania správ systému. Softvér NUBots je navrhnutý tak, aby nové tímy a členovia tímu vedeli ľahko pochopiť a inovovať existujúci kód. Pohybový engine robota je naprogramovaný v C++.

Systém videnia

Zrak je jeden z hlavných oblastí výskumu spojených s Newcastle laboratóriom robotiky. Skúmané bolo najmä rozpoznávanie objektov, určenie horizontu, detekcia hrán, úprava modelu a rozpoznávanie farieb použitím elipsovitej úpravy (elipse fitting), konvexná optimalizácia a ďaľšie. Tím skúšal aj nahradiť kameru a použiť novú ktorá poskytuje obraz 1080p čo zväčší veľkosť zorného poľa a umožňuje detekciu a klasifikáciu objektov vo väčších vzdialenostiach.

Lokalizácia

Na lokalizáciu tím používa Kalmanov filter. Výskum je zameraný na Bayesovské prístupy na lokalizáciu robota, vrátane Kalmanovho filtra a metódy založené na filtrovaní častíc. Tím sa zaoberal modifikovaním Kalmanovho filtra na zvládnutie neideálnych podmienok z obzoru, začlenením informácií od viacerých agentov.

Chôdza

Tím niekoľko rokov skúmal zlepšenie rýchlosti chôdze a stability na rôznych platformách. Na robotovi AIBO bola dosiahnutá jedna z najrýchlejších chôdzí v tom čas. Na Nao robotovi vylepšili existujúci engine pre chôdzu modifikovaním tuhosti. Vylepšenia pohybu boli vykonané v prvom radepomocou optimalizačných techník, s nedávnym zlepšením ich frameworku pre online optimalizáciu pohybu dvojnohého robota.

Učenie

Tím aplikoval posilnené učiace techniky pre optimalizáciu pohybu hlavy, poskytnutím robustného algoritmu ktorý robota učí vyberať si míľniky, pre efektívnu lokalizáciu počas futbalového zápasu. Tento algoritmus bol použitý v súťaži Robocup 2013.

Ďalšia práca

Veľa výskumov bolo zameraných na základnú softvérovú architektúru a externé nástroje umožňujúce flexibilitu a rozšíriteľnosť do budúcnosti pre budúci výskum. Medzi tieto projekty patrí vylepšenie konfigurovatelnosti softvéru, prostredníctvom konfiguračných aktualizácií v reálnom čase, vývoj webovej on-line vizualizácie a ladiacich nástrojov.

Zdroj: `http://arxiv.org/pdf/1403.6946.pdf`

Pohyb hlavy - vyberanie míľnikov pre efektívnu lokalizáciu robota počas zápasu

Schopnosť agenta lokalizovať sa v rámci svojho prostredia je kriticky závislá na jeho schopnosti presne pozorovať statické, charakteristické črty.

Keďže agent má obmedzené pole videnia, stanovenie optimálneho ovládania hlavy robota, by mohlo byť kľúčovým pre maximalizáciu poskytovaných informácií o jeho pozícii. Článok [4] podrobne opisuje aplikáciu vylepšeného učenia na základe pohybov hlavy, čo viedlo k zlepšeniu o 11% pri vlastnej lokalizácii, bez nutnosti ďalšej optimalizácie rozpoznávania objektov.

Efektívna lokalizácia je potrebná pri riešení úloh najmä pri robotickom futbale RoboCup-e. Správanie robota je silno závislé na lokalizácii robota. Nepresná lokalizácia vedie k neefektívnemu správaniu a zlému výkonu. Úspešná lokalizácia robota pritom závisí od kamerového systému robota a teda jeho rozhľadu. Ak nie je robot lokalizovaný, môže byť pre čiastočnú lokalizáciu použitý panoramatický pohyb hlavy. Vzhľadom k tomu, že existuje mnoho užitočných objektov, môžeme pomocou inverznej kinematiky sekvenčne zvoliť objekty na ktoré sa bude robot sústreďovať, čím minimalizujeme neistotu lokalizácie pri hre. Existujú dva typy objektov dôležitých pre lokalizáciu robota počas hrania futbalu - míľniky a objekty. Míľniky sú použité na lokalizáciu robota, zatiaľ čo objekty musia byť lokalizované v modeli sveta, meraním pozície vzhľadom na lokalizovaného robota. Informácie sú následne transformované do globálnych súradníc. Problém ovládania hlavy bol konštruovaný Markovým rozhodovacím procesom s využitím on-line vylepšených metód učenia. Markov rozhodovací proces bol postavený na vzorkovaní dát z infraštruktúry robota.

Akčný priestor je zbierka míľnikov a objektov, ktoré sú na poli a teda štyri bránkové žrde a lopta. Robot skenuje oblasť, v ktorej je pravdepodobné, že sa objekt nachádza, použitím Kalmanovho filtra. Toto skenovanie trvá určitý nastavený čas, alebo pokiaľ nenájde objekt. Nelokalizovaný robot skenuje celé pole rozhľadu, pre stanovenie dobrej počiatočnej polohy.

Algoritmus využívajúci spomínané učiace techniky pri pohybe hlavy robota a výbere míľnikov pre lokalizáciu robota, bol použitý zahraničným tímom NUBots. Bližší opis týchto techník je v článku [4].

4.9 Zahraničný tím rUNSWift

Posledná úprava	Miroslav Wolf
Platné od	23. október 2014
Poznámky	

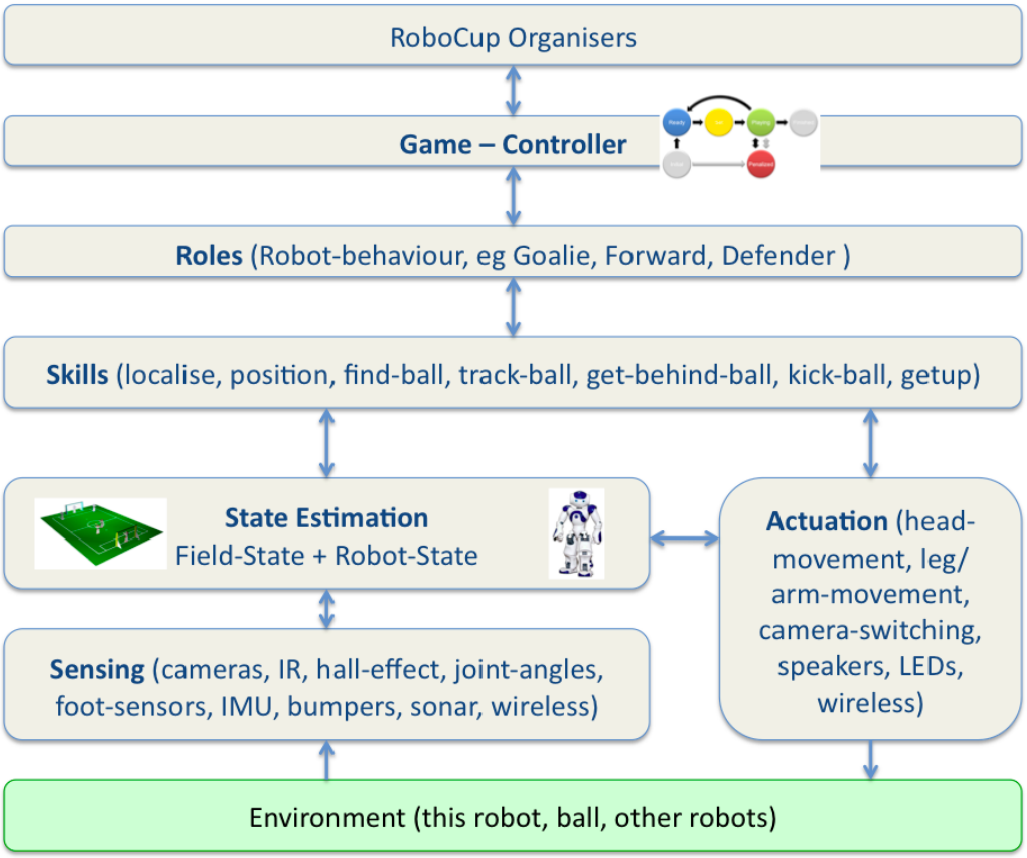
Jadro tímu je zložené z vysokoškolských študentov pod dozorom vedúcich univerzity UNSW (University of New South Wales), ktorí sa venujú Robocupu už mnoho rokov. Od roku 2012 začali spolupracovať aj s Vietnam National University. Univerzita sa venuje Robocupu už 14 rokov. Vývoj pre rok 2013 zahŕňa automatickú kalibráciu farieb, nový herný simulátor, lepšiu stabilitu, zlepšená detekcia robotov a zdokonalený prístup správania, ktorý bol navrhnutý v jazyku Python. Taktiež vylepšili základ pre prácu v tíme pri väčšom počte hráčov, určili nové ciele a zvýšili počet hráčov na 5 a zapracovali ďalšie zmeny pravidiel. Ich dlhodobým cieľom je rozvíjať všeobecné používanie inteligentných systémov, ktoré môžu učiť a byť učené na vykonávanie veľa rôznych úloh samostatne prostredníctvom interakcie s prostredím.

Architektúra

Tím rUNSWift používa sieťovo orientovanú architektúru, ktorá je odolná voči chybám. To znamená, že každý robot môže mať mierne rozličný pohľad na svet a podľa toho aj svoju rolu v tíme. Tento prístup má výhodu, že poskytuje určitú redundanciu v prípade, keď je iný robot diskvalifikovaný alebo keď prestane fungovať (môže ho nahradiť iný). Architektúra začína na koreňovej úrovni, kde "herný ovládač"vyvolá stavy pre vyššiu úroveň. Na nižších úrovniach generátor chôdze vykonáva fázy chôdze, ktoré vyvolávajú primitívne zmeny stavov robota ako sa menia jeho pózy 100 krát za sekundu.

Systém videnia

V roku 1999 bol použitý jednoduchý systém vzdelávania na trénovanie rozoznávania farieb. V roku 2001 už boli použité štandardy strojového učenia, C4.5, pre vytvorenie rozhodovacieho stromu rozpoznávania. Toto sa ukázalo ako veľmi dôležité, keďže osvetlenia v laboratóriu univerzity a haly, v ktorej tím súťažil bolo rozdielne a ich starý systém videnia si s tým nevedel poradiť. V roku 2000 bol systém videnia dostačujúci pre rozpoznanie robota a vyhnutiu sa tak zrážky so spoluhráčom. V posledných rokoch bol aktuali-



Obr. 2: Architektúra robota

zovaný systém videnia pre rozpoznanie ELD hranice, ELD značenia. Taktiež boli zapracované funkcie pre rozpoznávanie okrajov a robot sa už toľko nespolieha iba na farbu. Ďalej bol zapracovaný "natural landmark recognitionälgoritmus, ktorý je schopný lokalizovať robota v reálnom čase. Tento prístup je založený na jednorozmerných lokálnych obrazových funkciách a bol predstavený v posledných rokoch RoboCupu. Súčasný výskum a vývoj zahŕňa semi-automatickú korekciu farieb použitím templatov, rýchlejšiu detekciu čiar, vylepšenú detekciu robota a všeobecnú adaptáciu robota na nové rozmery a označenia.

Lokalizácia

Pre lokalizáciu bol v roku 2000 použitý jednoduchý Kalmanov filter. Systém lokalizácie sa vyvinul a zahŕňa multi-modálny filter a distribuované dátové spojenie cez robotov v sieti. V roku 2006 sa zmenilo správanie robotov (roboti individuálne zdieľali informácie) a začali pracovať ako jeden tím - jeden výpočet bol rozložený medzi viacero robotov. To umožnilo napríklad použiť loptu ako lokalizačnú informáciu. Použitie viac a viac robotov pre jeden Kalmanov filter sa neškáloval ľahko keďže počet módov exponenciálne rástol. V roku 2012 tak bol použitý algoritmus Iterative Closest Point (ICP), ktorý rozširuje predchádzajúcu prácu s porovnávaním čiar a ďalších vizuálnych prvkov a objektov.

Zdroj: `http://www.informatik.uni-bremen.de/spl/pub/Website/Teams2013/rUNSWift.pdf`

4.10 Zahraničný tím Apollo3D

Posledná úprava	Michal Segeč
Platné od	23. október 2014
Poznámky	

Tím Apollo3D pozostáva zo študentov z Nanjing Univerzity v Číne. Bol založený v roku 2006. Tím v roku 2010 a 2013 vyhral súťaž Robocup v 3D simulovanom futbale. V roku 2011 sa umiestnili na 3.mieste.

Sami autori povedali, že zabezpečenie chôdze humanoida je jednou z najťažších úloh a že neexistuje ideálny algoritmus, pre všeobecnú chôdzu.

Autori vytvorili nový druh flexibilnej chôdze, ktorá vyuzíva CMAC metódu (Cerebellar Model Articulation Controller), čo je vlastne typ neurónovej siete. Je to asociatívny typ pamäti. Táto metóda bola použitá už v roku 1975 pre robotické ovládače.

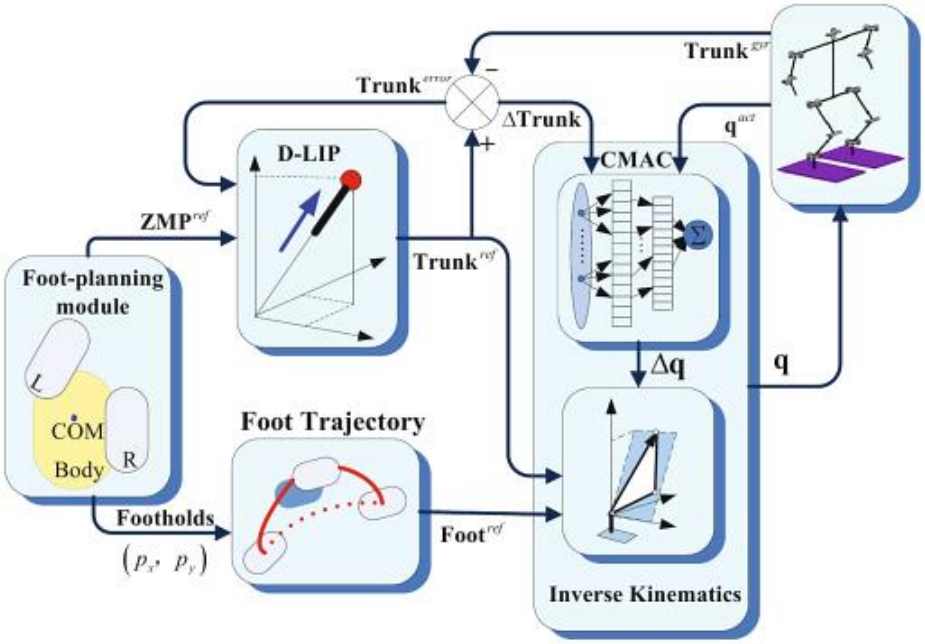
Taktiež táto flexibilná chôdza využíva lineárne invertované kyvadlo s prediktívnou kontrolou generovania trajektórie pohybu trupu agenta za predpokladu že sa zachová dynamická rovnováha agenta.

Tím Apollo3D zakomponoval tento pohyb a podarilo sa im vyhrať Robocup 2013. Cieľom robotického futbalu je, aby sa do roku 2050 podarilo vytvoriť tím autonómnych humanoidných robotov, ktorí budú schopní na šampionáte poraziť ľudský tím. Každý tím je zložený z 11 Nao robotov, ktorí medzi sebou komunikujú.

Výhodou CMAC metódy chôdze je, že agent sa dokáže pohybovať, otáčať, robiť úkroky smerom k cieľu bez straty stability. Na začiatku sa určí prijateľné držanie nohy a vypočíta sa ZMP. Zároveň sa vypočíta trajektória hrude agenta na základe dvojitého inverzného lineárneho kyvadla s prediktívnou metódou. Takýmto spôsobom je možné potom určiť trajektóriu oboch nôh. Okrem toho každý pohyb kĺbu je možné vypočítať pomocou inverznej kinematiky. Póza hrude agenta je jednoducho vypočítateľná na základe gyro senzoru. Na konci sa využije CMAC optimalizácia a korekčný algoritmus, aby sa zabezpečil správny pohyb kĺbov nohy.

Inverzná kinematika pomáha určiť polohu kĺbov, keď je známa finálna poloha objektu. Nao robot má 22 kĺbov. Maximálny uhol otočenia, ktorý bol dosiahnutý je 1.28 stupňa, a to v prípade, že sa zmenila podporná noha.

S daným algoritmom, ktorý tím vytvoril sa im podarilo umiestniť do nepriateľovej bránky 53 gólov, pričom prepustili 5 gólov.



Obr. 3: CMAC optimalizácia

Autori si uvedomujú, že úspešný humanoidný tím potrebuje aj nejaký vyšší rozhodovací modul, než len obyčajne nižšie akcie. A že na splnenie cieľa nestačí len obyčajná chôdza, ale táto chôdza musí byť stabilná a prispôsobivá hre.

Zdroje: `http://www.cs.utexas.edu/~todd/cs344m/resources/soccer-3dsim/Apollo3D_TDP.pdf` `http://books.google.sk/books?id=aJgrBAAAQBAJ&pg=PA112&lpg=PA112&dq=apollo3d&source=bl&ots=lEWluqSDC3&sig=Q6znoVlMZRAld49762jEgJPZc&hl=en&sa=X&ei=u8pEVLTJC8KtaczIgMAM&ved=0CFkQ6AEwBzgK#v=onepage&q=apollo3d&f=false`

4.11 Zahraničný tím Austin Villa

Posledná úprava	Michal Segeč
Platné od	23. október 2014
Poznámky	

Austin Villa – je tím robotického futbalu pochádzajúci z univerzity v Texase, Austine. Tento tím dosiahol četné úspechy, keďže sa im podarilo posledné 2 roky vyhrať v 3D simulovanom futbale na Robocupe v Brazílii a Iráne.

Členovia tímu si boli dobre vedomí problému nestabilnej chôdze pri vykonávaní pohybu agenta, čo aj označili za netriviálny problém. Snažili sa teda o vylepšenie optimalizácie agenta, aby sa dokázal pohybovať čo najrýchlejšie v každom smere a bez straty stability.

Agent, ktorý vytvorili bol taký úspešný, že zvíťazil na Robocupe 2011, kde vyhral všetkých 24 turnajov a strelil 136 gólov, pričom nepustil do bránky ani jeden. Agent je vytváraný podľa vzoru Aldebaran Nao robota, ktorý výšku cca. 57 cm a váži 4.5 kg. Každý agent má 22 kĺbov – 6 v každej nohe, 4 v ruke a 2 v krku. Vizuálne informácie sa agentovi posielajú cca. každých 60 ms v podobe vzdialenosti a uhlov k jednotlivým objektom v zornom uhle 120 stupňov. Agent je taktiež vybavený gyroskopom a accelerometrom. Každý agent môže komunikovať s ďalším každých 40 ms posielaním správy do veľkosti 20 Bytov.

Celý proces pohybu začína výpočtom sínusových funkcií pre vytvorenie pohybu v končatinách. Proces začne výberom druhu chôdzy, zvolí sa cieľ pre trup a nohy a následne sa použije inverzná kinematika pre určenie požadovaného pohybu kĺbov. PID kontrolery pre každý kĺb konvertujú tieto pozície do príkazov a sú odoslané do simulátoru.

Zaujímavé je taktiež, že väčšina týmov využíva simulované prostredie, aby neskôr bolo možné nasadiť reálneho agenta, zatiaľ čo tým Austin Villa najprv testovali pohyby na reálnom agentovi, aby ho mohli neskôr nasadiť do súťaže v simulovanom futbale.

Tím skúšal viaceré verzie agentov.

- 1.typ – agent, ktorý sa pohybuje smerom k cieľu ale neotáča sa pritom
- 2.typ – agent, ktorý sa pohybuje smerom k cieľu a otáča sa pritom

- 3.typ – agent, ktorý stoji na mieste a až po natočení smerom k cieľu sa rozbehne plnou rýchlosťou.

Všetky 3 typy agentov boli optimalizované pomocou algoritmov. Spôsob chôdze agenta je parametrizovaný cez viac ako 40 parametrov. Nastavenie iniciálnych parametrov malo ako dôsledok veľmi pomalú ale stabilnú chôdzu. Preto následne tím optimalizoval parametre použitím evolučného algoritmu CMA-ES. Je to algoritmus, ktorý generuje a vyhodnocujeset kandidátov z multivariančného gaussoveho rozdelenia. Každý z týchto kandidátov je ohodnotený pomocou fitness funkcie.

Spomedzi všetkých 40 parametrov bolo vybraný set 14 parametrov, určených pre optimalizáciu, ktoré boli dôkladne vybrané na základe pozorovania, že by mohli mať najväčší dopad na rýchlosť a stabilitu agenta.

Jednou z funkcií, ktoré optimalizovali je driveBallToGoal. Pri tejto funkcii je úlohou robota viesť loptu čo najďalej k cieľu počas 30 sekúnd. Fitness funkcia je vyhodnotená ako vzdialenosť, ktorú sa agentovi podarilo prejsť za tento čas. Po optimalizácií, agent prekonal vzdialenosť, ktorú prešiel pôvodný agent o 15 jednotiek. Pri simulovaní hry optimalizovaného vs. pôvodného agenta, optimal. agent vyhral o priemerne 5.54 gólov pri odohraných 100 hrách.

Druhou funkciou je goToTarget. Cieľom techniky je, aby sa agent dostal k cieľu za určitý čas. Týchto cieľov je viacero a vždy je aktívny v danej chvíli iba jeden na určitý čas. Agent je odmenený na základe vzdialenosti, ktorú prejde na ceste k cieľu. Ak agent dôjde k cielu, je extra odmenený v podobe extrapolácie celkového času, ktorý by mohol ísť k cieľu (čiže kým je cieľ aktívny) - čas, ktorý ostáva kým cieľ zmizne. Na druhú stranu v prípade že agent počas optimalizačného času padne na zem počas cesty k cieľu, je penalizovaný.

Ďalším prípadom optimalizácie bolo nastavenie setu parametrov pre zlepšenie rýchlosti agenta. Podarilo sa im síce zvýšiť rýchlosť agenta, avšak medzi prechodmi zo stavu rýchlej chôdze vpred a goToTarget setu, agent bol nestabilný a padal. Preto použili už existujúci set goToTarget parametrov a zlúčili ich s novými parametrami pre rýchly šprint, z čoho opäť vznikol vylepšený agent, ktorý bol schopný prepínať medzi 2 setmi bez straty stability.

Posledným setom je set parametrov pri nastavovaní pozície agenta. Preto tím vytvoril novú optimalizáciu nazvanú ako driveBallToGoal2, v ktorej agent je ohodnotený na základe toho, ako ďaleko dokáže viesť loptu za 15 sekúnd, pri rôzných štartovných pozíciach, čiže pri rôzných pozíciach a orien-

tácií pri lopte. Parameter pozície je použitý, ak je agent .8 metrov od lopty a je zlúčený s hodnotou zo setu goToTarget. Kombináciou všetkých 3 setov sa dosiahla požadovaná optimalizácia.

Zdroj: P. MacAlpine et. al. [12]

4.12 Zahraničný tím RoboCanes

Posledná úprava	Michal Segeč
Platné od	19. november 2014
Poznámky	

Tím vznikol v roku 2010 na Univerzite v Miami. Súťažia nielen v rámci robotického 3D simulovaného futbalu, ale aj rámci štandardnej platformy.

Vytvorenie základných skillov ich agenta je založené na základe agenta nemeckého humanoidného tímu B-Human.

Na začiatku teda zlúčili schopnosti B-Human agenta s ich vlastným agentom. Prvý skill, ktorý takto implementovali bolo chodenie. Keďže B-Human je humanoidný agent, museli podniknúť určité úpravy ako napr. namapovať efektory B-Human agenta na efektory agenta v simulovanom prostredí. Potom sa zamerali na úpravu parametrov chôdze, ako frekvencia, výška kroku, alebo umiestnenie ťažiska. Táto optimalizácia funguje veľmi dobre v simulovanom prostredí, ale nie je veľmi uplatniteľná pri štandardnej hre. Preto tím momentálne pracuje na vývoji chôdzi, ktorá by bola použiteľná rovnako v humanoidnom turnaji, tak aj v simulovanom 3D-futbale.

Túto optimalizáciu sa tímu podarilo dosiahnuť pomocou genetických algoritmov a tzv. "reinforcement learning", čo je druh učenia inšpirovaný psychológiou správania, ktoré sa zaoberá, ako by mal agent vykonať určité činnosti v prostredí, aby maximalizoval výšku odmeny. Q-Learning je typ tohto učenia.

Ďalším cieľom, ktorý sledujú je rýchlo generovať spoľahlivé pohyby agenta bez nejakého drahého a sprístupniteľného hardwaru. Tímu sa podarilo vytvoriť 4 robustné avšak nie úplne stabilné pohyby a následne aplikovali 3 algoritmy – CMA-ES, xNES, PSO a následne otestovali túto optimalizáciu na vyše 900 experimentoch. Týmto autori potvrdili, že je možné dosiahnuť stabilné a komplexné pohyby v pomerne krátkom čase.

Následne by chceli dané skúsenosti s optimalizáciou, integráciu preniesť aj na humanoidného robota, čo by mohlo zlepšiť fyziku robota.

Jedným z ďalších zámerov tímu je sa venovať aj rozpoznávaniu plánov a zámerov hráča. Čo už však patrí medzi zložitejšie high skilly. Toto nie je možné dosiahnuť bez koordinovanej kontroly agenta. Preto sa už v minulosti zamerali na vývoj viacerých techník ako pravdepodobnostné metódy, metódy založené na logike, neurónové siete.. Ich zámerom je momentálne sa venovať

kvalitatívnemu opisu dynamickej scény a to na základe mapovania informácií zosnímaných pomocou agenta z prostredia na kvalitatívne fakty určené pre spracovanie. Na základe symbolickej reprezentácie je možné definovať možné akcie na základe predpokladov a následkov.

Reinforcement learning (RL) je druh učenia, kde je agentovi daná odmena, ktorá ohodnocuje vykonanie jeho činností a tak zistíme optimálny postup pri ďalšom výbere akcií. Tím využil Q-Learninga SARSA učenie, ktoré zakomponovali do frameworku agenta. Zároveň plánujú využiť toto učenie 2 spôsobmi: 1. Preskúmať ako určité skilly môžu byť vylepšené pomocou tohto učenia, ako napr. rýchlejšia chôdza, alebo rýchlejšie vstanie zo zeme 2. Zameranie sa na správanie agenta. Keď vieme, ktorá stratégia sa môže použiť, tak by sa jednotlivé kroky, ktoré sa vykonajú v rámci stratégie mali vykonať vybrať pomocou tohto typu učenia

Pri RL využil tím taktiež algoritmy: GQ, Greedy-GQ a Off-Pac, ktoré sa približujú k aproximácií lineárnych funkcií a vykazujú oveľa lepšie výsledky v rámci predikcie a kontroly.

Reinforcement learning – je druh strojového učenia inšpirovaného psychológiou správania. Zaoberá sa tým, ako by mal softvérový agent vykonať určité akcie v prostredí tak, aby maximalizoval hodnotu získanej odmeny.

Základny model tohto učenia pozostáva z:

- množiny stavov S
- množiny akcií A
- pravidiel pre prechod medzi stavmi
- pravidiel, ktoré rozhodujú o okamžitej odmene
- pravidiel, ktoré popisujú, čo agent pozoruje

Q-learning – je metóda reinforcement learning učenia. Model pozostáva z agenta, stavu S a množiny akcií pre každý stav A. Vykonaním akcie a z množiny akcií A sa agent môže pohybovať zo stavu do stavu. Vykonaním akcie v určitom stave agent získa určitú odmenu. Cieľom je teda maximalizovať celkovú odmenu, ktorú môže agent získať pri prechode stavmi. Táto celková odmena je daná učením, čiže ktorá akcia je optimálna pre každý stav v zmysle očakávanej hodnoty celkovej odmeny, ktorú agent získa cez prechody medzi stavmi, začínajúc z momentálneho stavu.

Pri tomto type učenia nevieme okamžite vyhodnotiť, či sme zvolili vhodnú akciu. Nevieme ani určiť, aký bude nový stav po vykonaní danej akcie.

4.12.1 Princíp učenia

V každom kroku s sa zvolí akcia a, ktorá maximalizuje funkciu Q(s,a). Funkcia Q vyjadruje, ako vhodne je zvolená akcia pre daný stav.

$$Q(s,a) = r(s,a) + y.max'_a(Q(s',a')) \tag{1}$$

kde r(s,a) = okamžitá odmena, y – relatívna hodnota odloženej vs. okamžitej hodnoty (0,1), s' – nový stav po vykonaní akcie a, a,a' – akcie v stavoch s a s'. Zvolená akcia:

$$n(s) = argmax_a.Q(s,a) \tag{2}$$

Pre každý pár stav-akcia (s,a) sa inicializuje hodnota v tabuľke Q(s,a) na hodnotu 0.

V cykle sa následne vykonáva:

- výber akcie a a jej vykonanie
- obdržanie okamžitej odmeny r
- pozorovanie nového stavu s'
- úprava hodnoty v tabuľke pre Q(s,a) na základe výpočtu:

$$Q(s,a) = r + ymax'_aQ(s',a')$$

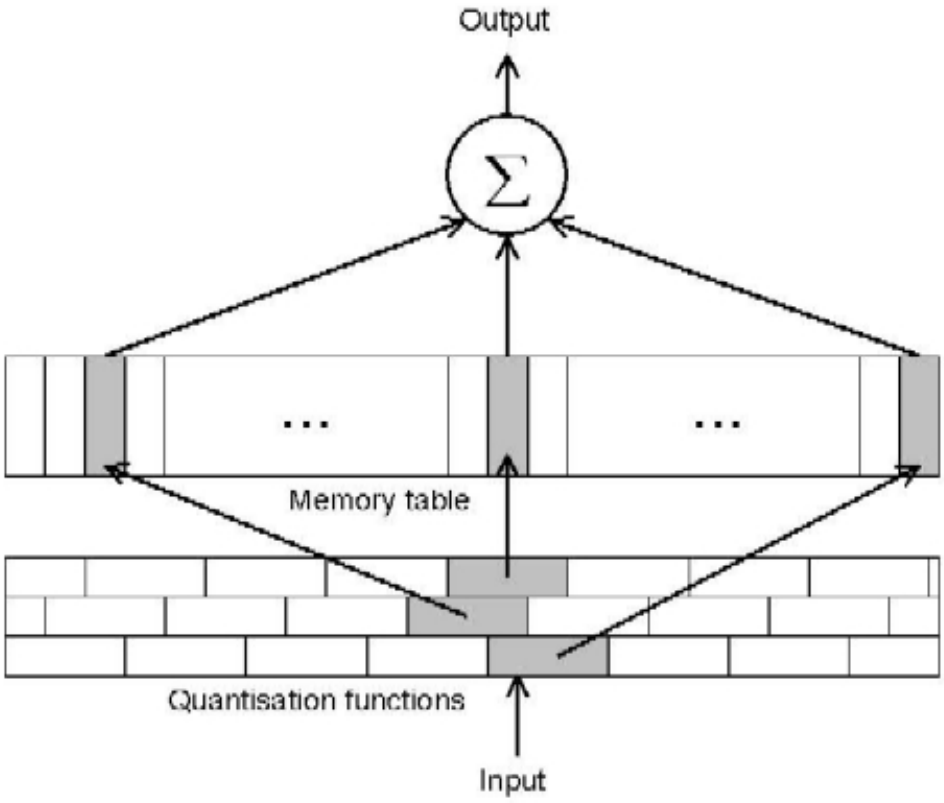
- s=s'

Príklad použitia algoritmu: `http://www.acm.uiuc.edu/sigart/docs/QLearning.pdf`

Najjednoduchšou metódou je pre túto metódu uchovávať dáta v tabuľke. Avšak so zvyšujúcou sa komplexitou sa systém stáva postupne neprehľadný. Druhou alternatívou pre toto učenie je využitie neurónových sietí namiesto tabuľky, kde vstupy predstavujú akcie a výstupom je hodnota v rozmedzí 0 a 1, ktorá reprezentuje užitočnosť.

CMAC - The Cerebellar Model Articulation Controller - je typ neurónovej siete, založený na modeli mozgu cicavcov. Je to asociatívny typ pamäte. Táto metóda bola prvýkrát navrhnutá ako funkčný model pre robotické ovládače v roku 1975. Je vysoko zaužívaná v reinforcement learning metóde. CMAC funkcia počíta funkciu f(x1.....xn), kde n je počet vstupných dimenzií. Vstupný

priestor je rozdelený na hypertrojuholníky, pričom každý z týchto trojuholníkov je asociovaný s určitou pamäťovou bunkou. Obsahom pamäte sú váhy, ktoré sú prispôsobené počas tréningu. Zvyčajne sa využíva viac ako jedna kvantizácia vstupného priestoru tak, že každý bod vstupného priestoru je asociovaný s počtom hypertrojuholníkov a tak aj s počtom pamäťových buniek. Výstupom CMAC metódy je algebraická suma váh vo všetkých aktivovaných pamäťových bunkách. Zmena hodnoty na vstupe vyústi v zmenu množiny aktivovaných hypertrojuholníkov, a tak aj v zmenu množiny pamäťových buniek, ktoré sa podieľajú na výstupe. (obr. 4)

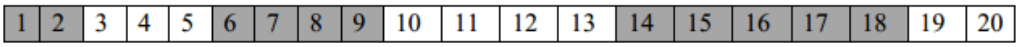


Obr. 4: Reprezentácia funkcie

Nevýhodou tejto metódy je objem požadovanej voľnej pamäte, ktorá je priamoúmerná s počtom aktivovaných pamäťových buniek. Zvyčajne sa tento

problem rieši použítím hashovacej funkcie, ktorá sprostredkuje iba toľko pamäte, koľko potrebuje aktuálny počet pamäťových buniek, aktivovaných zo vstupu.

V 3D simulácií robotického futbalu má každý z tímov k dispozícií 11 agentov, Títo agenti nesmú medzi sebou priamo komunikovať. Iba jeden z agentov môže poslať správu serveru raz za dva cykly pričom ostatní agenti ju príjmu nasledujúci cyklus. Zahraničný tím Apollo 3D, ktorý nejeden krát vyhral v celosvetovej súťaži Robocup využíva následujúci typ komunikácie medzi agentami. Veľkosť správy je limitovaná na 20 bajtov (obr. 5). Každý bajt môže kódovať jeden znak z ASCII pričom nie všetky znaky z ASCII môžu byt použité.



Obr. 5: Hash funkcia

- 1 a 2 bajt reprezentujú identifikáciu tímu
- 3 bajt reprezentuje číslo hráča ktorý správu vytvoril
- 4 a 5 bajt informujú o stave hráča ktorý správu zasielal (hráč padol / hráč vidí loptu ...)
- 6 až 9 bajt informuje o pozícií lopty z pohľadu hráča tvoriaceho správu
- 10 až 13 bajt informuje o pozícií hráča z jeho vlastného pohľadu
- 14 až 18 bajt popisuje úlohy zvyšných členov tímu z pohľadu odosielajúceho hráča
- 19 a 20 bajt tvorí rezervu

Zdroj: `http://fei.edu.br/rcs/2014/TeamDescriptionPapers/SoccerSimulation/Soccer3D/robocanes_TDP.pdf`

4.13 Zahraničný tím BahiaRT

Posledná úprava	Juraj Šimek
Platné od	27. október 2014
Poznámky	

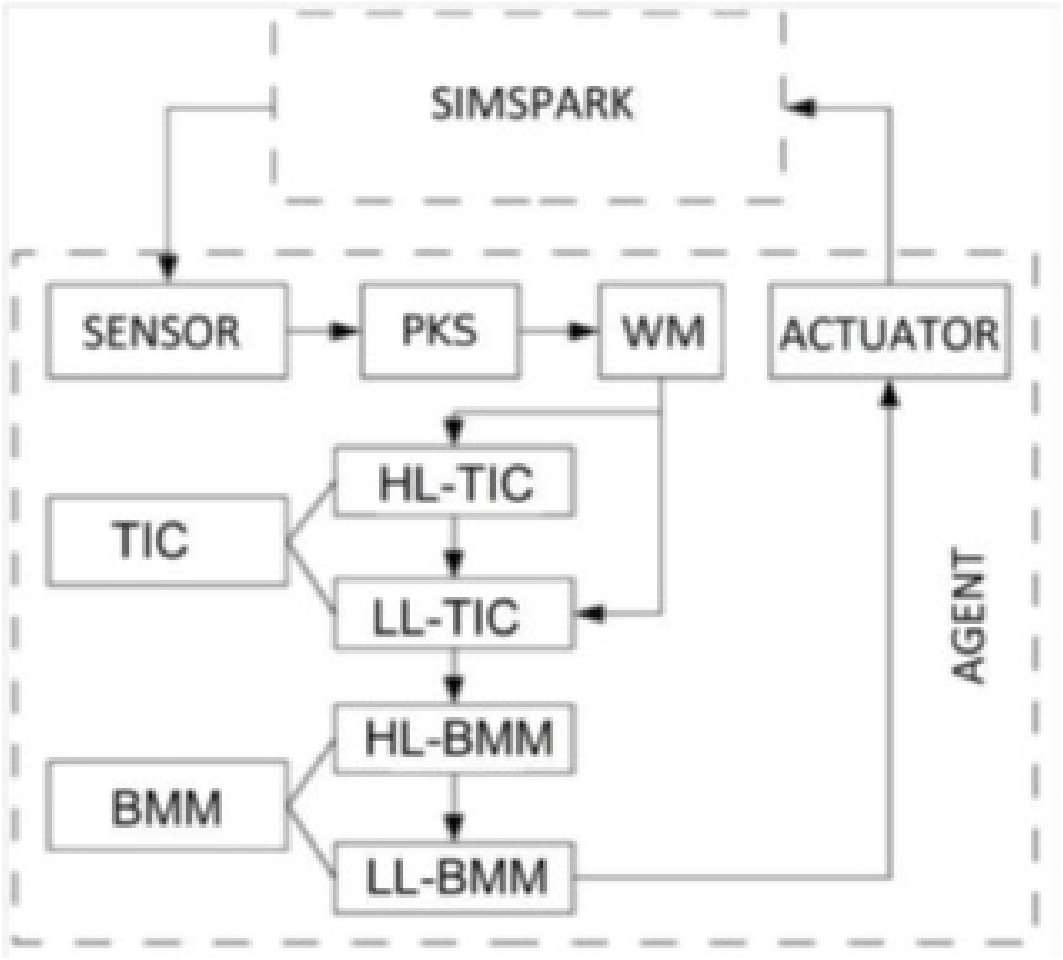
Tím s týmto názvom súťažil na RoboCup-e prvý krát v roku 2014, no jeho história je hlbšia. Roky pred tým súťažil pod názvom Bahia3D, čo bol tím, ktorý bol súčasťou skupiny študujúcej umelú inteligenciu s názvom BRT (Bahia Robotics Team). Najprv (v roku 2009) sa tím rozhodol použiť už existujúci kód použitím knižnice libbats tímu Little Green Bats, ktorá mala mnoho metód pre prácu s vektorovou algebrou a spoľahlivé komunikačné metódy. Keďže ale tím Little Green Bats neaktualizoval knižnicu na nový model videnia, neskôr začali vývojom vlastnej architektúry a knižníc. V roku 2009 skončili na 16 mieste. Pri analýze kopov do lopty čerpali z práce tímu Bahia2D (tiež časť BRT), kde bránka bola rozdelená na zóny a pre každú zónu sa počítala pravdepodobnosť úspechu skórovať. Agent sa rozhodoval na základe analýz. Použitý bol hierarchický behaviorálny model, kde je správanie robota určené postupnosťou krokov, ktoré sa môžu ďalej deliť [19].

V roku 2010 používali pohyby adaptované z logov získaných pomocou Microsoft Robotic Studio (MSRS), ktoré má predefinovaných množstvo pohybov. Vytvorili pohyby, ktoré odosielajú uhlové rýchlosti na server každých 0.02s. Kvôli aproximácii diskrétnych derivácií však bol v dátach šum, no po vyladení časových radov uhlových rýchlostí Nao kráčal stabilnejšie [8].

V roku 2011 sa tím rozhodol zlepšiť základné skilly robota. Implementovali jednoduchý 2D kinematický model. V scilabe tak spúšťali tieto modely generujúc XML skripty s uhlovými rýchlosťami kĺbov, ktoré sa zúčastňujú na pohybe. Tieto skripty spúšťali potom pomocou rozhrania v C++. Kinematický model pozostával z programu P, ktorý je matica n x m, kde m je počet zúčastnených kĺbov a n počet cyklov pohybu. Spúšťací program následne čítal riadky a generoval správy pre server. V každom kroku kontroloval odchýlku, snažil sa predpovedať nasledovnú odchýlku (tri druhy aproximácie), pričom posielal spätnú väzbu ďalšiemu kroku a tak sa snažil korigovať pohyb pre prípadný šum (nerovnosť terénu atď.) [8].

V roku 2013 tím predstavil novú architektúru. Pre úplnosť dokumentácie dopĺňame opis tejto architektúry z opisu tímu Gitmen v roku 2013, nakoľko TDP (Team Description Paper) tímu Bahia3D z roku 2013 už nie je na internete dostupný. Na obrázku 6 je táto architektúra názorne predstavená a skladá sa z nasledovných komponentov: *Simspark* (simulačný server), *Sensor* (poskytuje informácie o stave agenta), *Actuator* (posiela informácie s požiadavkami na zmenu stavu agenta na server), *PKS* (Perception Kinematic State – uchováva informácie o pohybe objektov v hre), *WM* (World Model – nespracované dáta v podobe, v akej prišli zo servera a ich očistená verzia, ktorá poskytuje lepšie informácie o okolitom svete), *TIC* (Tactical Intelligence Center – rozhoduje o tom, akú akciu má agent vykonať. Rozhoduje sa na základe agentovej role – útočník, brankár, obranca), *HL-TIC* (High Level TIC – analyzuje, čo robiť), *LL-TIC* (Low Level TIC – určuje sekvenciu pohybov, ktorú treba vykonať na základe presnosti a času vykonania), *BMM* (Body Movement Manager – vykonáva pohyby požadované vrstvou TIC), *HL-BMM* (High Level BMM – proces vykonávajúci kombináciu LL-BMB za účelom uvedenia agenta do stavu požadovaného LL-TIC), *LL-BMM* (Low Level BMM – nízkoúrovňové pohyby). Architektúra je modulárna a v budúcnosti tím uvažoval pridať vrstvu stratégií. Tím skončil v roku 2013 na treťom mieste.

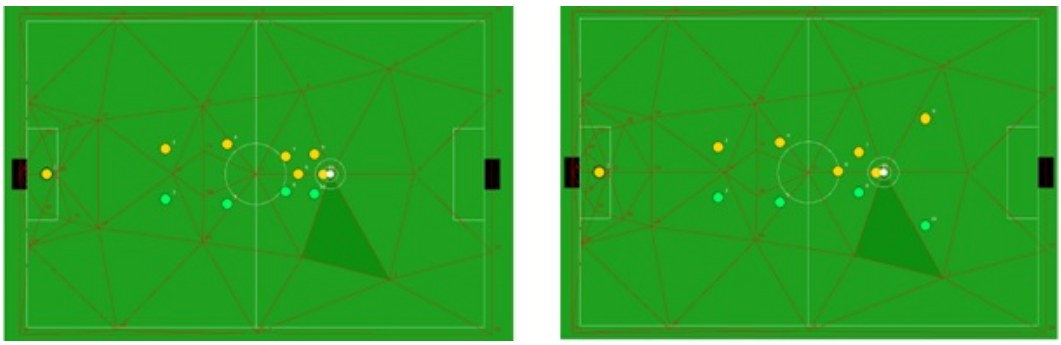
Po roku 2013 sa skupina sústredila na koordináciu medzi agentmi a plánovanie. Po RoboCupe 2013 vstúpili do partnerstva s FC Portugal a nad týmito tímami vytvorili základný tím FCPBase. Od FC Portugal čerpajú lepšie low skilly. V modeli sveta (komponent WM – World Model) je množstvo informácií užitočných pre rozhodovanie. Základný pohyb v FCPBase je vždy dopredu. Nie sú možné úkroky, čo robota spomaľuje oproti ostatným tímom. FCPBase má množstvo skriptov na kopanie, ktoré však neboli spúšťané a nie sú veľmi úspešné, ani keď sa ich spúšťanie povolí. Agent však vie obchádzať prekážky a sledovať špecifickú formáciu, no len sa snaží prejsť loptou za súperovu bránkovú čiaru ale nespúšťa skripty na kopanie. Hlavné správanie v základnom tíme FCPBase bolo teda také, že robot sa pokúšal zobrať súperovi loptu a zaniesť ju za bránkovú čiaru súpera, no neskóruje. Na vytváranie rozličných formácií tím BahiaRT použil nástroj MatchFlow. BahiaRT teda používa funkcionalitu z FCPBase. Vylepšili však model chôdze a stratégií. Aktuálne pohyby sú založené na kontrole rovnováhy a výpočte pozície každého kĺbu. Agentovi implementovali i nahrávky ostatným agentom. Nahrávku však môže zachytiť súper, preto sa spúšťa len keď je tím v súpe-



Obr. 6: Architektúra robota tímu BahiaRT od roku 2013.

rovej polovici ihriska, kde neohrozuje tak veľmi bránu. Na to, aby prihrávka bola úspešná, agent musí kopnúť správnym smerom na správnu vzdialenosť. Agent vyberá vhodný kop na to, aby kopol loptu čo najbližšie k vybratému spoluhráčovi. Aby nezachytili nahrávku súperi, je dôležité, aby každý agent vedel, či je vôbec možné niekomu nahrať. Ak hráči vyhodnotia, že je možné, aby ich spoluhráč nahral, zmení sa formácia tak ako to znázorňuje obrázok 7.

Dvaja útočníci sa presunú dopredu a stávajú sa možnými cieľmi prihrávky. Komunikačný protokol využíva round-robin algoritmus založený na



Obr. 7: Realizácia taktiky v prípade, že je možné prihrať loptu.

čase hry a počte hráčov. Pomocou správy tím posiela číslo druhého najbližšieho hráča k lopte a pozíciu, v ktorej sa dostane ku nahrávke. Táto správa sa replikuje všetkým agentom vrátane toho, ktorý má loptu. Ak hráč, ktorému sa prihráva čaká a za určitý čas neobdrží žiadnu správu od toho, ktorý mu nahráva, považuje prihrávku za neúspešnú [18]. Tím v roku 2014 napokon skončil na piatom mieste.

Z pohľadu nášho tímu je zaujímavá architektúra robota. Ako možno vidieť, architektúra Jima sa v niektorých oblastiach podobá architektúre robota tímu BahiaRT. Modul TIC predstavuje taktickú vstvru v Jimovi. Oproti tímu BahiaRT má Jim implementovaný aj modul spravujúci stratégie, ktorého funkčnosť však nie je ešte implementovaná dostatočne. Modul LL-BMM sa podobá na vrstvu low skillov, ktoré používa Jim a HL-BMM zase high skillom (HighSkillPlanner). Zaujímavé sú aj vylepšenia po roku 2013. Agent tímu BahiaRT využíva dynamické pohyby, čo v prípade Jima chýba a to sa negatívne odzrkadľuje na jeho stabilite. Preto je vhodné implementovať do Jima model, ktorý by bol schopný vykonávať dynamické pohyby a vedel by si poradiť s nerovnosťami terénu. Pomôcť by mohol nejaký vhodný kinematický model. O kinematike (či už doprednej alebo inverznej) robota NAO (práve tento robot je simulovaný pomocou simsparku) možno nájsť v článkoch [10] [9]. Zaujímavé sú aj nahrávky a kopnutia. Ak má Jim kopať na bránu, bude ju potrebné rozdeliť na viaceré segmenty, pre ktoré sa bude vyhodnocovať pravdepodobnosť skórovania, čo tím Bahia3D spravil už v roku 2009 a implementovať kop lopty vzduchom. Kop vzduchom umožní aj kopnutia na dlhé vzdialenosti. O kopoch na dlhé vzdialenosti hovorí napríklad

článok [2]. Taktika, ktorú tím používa na prihrávky sa javí byť vhodná a mohli by sme ju použiť alebo modifikovať, no na to, aby sa robotický tím držal formácie, aká je na obrázku 7, je potrebné implementovať rozhodovanie agenta na základe jeho úlohy v tíme. Projekt Jim nemá zatiaľ implementovaného ani brankára, čo by bolo vhodné v budúcnosti zmeniť. Pri prihrávkach je však dôležité najmä to, aby roboti medzi sebou komunikovali, čo Jim ešte nevie. Aj na základe analýz iných tímov je dôležité, aby mal robot čo najväčší prehľad o okolitom svete, a pokiaľ je to možné, aby bol tento prehľad rovnaký pre každého robota, čo možno dosiahnuť len vzájomnou komunikáciou medzi jednotlivými hráčmi a tým, že roboti si budú pamätať, čo sa odohralo v minulosti a pozíciu ostatných hráčov.

5 Úprava logovania

Posledná úprava	Juraj Šimek
Platné od	4. november 2014
Poznámky	

Po analýze kódu projektu Jim sme zistili, že bude potrebný jeho rozsiahly refaktoring. Jednou z oblastí, ktorú bolo treba zmeniť je logovanie. V projekte sa využívali viaceré štýly logovania udalostí. Pre uľahšenie čitateľnosti kódu a pre zjednodušenie ladenia chýb pomocou logov bolo potrebné tieto štýly zjednotiť do jedného. Používali sa nasledovné štýli logovania:

- *Logovanie pomocou štandardného výstupu.* V niektorých častiach projektu boli rôzne udalosti a ladiace výpisy vypisované priamo na konzolu pomocou System.out.println(), čo je nežiadúce. Takéto logy nemožno ukladať, len ak presmerovaním štandardného výstupu do súboru. Navyše v mnohých prípadoch boli takto vypisované epotrebné hlásenia.
- *Logovanie pomocou loggera v balíku sk.fiit.jim.log.* Tento balík slúžil na vypisovanie logov. Jeho analýzov sme zistili, že definuje niekoľko logovacích typov, ktoré slúžia na vypisovanie informácií, ktoré možno zoskupovať podľa toho, čoho sa týkajú. Tieto typy sme sa rozhodli zachovať v zjednotenom loggeri.
- *Logovanie pomocou loggera JavaAPI.* Nakoniec sa v poslednej dobe v projekte používalo logovanie pomocou štandardného API jazka Java.

Rozhodli sme sa preto vytvoriť nový logger, ktorý bude priamo využívať výhody loggera v Java API a zároveň bude podporovať aj používateľsky dodefinované typy logov pre zoskupovanie informácií, ktoré sa týkajú určitých oblastí v projekte Jim.

5.1 Logger

Keďže logovanie v projekte Jim nie je jednotné, rozhodli sme sa ho upraviť a vytvoriť nový logger, ktorý sa bude používať v novom kóde, ktorý bude do projektu doplnený. Napokon sme refaktorovali i pôvodný kód projektu Jim a prepísali logovanie na nový model logovania.

Posledná úprava	Juraj Šimek
Platné od	4. november 2014
Poznámky	

5.1.1 Analýza

V projekte Jim sa používa niekoľko metód logovania. Tím Gitmen používal na logovanie štandardný logger z Java API. Na mnohých miestach sa nachádzajú výpisy priamo na konzolu pomocou System.out.println(), čo je nežiaduce. V projekte sa nachádza i logger logujúci dodefinované udalosti v balíku sk.fiit.jim.log. Cieľom je zjednotiť rôzne metódy logovania do jednej.

5.1.2 Návrh

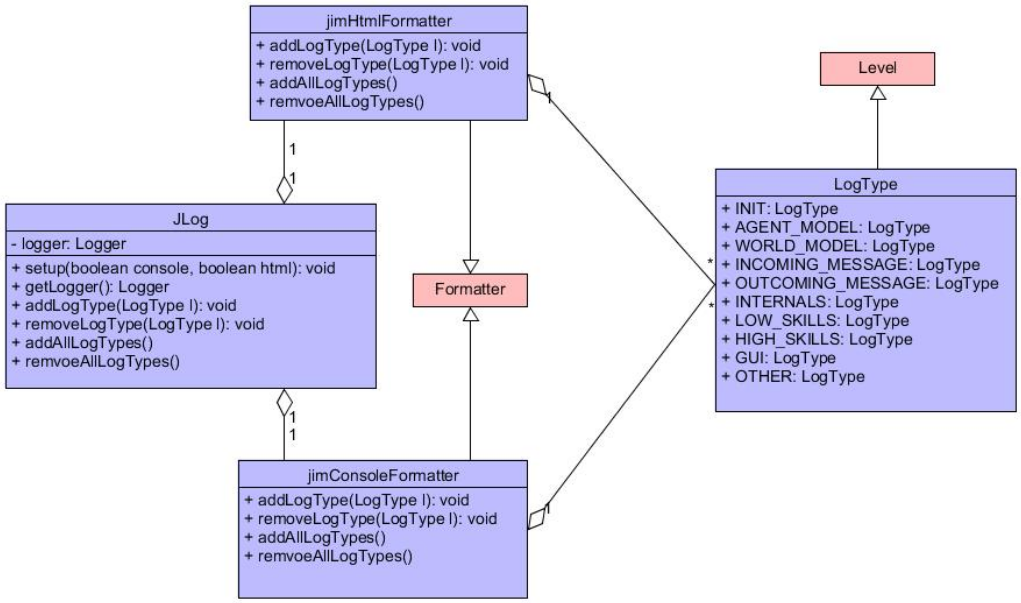
Jednotný Logger bude používať štandardný logger z JavaAPI. Logger definovaný v sk.fiit.jim.log bude odstránený, no dodatočné logovacie typy budú zakomponované do jednotného loggera, ktorý nakoniec nahradí balík sk.fiit.jim.log. Tieto typy bude predstavovať trieda LogType. Logovať bude možné do HTML súboru alebo na konzolu. Postrajú sa o to triedy JimConsoleFormatter a JimHtmFormatter odvodené od triedy Formatter, ktorá formátuje výstup loggera. Na obrázku 8 vidno navrhovanú architektúru nového loggera.

5.1.3 Implementácia

Logger používa ako základ štandardný logger z JavaAPI. Ten je obalený triedou JLog. Tu možno pomocou statickej metódy získať logger volaním getLogger(), no predtým musí byť konfigurovaný volaním JLog.setup(). Logger môže logovať aj na konzolu, aj do HTML tabuľky. Na formátovanie výstupu do konzoly a do HTML súboru sa používajú triedy JimConsoleFormatter a JimHtmlFormatter, ktoré sú odvodené od triedy Formatter slúžiacej na formátovanie výstupu loggera. Výstup v HTML znázorňuej obrázok 9.

Logger je rozšírený o nové typy v súbore LogType. Konkrétne ide o tieto typy:

```
LogType.INIT
LogType.AGENT_MODEL
LogType.WORLD_MODEL
LogType.INCOMING_MESSAGE
```



Obr. 8: UML diagram architektúry nového loggera používaného v projekte Jim.

Tue Oct 21 11:11:46 CEST 2014

Loglevel	Time	File	Log Message
FINEST	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
FINER	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
FINE	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
INFO	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
WARNING	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
SEVERE	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
INCOMING_MESSAGE	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.
GUI	okt 21,2014 11:11	sk.fiit.jim.log.LogTestMain : main()	Lorem ipsum dolor sit amet, consectetur adipiscing elit.

Obr. 9: HTML výstup nového loggera.

```
LogType.OUTCOMING_MESSAGE
LogType.LOW_SKILL
```

```
LogType.HIGH_SKILL
LogType.GUI
LogType.TACTIC
```

Nové typy možno ľubovoľne pridávať do triedy LogType. Triedy JimConsoleFormatter a JimHtmlFormatter majú zoznam týchto typov, ktoré sa logujú navyše oproti štandardným logovacím levelom. V prípade, že chceme logovať aj tieto typy, je nutné použiť metódy JLog.addLogType(LogType l), ktorá pridá daný level do zoznamu. Všetky levely definované v LogType možno pridať pomocou JLog.addAllLogTypes(), pričom sa netreba starať dopĺňaním kódu inde ako pridaním nového typu do LogTypes, nakoľko metóda využíva reflexiu z JavaAPI 10.

```
public void addAllLogTypes() {
    Field[] f =  LogType.class.getFields();
    for(int i = 0; i < f.length; i++)
    {
        Object o = new Object();
        try {
            o = f[i].get(o);
        } catch (IllegalArgumentException | IllegalAccessException e) {
            // TODO Auto-generated catch block
            e.printStackTrace();
        }
        if (o instanceof LogType) {
            levels.add((LogType) o);
        }
    }
}
```

Obr. 10: Metóda addAllLogTypes().

5.1.4 Testovanie

Keďže logger používa štandardný logger z JavaAPI, testovanie nebolo rozsiahle. Boli manuálne overené rôzne konfigurácie loggera a jeho výstupy boli porovnané s očakávaným výstupom. Počas testovania sme chybu neodhalili.

5.1.5 Používateľská dokumentácia

Používateľská dokumentácia k loggeru sa nachádza v Prílohe B 31.

5.2 Úprava logovania pomocou nového loggera

Posledná úprava	Juraj Šimek
Platné od	17. november 2014
Poznámky	

V tejto kapitole opisujeme ako sme upravovali logovanie v projekte Jim na jeden jednotný logger, ktorého implementácia je opísaná v kapitole Logger 5.1. Vytvorenie jednotného loggera bolo dôležité pre vyčistenie kódu, čo viedlo k zvýšeniu kvality kódu. Na dosiahnutie toho bolo potrebné prepísať logovanie v celom projekte do rozhrania nového loggera.

5.2.1 Analýza logovania v projekte Jim

Projekt Jim využíva rôzne metódy logovania. Na projekte pracovalo pred nami veľa tímov, čo viedlo k tomu, že logovanie nebolo jednotné, nakoľko si každý tím zvolil logovanie pomocou metódy, aká mu najviac vyhovovala. Takto nie sú logy umiestené na jednom mieste, čo výrazne sťažuje ich použitie pri ladení a hľadaní chýb. V projekte sú využívané tieto metódy logovania:

- *Logovanie pomocou štandardného výstupu*: V projekte sa na mnohých miestach logujú udalosti pomocou funkcie System.out.println() na štandardný výstup. Takéto výpisy sa neukladajú do žiadneho súboru a tak prístup k nim po vykonaní programu nie je možný, ak sme predtým sami štandardný výstup nepresmerovali do súboru. V prípade zachytávania výnimiek sa na niektorých miestach používa aj logovanie pomocou štandardného chybového výstupu použitím System.err.println(). Príklad možno vidieť v metóde getBestStrategyForSituation() triedy Selector:

```
try {
    actualSuitability =
        strategy.getSuitability(currentSituations);
} catch (Exception e) {
    System.err.println("GETTING␣BEST␣STRATEGY:"
                        +e.getMessage());
}
```

- *Logovanie pomocou loggera Java API*: Hlavne v minulom roku začalo byť používané logovanie pomocou štandardného loggera Java API. Rozhranie loggera Java API je opísané v [7]. Príklad logovania (navyše nevhodného nakoľko sa neuvádzajú žiadne informácie ale len null String) je v triede HighSkillPlanner v metóde control():

```
catch (Exception ex) {
    Logger.getLogger(Plan.class.getName())
        .log(Level.SEVERE, null, ex);
}
```

- *Logovanie pomocou logera v balíku log*: Tento logger nepoužíva rozhranie Java API. Bol vytvorený Marošom Urbancom. Definuje 3 úrovne logovania pomocou metód debug(), log() a error(). Metóda debug() slúži hlavne na ladiace výpisy, pomocou metódy error() sa zapisujú chybové hlásenia a pomocou metódy log() sa vypisujú obyčajné informácie. V type LogType sú navyše definované nasledovné typy logov:

```
public enum LogType{
    INIT,
    PLANNING,
    AGENT_MODEL,
    WORLD_MODEL,
    INCOMING_MESSAGE,
    OUTCOMING_MESSAGE,
    INTERNALS,
    LOW_SKILL,
    HIGH_SKILL,
    GUI,
    OTHER
}
```

  Tieto typy sa dajú do loggera pridávať, alebo z neho odoberať a tak sa logujú len určité typy. Typy boli prevzaté do nami vytvoreného nového loggera a rozširujú tak možnosti štandardného loggera Java API, ako je opísané v kapitole Logger 5.1. Príklad použitia logovania pomocou loggera v balíku log možno vidieť v triede Communication v metóde start():

```
catch (Exception e) {
```

```
        Log.error(LogType.INIT,
                  "Unable to connect. Cause: %s",
                  e.getMessage());
        System.exit(-1);
    }
```

Okrem toho sú mnohé logy uvádzané v projekte zbytočné. Mnohé slúžili len ako ladiace výpisy v čase programovania nejakej funkcionality a neboli vymazané. To sa týka najmä výpisov na štandardný výstup. Niektoré logy nemajú žiadny informačný charakter, nakoľko obsahujú nezmyselné hlásenia. Bude preto potrebné i vyfiltrovať nevhodné logy, aby sa tým sprehľadnil výstup loggera.

5.2.2 Realizácia

Postupne sme prešli všetky balíky projektu. Logovanie na štandardný výstup pomocou System.out.println() sme zamenili vo vhodných prípadoch za logovanie pomocou nového rozhrania nového loggera. Vo väčšine prípadov sme však takéto výpisy odstránili, nakoľko nemali význam. Prevod logovania pomocou štandardného loggera Java API nebol problém, nakoľko rozhranie nového loggera opísaného v 5.1 je rovnaké ako rozhranie štandardného loggera Java API. Stačilo len získať logger definovaný v triede JLog. Podobne sme upravili aj logovanie pomocou loggera definovaného v balíku log, ktorý bol presunutý do balíka log_old, nakoľko do balíka log sme umiestnili nový logger. Počas zmeny logovania sme starostlivo rozmýšľali nad jednotlivými logovacími hláseniami a zmazali sme tie, ktoré nemajú žiadny význam. Snažili sme sa takisto vhodne logovať výnimky tak, ako to predpisuje konvencia písania zdrojového kódu 31.

5.2.3 Zhodnotenie

Celkovo sme modifikovali približne 100 zápisov do logu pomocou starého loggera (ktorý bol presunutý do balíka log_old), približne 50 zápisov pomocou loggera Java API a približne 50 výpisov pomocou System.out.println(). Modifikácia zahŕňala úpravu starých logov do rozhrania nového loggera alebo iich zmazanie, ak boli nepotrebné. Modifikáciou sme dosiahli lepšiu kvalitu zdrojového kódu.

6 Odstránenie zakomentovaných častí kódu v projekte Jim

Posledná úprava	Miroslav Wolf
Platné od	16. november 2014
Poznámky	

V projekte Jim sa nachádza veľa zakomentovaných častí kódu, pričom nie všetky sú vhodne okomentované a tak nie je jasné na čo slúžili. Tento kód by bolo vhodné v rámci refaktoringu, dodržiavania konvencie, čitateľnosti a prehľadnosti kódu projektu vhodné otestovať a v prípade ponechania tejto časti vhodne okomentovať, prípadne aj zmazať. V rámci prvého šprintu sme daný kód najskôr analyzovali a označili si v ktorých častiach projektu sa zakomentované kódy nachádzajú (balíček, trieda). V druhom šprinte sme daný kód testovali, okomentovali alebo vymazali, pričom sme si do dokumentu zaznačili čo sme s daným kódom vykonali. Na tejto úlohe pracovali traja členovia tímu: Peter Filípek, Miroslav Wolf, Michal Segeč.

6.1 Analýza zakomentovaného kódu

Posledná úprava	Miroslav Wolf
Platné od	16. november 2014
Poznámky	

Najskôr sme celý kód prešli a zanalyzovali. Do dokumentu sme si značili kde sa zakomentovaný kód nachádza (balíček, trieda) a k danému kódu napísali krátky opis. Zoznam kódu je možné vidieť v tabuľke 2.

Tabuľka 2: Zoznam zakomentovaného kódu

Balíček	**Trieda**	**Opis**
sk.fiit.jim	Settings	Zmena ip adresy na monitor
sk.fiit.jim	Settings	Zmena ip adresy na server
sk.fiit.jim	Settings	Zmena nastavenia ID hráča
sk.fiit.jim.agent	AgentInfo.java	logovanie hodnoty premennej : IsUnderCover
sk.fiit.jim.agent	AgentInfo.java	logovanie hodnôt premenných - či hráč je blízko lopty a výpočet pozície lopty a hráča
sk.fiit.jim.agent	AgentInfo.java	logovanie polohy protihráča
sk.fiit.jim.agent	AgentInfo.java	logovanie premennej IsInRange
sk.fiit.jim.agent	AgentInfo.java	logovanie relativnej polohy lopty
sk.fiit.jim.agent	AgentInfo.java	logovanie vzdialenosti lopty a či ju má daný hráč
sk.fiit.jim.agent	AgentInfo.java	určenie polohy bránky
sk.fiit.jim.agent	AgentInfo.java	určenie polohy lopty
sk.fiit.jim.agent	AgentInfo.java	určenie pozície protihráča
sk.fiit.jim.agent	ParserToAgentModel IntegrationTest.java	vypis agentovej pozicie
sk.fiit.jim.agent	AgentInfo.java	Zisťovanie pozície bránky, hráča, lopty a logovanie zistených hodnôt
sk.fiit.jim.agent.communication	Communication	Metóda používaná na reštartovanie komunikácie medzi agentom a serverom - nefunguje správne

sk.fiit.jim.agent.communication.testframework	TestFrameworkCommunication	Zachytenie exception pri pripájaní na TestFramework monitor server
sk.fiit.jim.agent.highskill	GoToPosition.java	funkcia IsBack()
sk.fiit.jim.agent.highskill	Walk2.java	logovanie nižších skillov, ktoré sa majú vykonať
sk.fiit.jim.agent.highskill	Walk2Ball.java	logovanie nižších skillov, ktoré sa majú vykonať
sk.fiit.jim.agent.highskill	Walk2Ball.java	logovanie zóny
sk.fiit.jim.agent.highskill	Walk2BallTournament.java	logovanie zóny a pohybu, ktorý sa ide vykonať
sk.fiit.jim.agent.highskill	Walk2.java	logovanie zóny, čiže časť ihriska, kde sa nachádza agent
sk.fiit.jim.agent.highskill	Beam.java	nastavenie premennej ypos a jej pridanie do komunikačnej správy
sk.fiit.jim.agent.highskill	Walk2.java	návratová hodnota vracajúca low skill
sk.fiit.jim.agent.highskill	Kick.java	návratové hodnoty vracajúce low skills
sk.fiit.jim.agent.highskill	Beam.java	získanie aktuálneho low skillu, ak sa nejaký vykonáva
sk.fiit.jim.agent.highskill.kick	Kick	Logovanie stavu - "Horizontal range"
sk.fiit.jim.agent.highskill.kick	Kick	Logovanie stavu - "Move back"
sk.fiit.jim.agent.highskill.kick	Kick	Logovanie stavu - "Move forward"
sk.fiit.jim.agent.highskill.kick	Kick	Logovanie stavu - "Vertical range"
sk.fiit.jim.agent.highskill.kick	Kick	Logovanie stavu - premenná c1 a jej hodnota, premenná c2 a jej hodnota a premenná c3 a jej hodnota (c - case)

sk.fiit.jim.agent.highskill.kick	Kick	Logovanie stavu - premenná minX a jej hodnota a premenná rel Y a jej hodnota (rel Y - hodnota Y relativizovanej lopty)
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie a výpočet premennej maxX
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie a výpočet premennej minX
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie premennej bestLowSkill na "kick_left_faster"pre testovacie účely
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie premennej bestLowSkill na "kick_right_faster"pre testovacie účely
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie premennej bestLowSkill na "kick_right_normal"pre testovacie účely
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie premennej bestLowSkill na "kick_right_slow"pre testovacie účely
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie premennej bestLowSkill na "kick_right_template"pre testovacie účely
sk.fiit.jim.agent.highskill.kick	Kick	Nastavenie premennej bestLowSkill na "kick_step_strong_right"pre testovacie účely
sk.fiit.jim.agent.highskill.kick	KickSkills	Prázdna trieda - bez zakomentovaného kódu

sk.fiit.jim.agent.highskill.kick	Kick	Pridanie vztahov "kick_left_normal"do zoznamu anotacii pre pohyb
sk.fiit.jim.agent.highskill.kick	Kick	Pridanie vztahov "kick_left_slow"do zoznamu anotacii pre pohyb
sk.fiit.jim.agent.highskill.kick	Kick	Pridanie vztahov "kick_right_normal"do zoznamu anotacii pre pohyb
sk.fiit.jim.agent.highskill.kick	Kick	Pridanie vzťahov "kick_right_slow"do zoznamu anotácií pre pohyb
sk.fiit.jim.agent.highskill.move	MovementHighSkill	computeRelativeDistance, Vector3D position (NOT NEEDED)
sk.fiit.jim.agent.highskill.move	Walk	Funkcia logRelativeTarget() obsahuje zakomentované rozšírené info pre loger
sk.fiit.jim.agent.highskill.move	Walk	Funkcia pickLowSkill() plná zakomentovaných informácií pre loger
sk.fiit.jim.agent.highskill.move	WalkFast	Funkcia pickLowSkill() plná zakomentovaných informácií pre loger
sk.fiit.jim.agent.highskill.move	MovementHighSkill	ReplanWindow
sk.fiit.jim.agent.highskill.move	MovementSkills	Stará metóda pre getWalkSuitability()
sk.fiit.jim.agent.highskill.move	Walk	Volanie logRelativeTarget() nikde inde v kóde sa volanie nevyskytuje
sk.fiit.jim.agent.models	DynamicObject.java	funkcia pre predpovedanie pozície objektu

sk.fiit.jim.agent.models	DynamicObject.java	funkcia pre predpovedanie relatívnej pozície objektu
sk.fiit.jim.agent.models	AgentPositionCalculator.java	funkcia pre určenie pozície agenta
sk.fiit.jim.agent.models	AgentModel.java	logovanie hodnoty 3Dvectora gyroscope
sk.fiit.jim.agent.models	AgentPositionCalculator.java	logovanie údajov o agentovi
sk.fiit.jim.agent.models	WorldModel.java	logovanie údajov o lopte
sk.fiit.jim.agent.models	WorldModel.java	logovanie údajov o spoluhráčoch a protihráčoch, či sú v dosahu hráča
sk.fiit.jim.agent.models	AgentModel.java	metóda pre zistenie, či je hráč na zemi na základe accelerometra
sk.fiit.jim.agent.models	WorldModel.java	nastavenie hodnoty premennej angle
sk.fiit.jim.agent.models	WorldModel.java	nastavenie hodnoty premennej predikciaBall2
sk.fiit.jim.agent.models	AgentPositionCalculator.java	nedokončený kód
sk.fiit.jim.agent.models	WorldModel.java	výpis hodnoty premennej angle
sk.fiit.jim.agent.models	WorldModel.java	výpis low skillu, ktorý sa práve vykonáva
sk.fiit.jim.agent.models	AgentPositionCalculator.java	výpočet regresie
sk.fiit.jim.agent.models	AgentModel.java	výpočet relatívneho posunu lopty
sk.fiit.jim.agent.models	AgentModel.java	získanie pozície hráča a logovanie tejto hodnoty
sk.fiit.jim.agent.models.prediction	Prophet	Kontrolný výpis času hry
sk.fiit.jim.agent.models.prediction	Prophet	Kontrolný výpis pozície lopty

sk.fiit.jim.agent.models.prediction	Prophet	Kontrolný výpis predpokladanej polohy lopty
sk.fiit.jim.agent.models.prediction	Prophet	Kontrolný výpis predpokladanej polohy súperovho hráča
sk.fiit.jim.agent.models.prediction	Prophet	Kontrolný výpis predpokladanej polohy tímového hráča
sk.fiit.jim.agent.models.prediction	Prophet	Kus starého kódu zrejme na predpokladanie polohy lopty na základe jej rýchlosti
sk.fiit.jim.agent.models.prediction	Prophet	Nastavenie premennej predikciaBall2 na hodnotu predpokladanej pozície lopty
sk.fiit.jim.agent.parsing	Perceptors	Funkcia retrieveSeenData() obsahuje zakomentovanú časť, ktorá získavala informácie pre premenné ballRelativePosition, fixedObjects a otherplayers. Získavanie informácií je vyriešené inak a vzhľadom na to funkcia pushIdAndCoordinateToData() je nepotrebná. (Overiť)
sk.fiit.jim.agent.parsing	ParserTest	Funkcia shouldSeePlayers() vnútro celej funkcie je zakomentované, pre niektoré časti v kóde už nie je ani deklarácia

sk.fiit.jim.agent.parsing	SeenPerceptor	Podmienka na úpravu dát, ktoré už v kóde neexistujú, odkazuje na triedu ParsedData a premenné teammates a opponents.
sk.fiit.jim.agent.skills	HighSkill	Kontrolný výpis názvu skončeného pohybu
sk.fiit.jim.agent.skills	HighSkill	Kontrolný výpis názvu začatého pohybu
sk.fiit.jim.agent.skills	HighSkill	Zakomentované tagy s komentármi, zrejme vhodné vymazať - nespĺňa konvenciu
sk.fiit.jim.annotation.data	XMLCreator.java	premenná typu StreamResult pre definovanie výstupného súboru transformácie
sk.fiit.jim.anottation.gui	Window	Zakomentovaný kus kódu bez popisu - zrejme ide o testovací kód ktorý je už nepotrebný
sk.fiit.jim.decision.tactic	Goalie	Funkcia run() vnútro celej funkcie zakomentované, funkcia nie je kompletná, chýba Planner. Funkcia by mala do Planneru vkladať pozície, kam sa hráč bude presúvať.
sk.fiit.jim.decision.tactic.attack	AttackLeft	Nastavenie premennej meX - x-ová pozícia hráča

sk.fiit.jim.decision.tactic.attack	AttackLeft	Nastavenie premennej meY - y-ová pozícia hráča
sk.fiit.jim.decision.tactic.attack	AttackLeft	Povodne nastavenie súradníc lopty
sk.fiit.jim.decision.tactic.attack	AttackLeft	Povodne nastavenie súradníc lopty
sk.fiit.jim.decision.tactic.attack	AttackMid	Povodne nastavenie súradníc lopty
sk.fiit.jim.decision.tactic.attack	AttackMid	Povodne nastavenie súradníc lopty
sk.fiit.jim.decision.tactic.attack	AttackRight	Povodne nastavenie súradníc lopty
sk.fiit.jim.decision.tactic.attack	AttackRight	Povodne nastavenie súradníc lopty
sk.fiit.jim.decision.tactic.attack	AttackLeft_debug	Zakomentovaný switch - case - ktorý vykonával a logoval pohyby
sk.fiit.jim.decision.tactic.defense	DefendPosition.java	kontrolná podmienka, či sa lopta nachádza na území protivníkov
sk.fiit.jim.gui	ReplanWindow	reloadAllBtnActionPerformed(evt); funkcia sa v kóde nenachádza, možno to je pre niekoho iba ako komentár.
sk.fiit.jim.gui	ReplanWindow	reloadXmlBtnActionPerformed(evt); funkcia sa v kóde nenachádza, možno to je pre niekoho iba ako komentár.
sk.fiit.jim.gui	ReplanWindow	replanBtnActionPerformed(evt); funkcia sa v kóde nenachádza, možno to je pre niekoho iba ako komentár.
sk.fiit.jim.tests	GoalieTestCase	Trieda je celá zakomentovaná. Pravdepodobne sa jedná o test konkrétnej situácie. Trieda obsahuje aj init a ďalšie funkcie, ktoré tomu nasvedčujú.

sk.fiit.jim.tests	TestJim	Trieda je celá zakomentovaná. Pravdepodobne sa jedná o test na PlanningScript.
sk.fiit.jim.tests	GoalieTestCaseTest	Trieda je celá zakomentovaná. Pravdepodobne sa jedná o triedu určenú na spustenie testu, ktorý je v triede GoalieTestCase. Trieda obsahuje svoj main, kde sa nastavuje test.
sk.fiit.jim.tests.other	LogTest	Zakomentované volanie metódy ässertThat"ktorá zrejme slúžila pre overenie čo sa zapisovalo na výstup
sk.fiit.jim.tests.other	LogTest	Zakomentované volanie metódy ässertThat"ktorá zrejme slúžila pre zistenie čo je zapísané v súbore

6.2 Realizácia mazania zakomentovaného kódu

Posledná úprava	Miroslav Wolf
Platné od	16. november 2014
Poznámky	

Pri prechádzaní zakomentovaného kódu projektu Jim boli odstránené všetky testovacie výpisy, nastavovania premenných a hodnôt pre testovacie účely a logovanie stavov, ktoré boli zbytočné. Niektoré logy stavov boli ponechané a prepísané pre nový loger. Boli vymazané aj niektoré celé triedy a to:

- OldReplanWindow z balíčka sk.fiit.jim.gui - táto trieda bola nahradená novou triedou
- TestJim z balíčka sk.fiit.jim.tests - trieda sa nikde nepoužívala a nefungovala
- KickSkills z balíčka sk.fiit.jim.agen.kickskill.kick - trieda bola prázdna a nikde sa nepoužívala
- Prophecy z balíčka sk.fiit.jim.agent.models.prediction - triedu využívala iba zakomentovaná funkcia calculateBallPosition(), ktorá bola spolu s touto triedou vymazaná

Okrem toho bol vytvorený nový balík sk.fiit.jim.decision.tactic.oldgoalie, pre uchovanie starého brankára, do ktorého boli presunuté nasledujúce triedy:

- Goalie - presunuté z sk.fiit.jim.decision.tactic
- GoalieTest - presunuté z sk.fiit.jim.tests
- GoalieTestCase - presunuté z sk.fiit.jim.tests
- GoalieTestCaseTest - presunuté z sk.fiit.jim.tests

7 Odstránenie balíka sk.fiit.jim.garbage

Posledná úprava	Juraj Šimek
Platné od	16. november 2014
Poznámky	

V projekte Jim sa nachádza balík sk.fiit.jim.garbage, ktorý združuje nepotrebné balíky, ktorú sa už ďalej nepoužívajú. Tento balík vytvoril tím Gitmen a presunuli do neho všetky balíky, ktoré po prepísaní častí kódu agenta z Ruby do Javy nie je potrebné ďalej používať. Balíky však neboli kompletne odstránené, nakoľko neboli odstránené závislosti medzi nimi a inými balíkmi, ktoré sa v Jimovi používajú. Zo stretnutia s tímom Gitmen sme sa dozvedeli, že mali v pláne balík vyhodiť, ale neostal im na to čas. Pre zvýšenie čitateľnosti kódu v rámci refaktoringu bude preto tento balík odstránený a odstránené budú aj všetky závislosti.

7.1 Analýza jednotlivých podbalíkov balíka sk.fiit.jim.garbage

Posledná úprava	Juraj Šimek
Platné od	16. november 2014
Poznámky	

- sk.fiit.jim.garbage.build: Tento balík slúžil na zostavovanie spustiteľného kódu agenta v časoch, keď jeho súčasti boli okrem Javy implementované aj v Ruby. Balík už nie je potrebný, v projekte je však stále používaný v sk.fiit.jim.AllTests.java.
- sk.fiit.jim.garbage.plan: Balík združoval súbory pre plánovač využívajúci súčasti implementované v Ruby. Po prepísaní do Javy nie je potrebný, no stále sa používa v súboroch sk.fiit.jim.Settings.java, sk.fiit.jim.agent.highskill.runner.HighSkillPlanner.java a sk.fiit.jim.communication.CommunicationThread.java.
- sk.fiit.jim.garbage.code_review: Tento balík obsahuje anotácie s poznámkami k úprave kódu. Tieto anotácie sa používajú takmer v celom projekte kde slúžia na značkovanie kódu.

7.2 Realizácia odstránenia balíka

Posledná úprava	Juraj Šimek
Platné od	16. november 2014
Poznámky	

Pri odstraňovaní sk.fiit.jim.garbage.build bolo potrebné odobrať zo súboru sk.fiit.jim.AllTests.java importy viažuce sa na tento balík a testy, ktoré slúžili na jeho testovanie. Potom bolo možné balík odstrániť. Pri odstraňovaní sk.fiit.jim.garbage.plan bolo treba zo súboru sk.fiit.jim.Settings.java odstrániť importy odkazujúce na tento balík a riadok, ktorý nastavoval meno triedy plánovača PlanTactic nachádzajúci sa v odstraňovanom balíku. Trieda CommunicationThread.java bola vyhodená, nakoľko už nebola ďalej používaná. V triede HighSkillPlanner.java stačilo odstrániť nepoužívané importy odkazujúce na odstraňovaný balík, potom bolo možné tento balík odobrať. Balík sk.fiit.jim.garbage.code_review bol ponechaný a premenovaný na sk.fiit.jim.code_review, nakoľko sa anotácie môžu zísť pri posudzovaní ostatných častí kódu.

8 Úprava pôvodného zdrojového kódu podľa konvencií písania zdrojového kódu

Posledná úprava	Juraj Šimek
Platné od	19. november 2014
Poznámky	

8.1 Analýza

V projekte Jim neboli dodržiavané jednotlivé konvencie písania zdrojového kódu. Bolo to zapríčinené tým, že každý tím písal podľa vlastných konvencií. Práve kvôli tomu sme vytvorili metodiku písania zdrojového kódu 31, pomocou ktorej sme analyzovali pôvodný zdrojový kód. Zistili sme mnohé porušenia pravidiel, ktoré opisuje metodika. Mnoho metód a tried nemá uvedené dokumentačné komentáre. Nie sú správne uvádzané medzery medzi jednotlivými časťami príkazov, čo zneprehľadňuje zdrojový kód (napr `x=0` namiesto `x = 0`). Bloky nie sú formátované správne. V mnohých prípadoch telo cyklu alebo podmieneného vetvenia nie je uvedené v bloku, čo môže viesť ku chybám v prípade, že sa niečo bude v týchto častiach kódu meniť. V projekte chýbajú aj komentáre ku jednotlivým blokom kódu a je tak ťažké zistiť na čo slúžia. Rozhodli sme sa teda prepísať projekt pomocou uvedenej metodiky a túto metodiku potom verejne publikovať pre ostatné tímy, ktoré budú pracovať na projekte po nás.

8.2 Realizácia a zhodnotenie

Postupne sme prešli všetky balíky projektu Jim. Hľadali sme porušenia pravidiel dané metodikou písania zdrojových kódov a snažili sme sa ich upraviť. Tam, kde to bolo možné, sme uviedli dokumentačné komentáre. Pri mnohých metódach a to však nepodarilo lebo nebolo jasné, na čo slúžia. Telá podmienených príkazov a cyklov sme uvádzali do blokov tam, kde v blokoch neboli. Správne sme formátovali kód, čím kód dosiahol lepšiu prehľadnosť a tým sa zvýšila aj jeho kvalita. Celkovo sme zaznamenali nárast celkového počtu riadkov z 15860 na 15938, čo má súvis práve s dopĺňaním dokumentačných komentárov a úpravou tiel podmienených príkazov a cyklov do blokov.

9 Odstránenie nepoužívaných highskillov

Posledná úprava	Juraj Šimek
Platné od	3. decembra 2014
Poznámky	

9.1 Analýza

Projekt Jim obsahuje množstvo HighSkillov, ktoré sa v ňom nepoužívajú. Fakt, že sa v ňom nepoužívajú sme zistili pomocou výstupu pluginu Unnecesary Code Detector, ktorý ukázal, že mnohé highskilly nie sú nikde volané. Highskilly sa nachádzajú v balíku sk.fiit.jim.highskill. CodeDetector označil za nepoužívané tieto HighSKilly:

- `CyclicHighSkill` – cyklicky vykonáva lowskill, ktorý mu je zadaný ako parameter.
- `FormationHelper` – pomocná metóda getHighSkillToGoToFormation() pre ostatné highskilly, ktoré však nie sú používané.
- `FreeRide` – highskill pre disciplínu FreeRide v rámci RoboCup turnaja na škole. Tento HighSkill je nahradený v Taktikách.
- `GoTo` – chôdza agenta na určitú zadanú pozíciu.
- `GotoBall` – chôdza agenta smerom ku lopte.
- `GoToPosition` – chôdza agenta na určitú bližšie špecifikovanú pozíciu (podobne ako GoTo). Ide však o modifikovanú chôdzu.
- `Kick` – kopnutie do lopty špecifikovaným semrom.
- `KickAccuracyTournament` – kop loptou na presnosť, čo sa používa na turnaji RoboCupu na našej škole. Tento HighSkill je nahradený v taktikách.
- `KickTournamennt` – turnajový HighSKill pre kopnutie lopty.
- `LinkedHighSKill` – cykylicky vyberá lowskilly z preddefinovaného zoznamu lowskillov.

- `LowSkillHighSkill` – vykonáva vybratý lowskill, pričom kontroluje to, či daný lowskill beží. Ak beží, tak vracia null namiesto lowskillu.
- `Trajectory` – zo zdrojového kódu nie je jasné, na čo highskill Trajectory slúži.
- `Turn` – otáčanie agenta ku nejakej zadanej pozícii.
- `TurnToPosition` – vylepšené otáčanie agenta ku nejakej zadanej pozícii.
- `VedenieLopty` – vedenie lopty agentom.
- `Walk2` – chôdza ku nejakej zadanej pozícii (ide o vylepšenú chôdzu).
- `Walk2Ball` – chôdza ku lopte (vylepšená).
- `Walk2BallTournament` – chôdza ku lopte používaná na turnaji.
- `WalkNew` – nový vylepšený model chôdze.
- `Walk` – zastaraný model chôdze.

Ako vidno už z tejto analýzy mnohé highskilly aj medzi týmito nepoužívanými highskillmi sú duplicitné. Keď nejaký tím naprogramoval nový highskill, starý nevymazal, čím sa dostal aj do projektu Jim, keď minulý rok boli highskilly reimplementované z Ruby do Javy. Highskilly a s nimi súvisiace triedy z balíka sk.fiit.jim.highskill, ktoré sa však v projekte používajú sú tieto:

- `AbstractHighSkill` – identifikácia toho, že v odvodenej triede ide o HighSkill.
- `Beam` – začiatočné umiestnenie agenta na štartovaciu pozíciu.
- `BeamHighSkill` – obaľuje triedu Beam a spôsobuje, že agent sa po pripojení na server umiestni na určitú pozíciu hracej plochy.
- `DefaulHighSkill` – defaultný highskill vykonávajúci lowskill zadaný konštruktorom. Slúži hlavne na testovacie účely.
- `GetUp` – highskill implementujúci vstávanie agenta zo zeme.
- `Localize` – lokalizácia hráčov v závislosti na polohe lopty.

Ďalšie používané highskilly sú v balíku sk.fiit.jim.agent.highskill.kick:

- `Kick` – kopanie do lopty zadaným smerom.
- `KickHighSkill` – highskill pre kopanie do lopty, ktorý vyberá vhodné štýly kopov.

Kopanie do lopty ešte nie je dokončené, kvalita kopu nie je vysoká a v budúcnosti sa budú zrejme pridávať ďalšie highskilly umožňujúce agentovi vykonávať lepšie a presnejšie kopy.

Posledným balíkom reprezentujúcim používané highskilly je balík sk.fiit.jim.agent.highskill.move, ktorý slúži na pohybovanie agenta. Dôležité sú predovšetkým triedy:

- `Walk` – abstraktná trieda pre implementovanie nejakého štýlu chôdze.
- `WalkFast` – rýchla chôdza na stanovenú pozíciu.
- `WalkMedium` – stredná schôdza na stanovenú pozíciu.
- `WalkSlow` – pomalá chôdza na stanovenú pozíciu.
- `MovementHighSkill` – trieda vyberajúca druh chôdze na základe ohodnotenia vhodnosti daných druhov chôdze.

Na základe tejto analýzy sme sa rozhodli presunúť nepoužívané highskilly do nejakého priečinku mimo hlavný projekt, aby sme ich zachovali pre budúce tím, ktoré by mohli v nich hľadať inšpiráciu pre tvorbu vlastných highskillov. Ďalšia analýza ukázala, že nepoužívané highskilly využívajú mnohé z metód triedy AgentInfo, ktoré už nevyužíva žiadna iná trieda. Rozhodli sme sa preto tieto metódy odstrániť a tak túto triedu výrazne sprehľadniť.

9.2 Realizácia

Menované nepoužívané highskilly boli odstránené z projektu a zdrojové súbory boli presunuté do priečinka *OLD FILES* mimo hlavný projekt agenta. Následne bol spustený plugin Unnecessary Code Detector nad triedou AgentInfo, ktorého výstup nám ukázal nové nepoužívané metódy. Tieto metódy boli zmazané, ale predtým sme skopírovali pôvodnú neupravenú verziu triedy AgentInfo do priečinka OLD FILES, aby sa zachovali v prípade, že by sme sa rozhodli nejaký vyhodený highskill v budúcnosti modifikovať a použiť.

9.3 Zhodnotenie

Po vyhodení nepoužívaných highskillov sme v projekte nezaznamenali žiadne chyby. Všetky definované testy zbehli rovnako ako pred vyhodením týchto highskillov a správanie agenta v simulačnom prostredí ostalo nezmenené. Počet riadkov zdrojového kódu poklesol z 15938 po 3. šprinte na 14579 po 4. Šprinte. Odstránili sme tak okolo 2000 nepotrebných riadkov zdrojového kódu (niekoľko sme pridali v iných častiach, ale z hľadiska odstraňovania highskillov bolo zmazaných zhruba 2000 riadkov). Týmto sa výrazne zvýšila kvalita zdrojového kódu a zvýšila sa jeho čitateľnosť.

10 Úprava konfigurácie nastavení agenta

Posledná úprava	Juraj Šimek
Platné od	9. decembra 2014
Poznámky	

Keďže prechodom z Ruby do Javy sa prepísali nastavenia agenta napevno do súboru Settings.java, nebolo ich možné už načítavať zo súboru. Po každej zmene nastavení preto bolo potrebné kompilovať nanovo projekt, čo bolo nepriaznivé a preto sme sa rozhodli reimplementovať nastavenia tak, aby ich bolo možné načítať z konfiguračného súboru.

10.1 Analýza

Súbor `Settings.java` obsahuje množstvo nastavení, ktoré sú v ňom napevno zapísané. Súbor slúžil na načítanie defaultných nastavení (tie sa načítavajú aj teraz a každú zmenu v nastaveniach treba vykonať zmenou týchto defaultných nastavení) a potom sa tieto nastavenia prekryli nastaveniami zo z ruby súboru `settings.rb`. Po reimplementácii agenta do javy už nie je možné načítavať nastavenia zo súboru. Nastavenia sú uložené v mape, kde kľúče tvoria reťazcové identifikátory a hodnoty sú typu `Object`. Metóda `setDefaults()` nastavuje defaultné nastavenia a momentálne sa ako jediná používa k nastaveniu agenta. Metóda `initDecisionObjects()` inicializuje objedkty rozhodovania používané v taktikách a stratégiách. Metóda `parseCommandLine()` parsuje nastavenia z príkazového riadku a ukladá ich do mapy override, ktorou sa potom nahradí pôvodná mapa nastavení pomocou metódy `setCommandLineOverrides()`. Metóda slúži na zmenu nastavení pomocou príkazového riadku, čo používal testovací framework. Nie je vcelku jasné, prečo je táto zmena riešená takto a prečo sa rovno nemenia hodnoty v hlavnej mape nastavení. Ak chceme zmeniť nastavenia pomocou príkazového riadku, treba najskôr zavolať metódu `parseCommandLine()` a hneď po nej `setCommandLineEverrides()`. Trieda Settings.java používa statické metódy a nie je riešená ako singleton. Na získavanie hodnôt z mapy sa používajú metódy `getBoolean()`, `getInt()`, `getDouble()` a `getString()`, ktoré pretypujú objekt typu `Object` na príslušný dátový typ.

10.2 Návrh

Triedu implementujeme pomocou návrhového vzoru singleton. Nastavenia budú opäť uložené v mape. Tentoraz bude implementovaná iba jedna mapa nastavení, ktorej kľúče budú typu `String` a rovnako aj hodnoty budú typu `String`. Učinili sme tak, aby sme nemuseli meniť triedy, ktoré túto triedu používajú. Navonok preto musí ostať rozhranie triedy Settings.java nezmenené. Metódy `getBoolean()`, `getInt()`, `getDouble()` a `getString()` tak budú vracať daný objekt pretypovaním reťazcovej hodnoty z mapy nastavení (a nie pretypovaním z Object, ako tomu bolo doteraz). Metódy je nutné zachovať pre zachovanie pôvodného rozhrania. Privátna metóda `loadFromFile()` bude načítavať nastavenia zo súboru pomocou triedy `Properties` štandardného Java API. Nastavenia zo súboru sa budú vykonávať pomocou metódy `laodSettings()`. Metódu `parseCommandLine()` ponecháme, nakoľko je jedinou možnosťou, ako komunikovať s testovacím frameworkom. Treba ponechať aj metódu `initDecisionObject()`, ktorá sa bude volať po načítaní nastavení v `loadFromFile()`.

10.3 Implementácia

Funkcionalitu triedy Settings sme reimplementovali tak, ako je to uvádzané v návrhu. Privátna metóda `loadFromFile()` načíta nastavenia zo súboru pomocou triedy `Properties`. Nastavenia konfiguračného súboru (settings.properties) sú definované nasledovne:

```
# SETTINGS FILE OF JIM AGENT
#
# author Juraj Simek (Infinity - 2014)
#

# Team name
TEAM_NAME = Infinity

# Kalman filter setttings
KALMAN_USE_FILTER = true
KALMAN_DEFAULT_Q = 0.475
KALMAN_DEFAULT_R = 0.375

# Garbage Collection
```

```
RUN_GC_ON_PHASE_START = true

# Launch GUI
RUN_GUI = true

# Physical constants settings and settings of receptors
GRAVITY_ACCELARATION = 9.81
MAXIMUM_ANGULAR_CHANGE_PER_QNT = 7.0
IGNORE_ACCELEROMETER = false

# Test Framework Settings
RUN_TFTP_SERVER = true
TEST_FW_MONITOR_ENABLE = true
TEST_FW_MONITOR_PORT = 8000
TEST_FW_MONITOR_ADDRESS = 127.0.0.1
TFTP_ENABLE = false
TFTP_PORT = 3073

# Root path
#JIM_ROOT_PATH =

# Server configuration
RCSSSERVER_IP = 127.0.0.1
RCSSSERVER_PORT = 3100

# Debug tactic
DEBUG_TACTIC_ENABLE = false
DEBUG_TACTIC_NAME = DefaultTactic
```

Každá vlastnosť sa ukladá do mapy a buď sa pre ňu uloží hodnota daná súborom, alebo defaultná hodnota, ak sa v konfiguračnom súbore táto vlastnosť nenachádza:

```
settings.put("Tftp_port",
  properties.getProperty("TFTP_PORT", "3073").trim());
```

Metóda `loadSettings(String file)` načíta nastavenia zo zadaného proeprties súboru, metóda `loadSettings()` načíta defaultné nastavenia pomocou metódy `loadFromFile()`, do ktorej sa ako argument uvedie prázdny reťazec. Ak v metódach `getInt()`, `getBoolean()`, `getString()`, či `getDouble()`

žiadame hodnotu pre neexistujúci kľúč, vyhodí sa `IllegalArgumentException`. Metóda `setValue(String key, String value)` slúži na dodatočné vkladanie alebo prekrývanie nastavení v hlavnej mape nastavení. Celkovo sa logika triedy výrazne zjednodušila a rozsah zmenšil (až o 500 riadkov), pričom jej funkcionalita ostala pôvodná, rozhranie nezmenené a navyše je možné načítať nastavenia zo súboru.

10.4 Testovanie

Pre účely testovania bol vytvorený test `SettingsTest.java`, ktorý testuje vkladanie a výber dát z nastavení a to, či po načítaní nastavení z testovacieho súboru dostaneme očakávané výsledky. Reimplementovaná trieda testom prešla úspešne.

11 Sprístupnenie MediaWiki

Posledná úprava	Peter Filípek
Platné od	10. decembra 2014
Poznámky	

Vzhľadom na skutočnosť, že MediaWiki všetkých predošlých tímov, ktoré pracovali na tomto projekte boli začiatkom semestra odstavené, rozhodlo sa vytvoriť vlastnú MediaWiki s využitím dát, ktoré odovzdal predošlý tím. Úlohou bolo najprv zistiť v akom stave sú dáta od predošlého tímu a následne podľa možnosti využiť ich pri tvorbe vlastnej MediaWiki. Na úlohe pracoval Peter Filípek a Martin Vrabec.

Dáta od predošlého tímu obsahovali dump databázy ich MediaWiki a súbory samotnej MediaWiki pre server. Pre analýzu sa vytvorila na lokálnom serveri kópia tejto MediaWiki. Pri vytváraní sa zistilo, že je nutné upraviť základné nastavenia serveru pre možnosť nahrávania väčších súborov vzhľadom na to, že dump databázy predošlého tímu mal skoro 30MB, teda niekoľko násobne prevyšoval základné hodnoty servera. Po úprave servera a nahratí databázy sa ukázalo, že MediaWiki predošlého tímu musí byť aktualizovaná na novšiu verziu lebo v aktuálnej verzii nekomunikuje s novšou verziou PHP ktorá je na serveri. Po aktualizácii MediaWiki nastal problém so zobrazovaním obrázkov a niektorých častí ovládacích prvkov a odkazov na stránky. Pre odstránenie tohto problému sa nainštalovala najnovšia verzia MediaWiki a následne sa do nej nahrali len potrebné dáta od predošlého tímu. Po týchto úpravách bežala MediaWiki na lokálnom serveri bez problémov.

Upravilo sa nastavenie servera pre potreby nahratia dump databázy. Vytvorila sa nová databáza, kde sa daný dump nahral. Pre danú databázu sa vytvorili potrební používatelia s prístupom k nej, aby MediaWiki správne fungovala. Upravili sa nastavenia v MediaWiki pre komunikáciu zo serverom. MediaWiki sa nahrala na server. Pri nahrávaní na server nastal problém pri niektorých názvoch súborov, ktoré obsahovali diakritiku, museli sa upraviť, aby sa správne nahrali. Po úspešnom nahratí MediaWiki na server sa vytvoril používateľ, aby bolo možné pridávať nové dáta a prebehlo rýchle testovanie jej funkcií.

Aktuálna MediaWiki je na adrese `http://labss2.fiit.stuba.sk/TeamProject/2014/team08is-si/wiki`. Informácie na MediaWiki pochádzajú od predošlých tímov. Jediný problém, ktorý sa zatiaľ vyskytol, je pri odkazoch, ktoré

odkazujú na dokumenty, ktoré sa majú nachádzať naserveri. Vzhľadom na skutočnosť, že dané dokumenty neboli súčasťou dát získaných od predošlého tímu je pravdepodobné, že sa ich nepodarí sprístupniť.

12 Oddelenie testov od zdrojových kódov

Posledná úprava	Peter Filípek
Platné od	10. decembra 2014
Poznámky	

Vzhľadom na rozsiahlosť projektu sa rozhodlo v rámci sprehľadnenia kódu oddeliť testovacie triedy podľa Metodiky tvorby testov v projekte JIM. Testovacie triedy sa presunuli do nového zdrojového priečinka určeného pre testy. Testovacie triedy dodržujú rovnakú hierarchiu balíkov ako hlavný zdrojový priečinok. Na úlohe pracoval Peter Filípek.

Postupne sa prešli všetky balíky projektu JIM, pri prehľadávaní sa hľadali testovacie triedy JUnit testov.

V projekte JIM sa vytvoril nový zdrojový priečinok Test. V zdrojovom priečinku sa vytvorila rovnaká hierarchia balíkov ako v hlavnom zdrojovom priečinku, vytvorili sa len potrebné balíky pre testovacie triedy. Následne sa presunuli všetky testovacie triedy JUnit testov do príslušných balíkov zdrojového priečinka Test. Trieda AllTests.java, ktorá sa stará o spúšťanie vybraných testov, zostala pre potreby automatického testovania v hlavnom zdrojovom priečinku v balíčku sk.fiit.jim.

Oddelenie JUnit testov prebehlo bez väčších problémov. Po presunutí testov do nového zdrojového priečinka prebehlo lokálne spustenie všetkých testov. Výsledok testov bol rovnaký ako pred oddelením s tým rozdielom, že pri jednom teste sa odstránila chyba spôsobená pravdepodobne závislosťou s inými triedami. Po oddelení bolo upravené aj automatické testovanie pre nové umiestnenie testov.

13 Odstraňovanie warning hlásení v projekte Jim

Posledná úprava	Miroslav Wolf
Platné od	10. decembra 2014
Poznámky	

V projekte Jim sa nachádzalo mnoho warning hlásení. Kód sme zanalyzovali pomocou eclipse pluginu UCD (Unnecessary code detector), ktorý do projektu pridal ďalšie upozornenia (warningy). V rámci úlohy sme odstraňovali najmä pôvodné warningy z eclipsu, ale aj niektoré z pluginu UCD. Na úlohe pracovali traja členovia tímu: Miroslav Wolf, Michal Segeč a Martin Vrabec.

Postupne sme prechádzali všetky balíčky a do google dokumentu sme značili ktorý člen tímu ktorý balíček prechádza aby nedošlo ku kolíziám. Okrem toho sa neriešili warningy spojené s vector3 a vector3d, keďže tu ide o závažnejší problém, ktorý sa bude riešiť zrejme v rámci inej úlohy. Taktiež sme neriešili ani niektoré warningy z UCD, ktoré sa tiež riešia mimo tejto úlohy.

Pri prechádzaní warningov sme narazili najmä na nepoužité premenné, či nepoužité metódy, ktoré boli v prípade že nemali hlbší zmysel vymazané. Okrem toho sme niektorým metódam a premenným, tam kde to bolo potrebné, zmenili prístup z public na private, prípadne opačne. Väčšina warningov sa dala upraviť automaticky, pomocou navrhnutých riešení eclipsu. Veľká väčšina warningov je tiež z projektov RoboCupLibrary a TestFramework. Warningy z týchto projektov sme neopravovali.

Celkový počet warningov (v rátane RoboCupLibrary, TestFramework a warningov z UCD) pred vykonaním tejto úlohy bolo 936. Po dokončení úlohy ich ostalo 582, z ktorých prevažná väčšina je zo spomínaných projektov, ktoré sa neprechádzali a z UCD detectora.

14 Implementácia Zero moment point

Posledná úprava	Metod Rybár
Platné od	10. decembra 2014
Poznámky	

Cieľom tejto úlohy bolo implementovať metódu Zero moment point z práce Hudec [6] spomínanej v kapitole 4.3. Rovnako sa pri tabilizácií využívajú perceptory odporu na nohe, ktoré sa predtým v hráčovi Jim nevyužívali.

Hlavná funkcia pre výpočet Zero moment point sa zakomponovala do triedy *AgentModel* a ide o triedu *private void updateZeroMomentPoint()*. V nej sa počíta bod, v ktorom by mal hráč byť stabilný. Ďalšie pridané metódy do *AgentModel* boli *private void updateCenterOfMass()*, *private void updateBodyPartsPositions2()*, *private void updateFeetForce(ParsedData data)* a *public boolean isOnGround()*, ktorá nahradila pôvodnú funkciu kontrolujúcu či je hráč na zemi, kedže pri Zero moment point spôsobuje, že hráč nie je v každej fáze stabilný, ale napriek tomu nemusí spadnúť.

Tieto metódy z triedy *AgentModel* slúžia na obnovovanie informácií o hráčovi v každom cykle a teda sa dajú informácie nimi vypočítané využívať na stabilizáciu pohybov. Na otestovanie Zero moment point sa vytvoril nový HighSkill *WalkFastZMP*, ktorý je reimplementáciou triedy z Ruby, ktorú vytvoril Hudec vo svojej diplomovej práci. HighSkill sa musel vytvoriť nanovo v Jave a upraviť, kedže minuloročný tím prerobil hráča kompletne do Javy.

Rovnako bolo treba pridať z verzie hráča z diplomovej práce Hudec [6] niektoré triedy, ktoré sa v našej verzií hráča nenachádzali. Ide o triedy *ComputedValues a ComputedValue*, ktoré slúžia na uchovávanie vypočítaných hodnôt pre kĺby.

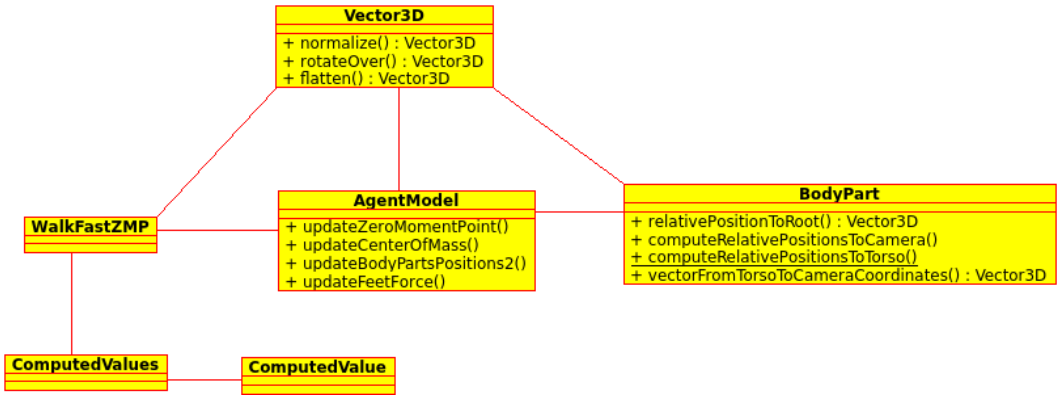
Trieda *BodyPart*, respektívne jej hlavný enum BodyPart bola rozšírená o hodnoty pre kĺby z diplomovej práce od Hudeca. Zároveň v triede pribudli metódy

```
public static Vector3D relativePositionToRoot
(BodyPart bodyPart,
Map<Joint, Double> jointAngles, Vector3D position)

public static void computeRelativePositionsToCamera
(Map<Joint, Double> jointAngles,
Map<BodyPart, Vector3D> relPositions)
```

```
public static void computeRelativePositionsToTorso
(Map<Joint, Double> jointAngles,
Map<BodyPart, Vector3D> relPositions)

public static Vector3D vectorFromTorsoToCameraCoordinates
(Vector3D vector, double h1Angle, double h2Angle)
```

V triede Vector3D pribudli metódy *public Vector3D normalize()*, *public Vector3D rotateOver(Vector3D axis, double angleInRad)* a *public Vector3D flatten()*, ktoré sa využívajú pri práci s vektormi v Zero moment point.

Hlavné triedy využívané pri Zero moment point a hlavné metódy súvisiace s touto metódou sú zobrazené na obrázku 11.



Obr. 11: Diagram tried pre Zero moment point

Stabilizácia po testovaní funguje, ale vyskytli sa problémy s vyberaním HighSkillu, anotáciami a plynulým prechádzaním medzi fázami, ktoré zrejme nastali kvôli zmenám uskutočneným po prechode z Ruby na Javu. Stabilizácia sa preto môže v pohyboch využívať, ale treba upraviť HighSkill *WalkFastZMP* tak, aby dokázal správne pracovať s našou verziou hráča.

15 Zámena triedy Vector3 za Vector3D

Posledná úprava	Juraj Šimek
Platné od	28. február 2015
Poznámky	

V projekte Jim, ale hlavne v projekte TestFramework sa vyskytuje veľké množstvo použití metód triedy Vector3. Trieda reprezentuje vektory v trojrozmernom priestore, no je prostredníctvom anotácie jazyka Java označená ako deprecated, čo vedie ku zbytočným varovaniam v projekte. Jej funkcionalitu úspešne prevzala a vhodne doplnila trieda Vector3D, ktorá je používaná najmä v projekte Jim. Obe triedy sú umiestnené v projekte RoboCupLibrary. Z týchto dôvodov je preto vhodné prepísať všetky projekty robotického hráča tak, aby sa používala už len novšia verzia – trieda Vecotr3D.

15.1 Analýza

Táto kapitola analyzuje podstatné rozdiely medzi triedami Vector3 a Vector3D.

15.1.1 Trieda Vector3.java

Trieda reprezentuje vektor pomocou karteziánskej súradnicovej sústavy. Trieda poskytuje tri konštruktory:

- *Prázdny konštruktor Vector3()* – vytvorí nulový vektor
- *Kopírovací konštruktor Vector3(Vector3 from)* – vytvorí nový vektor ako kópiu vektora v parametri
- *Konštruktor Vector3(double x, double y, double z)* – vytvorí nový vektor so zadanými súradnicami

Trieda obsahuje gettery a settery pre súradnice v tvare *getX()* a *setX(double x)*. Tieto metódy menia členské premenné triedy. Ďalšie metódy, ktoré poskytuje trieda Vector3 sú metódy *subtract(Vector3 b)* a *addition(Vector3 b)*,

ktoré vykonávajú vektorový rozdiel a súčet, vracajú však nový objekt, nemenia vlastnosti volajúceho objektu. Metóda *division(double x)* delí súradnice vektora daným číslom, metóda *getXYDistanceFrom(Vector3 from)* vracia vzdialenosť volajúceho vektora od vektora v argumente. Pomocou metódy *asPoint3D()* možno vrátiť súradnice vektora ako objekt triedy *Point3D*. Na záver, metóda *toString()* vracia vektor ako reťazec pre účely výpisu.

15.1.2 Trieda Vector3D.java

Trieda reprezentuje vektor pomocou karteziánskych súradníc, aj pomocou polárnych súradníc. Trieda obsahuje statické premennú *ZERO_VECTOR*, ktorá reprezentuje nulový vektor. Trieda je navrhnutá podľa vzoru immutable class a nemá žiadne konštruktory. Jediný konštruktor je uvedený ako privátny, nové objekty možno vytvárať len nasledovnými statickými metódami:

- *fromVector3(Vector3 v)* – konverzná metóda z Vector3 na Vector3D
- *cartesian(Vector2, double z)* – vytvorí trojrozmerný vektor z dvojrozmerného vektora doplneného o tretiu súradnicu
- *cartesian(double x, double y, double z)* – vytvorí vektor podľa karteziánskych súradníc
- *spherical(double r, double phi, double theta)* – vytvorí vektor podľa polárnych súradníc

Gettery a settery sú uvedené pre polárne aj karteziánske súradnice a vždy vracajú nový objekt, nemenia vlastnosti volajúceho objektu. Metódy *addX(), addY()* a *addZ()* pridávajú dĺžku k danej karteziánskej súradnici. Metódy *add(Vector3D)* a *subtract(Vector3D)* realizujú sčítanie vektorov a vracajú nový objekt. Metódy *multiply(Number)* a *divide(Number)* vykonávajú násobenie súradníc vektora. Metóda *negate()* otočí vektor, *toUnitVector()* vracia jednotkový vektor a *rotateOverX(), rotateOverY(), rotateOverZ()* vykonávajú rotácie o zadaný uhol. Metódy *crossProduct()* a *dotProduct()* vykonávajú vektorový a skalárny súčin vektorov. Trieda taktiež obsahuje metódy *asPoint3D(), getXYDistanceFrom(), toString(), hashCode()* a *equals()* pre porovnanie dvoch vektorov.

15.1.3 Porovnanie

Z hľadiska výmeny triedy Vector3 za triedu Vector3D nás zaujíma v čom sa líši trieda Vector3D od triedy Vector3. To uvádza nasledovná tabuľka.

Vlastnosť	Vector3	Vector3D
kopírovací konštruktor	má	nie je potrebný, trieda je immutable
verejný konštruktor	má	nemá
prázdny konštruktor	má	nemá
statické metódy pre vytvorenie vektorov	nemá	má, nahrázdajú konštruktory
karteziánska reprezentácia	má	má
settery a gettery	má, menia vlastnosť objektu	má, nemenia vlastnosť objektu
odčítanie vektorov	subtract()	subtract()
sčítanie vektorov	addition()	add()
delenie súradníc číslom	division()	division()
getXYDistanceFrom()	má	Má
asPoint3D()	má	má
toString()	má	má

15.2 Návrh

Postupne sa prejdú balíky projektov *Jim* a *TestFramework* a zamenia sa úseky zdrojového kódu s Vector3 na kód s použitím Vector3D. Treba si dávať však pozor na to, že tam, kde sa vektor vytváral konštruktorom sa bude teraz vytvárať pomocou statickej metódy. Tam, kde sa sčítavali vektory, treba zameniť metódu *addition()* za metódu *add()*, nakoľko obe vracajú nový objekt, správajú sa rovnako a ďalšie úpravy nie sú nutné. Pozor si však bude treba dávať pri *setteroch*, kde settery triedy Vector3 menili vlastnosti daného objektu, ale settery triedy Vector3D vracajú nový objekt. Treba preto vrátený objekt vhodne priradiť. Použitie ostatných metód je pre obe triedy rovnaké, nadbytočné metódy triedy Vector3D nespôsobia problém, nakoľko sa v prípade Vector3 nepoužívajú. Nakoniec bude možné odstrániť konverznú metódu *fromVector3()* z triedy Vector3D. Trieda Vector3D taktiež nie je oko-

mentovaná a preto v rámci prepracovania bude aj vhodne zdokumentovaná dokumentačnými komentármi.

15.3 Implementácia

V projekte Jim a v testovacom frameworku v projekte TestFramework sa nachádzalo približne 300 použití triedy Vector3. Všetky použitia boli úspešne nahradené za Vector3D a trieda Vector3 bola odstránená z projektu. Z triedy Vector3D bola na záver odstránená konverzná metóda z triedy Vector3.

15.4 Testovanie

Každé použitie setterov triedy Vector3D bolo starostlivo skontrolované a boli spustené testy, ktoré nehlásili žiadne nové chyby. Robot sa po supstení správal rovnako ako pred úpravou a testovací framework fungoval rovnako ako pred zmenou.

16 Pridávanie hráčov do testovacieho frameworku

Posledná úprava	Juraj Šimek
Platné od	8. marec 2015
Poznámky	

Nakoľko pridávanie hráča do testovacieho frameworku nefunguje, rozhodli sme sa zanalyzovať príčinu tohto problému a odstrániť ho. Cieľom je teda umožniť testovaciemu frameworku pridávať hráčov v rámci dvoch rôznych tímov. Hráči pridaný pomocou testovacieho frameworku tak budú môcť mať presmerovaný výstup priamo do testovacieho frameworku a tak bude možné kontrolovať činnosť každého hráča.

16.1 Analýza

Hráči sa pridávajú v testovacom frameworku pomocou karty *Manage agents*. V časti *Add new agent* sa vyberie názov tímu a kliknutím na tlačidlo *Add* by sa mal pridať nový hráč. Kliknutím na tlačidlo však nedôjde k pridaniu žiadneho hráča a toto tlačidlo sa zablokuje a ďalej nie je použiteľné. Tlačidlo sa stane neaktívne a jeho názov sa zmení na *wait...* . Do testovacieho frameworku možno ale pridávať hráčov tak, že sa spustí framework a potom sa samostatne spustí aj hráč. Takto vytvorený hráč ale nemá presmerovaný výstup na testovací framework.

Pomocou debuggera z prostredia Eclipse IDE sme zistili, že po kliknutí na tlačidlo *Add*, ktoré je definované v rámci triedy *MainFrame* balíka *testframework.ui*, sa vyvolá metóda *btnAddAgentClicked()*. V nej sa nastaví blokovanie príslušného tlačidla až do chvíle, keď sa podarí frameworku nadviazať spojenie s očakávaným agentom. Agent je identifikovaný pomocou čísla jeho dresu a názvy tímu, očakávaný agent je agent, ktorého číslo a názov tímu sa zhodujú s číslom a názvom tímu agenta, ktorého pridanie sme si vyžiadali. Za účelom odblokovania tlačidla trieda implementuje rozhranie *IAgentManagerListener*, z ktorého implementuje metódu *agentAdded()*. Keď sa podarí nadviazať spojenie s novo pridaným agentom, agent bude komunikovať s frameworkom pomocou správ, ktoré spravuje trieda *AgentMonitorMessage*. Za týmto účelom trieda *AgentManager* implementuje metódu *receivedMessage()* predpísanú rozhraním *IAgentMonitorMessage*, ktorá spracováva 3 typy správ.

Ide o správy TYPE_DESTROY, ktoré sú posielané, keď sa ukončuje beh daného agenta, TYPE_WORLD_MODEL, ktoré predstavujú správy spojené s modelom samotného agenta a o správy typu TYPE_INIT, ktoré sa posielajú pri spustení agenta. Práve zachytenie správy TYPE_INIT má význame pre identifikáciu toho, či je agent vytvorený a či sa s ním podarilo nadviazať spojenie. Metóda *receivedMessage()* preto po obdržaní tohto typu správy notifikuje GUI prostredníctvom metódy *agentAdded()* o tom, že agent bol pridaný a tlačidlo na pridávanie nových agentov môže byť uvoľnené. Metóda *agentAdded()* sa potom postará o samotné uvoľnenie tlačidla *Add*. Príčina toho, prečo sa tlačidlo *Add* neodblokuje je v tom, že proces želaného agenta nie je spustený a preto sa nikdy nezavolá metóda *agentAdded()*.

Metóda *btnAddAgentClicked()* vyvolá pridávanie hráča volaním metódy *AgentManager.getManager().getAgent()*. Trieda *AgentManager* sa stará o pridávanie hráča a komunikáciu s ním a je implementovaná ako singleton. Metóda *getAgent()* vracia už pridaného hráča, alebo, ak je jej posledný parameter (blocking) nastavený na true a v prípade, že parametre uniform a team nepredstavujú číslo a tím žiadneho pridaného hráča, pridá nového hráča volaním metódy *startAgent()*.

Metóda *startAgent()* načíta príslušný príkaz pre spustenie agenta z triedy *C*, ktorá združuje všetky podstatné nastavenia pre testovací framework. Pri bližšom pohľade sme zistili, že príkaz, ktorý je uvedený v príslušnom konfiguračnom súbore *(src/sk/fiit/testframework/init/default.properties)*, nefunguje. Bude preto potrebné ho upraviť tak, aby sa stal funkčným a spustil agenta. Metóda k tomuto príkazu následne pripojí informácie o agentovom tíme, čísle a komunikácii pomocou TFTP servera. Následne metóda daný príkaz spustí a spustí vlákno na čítanie výstupu pridaného agenta.

16.2 Riešenie

Prvým problémom, ktorý bolo potrebné vyriešiť bolo upraviť príkaz na pridávanie nového hráča. Príkaz sa vykoná v metóde *startAgent()* triedy *AgentManager* prostredníctvom triedy *ProcessBuilder*. Trieda *C*, ktorá združuje nastavenia testovacieho frameworku načítava defaultné nastavenia zo súboru *default.properties*, ak v hlavnom adresári projektu testovacieho frameworku nie je prítomný súbor *configuration.properties*. Radšej ako upravovať defaultné nastavenia sme sa rozhodli tento súbor vytvoriť a upravený príkaz zmeniť v ňom. Príkaz upravuje položka *robocup.command.player*. Pôvodne mal nasledovný tvar:

```
java
-classpath ../RoboCupLibrary/bin;bin;
  lib/aspectjrt.jar;lib/bsf.jar;
  lib/jruby-complete-1.4.0.jar;
  lib/commons-logging-1.1.jar;
  lib/commons-net-2.2.jar
sk.fiit.jim.init.Main
```

Po niekoľkých pokusoch sme príkaz upravili do funkčného tvaru:

```
java
-classpath "bin;../RoboCupLibrary/bin;
  ../TestFramework/bin;
  lib/reflections-0.9.9-RC1.jar;
  lib/javassist-2.6.jar;
  lib/google-collections-0.9.jar;
  lib/commons-net-2.2.jar;
  lib/mockito-all-1.8.5.jar;
  lib/hamcrest-all-1.3.0RC1.jar;
  lib/junit.jar;lib/commons-logging-1.1.jar;
  lib/jruby-complete-1.4.0.jar;lib/bsf.jar"
sk.fiit.jim.init.Main
```

Príkaz treba spúšťať z koreňového adresára projektu Jim. Príkaz sa úspešne vykoná a vyvolá spustenie nového agenta. Triede *ProcessBuilder* je zmenený pracovný adresár pomocou metódy *directory()* na adresár v položke *robocup.player.dir* konfiguračného súboru, ktorá je nastavená na koreňový adresár projektu Jim. Po jeho vložení do konfiguračného súboru testovacieho frameworku však agent stále nebol pridávaný. Zistili sme, že v prípade povolenia TFTP servera na strane agenta (parameter TFTP_ENABLED) agent vyhodí výnimku a nespustí sa. Riešenie problému nebolo náročné a vyžadovalo len vytvorenie hierarchie adresárov *scripts/testframework* v koreňovom adresári projektu Jim. Po týchto úpravách už bol agent do testovacieho frameworku úspešne pridaný.

Pri úprave sme sa stretli s problémom, kedy agenti s iným názvom tímu ako ANDROIDS neboli zaregistrovaní do testovacieho frameworku. Po dlhšom pátraní sa však ukázalo, že chyba bola na strane servera (rcssserver) a po jeho reštarte bolo možné pridávať agentov s ľubovoľným názvom tímu.

V rámci úprav pridávania agentov do frameworku bola zrefaktorovaná aj celá trieda *AgentManager* a trieda *IAgentManagerListener*.

16.3 Overenie riešenia

Po úpravách bol agent úspešne pridaný a zaregistrovaný medzi bežiacimi agentmi v testovacom frameworku. Agent s frameworkom bez problémov komunikoval.

17 Refaktorovanie projektu RoboCupLibrary

Posledná úprava	Juraj Šimek
Platné od	12. marec 2015
Poznámky	

Balík RoboCup library obsahuje mnohé triedy, ktoré do neho boli v minulosti presunuté v rámci vytvorenia spoločnej knižnice tried a metód pre všetky projekty, ktorých účelom je vývoj, údržba a testovanie robotických hráčov pre robotický 3D futbal. Triedy však nie sú zdokumentované a neboli v nich dodržiavané jednotné pravidlá pre písanie zdrojového kódu. Preto je potrebné zvýšiť ich prehľadnosť a vhodne ich zdokumentovať.

17.1 Analýza

Ako prvý krok, ktorý sme vykonali, bolo spustenie pluginu *Unnecessary Code Detector* na celý projekt. V niektorých prípadoch boli označené metódy, ktoré nie sú nikde používané. Analýza však ukázala, že ide o dôležité metódy, ktoré môžu byť niekedy použité a nie je vhodné ich z tried odstrániť. Nakoniec, cieľom tohto balíka je poskytovať knižnicu metód a tried pre vývoj hráča, nie každá metóda preto musí byť použitá.

UCD označil mnoho členských premenných, ktoré mali viditeľnosť *package* a bolo možné ich zmeniť na *private*. Budeme sa preto snažiť o dodržanie objektového prístupu a zabezpečíme patričné zapúzdrenie v rámci tried.

Niektoré nezrovnalosti sa týkali aj názvov tried. Jednou z nich bolo pomenovanie triedy *Vector2*, ktorá reprezentovala dvojrozmerný bod. Premenujeme ju preto podľa vzoru jej obdoby pre trojrozmerný priestor, na *Point2D*. Trieda neobsahuje žiadnu metódu pre výpočet vzdialenosti medzi dvoma bodmi. Táto metóda sa v rámci samotnej knižnice používa, no je implementovaná na iných miestach, ako napríklad v triede *MEC*, ktorá pre danú množinu bodov vypočíta najmenšiu kružnicu, ktorá ich uzaviera. Túto metódy preto presunieme do triedy Point2D.

Pri prehliadke zdrojového kódu projektu *Jim* sme zistili, že trieda MEC je v ňom taktiež implementovaná (má dokonca rovnakú formu ako v RoboCupLibrary), ale nepoužíva sa. Jim používa verziu MEC z RoboCupLibrary. Preto sme sa rozhodli, že túto triedu z projektu Jim odstránime.

V balíku sk.fiit.robocup.library.review sa nachádza jediná trieda *ReviewOk*. Táto trieda (anotácia) označuje úsek zdrojového kódu, ktorý bol posúdený a zrevidovaný nejakým členom tímu a je bezchybný. Triedou sú však označené len 3 triedy a 3 anotácie. Nakoľko sa trieda s rovnakým účelom (*Reviewed*) vyskytuje aj v balíku sk.fiit.robocup.library.annotations, túto triedu odstránime.

Všetky *testy* v RoboCupLibrary zbehnú bez chyby, sú však umiestnené priamo pri zdrojových kódoch testovaných tried. Pri refaktorovaní ich preto umiestnime do osobitného zdrojového priečinka s názvom tests.

Trieda *Vector3D* už bola zrefaktorovaná v úlohe Zámena triedy Vector3 za Vector3D 15.

17.2 Realizácia

Ako prvé sme presunuli *testy* do samostatného zdrojového súboru. Testy sme neskôr aj zrefaktorovali tak, že sme ich správne naformátovali a pridali popis k jednotlivým triedam, ktoré ich realizujú. Následne sme pristúpili k *refaktorovaniu* samotných tried projektu RoboCupLibrary. Všetky členské premenné, ktoré bolo možné označiť ako *private* sme takto označili. V niektorých triedach boli vytvorené aj príslušné *gettery* a *settery*.

Triedu *Vector2* sme premenovali na *Point2D* a vložili sme do nej metódu pre výpočet vzdialenosti dvoch bodov, ktorú sme zase odstránili z triedy *MEC*. Duplicitnú triedu MEC z projektu Jim sme vymazali. Vymazali sme aj anotáciu *ReviewOk* z dôvodov uvedených v analýze.

V rámci refaktoringu, ktorý sledoval výstup analýzy, boli *zdokumentované* všetky triedy a metódy a zdrojový kód bol správne naformátovaný podľa metodiky pre písanie zdrojového kódu.

17.3 Testovanie

Všetky testy v projekte Jim a RoboCupLibrary po refaktorovaní úspešne zbehli. Preto usudzujeme, že pri refaktorovaní nedošlo k žiadnej chybe na funkčnosti kódu.

18 Grafické rozhranie pre kontrolu loggera

Posledná úprava	Juraj Šimek
Platné od	19. marec 2015
Poznámky	

V priebehu riešenia projektu sa najmä v súvislosti s debugovaním úlohy riešiacej KPM ukázalo, že je vhodné kontrolovať úrovne logovania počas behu programu. To by malo byť možné pomocou samostatného okna, kde bude možné nastavovať dané úrovne logovania.

18.1 Analýza

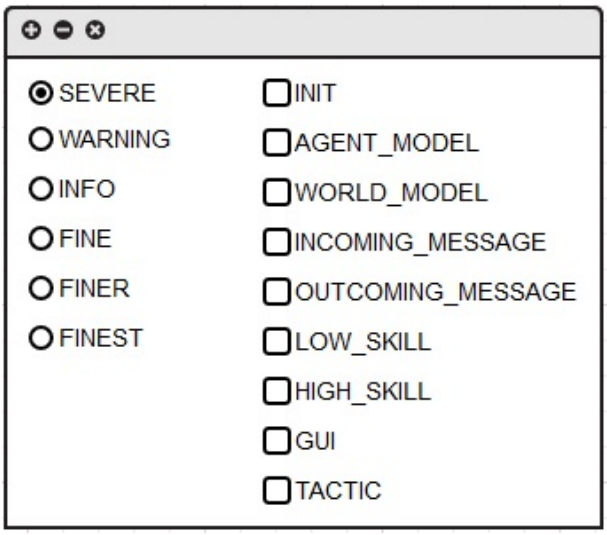
Logger bol predstavený v predchádzajúcom texte 5.1. Logger možno získať pomocou príkazu *JLog.getLogger()*, pričom tento príkaz vracia štandardný logger Java API. Pomocou metódy *setLevel()* možno nastaviť jednu z úrovní *Level.SEVERE* až *Level.FINEST*. Logger používaný v projekte Jim bol rozšírený o niektoré dodatočné úrovne logovania. Tieto úrovne sú realizované triedou *LogType*, pričom možno do nej dodatočne dopĺňať ďalšie úrovne pre logovanie. Úrovne typu LogType sú logované dodatočne, bez ovplyvnenia vzhľadom na nastavenie setLevel(). Tieto úrovne sa do loggera dodávajú pomocou metódy *JLog.addLogType()* a odoberajú sa z neho pomocou metódy *JLog.removeLogType()*.

Projekt Jim má vo funkcii main() triedy Main inicializovaný logger tak, aby logoval všetky štandardné aj dodatočné úrovne. Keďže pri debugovaní by bolo zbytočne komplikované prepisovať zdrojový kód, doplníme grafické používateľské rozhranie, ktoré umožní nastavovať úrovne logovania za behu programu.

18.2 Návrh

Do nastavení pridáme novú položku, ktorá umožní zapnúť alebo vypnúť grafické rozhranie pre ovládanie úrovní logovania. Bude preto potrebné vykonať zmenu v triede *Settings*, aby načítavala túto novú položku.

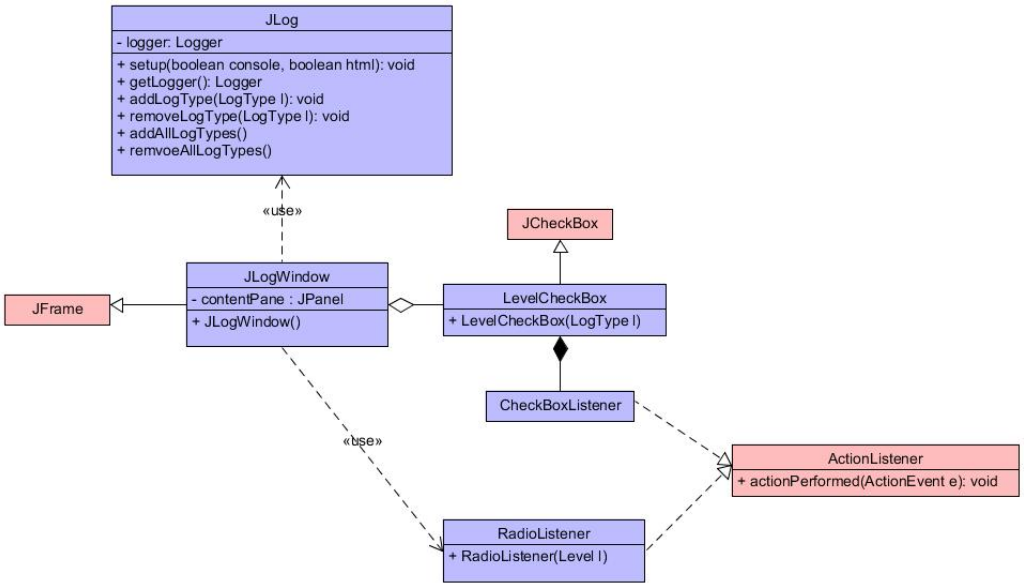
Samotné *okno* na ovládanie úrovní logovania bude rozdelené do dvoch stĺpcov. V ľavom sa budú ovládať štandardné úrovne logovania, v pravom sa budú ovládať dodatočné úrovne. Jeho návrh je možné vidieť na obrázku 12.



Obr. 12: Návrh okna na kontrolovanie úrovní ovládania

Okno bude realizované triedou *JLogWindow* s použitím knižnice *Swing*, trieda teda bude dediť od triedy *JFrame*. Radio buttony budú slúžiť na ovládanie základných úrovní logovania a budú spravované triedou *RadioListener*, ktorá bude vykonávať príslušné nastavenia loggera podľa toho, ktorý radio button je zaškrtnutý. Nastavenie prostredníctvom radio buttonov sa vykoná použitím metódy setLevel() opísanej v analýze. Samotné check boxy riadiace dodatočné úrovne logovania budú reprezentované triedou *LevelCheckBox*, v rámci ktorej sa budú prostredníctvom vnorenej triedy *CheckBoxListener* vykonávať aj jednotlivé nastavenia v závislosti od toho, ktorý check box je zaškrtnutý. Každý check box bude spárovaný s jednou dodatočnou úrovňou logovania a v prípade, že bol zaškrtnutý, táto úroveň sa pridá do logov pomocou metódy addLogType(), v prípade odškrtnutia sa odoberie volaním metódy removeLogType(). Dodatočné úrovne logovania budú získané pomocou reflexie vykonanej nad triedou LogType. Diagram tried na obrázku 13 zachytáva detaily na dostatočnej úrovni podrobnosti.

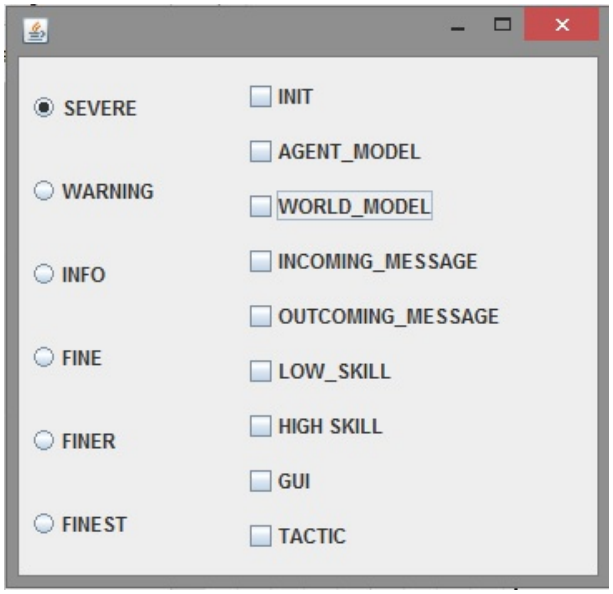
Inicializačná úroveň logovania pri vyvolaní kontrolného okna bude nastavená na logovanie úrovne SEVERE bez logovania dodatočných úrovní.



Obr. 13: Diagram tried pre okno na ovládanie úrovní logovania

18.3 Implementácia

Používateľské rozhranie pre podporu kontroly logovania bolo implementované v zmysle opísanom v návrhu. Do nastavení bolo pridané pole LOGGER_CONTROL_WINDOW, ktoré možno nastaviť na hodnotu *true* alebo *false* v závislosti od toho, či si prajeme, aby sa zobrazilo okno na kontrolu logovania. Výsledné okno je na obrázku 14.



Obr. 14: Finálne okno na ovládanie úrovní logovania

18.4 Testovanie

Okno sa správa podľa očakávaní, pri výbere danej úrovne logovania sa loguje len daná úroveň. Zmeny možno vykonávať za behu agenta. Všetky testy prešli bez nájdených chýb.

19 Ukladanie nastavení GUI loggera

Posledná úprava	Juraj Šimek
Platné od	4. apríl 2015
Poznámky	

Skúsenosti s novým grafickým používateľským rozhraním loggera predstaveným v 18 ukázali, že by bolo vhodné, keby toto GUI ukladalo poslednú svoju konfiguráciu, aby nebolo potrebné pri každom jeho spustení vyklikávať jednotlivé úrovne logovania, ktoré chceme logovať. Túto funkcionalitu sme preto do GUI doplnili.

19.1 Analýza a návrh

O GUI pre logger sa stará trieda *JLogWindow*. V tejto triede preto implementujeme zmeny, ktoré umožnia ukladať najnovšiu konfiguráciu GUI a načítavať takto naposledy uloženú konfiguráciu. Do úvahy pripadajú viaceré možnosti, od obyčajného *textového súboru*, *serializácie*, *XML súboru*, či *properties* súboru. Aj keď by sa *serializácia* mohla javiť ako vhodné riešenie, ukázalo sa, že je s ňou spojených množstvo problémov hlavne kvôli premenlivému počtu dodatočných logovacích úrovní. Práve súbory *properties* sa ukazujú ako najlepšia voľba, nakoľko s nimi Java vie prirodzene pracovať, preto sme sa ich rozhodli použiť.

Do príslušného properties súboru budú ukladané dvojice NÁZOV_ÚROVNE={ true | false }. Ak bude pri názve úrovne uvedená hodnota *true*, znamená to, že v GUI bude zaškrtnutá odpovedajúca položka. To platí pre základné aj dodatočné úrovne logovania. Ak by medzi časom pribudla dodatočná úroveň logovania, bude chápaná ako nezaškrtnutá a pri ďalšom uložení konfigurácie bude do properties súboru vložená. Takisto, ak by nejaká úroveň odbudla, nebude jej nastavenie z properties načítané.

Pre každé zaškrtávacie pole sme sa preto rozhodli pridať nový *ActionListener*, ktorý túto funkcionalitu zabezpečí. Ukladanie nastavení GUI loggera bude vykonávané len vtedy, ak v súbore *settings.properties* bude nastavená hodnota pre ukladanie konfigurácie GUI loggera na true.

19.2 Implementácia a testovanie

V *settings.properties* sme vytvorili položku LOGGER_SAVE_CONFIGURATION, ktorá hovorí o tom, či sa bude načítavať a ukladať konfigurácia GUI loggera. Do triedy *Settings* sme pridali načítanie tejto položky z nastavení.

V triede *JLogWindow* došlo k rozsiahlym zmenám. Všetky *radiobuttony* pre základné úrovne logovania boli vytiahnuté z konštruktora a vložené do triedy ako súkromné členské premenné. Vytvorili sme súbor *logger_gui_config.properties*, ktorý bude uchovávať informácie o tom, ktoré položky sú vybraté. Pri inicializácii okna sa z neho načítajú posledné nastavenia. Ako v prípade základných úrovní, tak aj v prípade doplnkových úrovní sa k nastaveniam toho, či majú byť dané zaškrtávacie polia zaškrtnuté pristupuje pomocou ich názvov. V triede JLogWindows sme ďalej vytvorili metódu *saveConfiguration()*, ktorá do logger_gui_config.properties ukladá aktuálne hodnoty jednotlivých zaškrtávacích tlačidiel. Kvôli prístupu k dodatočným úrovniam logovania sme upravili kód tak, že sú im odpovedajúce checkboxy sú pridávané do zoznamu typu *LevelCheckBox*. Zoznam sa potom prejde pomocou foreach cyklu. Na záver sme pridali vnútornú triedu *SaveConfigurationListener* dediacu od ActionListener, ktorá vykoná uloženie aktuálnej konfigurácie pomocou saveConfiguration() vždy, keď dôjde k nejakej zmene v GUI.

Implementáciu sme úspešne otestovali a videli, že nastavenia GUI sa medzi behmi programu zachovávajú. Implementácia neovplyvnila výsledky existujúcich testov.

20 Príprava taktík pre RoboCup turnaj

Posledná úprava	Juraj Šimek
Platné od	7. apríl 2015
Poznámky	

V tejto kapitole sme sa venovali analýze fakultného RoboCup turnaja. Po úvodnej analýze jednotlivých súťažných disciplín sme upravovali už existujúce turnajové taktiky tak, aby zodpovedali pravidlám. Tie taktiky, ktoré neexistovali sme vhodne doplnili.

20.1 Analýza pohybov pre RoboCup turnaj na FIIT

Posledná úprava	Juraj Šimek
Platné od	7. apríl 2015
Poznámky	

V RoboCup turnaji organizovanom na FIIT sa hodnotia nasledovné disciplíny:

- Základné pohyby
  - Vstávanie z brucha
  - Vstávanie z chrbta
  - Chôdza (stabilita)
  - Chôdza (rýchlosť)
  - Kopanie do lopty (vzdialenosť)
  - Kopanie do lopty (presnosť)
  - Otočenie hráča
- Pohyb po ihrisku
  - Kop na bránu s orientáciou
  - Kop na určený bod na presnosť
  - Obchádzanie prekážok

- Voľná jazda

Všeobecné pravidlá turnaja sú nasledovné:

- Bude použitý server s modelom NAO
- Začne sa v hracom režime *BeforeKickOff* hráč je povinný teleportovať (alebo inak dopraviť) sa na počiatočnú pozíciu definovanú pri danej disciplíne. Hráč je povinný zaujať príslušnú počiatočnú polohu a potom sa prestať hýbať.
- Hráč dostane pokyn na začiatok pohybu zmenou hracieho režimu na *PlayOn*. Čas zmeny hracieho režimu na PlayOn je zároveň okamihom začiatku merania času. Jedinou výnimkou sú disciplíny kopania, kedy sa pokyn na začiatok pohybu dá hracím režimom *KickOff_Left*.
- V prípade, že hráč spadne, nebude sa vedieť postaviť a server ho z toho dôvodu automaticky presunie na okraj ihriska, bude hráč hodnotený ako keby ostal ležať na mieste až do skončenia časového limitu určeného na danú disciplínu.
- Organizátori si vyhradzujú právo na doplnenie pravidiel, ak bude niektorý z hráčov úmyselne využívať neúplné alebo nepresné zadefinovanie pravidiel, pri čom poruší základnú myšlienku danej disciplíny (napríklad "kopanie" rukou a podobne).
- Počas celej súťaže platia pravidlá športového, čestného a slušného správania.

Víťazí tím, ktorý získal najlepšie umiestnenie vo všetkých disciplínach. Umiestnenie v každej disciplíne sa hodnotí bodmi, za 1. miesto 3 body, za 2. miesto 2 body, za 3. miesto 1 bod. *Výsledné poradie* sa určí na základe súčtu bodov za všetky disciplíny.

Ďalej uvádzame opis jednotlivých disciplín a to, ako ich implementuje projekt *Jim*. Jednotlivé taktiky pre turnaj sa nachádzajú v balíku *sk.fiit.jim.decision.tactic.tournament*.

20.1.1 Vstávanie z brucha

Hráč začína na pozícii $(-3, 0)$, pričom leží na bruchu, kĺby má nastavené v polohe, ako pri prihlásení na server, ale ruky musí mať pripažené. Meria sa

čas, za ktorý robot vstane zo zeme, spraví sa 5 pokusov, za každý neúspešný pokus sa pripočíta 20s. Meranie sa zastaví po 20s alebo sekundu po tom, ako sa robot dostane do vzpriamenej polohy (dotýka sa zeme len chodidlami a ťažisko má aspoň 38cm nad zemou) a prestane sa viditeľne hýbať.

Táto disciplína je implementovaná taktikou *GetUpFromStomach*. Je v nej nastavená nesprávna pozícia na hodnotu $(-5, 1)$, opis disciplíny vyžaduje umiestnenie robota na pozíciu $(-3, 0)$, čo bude potrebné opraviť. Agent navyše padá a neustále vstáva aj v režime *BeforeKickOff*.

20.1.2 Vstávanie z chrbta

Táto disciplína má rovnaké požiadavky a vyhodnocuje sa rovnako ako disciplína Vstávanie z brucha s rozdielom, že agent na začiatku musí ležať na chrbte. Disciplínu realizuje taktika *GetUpFromBacks*. Agent v nej začína na správnej pozícii, ale rovnako, ako v prípade *GetUpFromStomach* neustále padá a vstáva aj v režime *BeforeKickOff*.

20.1.3 Chôdza (stabilita)

Počiatočná pozícia agenta je $(-5, 1)$, pričom hráč stojí na nohách a všetky kĺby má v polohe v akej boli po prihlásení na server. V rámci disciplíny sa meria čas, za ktorý sa robot dostane na druhú polovicu ihriska. Z 2 pokusov sa započíta ten rýchlejší. Meranie času sa zastaví vtedy, keď sa ťažisko robota dostane na vzdialenejšiu pozíciu ihriska oproti pozícii, kde s chôdzou začal. Za každý dotyk hracej plochy inou časťou robota ako je chodidlo sa k výslednému času pripočíta 10 trestných sekúnd (viacero dotykov behom 1 sekundy sa počíta ako 1 dotyk). Ak robot nepríde do cieľa za 180 sekúnd, vypočíta sa výsledný čas lineárnou aproximáciou podľa prejdenej vzdialenosti (so zarátaním trestných sekúnd).

Chôdza je realizovaná taktikou *StabilityWalkTactic*, pričom je použitá chôdza *WALK_MEDIUM*. Agent začína na pravidlami definovanej pozícii $(-5, 1)$ a pomocou chôdze *WALK_MEDIUM* kráča na pozíciu $(6, 1)$, teda na vzdialenejšiu pozíciu vzdialenejšej polovice ihriska. Chôdzu bude možné zrýchliť použitím chôdze definovanej pomocou *ZMP*.

20.1.4 Chôdza (rýchlosť)

Pravidlá pre túto disciplínu sú rovnaké ako v prípade hodnotenia stability chôdze, ale s rozdielom, že za dotyky hráča hracej plochy inou časťou robota,

ako sú chodidlá sa trestné sekundy nepripočítavajú.

Rýchla chôdza je realizovaná pomocou taktiky *FastWalkTactic*. Agent začína na pravidlami definovanej pozícii. Potom sa spustí chôdza, pričom na pozíciu $(1, 1)$ kráča pomocou chôdze *WALK_FAST*, na pozíciu $(1.25, 1)$ pomocou *WALK_MEDIUM* na na pozíciu $(6, 1)$ opäť pomocou chôdze *WALK_FAST*. Celková chôdza je však nestabilná a javí sa v dôsledku častých pádov robota ešte pomalšia ako chôdza v taktike *StabilityWalkTactic*. Prečo autori taktiky zvolili postupnosť týchto chôdzí nie je zrejmé. Keď nastavíme agenta tak, aby do cieľa $(6, 1)$ kráčal iba pomocou chôdze *WALK_FAST*, je jeho chôdza oveľa stabilnejšia a do cieľa sa dostane rýchlejšie s menším počtom pádov (priemerne v rozmedzí $0 - 3$ pády). Zakomponovaním ZMP očakávame zefektívnenie rýchlej chôdze v turnajovej taktike.

20.1.5 Kopanie do lopty (vzdialenosť)

Počiatočná pozícia hráča v tejto disciplíne je ľubovoľná, no hráč musí stáť na nohách. Meria sa vzdialenosť, do akej sa lopta po kopnutí dostane. Vykonáva sa 5 pokusov, z nich sa potom vyberie najlepší. Ak hráč počas kopnutia spadne, do úvahy sa berie len polovičná vzdialenosť.

O túto disciplínu sa stará taktika *KickDistance*. Hráč začína na pozícii $(-0.3, 0)$, pričom lopta je na pozícii $(0, 0)$. Hráč do lopty kopne použitím lowskillu *kick_faster_left*, ktorý sa volá pomocou triedy *DefaultHighSkill*. Testovanie však odhalilo, že taktika nefunguje a hráč sa do lopty ani nepokúsi kopnúť. Volaný lowskill je však spustený, no hráč aj tak nevykonáva žiadny pohyb. Taktiku bude potrebné vytvoriť nanovo. Taktiku *KickDistance* je možné nahradiť aj taktikou *KickOnXY*.

20.1.6 Kopanie do lopty (presnosť)

Počiatočná pozícia agenta je $(X, 0)$, prvú súradnicu si môže zvoliť tím, hráč sa ale nesmie dotýkať lopty. Hráč stojí na nohách a je otočený čelom k stredu súperovej brány. V tejto disciplíne sa meria odchýlka smeru kopnutia od smeru k niektorému z rohov ihriska, pričom roh si vyberie hráč *(F1R, F2R, F1L, F2L)*. Celkovo sa vykoná 5 pokusov, započítava sa pokus s najmenšou odchýlkou smeru lopty od smeru k niektorému z rohov ihriska. Ak hráč počas kopnutia spadne (tzn. dotkne sa ihriska inou časťou tela ako chodidlami), vynásobí sa výsledný uhol dvomi. Ak lopta po kopnutí neprejde minimálne 1,5 metra, pokus sa považuje za neplatný.

Pre túto disciplínu neexistuje osobitná taktika. Možno však využiť taktiku *KickOnXY*, ktorú opisujeme nižšie. Možnosťou je však aj implementácia novej osobitnej taktiky.

20.1.7 Otáčanie hráča

Počiatočná pozícia hráča je $(-3, 0)$, hráč stojí na nohách a je otočený čelom k súperovej bráne. Meria sa čas, za ktorý sa hráč otočí o 180 stupňov s toleranciou 5 stupňov. Hráč má na otočenie 2 pokusy, započítava sa rýchlejší. Meranie času sa zastaví vo chvíli, keď hráč dosiahne predpísané otočenie a je pri tom vo vzpriamenej polohe.

Pre túto disciplínu nie je implementovaná žiadna taktika. Taktiku sa preto pokúsime implementovať sami.

20.1.8 Kop na bránu s orientáciou

Hráč začína na pozícii $(-1, 0)$, pričom lopta sa vyskytuje postupne na troch rôznych pozíciách (pozície určia organizátori pred súťažou pre všetky tímy rovnaké). Meria sa čas od zapnutia režimu *PlayOn* po zmenu polohy lopty. Hráč má 3 pokusy s rôznou polohou lopty, počíta sa priemer časov. Hráč musí zistiť svoju pozíciu, pozíciu lopty, prísť k nej bez toho, aby ju posunul a kopnúť smerom na súperovu bránu. Kop musí mať aspoň 1,5 metra a nesmie sa dochýliť o viac ako +/- 20 stupňov od smeru na stred brány. Ak je pokus z akéhokoľvek dôvodu neúspešný, počíta sa 120 sekúnd.

Táto disciplína je realizovaná taktikou *KickOnGoal*. Hráč je pomocou nej umiestnený na pozíciu $(-1, 0)$, potom sa stredne rýchlou chôdzou (*WALK_MEDIUM*) priblíži k lopte a kopne normálnou silou (*KICK_NORMAL*). Spustenie hráča však ukázalo, že hráč má pomerne veľké problémy so stabilitou a so samotným nasmerovaním na loptu. Robot navyše po prepnutí z režimu *PlayOn* späť do *BeforeKickOff* sa stále pokúša kopnúť do lopty.

20.1.9 Kop na určený bod na presnosť

Počiatočná pozícia hráča je $(-3, 1)$, pričom je otočený smerom k súperovej bráne. Počiatočnú pozíciu lopty určia organizátori pre všetky tímy rovnako, rovnako aj cieľovú pozíciu lopty. Meria sa absolútna vzdialenosť výslednej pozície lopty od určenej cieľovej polohy. Z 5 pokusov sa berie priemer najlepších 3.

Túto disciplínu realizuje taktika *KickOnXY*, ktorá na začiatku disciplíny umiestni hráča na požadovanú polohu v rámci ihriska. Taktika definuje vektor *TournamentPosition*, ktorý predstavuje cieľovú polohu lopty (kde chceme kopať). Pohybom *WALK_MEDIUM* agent príde k lopte a kopne ho kopom *KICK_STRONG* na pozíciu určenú pomocou *TournamentPosition*. Robot má však rovnaké problémy so stabilitou a nájdením lopty ako v prípade taktiky *KickOnGoal*. Problém je aj s pokračovaním činnosti po opätovnom prepnutí do režimu *BeforeKickOff*.

20.1.10 Obchádzanie prekážok

Počiatočná pozícia hráča je $(-0.2, 0)$. Na plochu sa umiestnia 4 súperi, ktorých poloha je určená organizátormi na začiatku súťaže pre všetky tímy rovnako. Cieľová pozícia hráča je $(7.2, 0)$. Meria sa čas po zastavenie hráča v cieľovej pozícii. Hráč má 4 pokusy, berie sa priemer najlepších 2. Hráč sa musí dostať z počiatočnej pozície do finálnej pozície. Súperovi hráči sú nehybní a otočení štandardne k bránke testovaného hráča. Hráč musí prísť na cieľovú pozíciu s presnosťou v absolútnej hodnote 0,2. Za dotyk každého súperovho hráča je penalizácia 120 sekúnd. V prípade neúspešného pokusu sa berie hodnota 240 sekúnd. Časový limit je 120 sekúnd. Sú dva pokusy pre prvú kombináciu súpera a dva pre druhú. Výsledkom je priemer z lepších pokusov v prvom a druhom prípade.

Túto disciplínu spravuje taktika *AvoidEnemy*. Hráč však v nej nie je nastavený na pravidlami určenú pozíciu, čo treba opraviť. Hráč sa následne snaží vyhnúť ostatným hráčom tak, že zistí ich y-ové osi a pamätá si najvzdialenejšiu z nich. Cez obsadené územie prejde chôdzou *WALK_MEDIUM*, pričom sa snaží vyhýbať hráčom. Keď prejde za posledného súpera, kráča znova chôdzou *WALK_MEDIUM* až po cieľový bod.

20.1.11 Voľná jazda

Počiatočná pozícia agenta je ľubovoľná. Vyhodnocuje sa umelecký dojem, estetika pohybov, zábavnosť predvedeného výkonu, náročnosť technického riešenia. Robot má 2 minúty na to, aby na ihrisku niečo predviedol, organizátori súťaže môžu tento čas v odôvodnených prípadoch predĺžiť. Je možné spustiť naraz aj viacero robotov (treba ale počítať so zníženým výkonom servera). Hodnotenie sa uskutoční na základe hlasovania súťažiacich a divákov (pre súťažiacich platí pravidlo: 1 hlasovací lístok na tím).

Túto disciplínu nerealizuje žiadna z prítomných taktík. Bude ju preto potrebné doplniť.

20.2 Úprava a implementácia nových turnajových taktík

Posledná úprava	Juraj Šimek
Platné od	26. apríl 2015
Poznámky	

V tejto časti sme sa venovali oprave chýb existujúcich turnajových taktík nadväzujúc na analýzu turnajov pre RoboCup turnaj na FIIT. Tie turnajové taktiky, ktoré ešte neboli implementované sme doplnili.

20.2.1 GetUpFromStomach

V tejto taktike bola nastavená nesprávna pozícia na hodnotu $(-5, 1)$, opis disciplíny vyžaduje umiestnenie robota na pozíciu $(-3, 0)$, preto sme túto pozíciu zmenili. Neustále padanie a vstávanie agenta počas režimu *BeforeKickOff* bolo spôsobené nesprávnym plánovaním a pridávaním lowskillu *fall_front* a highskillu *GetUp* v metóde *startTactic()*, ktorá je uvedená nižšie. Na začiatku tejto metódy agent beamoval na dané súradnice ale okamžite naplánoval spomenuté highskilly. Preto sme pridanie týchto highskillov odstránili a ponechali iba beamovanie a spadnutie agenta pomocou lowskillu *fall_front*. Po tom ako agent spadne už nebeamuje, len čaká na režim *PlayOn* a nastaví nami pridanú premennú *inStartPotition* na *true*. Neskôr, v momente, keď sa zmení hrací režim a *isStartPosition* je true, spustí sa metóda *run*, do ktorej sme premiestnili highskill vstávania a zavolali lowskill *nothing* použitím triedy *DefaultHighSkill*. Lowskill *nothing* sme vytvorili na základe lowskillu *head_ left_ 120*, pričom sme ho upravili tak, aby agent hlavu neotáčal. Pridanie tohto lowskillu je potrebné na stabilizáciu agenta po jeho postavení. Po vykonaní týchto highskillov sa nastaví *inStartPosition* na *false*, čo umožní opakovať vstávanie len zmenou režimu na *BeforeKickOff*.

Listing 1: Pôvodná implementácia metódy startTactic()

```
public void startTactic(List<String> currentSituations) {
  if (EnvironmentModel.beamablePlayMode() == true) {
    this.beamExec.BeamAgent(Vector3D.cartesian(START_X,
```

```
      START_Y, 0.1) );
    HighSkillPlanner planner =   HighSkillRunner.getPlanner();
    planner.addHighskillToQueue(
      new DefaultHighSkill("fall_front")
    );
    planner.addHighskillToQueue(new GetUp());
    return;
  }
  if(currentState == 1) {
    this.run();
  }
}
```

20.2.2 GetUpFromBacks

Príčina neustáleho vstávania a padania agenta je rovnaká ako v prípade taktiky *GetUpFromStomach*. Preto sme taktiku GetUpFromBacks upravovali rovnakým spôsobom ako taktiku GetUpFromStomach. Rozdiel je v tom, že na začiatku sa volá lowskill *fall_back*, ktorý spôsobí pád agenta na chrbát.

20.2.3 KickDistance

Taktika *KickDistance* slúži na kopanie do lopty, pričom sa hodnotí vzdialenosť, ktorú lopta prejde. V Taktike bol volaný lowskill *kick_faster_left*, ktorý však nevykonával žiadny pohyb. Taktika je preto nefunkčná. Taktiku sme opravili tak, že voláme kopanie typu *KICK_STRONG* priamo z triedy *KickHighSkill*, ktorej objekt je súčasťou triedy *Tactic*. Agent po tejto zmene úspešne kopne do lopty.

20.2.4 Turn180

Táto taktika bola vytvorená pre potreby hodnotenia disciplíny otáčania. Agent je umiestnený na začiatočnú pozíciu a po zmene režimu na *PlayOn* vykoná dve otočenia doprava pomocou highskillu *DefaultHighSkill*, ktorý volá lowskill *turn_right_90*. Potom sa agent stabilizuje lowskillom *nothing*. Problém je ale v tom, že niekedy sa agent otočí viac ako o 180 stupňov, čomu sme sa venovali v ďalšej práci.

Taktiku Turn180 sme teda prerobili nasledovným spôsobom. Vytvorili sme highskill *TurnToVector*, ktorý zistí uhol hráča (pomocou *getRotationZ()*) a φ uhol vektora, ku ktorému sa má otáčať (pomocou *getPhi()*). Odčítaním

týchto uhlov získame uhol, o ktorý sa má agent otočiť. Na základe toho, či je rozdiel kladný alebo záporný a na základe toho, na ktorej strane agenta je uhol otočenia menší, sa rozhodne o tom, či sa agent bude otáčať doprava alebo doľava. Potom sa na základe veľkosti uhla otočenia rozhodne o tom, ktorý lowskill sa vyberie, agent sa môže otáčať o 90, 45, 20, 10 alebo 4.5 stupňa. Do taktiky sme pridali statickú konštantu *TWO_TURN_ROUND*, ktorá, keď je nastavená na true, otočí hráča v zmysle popísanom v prvom odstavci, ak je false, hráč sa otočí použitím TurnToVector ku vlastnej bráne, ktorej smer je zadaný vektorom *ourGoal*. Highskill TurnToVector je opísaný v časti 28. Keďže stabilizácia po každej elementárnej otáčke trvala dosť dlho (1s), doplnili sme do TurnToVector úrovne stabilizácie a vybrali sme úroveň s časom stabilizovanie 0,6s.

20.2.5 StabilityWalkTactic

Agent kráča do cieľa daného súradnicami $(6, 1)$, ktorý leží na vzdialenejšej časti náprotivnej strany ihriska pomocou chôdze WALK_MEDIUM. Taktika obsahuje premennú RUN_ZMP, ktorá, keď je aktivovaná, robot kráča namiesto chôdze WALK_MEDIUM pomocou chôdze implementujúcej pôvodné ZMP (bez otáčania sa za cieľom) v triede WalkFastZMPOld. ZMP pritom používa chôdzu z lowskillu walk_turbo. Testy však ukázali, že chôdza pomocou ZMP je pomalšia, to je dôvod, prečo sme ponechali i možnosť chôdze pomocou WALK_MEDIUM.

20.2.6 FastWalkTactic

Agent kráča do cieľa daného súradnicami $(6, 1)$, ktorý leží na vzdialenejšej časti náprotivnej strany ihriska pomocou chôdze WALK_FAST. Taktika obsahuje premennú RUN_ZMP, ktorá, keď je aktivovaná, robot kráča namiesto chôdze WALK_FAST pomocou chôdze implementujúcej pôvodné ZMP (bez otáčania sa za cieľom) v triede WalkFastZMPOld. ZMP pritom používa chôdzu z lowskillu walk_turbo. Testy však ukázali, že chôdza pomocou ZMP je pomalšia, to je dôvod, prečo sme ponechali i možnosť chôdze pomocou WALK_FAST. Chôdza WALK_FAST je zase ale menej stabilná.

20.2.7 KickAccuracy

Táto taktika implementuje disciplínu na kopanie na presnosť. Obsahuje premennú randomSelection, ktorá, ak je nastavená na true, umožní agentovi

rozhodnúť sa, či bude kopať na roh F1R alebo roh F2R. Ak je randomSelection nastavená na false, agent kope na roh, ktorý je zadaný premennou preferredKick. Agent začína na pozícii $(-0.18, 0)$, pričom následne pri vykonávaní taktiky vykoná pre kopanie na F1R nasledovnú postupnosť krokov:

```
// step right to avoid contact with ball and stabilize
planner.addHighskillToQueue(new DefaultHighSkill("step_right"));
planner.addHighskillToQueue(new DefaultHighSkill("stabilize_normal"));
// turn to F1R
planner.addHighskillToQueue(new TurnToVector(f1r));
// step left to get to position suitable for kicking
planner.addHighskillToQueue(new DefaultHighSkill("step_left_small"));
planner.addHighskillToQueue(new DefaultHighSkill("step_left_very_small"));
// step closer to ball and compensate unwanted shift
planner.addHighskillToQueue(new DefaultHighSkill("walk_slow"));
planner.addHighskillToQueue(new DefaultHighSkill("step_left_very_small"));
// correction turn to F1R
planner.addHighskillToQueue(new TurnToVector(f1r));
// kick to ball with strong kick
planner.addHighskillToQueue(new DefaultHighSkill("kick_step_strong_right"));
planner.addHighskillToQueue(new DefaultHighSkill("stabilize_extra"));
```

Pre F2R je postupnosť krokov vykonaná symetricky. Túto postupnosť krokov sme zvolili po rôznych pokusoch s voľbami rôznych postupností highskillov.

20.2.8 KickOnGoal a KickOnXY

Tieto taktiky využívajú highskilly Walk a Kick a oproti pôvodnej verzii neboli nijako výnimočne zmenené. Došlo len k ich refaktorovaniu.

21 Analýza chybných JUnit testov

Posledná úprava	Peter Filípek
Platné od	12. apríl 2015
Poznámky	

Pri refaktorovaní kódu a nasadení automatického testovania pomocou Bamboo v predchádzajúcom semestri sme objavili 2 chybné testy a 2 testy, ktoré v určitých situáciách sú vyhodnotené negatívne. Úlohou bolo zanalyzovať chyby v testoch a situácie, pri ktorých testy dávajú negatívy výsledok. Následne podľa výsledkov analýzy vyvodiť záver a reagovať zmenou v kóde. Na úlohe pracoval Peter Filípek.

21.0.1 Analýza

Tabuľka 4: Zoznam testov

Balík	Trieda	Názov testu	Problém
sk.fiit.jim.agent.models	AgentModelTest	rotationInfer	Error
sk.fiit.jim.agent.models	AgentModelTest	positionCalculation	Error
sk.fiit.robocup.library.math	KalmanTest	reasonableNoiseCovariances	Failure
sk.fiit.robocup.library.math	KalmanTest	testCorrections	Failure

Označenie Error znamená, že počas testu dôjde k neočakávanej chybe a test je prerušený. V prostredí Eclipse sú testy s takýmto výsledkom označené červeným symbolom.

Označenie Failure znamená, že test prebehol bez chyby ale pri porovnávaní (funkcie typu assert) sa očakávaný a dosiahnutý výsledok nezhodujú. V prostredí Eclipse sú testy s takýmto výsledkom označené modrým symbolom.

Analýza testov s označením Error: Oba testy sú si podobné a nachádzajú sa v projekte Jim v triede AgentModelTest . Situácia, ktorá je testom testovaná nie je nijak okomentovaná a okrem zjavného, že sa jedná o testy rotácie a výpočtu pozície, nie sú jasné presné parametre testovanej situácie.

Oba testy boli vytvorené tímom Androids, pri hľadaní v ich dokumentácii sme nenašli presný popis testov, ktorý by nám pomohol pri riešení problému. Po stiahnutí a spustení zdrojových súborov odovzdaných na konci semestra

tímom Androids sa ukázalo, že testy boli odovzdané nefunkčné, pravdepodobne neboli včas dokončené pre iné povinnosti. Analýza pomocou nástroja na degub ukázala, že chyba nastáva v obidvoch testoch pri

- podmienke: if (b.name().equals(PLAY_MODE.name()))
- funkcia: beamablePlayMode()
- trieda: EnvironmentModel.java
- balík: sk.fiit.jim.agent.models
- hláška: <terminated, exit value: 0>

Analýza testov s označením Failure: Tieto testy sa nachádzajú ešte v nerefaktorovanom projekte RoboCupLibrary v triede KalmanTest. Problém s negatívnym vyhodnotením testov sa objavil pri automatickom testovaní pomocou Bamboo. Pri samostatnom spúšťaní testov sa negatívny výsledok neobjavil. Pri automatickom testovaní sa negatívny výsledok objavuje sporadicky a pri oboch testoch nezávisle niekedy vyjde negatívne jeden, niekedy druhý, niekedy oba.

21.0.2 Realizácia

Testy s označením Error boli z kódu odstránené.

Testy s označením Failure boli v kóde ponechané.

Vznikol tento dokument, aby informoval o stave týchto problémových testov v projekte.

21.0.3 Záver

Vzhľadom na nejasnú testovanú situáciu a dlhodobú nefunkčnosť testov bolo rozhodnuté, že testy z triedy AgentModelTest s označením Error budú z kódu odstránené. Ich oprava by bola časovo príliš náročná a vzhľadom na nejasnosť testovanej situácie tvorba nového testu nie je možná. Testy z triedy KalmanTest s označením Failure boli v projekte ponechané vzhľadom na ich funkčnosť pri samostatnom spustení. Náhodný výskyt negatívneho výsledku pri automatickom testovaní je pravdepodobne spôsobený náhodnou povahou testu, kde sa niektoré údaje náhodne generujú a následne sa skúma napríklad úspešnosť odfiltrovania šumu. V prípade že sa náhodne vygenerované veličiny

vygenerujú nevhodne, môže pravdepodobne dôjsť k negatívnemu vyhodnoteniu testu.

22 Detekcia pádu robota s využitím akcelerometra

Posledná úprava	Peter Filípek
Platné od	12. apríl 2015
Poznámky	

Na detekciu pádu robota som sa rozhodol využiť akcelerometer a pomocou údajov z neho rozhodnúť či robot padol.

V prvom rade je nutné sa pozrieť na to, aké údaje nám ponúka akcelerometer. Z akcelerometra vieme získať hodnoty x, y a z, tieto hodnoty nám hovoria o sile zrýchlenia, ktoré pôsobia na danú os a súčasne tvoria súradnice bodu kde pôsobí celková sila zrýchlenia. Akcelerometer nám poskytuje aj informácie r, phi, theta, hodnota theta je uhol azimutu, hodnota phi je polárny uhol a r je vzdialenosť daného bodu od bodu 0, 0, 0. V kľude, keď sa robot nehýbe, sú tieto hodnoty x, y, z: [0,00, 0,00, 9,81] r, phi, theta: [9,81, 4,71, 1,57]. Z týchto hodnôt je vidno, že sila gravitačného zrýchlenia pôsobí v kľude len na os Z a theta je 1,57 radiánov čo je 90 stupňov. Pri pohybe sa tieto hodnoty menia v určitom rozsahu. Najprv som analyzoval hodnoty x, y a z a vytvoril na základe nich prototyp, ktorý som následne testoval.

Prototyp na detekciu pádu podľa x, y a z som vytvoril pomocou pôvodnej funkcie, ktorá na detekciu využívala hľadanie orientačných bodov na ihrisku. Spustil som hráča s touto funkciou a nechal som si logovať údaje z akcelerometra a pôvodnú funkciu som nechával rozhodnúť o tom či robot kráča alebo spadol. Následne som tieto logy analyzoval a snažil sa nájsť thresholdy pre hodnoty y a z, pri ktorých jednoznačne viem povedať či robot stojí alebo padol. Pri hľadaní tresholdov pre tieto hodnoty som si spravil rôzne štatistiky nad logmi ako napr. priemerné hodnoty, medián, min, max, atď.

Následne som odstránil niektoré extrémne hodnoty, ktoré som považoval za chybu a znova spravil rovnaké štatistiky nad upravenými dátami. Štatistiky som robil zvlášť pre prípady kedy robot chodil a kedy robot spadol pre nájdenie rozdielov a závislostí rôznych hodnôt z akcelerometra pre tieto dva stavy.

Následne som vytvoril funkciu, ktorá detekovala pád robota podľa hodnôt Y a Z, pričom to či robot kráča sa overovalo pomocou hodnoty theta.

Táto funkcia úspešne detekovala pády hráča pri pôvodnom nastavení bez ZMP. Pri testovaní na hráčovi, ktorý pri pohybe využíva ZMP nastali prob-

Tabuľka 5: Štatistické údaje pre prípady kedy robot podľa pôvodnej funkcie kráčal s extrémnymi hodnotami

	priemer	median	smer. odch.	MIN	MAX
x	0,92028	0,31	1,323288	0	6,13
y	1,443566	0,43	2,466997	0,01	22,12
z	7,610536	9,16	4,067693	0,38	28,85
r	8,070023	9,22	4,482558	1,07	29,5
phi	2,233636	2,37	1,499515	0	6,22
tetha	1,67965	1,37	1,160849	0,71	6

Tabuľka 6: Štatistické údaje pre prípady kedy robot podľa pôvodnej funkcie kráčal s odstránenými extrémnymi hodnotami

	priemer	median	smer. odch.	MIN	MAX
x	0,487971	0,22	0,612561659	0	2,1
y	0,58	0,15	0,748228977	0,01	2,48
z	9,461449	9,4	0,418983775	9,01	11,93
r	9,537391	9,52	0,480575057	9,01	12,21
phi	2,175217	1,61	1,687426899	0	6,22
tetha	1,47913	1,53	0,083968426	1,27	1,57

Tabuľka 7: Štatistické údaje pre prípady kedy robot podľa pôvodnej funkcie spadol s odstránenými extrémnymi hodnotami

	priemer	median	smer. odch.	MIN	MAX
x	0,256875	0,115	0,285955529	0	1,26
y	9,594375	9,77	0,663489979	8,38	10,57
z	1,330938	1,11	0,783624167	0,07	3,05
r	9,72375	9,89	0,669326816	8,47	10,83
phi	3,121563	3,14	0,035388113	3,02	3,17
tetha	6,14625	6,15	0,07885266	5,97	6,28

lémy. Pridanie ZMP do pohybov hráča spôsobilo nárast extrémnych hodnôt a robilo veľký problém pri detekcii keď pri niektorých prípadoch sa na osi Y alebo Z objavovali až hodnoty cez 140. Pre tieto extrémne hodnoty Y a Z som sa rozhodol riešiť celú detekciu cez hodnotu tetha, ktorá aj pri ZMP dávala stabilné hodnoty. Po analýze hodnôt tetha a jej významu som sa rozhodol nastaviť funkciu, ktorá keď robot prekoná hodnotu 30 stupňov bude považovať robota za spadnutého, s touto hodnotou sa dá jednoducho manipulovať a výsledky sú dobré aj napríklad pre 10stupňov. Pre porozumenie hodnotám theta je lepšie keď sa pozrieme v akom rozsahu sa tieto hodnoty pohybujú pri páde a chôdzi. Podľa zalogovaných hodnôt, ktorých časť je vidno aj v Tabuľke 8 a 9 sa hodnoty pohybujú buď okolo 0.0 – 0.1 alebo 6.0 – 6.2 a naopak pri chôdzi sú približne od 0.85 po 5.6.

Tabuľka 8: Hodnoty tetha pri páde

6,06	6,06	0,03	0,03	0,03	0,03	6,02	6,19	6,19	6,19	6,07	0,08	0,08	0,08	0,05	0,09	0,09

Tabuľka 9: Hodnoty tetha pri chôdzi

0,93	0,93	0,93	1,24	1,24	1,24	1,22	1,22	1,22	1,14	1,14	1,14	1,19	1,19	1,19	1,1	1,1

23 Implementácia pohybu lopty v TestFrameworku

Posledná úprava	Miroslav Wolf
Platné od	12. apríl 2015
Poznámky	

V rámci tejto úlohy Miroslav Wolf naimplementoval do TestFrameworku simulovanie strelby. Kód bol vytvorený diplomantom Matejom Kováčom od ktorého sme kód prevzali. Simulovaná streľba môže slúžiť najmä na otestovanie brankára, ale aj na iné účely.

V rámci úlohy boli vytvorené v TestFrameworku v balíčku sk.fiit.testframework.iu dve triedy a to BallState a ShootingSimulation. Trieda BallState reprezentuje stav lopty, jej pozíciu a rýchlosť. V triede ShootingSimulation je sa nachádzajú metódy použité pre streľbu lopty z rôznych pozícií.

Okrem toho bol v triede MainFrame vytvorený kombobox na výber pozície streľby a tlačidlo "Shoot ball"pre vystrelenie lopty. Loptu je možné strieľať z nasledujúcich pozícií - náhodne, priamo, zľava z blízka, sprava z blízka, zľava z ďaleka a sprava z ďaleka. Tlačidlo a kombobox sa nachádzajú pri nastavovaní pozície lopty v karte "Server monitoring"grafického rozhrania TestFrameworku.

24 Analýza TestFrameworku

Posledná úprava	Miroslav Wolf
Platné od	12. apríl 2015
Poznámky	

Úlohe sa venovali všetci členovia tímu okrem Metoda, ktorý sa venoval implementácii ZMP.

V rámci úlohy sme sa oboznámili s testovacím frameworkom pomocou dokumentácií na wiki. Každý si prešiel dokumentáciu a následne testoval jeho funkčnosť. Od vytvorenia testovacieho frameworku však zrejme prebehlo niekoľko zmien, keďže niektoré ukážky v dokumentácii nezodpovedali nášmu testovaciemu frameworku. Keďže je kód frameworku neprehľadný, zhodli sme sa, že podobne ako v Jimovi by bolo vhodné kód testovacieho frameworku upraviť do konvencie, refaktorovať a vymazať nepotrebné časti. Padol aj návrh celý testovací framework prerobiť odznovu, keďže veľa jeho častí nefunguje alebo nepracuje správne.

V testframeworku vieme sledovať počiatočnú pozíciu lopty a hráča, aktuálnu pozíciu lopty a hráča, vzdialenosť hráča od lopty, vzdialenosť lopty od stredu ihriska, aktuálne natočenie hráča a počiatočné natočenie hráča. Vieme tiež určiť pozíciu lopty a agenta. Funkčné je tiež sledovanie hracej plochy 2D z pohľadu z hora.

Agenta sa nám však podarilo pridať na hraciu plochu iba spustením projektu Jim. V testovacom frameworku je funkcia pre pridanie hráča, no zdá sa že nefunguje. Po stlačení tlačidla pre pridanie hráča sa tlačidlo prepne do wait.. a tak už ostane.

Agenta pridáva metóda getAgent(), ak agent neexistuje, cez synchronizáciu sa získa voľný ftp port a zavolá sa metóda AgentManager : StartAgent(). Získa sa nastavenie prostredia a zadane príkazy pre robota "properties_robocup_player_command"sa načítajú do príkazov na vykonanie. Vytvorí sa nový processBuilder s príkazmi a priečinok pre agenta PROPERTIES_ROBOCUP_PLAYER_DIR a cez vlákno sa snaží pridať robota. Debugger sa po prvom kroku synchronizácie vráti ešte zopakovať synchronizáciu a ako ďalší krok vráti robota ako null.

Analyzovali sme aj časti z ruby, kde sme našli iba časť, ktorá má za úlohu automaticky spúšťať monitor a server pre rôzne operačné systémy. Táto časť sa však buď nevyužíva, alebo nefunguje. Rovnako zrejme nefungujú

ani testovacie prípady, kde po vybratí a zapnutí testovacieho prípadu sa nič nedeje.

Juraj Šimek analyzoval spúšťanie Jima, kde zistil že problém bude zrejme v príkaze ktorý spúšťa trieda ProcessBuilder a má inicializovať Jima. Testoval ho pomocou príkazového riadku. K analýze vypracoval nasledovnú správu:

- getAgent() - agentManager.java testframework.communication.agent int uniform - číslo hráča String team - meno tímu boolean blocking - TRUE ak sa nový agent spustí lokálne, v prípade že agent daného čísla pre tím neexistuje
  - získa sa voľný FTP port - getFreeTFTPPort() z localhostu a uloží do premennej port.
  - startAgent(uniform, team, true, port) - AgentManager.java int uniform - číslo hráča String team - meno tímu boolean tftp_enabled - povolí TFTP server na agentovi? int tftp_port - port na ktorom bude agentov TFTP server počúvať
  - properties_robocup_player_command - c.getProperty vyzerá to tak, že trieda C uchováva pravdepodobne nastavenia (C.java v balíku init) properties_robocup_player_command = java -classpath ../RoboCupLibrary/bin;bin;lib/aspectjrt.jar;lib/bsf.jar;lib/jruby-complete-1.4.0.jar;lib/commons-logging-1.1.jar;lib/commons-net-2.2.jar sk.fiit.jim.init.Main

    samotný príkaz v konzole nefunguje a vyhadzuje chyby, prinajlepšom chybu, že nemôže nájsť Main funkciu. Príkaz je uložený v súbore src/sk/fiit/testframework/init/default.properties.
  - List<String> command - rozbije sa príkaz v properties_robocup_player_command pomocou StringTokenizer podľa medzier
  - do command sa pridajú nastavenia pre agenta (tie, ktoré možno preťažiť volaním spustiteľnej inštancie v jej parametroch)
  - použije sa ProcessBuilder na vytvorenie príkazov pre OS (premenná builder)
  - do triedy Process sa uloží spustiteľný príkaz z builder.start() výsledok sa pridá aj do mapy bežiacich procesov zasekne sa to v EventDispatchThread.pumpEventsForFilter(int, Conditional, EventFilter)

Analýza príkazu: treba sa dostať do priečinku TestFramework a tam spúšťať príkazy v príkazovom riadku. Chyba je pravdepodobne v samotnom konzolovom príkaze. Tieto úpravy hádžu chyby:

```
java -classpath ../RoboCupLibrary/bin;bin;lib/aspectjrt.jar;lib/bsf.jar;
lib/jruby-complete-1.4.0.jar;lib/commons-logging-1.1.jar;lib/commons-net-2.2.jar
sk.fiit.jim.init.Main -runGui=true -uniform=1 -team=ANDROIDS
-TestFramework_monitor_enable=true -TestFramework_monitor_address=
127.0.0.1 -TestFramework_monitor_port=8000 -Tftp_enable=true -Tftp_port=3071
java -classpath
"..\Jim\bin;..\RoboCupLibrary\bin;bin;lib\aspectjrt.jar;lib\bsf.jar;lib\jruby
-complete-1.4.0.jar;lib\commons-logging-1.1.jar;lib\commons-net-2.2.jar"
sk.fiit.jim.init.Main -runGui=true -uniform=1 -team=ANDROIDS
-TestFramework_monitor_enable=true -TestFramework_monitor_address=
127.0.0.1 -TestFramework_monitor_port=8000 -Tftp_enable=true -Tftp_port=3071
java -classpath "..\Jim;..\Jim\bin;..\RoboCupLibrary\bin;bin;lib\
aspectjrt.jar;lib\bsf.jar;lib\jruby-complete
-1.4.0.jar;lib\commons-logging-1.1.jar;lib\commons-net-2.2.jar"
sk.fiit.jim.init.Main java -classpath
"..\Jim;..\Jim\bin;..\RoboCupLibrary\bin;bin;lib\aspectjrt.jar;lib\
bsf.jar;lib\jruby-complete
-1.4.0.jar;lib\commons-logging-1.1.jar;lib\
commons-net-2.2.jar" sk.fiit.jim.init.Main
```

treba skopírovať settings.properties z Jima aj do TestFramework, tento pokus už spustí Jima, ale vyhodí výnimku keď je Jim vykonávaný

```
java -classpath "..\Jim;..\Jim\bin;..\RoboCupLibrary\bin;bin;lib\aspectjrt.jar;lib
\bsf.jar;lib\jruby-complete-1.4.0.jar;lib\commons-logging-1.1.jar;lib\commons-net-2.2.jar"
sk.fiit.jim.init.Main -Jim_root_path='./Jim' -runGui=true -uniform=1 -team=ANDROIDS
-TestFramework_monitor_enable=true
-TestFramework_monitor_address=127.0.0.1 -TestFramework_monitor_port=8000 -Tftp_enable=true
-Tftp_port=3071
```

Zistenie: treba možno upraviť PROPERTIES_ROBOCUP_PLAYER_DIR.

24.1 Hlbšia analýza anotácií v TestFrameworku

Posledná úprava	Martin Vrabec
Platné od	2. máj 2015
Poznámky	

Aby sa vedel hráč rozhodovať, ktorý pohyb naplánovať v danej hernej situácii, musí byť informovaný o tom, čo robí ktorý pohyb (aký bude mať vplyv jeho naplánovanie na budúcnosť, ktorý z pohybov je vhodné vybrať v danej situácii). Preto bola v predchádzajúcich verziách vytvorená anotácia pre pohyby, vďaka ktorej bude možné automatizovať ich plánovanie.

Anotácie k pohybom sú ukladané v samostatných súboroch typu XML. Takto dostupné anotácie je možné pomerne jednoducho spracovávať a meniť tak človekom ako i strojom. V prípade potreby je možné takúto reprezentáciu anotácie vložiť priamo do XML súboru pohybu.

Životný cyklus anotácií

Z informácii od predchádzajúcich tímov sú známe nasledovné poznatky o procese anotácií:

1. Zbieranie údajov

   Framework pozná na základe monitorovania servera presné údaje o stave ihriska (kde je hráč, ako je otočený a pod.)

   Hráč plánuje svoje pohyby, preto vie presne povedať, kedy začal a kedy skončil vykonávanie akéhokoľvek pohybu

2. Vyhodnocovanie údajov - Testovací framework takéto informácie z oboch zdrojov dokáže spájať v zmysle zisťovania vlastností jednotlivých pohybov a exportovať na základe nich súbory s XML anotáciou pohybu (priemerné hodnoty pre každý použitý pohyb).

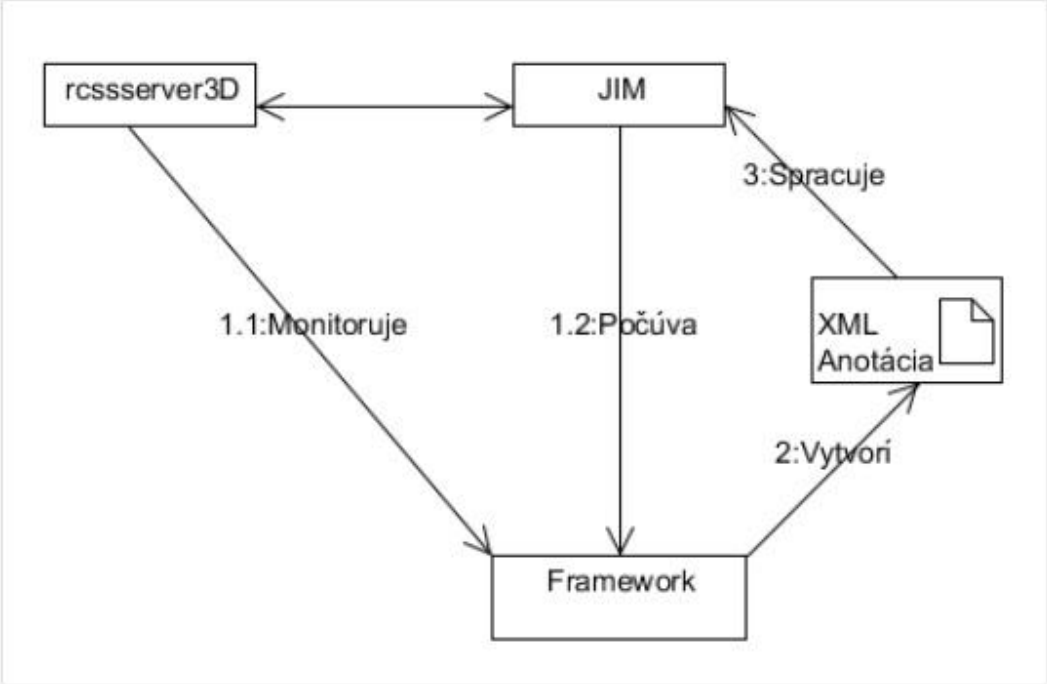
3. Spracovanie anotácie - Hráč dokáže takto vytvorenú XML anotáciu využiť pri plánovaní pohybov.

Analyzovanie funkčnosti

Pokyn na pridanie anotácie do testovacieho frameworku vykonáva používateľ kliknutím na tlačidlo Annotate na karte Annotations. Kliknutie na tlačidlo vyvolá metódu btnAnnotateClicked(), kde sa vytvára nová inštancia triedy Annotator so zadanými hodnotami z GUI okna. Na túto triedu sa zavolá metóda annotate().

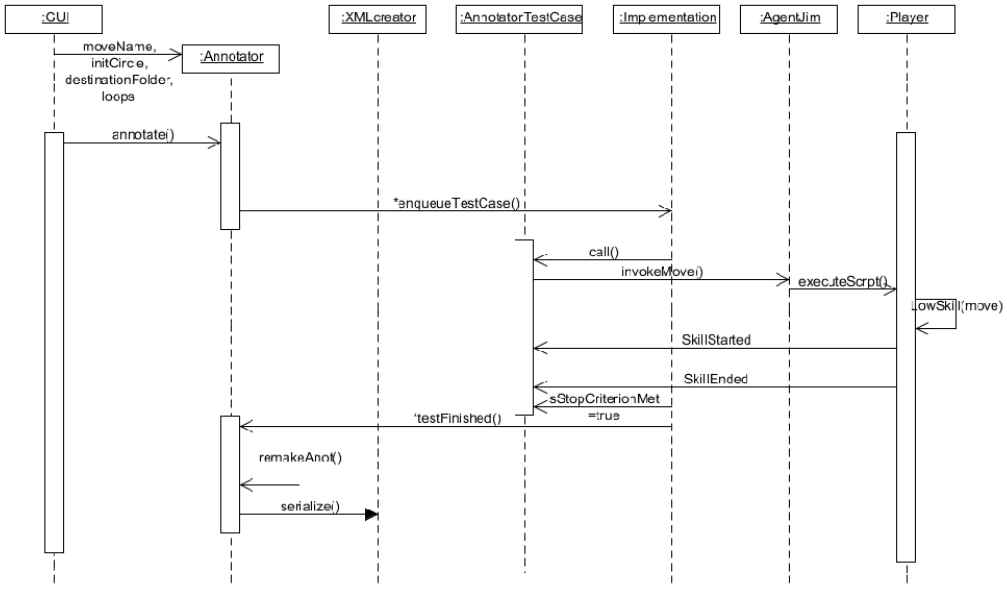
V nej sa nastaví cieľový adresár pre vytvorenú anotáciu a nasleduje blok, ktorý vytvára anotáciu, ak už rovnaká neexistuje. Nasleduje nastavenie pozície lopty pre test anotácie a na konci sa vyvolá metóda test() triedy Annotator, ktorá vytvára prostredie pre testcase pre anotáciu. Najskôr získa inštanciu triedy Implementation, ktorá je singleton a následne v cykle inicializovaných pozícií lôpt, spustí cyklus podľa nastavenej hodnoty počtu opakovaní testu anotácie, kde vytvára testCase s parametrami inicializovanej pozície lopty a názvu pohybu. Následne sa vyvolá metóda init(), ktorej úlohou je inicializovať TestCase, získať agenta a inicializovať test.Test Case sa inicializuje tak, že sa získajú inštancie monitoru, servera a servera agenta. Následne je získavaný agent, ktorý sa získava pomocou metódy getManager() triedy AgentManager, výsledkom tejto metódy je hodnota NULL, na základe čoho testframework vypíše log Init test case Failed!

Vzhľadom k tomu, že inštancia triedy Agent, ktorá vracia získaného hráča, má hodnotu NULL a s týmto hráčom sa ďalej pracuje, beh programu



Obr. 15: Životný cyklus anotovania (zdroj FIIT Robocup wiki)

nemôže ďalej pokračovať. Ak by metóda getManager() triedy AgentManager vrátila vytvoreného agenta, beh programu by pokračoval ďalej v metóde Init() triedy AnnotatorTestCase a to nastavením monitoru na vypisovanie logov typu High skill a try – catch blokom, kde sa na serveri (inštancia RobuCupServer) nastavuje hrací mód metódou setPlayMode(), pozícia lopty metódou setBallPosition() a pozícia agenta metódou setAgentPosition(), ktorá potrebuje inštanciu agenta na metódu setAgentPosition() na získanie údajov o agentovi z triedy AgentData, ktorá získa tzv. uniform číslo, stranu tímu, v ktorej je hráč (buď hrá tím sprava do ľava alebo naopak), tftp port a ip adresu tfpt. Na záver try-catch bloku sa zavolá na agentovi (inštancia triedy AgentJim) metóda invokeMove(), ktorá by mala vykonať želaný pohyb, ktorý je uvedený ako jej parameter. V tejto metóde sa vytvorí nový string builder, do ktorého sa pridá položka Plan.instance.change_skill(\ + [názov pohybu] + "\")". Následne je zavolaná metóda executeRubyScript() s touto položkou. Tento kód, ktorý požaduje jazyk Ruby, otvorí klienta tftp a odošle mu



Obr. 16: Implementácia anotácie - sekvenčný diagram (zdroj: FIIT Robocup wiki)

súbor ruby.exec s hore uvedenou požiadavkou a IP adresou tftp servera a jeho portom.Po vykonaní vyššie uvedeného skriptu by mal hráč začať vykonávať uvedený pohyb, pričom v anotácii by sa malo testovať, či boli splnené kritériá na ukončenie Test Case. Túto metódu je možné napísať pre každý Test Case a nastaviť napríklad, že ak hráč padne tak sa Test Case zruší. Pomocou triedy AnnotatorTestCaseResult sa vyhodnotia výsledky Test Case-u, pomocou ktorých sa vytvorí nová anotácia a tá sa odošle triede XMLCreator, ktorá z nej vytvorí xml súbor.

25 Detekcia pádu pomocou force receptorov na nohách agenta

Posledná úprava	Miroslav Wolf
Platné od	12. apríl 2015
Poznámky	

Agent má na obidvoch nohách senzor z ktorého vieme zistiť bod na ktorý pôsobí sila (point) a vektor sily (force). Tieto senzory vracajú hodnoty x, y, z, r, phi a theta, pričom x, y a z znázorňujú súradnice bodu, r znázorňuje vzdialenosť bodu od bodu 0, 0, 0, phi znázorňuje polárny uhol a theta znázorňuje uhol azimutu.

Pri návrhu funkcie na detekciu pádu pomocou týchto hodnôt, bolo nutné zistiť aké hodnoty nadobúdajú pri chôdzi a aké pri páde robota. S kolegom Petrom Filípkom sme teda vytvorili viacero logov z ktorých sme vybrali len pre nás zaujímavé údaje. Z týchto logov som následne vytvoril viacero štatistík - medián, minimálna hodnota, maximálna hodnota a priemerná hodnota, zapísal ich do tabuľky a hľadal zaujímavé odlišnosti medzi jednotlivými prípadmi.

Z týchto štatistík je vidno, že Y z force nedosahuje také vysoké hodnoty ako pri chôdzi, z daných logov je tiež vidno že hodnoty Y z point sú pri páde takmer vždy 0,08 (zrejme pri páde na brucho) a hodnoty phi tiež dosahujú menšie hodnoty ako pri chôdzi. Podľa týchto údajov som teda vytvorenej funkcii nastavil hraničné hodnoty.

Funkcia fungovala pri agentovi bez použitia ZMP, no mal problém so vstávaním z chrbta. So ZMP funkcia nefungovala, čo bolo zapríčinené zrejme tým, že som hraničné hodnoty nastavoval podľa logov agenta bez ZMP a pri ZMP sú zrejme odlišné.

Tabuľka 10: Údaje z Force receptorov - Force

Ľavá noha pri páde - FORCE				
MEDIAN	MIN	MAX	AVERAGE	
X	0	-28,58	6,23	-4227
Y	-4,79	-20,62	0,28	-6012
Z	-1,98	-20,89	37,12	1059
R	16,81	3,47	46,87	18436
Phi	3,14	1,62	5,03	3088
Theta	5	0,01	6,25	3851
Pravá noha pri páde - FORCE				
MEDIAN	MIN	MAX	AVERAGE	
X	-0,09	-6,56	38,8	4823
Y	-16,41	-45,35	-0,43	-13103
Z	1,09	-15,28	56,6	5979
R	18,8	5,19	66,25	22642
Phi	3,09	2,37	4,43	3234
Theta	1,35	0,08	6,28	2654
Obe nohy pri páde - FORCE				
MEDIAN	MIN	MAX	AVERAGE	
X	-0,09	-28,58	38,8	314
Y	-5,17	-45,35	0,28	-9557
Z	-0,06	-20,89	56,6	3519
R	18,22	3,47	66,25	20539
Phi	3,11	1,62	5,03	3161
Theta	4,76	0,01	6,28	3253
Ľavá noha pri chôdzi - FORCE				
MEDIAN	MIN	MAX	AVERAGE	
X	-4,43	-47,49	27,77	-10562
Y	635	-32,47	89,15	13994
Z	50765	-6,32	226,18	65977
R	52205	1,55	233,03	75613
Phi	1605	0,03	6,25	1873
Theta	1,29	0	6,28	1793
Pravá noha pri chôdzi - FORCE				
MEDIAN	MIN	MAX	AVERAGE	
X	1,54	-10,24	47,57	13684
Y	-1,75	-59,2	65,41	-2411
Z	31,53	-0,83	75,48	32985
R	44,87	10,92	103,12	47363
Phi	3,41	0,04	5,91	3153
Theta	1,22	0,01	6,26	994

Tabuľka 11: Údaje z Force receptorov - Point

Ľavá noha pri páde - POINT				
MEDIAN	MIN	MAX	AVERAGE	
X	0	-0,04	0,04	-5
Y	0,08	0,08	0,08	80
Z	0	-0,01	0,01	0
R	0,08	0,08	0,09	82
Phi	0,12	0	6,04	923
Theta	0,12	0	6,17	3032
Pravá noha pri páde - POINT				
MEDIAN	MIN	MAX	AVERAGE	
X	0	-0,04	0,04	-5
Y	0,08	0,08	0,08	80
Z	0	-0,01	0,01	0
R	0,09	0,08	0,09	86
Phi	0,46	0	6,16	1792
Theta	0,12	0	6,17	3031
Obe nohy pri páde - POINT				
MEDIAN	MIN	MAX	AVERAGE	
X	0	-0,04	0,04	-5
Y	0,08	0,08	0,08	80
Z	0	-0,01	0,01	0
R	0,08	0,08	0,09	84
Phi	0,12	0	6,16	1358
Theta	0,12	0	6,17	3032
Ľavá noha pri chôdzi - POINT				
MEDIAN	MIN	MAX	AVERAGE	
X	0,04	-0,04	0,04	-6
Y	-0,01	-0,08	0,08	-6
Z	-0,01	-0,02	0	-10
R	75	0,02	0,09	72
Phi	3,61	0	5,82	4133
Theta	6,15	0	6,17	6074
Pravá noha pri chôdzi - POINT				
MEDIAN	MIN	MAX	AVERAGE	
X	0,01	-0,04	0,04	3
Y	-0,01	-0,09	0,08	-4
Z	-0,01	-0,02	0	-10
R	0,08	0,02	0,09	65
Phi	3,27	0	6,04	3397
Theta	6,15	0	6,17	5995

Zdrojový kód funkcie na zistenie pádu pomocou force receptorov:

```
public boolean isOnGround() {

if (lastDataReceived.forceReceptor != null) {

if(( lastDataReceived.forceReceptor.leftFootForce.getY()
        < 5.0
&& lastDataReceived.forceReceptor.rightFootForce.getY()
        < 5.0)
&& (lastDataReceived.forceReceptor.leftFootPoint.getY()
        == 0.08
&& lastDataReceived.forceReceptor.rightFootPoint.getY()
        == 0.08)
&& (lastDataReceived.forceReceptor.leftFootPoint.getPhi()
        < 1.0
|| lastDataReceived.forceReceptor.leftFootPoint.getTheta()
        < 1.0
|| lastDataReceived.forceReceptor.rightFootPoint.getPhi()
        < 1.0
|| lastDataReceived.forceReceptor.rightFootPoint.getTheta()
        < 1.0))
{
        return true;
}
else return false;
}

return Angles.angleDiff(rotationX, 0.0) > (Math.PI / 4.0)
|| Angles.angleDiff(rotationY, 0.0) > (Math.PI / 4.0);
}
```

26 Analýza planera HighSkills a execute()

Posledná úprava	Metod Rybár
Platné od	12. apríl 2015
Poznámky	

V rámci tejto úlohy sme analyzovali planer, ktorý slúži na plánovanie highskillov, keďže sa vyskytol problém s plánovaním pri použití ZMP. Planer bol minulí rok upravený tímom Gitman, ktorý ho vylepšovali a najmä odstránili časť s ruby. Planner sa nachádza v balíčku sk.fiit.jim.agent.highskill.runner v triede HighSkillPlanner. Tiež sme sa pozreli na execute() metódu, ktorá vykonáva HighSkilly zaradené vo fronte. Úlohe sa venovali Miroslav Wolf a Metod Rybár.

26.0.1 Planner

Kedysi sa ako rad na plánovanie úloh používal ArrayList, no ten bol nahradený LinkedBlockingDeque radom, čo umožnilo pridávať highskilly aj na začiatok aj na koniec radu. Táto zmena bola vhodná pre pridávanie highskillu na začiatok radu, napríklad keď robot musí vstať tak na začiatok radu sa naplánuje highskill getup a predbehne tak ostatné highskilly. Plánovač beží v samostatnom vlákne a volá sa v každej iterácii cyklu ak sú na vstupe dáta zo servera.

V dokumentácii k inžinierskemu dielu tímu Gitman sa okrem iného nachádza aj opis jednotlivých metód planera:

Control() - spúšťa sa keď príde správa zo servera. Kontroluje príznak abort na prerušenie vykonávaného highskillu. Vyberá z radu nasledujúce highskilly na vykonanie a začne ich vykonávať.

addHighskillToQueue(Highskill highskill) - metóda na pridávanie highskillu na koniec radu pre taktickú vrstvu.

addHighskillAsFirst(Highskill highskill) - metóda na pridávanie highskillu na začiatok radu pre taktickú vrstvu.

getCurrentHighskill() - vráti aktúalne vykonávaný highskill.

getNextHighskill() - vráti nasledujúci naplánovaný highskill.

abortPlanedHighskills() - ukončí práve vykonávaný highskill a rad vymaže.

Vykonávané highskilly sa musia vždy dokončiť, aby nenastala situácia, že robot sa v polovici vykonávania highskillu zasekol a začal vykonávať iný highskill - čo by mohlo spôsobiť jeho pád. Na vykonanie a ukončenie highskillu slúži metóda execute().

V rámci úlohy sme sledovali logy, ako sa planer správa v rôznych situáciách. Z logov je vidno, že v planery je väčšinou 0 až 3 highskillov a to higskilly ako "walkfast", "walkslow", "getup", "kick", "localizeä "beam". V logoch sme problém nezaznamenali a zdá sa, že highskilly sa plánujú korektne. Po vykonaní highskillu ktorý je prvý je tento highskill vymazaný z radu a na jeho mieste je následne druhý highskill.

26.0.2 HighSkill execute()

Pri vykonávaní stavu sa využívajú stavy zadefinované enumom private enum HighSkillState { INITIAL_STATE, EXECUTING_STATE, FINALIZING_STATE, END_STATE }.

Tieto stavy sa následne v metóde execute() využívajú na určenie ako sa bude pri vykonávani HighSkillu ďalej vykonávať.

case INITIAL_STATE: - Pri inicializácií sa zavolá pickLowskill(). Vrátený LowSkill sa zresetuje a začne sa vykonávať. Stav HighSkillu sa mení na EXECUTING_STATE.

case EXECUTING_STATE: - Ak sa HighSkill vykonáva a aktuálna fáza LowSkillu má nastavený parameter isFinal=True, vyberie sa ďalší LowSkill volaním pickLowskill(), ak ide o rovnaký LowSkill pokračuje sa ďalej, ak ide o iný LowSkill, prechádza HighSkill do stavu FINALIZING_STATE a spúšťa finalizáciu LowSkillu pomocou executeFinalisation(). Rovnako sa LowSkill začne finalizovať, ak isStoppedHighSkill() vráti pravdu. Ak isFinal=True nie je parametrom aktuálnej fázy, pokračuje sa vo vykonávaní LowSkillu a HighSkill zostáva vo fáze EXECUTING_STATE.

case FINALIZING_STATE: - Ak je HighSkill vo finalizačnom stave, začne sa vykonávať nový vybraný LowSkill a prechádza naspäť do EXECUTING_STATE.

V stave EXECUTING_STATE sa volá funkcia checkProgress(), ktorá ale nemá implementované zachytávanie výnimiek. Ak funkcia checkProgress() zachytí výnimku, mala by umožniť ukončiť aktuálny LowSkill a vynútiť si vybratie nového.

27 Tvorba HighSkills a zakomponovanie Zero moment point

Posledná úprava	Metod Rybár
Platné od	24. apríl 2015
Poznámky	

27.0.1 Základná štruktúra

HighSkills musí dediť od triedy HighSkill z balíka sk.jim.fiit.agent.skills prípadne od triedy ComplexHighSkill z rovnakého balíka. Tieto abstraktné triedy ponúkajú základnú funkcionalitu vyžadovanú od každého HighSkillu.

Medzi najdôležitejšie metódy patrí funkcia pickLowSkill() a checkProgress(). Ďalej medzi hlavné elementy ktoré HighSkill potrebuje patria

private Logger LOG = JLog.getLogger();

protected AgentInfo agentInfo = AgentInfo.getInstance();

protected AgentModel agentModel = AgentModel.getInstance();

protected WorldModel worldModel = WorldModel.getInstance();

protected HighSkillPlanner planner = HighSkillPlanner.getInstance();

Hlavnou úlohou je implementácia metódy pickLowSkill(). Táto metóda sa volá zakaždým, keď skončí vykonávanie aktuálne vybraného pohybu, čo nastane vtedy, kedy je pri vykonávaní splnená podmienka isFinal=True.

Metóda pickLowSkill() musí vždy vrátiť objekt typu LowSkill(). LowSkill ktorý je vrátený je následne vykonávaný až do jeho ukončenia alebo pokiaľ sa znova nezavolá z plánovača metóda pickLowSkill().

Preto by mali byť v tejto metóde definované pravidlá, ktoré definujú kedy sa agent otáča tak aby sa stále pohyboval za cieľom alebo iné potrebné vlastnosti. Základná štruktúra je napríklad v triedach Walk alebo WalkFast.

Metóda checkProgress() je volaná počas vykonávania aktuálnej fázy HighSkillu, teda aktuálneho LowSkillu. Môžeme v nej definovať pravidlá, ktoré ovplyvňujú tento pohyb, ako napríklad dynamická zmena polohy kĺbov. Táto metóda sa teda využíva aj na stabilizáciu pohybu pomocou Zero moment point.

27.0.2 Postupnosť vykonávania HighSkillu a jeho implementácia

V metóde pickLowSkill() sa vyberá LowSkill, ktorý sa bude vykonávať. Rovnako by tu mala byť zakomponovaná kontrola toho, či je agent na zemi pomocou zistenia tohto stavu cez agentModel.isOnGround(), kedy treba zavolať HighSkill GetUp(). Toto vykonáme jeho priradením na začiatok fronty HighSkills pomocou planner.addHighskillAsFirst(new GetUp()).

Ak chceme agenta dostať do iniciálnej stojacej polohy, môžeme vrátiť LowSkill rollback cez return LowSkills.get("rollback").

Rovnako je dôležité kontrolovať agentovu polohu a smer pohybu vzhľadom k jeho cieľu. Preto by sa v pickLowSkill() mala volať metóda computeRelativeTarget(), aby sme následne vedeli používať hodnoty ako x-ové a y-ové vzdialenosti k cieľu.

Tiež je dôležité do metódy getLowSkillName() zadať názov defaultného LowSkillu pre daný HighSkill. Napríklad

```
@Override
protected String getLowSkillName() {
return "walk_dynamic";
}
```

27.0.3 Stabilizácia pomocou Zero moment point

Stabilizáciu pohybu pomocou Zero moment point môžeme vykonávať v metóde checkProgress(). Najskôr si z AgentModel získame Zero moment point informácie pomocou Vector3D zmp = agentModel.getZeroMomentPoint().

Následne si nainicializujeme receptory na nohe ak ešte inicializované neboli.

```
if (agentModel.getForceReceptor().rightFootForce == null)
{
rfFRP = zero_vector;
} else {
rfFRP = agentModel.getForceReceptor().rightFootForce;
}

if (agentModel.getForceReceptor().leftFootForce == null)
{
lfFRP = zero_vector;
```

```
} else {
lfFRP = agentModel.getForceReceptor().leftFootForce;
}

if (agentModel.getForceReceptor().rightFootPoint == null)
{
rpFRP = zero_vector;
} else {
rpFRP = agentModel.getForceReceptor().rightFootPoint;
}

if (agentModel.getForceReceptor().leftFootPoint == null)
{
lpFRP = zero_vector;
} else {
lpFRP = agentModel.getForceReceptor().leftFootPoint;
}
```

Následne v metóde checkProgress() zavoláme metódu angle_proc(), ktorá vypočíta uhol k cieľu a využíva sa pri modifikácií uhlov na kĺboch na rukách agenta.

```
private double angle_proc() {
angle_to_target = 0;
Vector3D target = Vector3D.cartesian(0, 0, 0);
Vector3D vector_to_target = (target.subtract
(agentModel.getPosition())).flatten();
Vector3D vector_forward = agentModel.
globalize(Vector3D.Y_AXIS).flatten();
Vector3D vector_diff =
(vector_to_target.subtract(vector_forward)).flatten();
Vector3D vector_diff_rel =
agentModel.relativizeVector(vector_diff).flatten();
angle_to_target = vector_diff_rel.getPhi();
if (angle_to_target >= PI) {
angle_to_target = (angle_to_target - 2 * PI);
}
angle_to_target = angle_to_target * 180 / PI;
return angle_to_target;
```

```
}
```

Ďalej si vypočítame rollX a pitchY potrebné pri stabilizácií.

```
double rollX = rfFRP.getX() + lfFRP.getX();
double pitchY = rfFRP.getY() + lfFRP.getY();
```

Nakoniec vykonáme stabilizáciu. Tu môžeme pre každú vázu vykonávať buď stabilizáciu rúk alebo stabilizáciu nôh, prípadne obe naraz. Preto je vhodné využiť switch - case statment ako napríklad:

```
switch (active_phase.name) {
        case "wd_1":
        handsStabilization(rollX, pitchY);
        case "wd_2":
        handsStabilization(rollX, pitchY);
        footStabilization(zmp);
        case "wd_3":
        handsStabilization(rollX, pitchY);
        case "wd_4":
        handsStabilization(rollX, pitchY);
        footStabilization(zmp);
}
```

Na záver si zapamätáme aktuálne údaje zo senzorov

```
prev_rfFRP = rfFRP;
prev_lfFRP = lfFRP;
prev_rpFRP = rpFRP;
prev_lpFRP = lpFRP;
```

Metódy pre stabilizáciu rúk a nôh vyzerajú nasledovne

```
private void handsStabilization(double rollX, double pitchY)
{
double rae2 = agentModel.getRAE2() - rollX * 2;
double lae2 = agentModel.getLAE2() - rollX * 2;

rae2 += (abs(rollX) < 3) ? 1.5 : 0;
lae2 -= (abs(rollX) < 3) ? 1.5 : 0;

// handle rotation to target by adjusting arm positions
if (abs(angle_to_target) >= 10) {
```

```
if (angle_to_target < 0) {
rae2 -= 25;
lae2 -= 10;
} else {
lae2 += 15;
rae2 += 10;
}
}

if (rae2 < -95) {
rae2 = -95;
} else {
if (rae2 > 1) {
rae2 = 1;
}
}

if (lae2 > 95) {
lae2 = 95;
} else {
if (lae2 < -1) {
lae2 = -1;
}
}

ComputedValues.set(new ComputedValue("rae2"), rae2);
ComputedValues.set(new ComputedValue("lae2"), lae2);
}

private void footStabilization(Vector3D zmp) {
double correction = -0.25;
double side_correction = 0.01;
double threshold = 0.04;
double side_threshold = 0.03;

double foot_coorection_angle = 0;
double foot_coorection_side_angle = 0;
```

```
double rle2 = 0;
double rle3 = 0;
double lle2 = 0;
double lle3 = 0;

if (rfFRP.getZ() > 0) {
// prava je na zemi
if (abs(rpFRP.getY()) >= threshold) {
foot_coorection_angle = correction
* (rpFRP.getY() > 0 ? 1.0 : -1.0);
}
if (abs(rpFRP.getX()) >= side_threshold) {
foot_coorection_side_angle = side_correction
* (rpFRP.getX() > 0 ? 1.0 : -1.0);
}
if (abs(zmp.getY()) > 0.03) {
rle3 = agentModel.getRLE3() + 1;
rle2 = agentModel.getRLE2() + 0.03;
}
} else {
if (lfFRP.getZ() > 0) {
// lava je na zemi
if (abs(lpFRP.getY()) >= threshold) {
foot_coorection_angle = correction
* (lpFRP.getY() > 0 ? 1.0 : -1.0);
}
if (abs(lpFRP.getX()) >= side_threshold) {
foot_coorection_side_angle = side_correction
* (lpFRP.getX() > 0 ? 1.0 : -1.0);
}
if (abs(zmp.getY()) > 0.03) {
lle3 = agentModel.getRLE3() + 1;
lle2 = agentModel.getRLE2() + 0.03;
}
} else {
foot_coorection_angle = 0;
foot_coorection_side_angle = 0;
}
```

```
}

ComputedValues.set(new ComputedValue("rle2"), rle2);
ComputedValues.set(new ComputedValue("lle2"), lle2);
ComputedValues.set(new ComputedValue("rle3"), rle3);
ComputedValues.set(new ComputedValue("lle3"), lle3);
// ready to phases 2 and 4 - no more
ComputedValues.set(new ComputedValue("rle5"),
(40.0 + foot_coorection_angle));
ComputedValues.set(new ComputedValue("lle5"),
(40.0 + foot_coorection_angle));
ComputedValues.set(new ComputedValue("rle6"),
(-7.0 + foot_coorection_side_angle));
ComputedValues.set(new ComputedValue("lle6"),
(-7.0 + foot_coorection_side_angle));

}
```

Implementáciu stabilizácie pomocou Zero moment point je možné vidieť v HighSkille WalkFastZMP.

27.0.4 Testovanie HighSkills

HighSkill sa dajú testovať pomocou nastavenia si testovacej taktiky. Tá sa dá zapnúť nastavením debugTactic na true v triede Settings a nastavením planner.addHighskillToQueue(HighSkill) v metóde run() triedy DefaultTactic s HighSkillom, ktorý chceme testovať. Následne sa bude vykonávať vždy len nami definovaný HighSkill.

27.0.5 Testovanie pádu

Trieda AgentModel ponúka metódu isOnGround opísanú v 22. Pre ZMP je nutné nastaviť extrémnejšie parametre ako pre nedynamické chôdze, a to z dôvodu, že pri dynamickej chôdzi je agent viac rozkývaný a teda akcelerometer zaznamenáva vyššie hodnoty. Ako funkčné boli testované hodnoty *lastDataReceived.accelerometer.getTheta()) < 0.04* a *lastDataReceived.accelerometer.getTheta()) > 6.23*, ale pre každý nový dynamický pohyb môžu byť hodnoty iné.

28 Otáčanie hráča

Posledná úprava	Juraj Šimek
Platné od	18. apríl 2015
Poznámky	

Keďže pre potreby otáčania hráča nebol implementovaný žiadny spoľahlivý highskill, rozhodli sme sa ho implementovať. Otáčanie hráča sme implementovali do highskillu *TurnToVector* tak, ako je opísané v ďalších častiach.

28.0.6 Vykonávanie otáčania hráča

Agentovi je pomocou konštruktora highskillu zadaný vektor typu *Vector3D*, ku ktorému sa má otočiť. Highskill sa vykonáva pomocou metódy *pickLowSkill()*, ktorá vracia lowskill, ktorý sa má aktuálne vykonávať. Agent pomocou metódy *getPosition()* triedy *AgentModel* zistí svoju pozíciu. Keď odpočítame agentovu pozíciu od pozície, ku ktorej sa má otočiť, získame *vektor otočenia*, ku ktorému sa má agent otočiť vzhľadom na svoju polohu. *Uhol otočenia* potom získame ako rozdiel rotácie agenta, ktorú získame pomocou metódy *getRotationZ()* triedy AgentModel a uhla φ vektora otočenia, ktorý sme vypočítali:

```
Vector3D agentPosition = agentModel.getPosition().setZ(0.0);
Vector3D direction = position.subtract(agentPosition);
double difference = Math.toDegrees(agentModel.getRotationZ()
                  - direction.getPhi());
```

Následne sa zistí, do ktorej strany je výhodnejšie, aby sa agent otočil. Otočenie sa vykonáva do tej strany, do ktorej agent musí vykonať menší uhol otočenia k želanému vektoru. Toto rozhodovanie zachytáva nasledovný úsek kódu:

```
String lsInfix = "left";

// find out whether turn left or right (what is better)
if (difference >= 0) {
        if (difference > 180) {
                difference = 360.0 - difference;
                lsInfix = "left";
        }
        else {
```

```
                lsInfix = "right";
        }
}
else {
        difference = Math.abs(difference);
        if (difference > 180) {
                difference = 360.0 - difference;
                lsInfix = "right";
        }
        else {
                lsInfix = "left";
        }
}
```

Po zistení strany otáčania sa vykonáva samostatné otáčanie pomocou lowskillov podľa veľkosti ostávajúceho uhla otočenia. Agent sa vie otáčať o *90, 45, 20, 10* a *4.5* stupňa. Agent sa k želanému vektoru otočí *s presnosťou 5 stupňov*. V prípade sekvencie otočení o 4.5 stupňa sa počíta počet týchto malých otočení. Agent sa niekedy kvôli nim zasekne a nevie sa trafiť do 5 stupňovej tolerančnej hranice. Ak agent vykoná viac ako 5 malých otočení, vykoná jedno 20 stupňové otočenie a pokúsi sa na vektor nasmerovať odznova.

28.0.7 Stabilizovanie hráča počas otáčania

Pri testovaní agenta sme zistili, že je dosť nestabilný, najmä pri opakovaní častých malých otočení. Túto skutočnosť sme sa rozhodli vyriešiť tak, že po každom otočení pomocou nejakého lowskillu sa vykoná stabilizačná fáza pomocou lowskillu nothing, ktorý už bol opísaný v časti 20.2. Táto stabilizácia ale trvá vždy 1s a tak sme sa rozhodli doplniť lowskilly *stabilize_low (0,47s), stabilize_normal (0,6s), stabilize_high (1s)* a *stabilize_extra(2s)*, ktoré agenta stabilizujú s rôznymi časmi ich vykonávania uvedenými v zátvorkách. Lowskilly boli vytvorené po lepšej analýze lowskillu nothing, na jeho základe. Čím dlhší je čas vykonávania lowskillu stabilize, tým stabilnejší agent bude. Vytvorili sme druhý konštruktor, kde možno nastaviť úroveň stabilizácie v rozmedzí 0 – 4. V prípade, že je úroveň 0, nevykonáva sa žiadna stabilizácia, v ostatných prípadoch sa vyberú opísané lowskilly stabilizácie. Defaultná hodnota pri volaní konštruktora bez zadania hodnoty stabilizácie je stabilizovanie pomocou lowskillu *stabilize_high*.

28.0.8 Testovanie

Agenta sme testovali manuálne tak, že sme mu uviedli vektory, ku ktorým sa má otáčať v každom kvartáli ihriska, i na hraniciach súradnicovej osi. Vždy sa mu podarilo otočiť sa daným smerom s 5 stupňovou toleranciou. Pri stabilizovanom otočení musel vykonať menej lowskillov otočenia a pády boli skôr výnimočné. Implementácia highskillu TurnToVector nespôsobila nefunkčnosť žiadnych unit testov.

29 Nefunkčnosť anotovania hráčov

Posledná úprava	Martin Vrabec
Platné od	23. apríl 2015
Poznámky	

Keďže anotovania hráčov v testframeworku nie je funkčná, rozhodli sme sa zanalyzovať tento problém a prípadne ho odstrániť. Anotácia umožňuje vytvárať pravidlá pre pohyby, vďaka ktorým je možné automatizovať plánovanie pohybu hráčov podľa danej hernej situácie. Anotácia je zapísaná v XML súbore, ktorý vytvorí testovací framework a spracuje samotný agent Jim.

29.0.1 Analýza

Anotovať hráča je možné v testovacom frameworku na karte Annotations, kde je možné vybrať názov pohybu, cieľový priečinok anotácie, počet opakovaní testu a nastavenie pozície lopty, ktorá je reprezentovaná kruhom s určitým polomerom a hodnotami osí X a Y. Kliknutie na tlačidlo Annotate by malo spustiť automatickú anotáciu hráča. Po stlačení tlačidla sa však vo výstupe logov objaví najskôr hláška, že bola vytvorená čistá anotácia (Created blank annotation), nasledujúca hláška z logov je, že test case bol supstený (sk.fiit.testframework.annotator.AnnotatorTestCase: Running test case), ďalej informačný hláška, že agent je vkladaný do test frameworku a napokon varovanie, že test case zlyhal. Pri použití debuggeru z Eclipse IDE bolo zistené, že po kliknutí na tlačidlo Annotate sa vyvolá metóda btnAnnotateClicked(), kde sa získajú a nastavia hodnoty pre anotáciu a vytvára sa nová inštancia Annotator() s týmito nastavenými hodnotami, na ktorú sa zavolá metóda annotate(). V nej sa nastaví cieľový adresár pre vytvorenú anotáciu a nasleduje blok, ktorý vytvára anotáciu, ak už rovnaká neexistuje. Nasleduje nastavenie pozície lopty pre test anotácie a na konci sa vyvolá metóda test() triedy Annotator, ktorá vytvára prostredie pre testcase pre anotáciu. Najskôr získa inštanciu triedy Implementation, ktorá je singleton a následne v cykle incializovaných pozícií lôpt, spustí cyklus podľa nastavenej hodnoty počtu opakovaní testu anotácie, kde vytvára testCase s parametrami inicializovanej pozície lopty a názvu pohybu. Následne sa vyvolá metóda init(), ktorej úlohou je inicializovať TestCase, získať agenta a inicializovať test.Test Case

sa inicializuje tak, že sa získajú inštancie monitoru, servera a servera agenta. Následne je získavaný agent, ktorý sa získava pomocou metódy getManager() triedy AgentManager, výsledkom tejto metódy je hodnota NULL, na základe čoho testframework vypíše log Init test case Failed!

29.0.2 Riešenie

Podobný problém s nefunkčným pridávaním hráča sa vyskytol aj pri samotnom pridávaní hráča do test frameworku. Tento problém bol vyriešený úpravou konfiguračných parametrov. Keďže pridávanie hráčov do test frameworku a pridávanie hráčov pre anotovanie používajú rovnakú metódu, len pri anotovaní hráča sa kontroluje, či agent existuje, a keďže metóda vytvárajúca agenta stále vracia hodnotu null, čo sa pri samotnom pridávaní hráča do test frameworku nekontroluje, anotovanie nefunguje. Predpokladal som, že úpravou nastavení sa chovanie metódy na pridanie hráča do test frameworku podarí zmeniť, aby metóda vracala vytvoreného agenta. Ani po niekoľkých pokusoch so zmenou nastavení sa však funkčnosť nepodarilo zmeniť.

30 Pridanie vynúteného prerušenia HighSkills

Posledná úprava	Metod Rybár
Platné od	24. apríl 2015
Poznámky	

Pri analýze predchádzajúcich tímov a diplomových prác sme odhalili, že hráč vo verzii kedy taktiky a manažment HighSkills bol implementovaný v Ruby, existovala možnosť počas behu LowSkill vo vykonávanom HighSkille tento LowSkill prerušiť. Toto je vhodné napríklad ak vykonávaný LowSkill je dlhší a nemá veľa fáz, ktoré sú finalize, a teda jeho priebeh musí byť neprerušený. Ak však počas takéhoto vykonávania agent spadne, zbytočne bude na zemi vykonávať aktuálny LowSkill, a preto je vhodné ho prerušiť a dať agentovi pokyn aby vstal. Podobných situácii na využitie tejto možnosti môže byť viacero.

Samotné prerušenie vykonávaného LowSkillu možeme vykonať ak vyhodíme Exception v checkProgress() v danom HighSkille. Táto metóda beží počas vykonávania LowSkillu a teda vieme kontrolovať jeho priebeh a stav agenta. Príklad ako to môžme vykonať

```
if(agentModel.isOnGround()){
        throw new Exception("Agent has fallen");
}
```

Je vhodné v Exception popísať dôvod vynúteného prerušenia LowSkillu.

Samotné prerušenie LowSkillu prebieha v metóde execute() v triede HighSkill, ktorá slúži na výber a vykonávanie LowSkillow pre všetky HighSkills. Vyhodená Exception sa zaloguje a vynúti sa vybratie nového LowSkillu a prerušenie aktuálneho.

```
try {
checkProgress();
} catch (Exception e) {
LOG.log(LogType.HIGH_SKILL,
"HighSkill "
+ this.getName()
+ " was interupted while checkProgress() with exception "
+ e);
        pickLowSkill();
```

```
            break;
}
```

Netreba zabúdať, že v metóde pickLowSkill() v danom HighSkille musí byť napríklad spôsob, ktorým sa agent postaví alebo napraví situáciu, pre ktorý bol predchádzajúci LowSkill prerušený, a to napríklad takto

```
if (agentModel.isOnGround() || agentModel.isLyingOnBack()
|| agentModel.isLyingOnBelly()) {
            planner.addHighskillAsFirst(new GetUp());
            return LowSkills.get("rollback");
}
```

31 Návrh na ďalšiu prácu na hráčovi Jim

Posledná úprava	Miroslav Wolf
Platné od	16. máj 2015
Poznámky	

Pokračovanie v práci Keďže projekt RoboCup sa stále vyvíja, je na tomto projekte ešte mnoho vecí nedotiahnutých a tak budúcich vývojárov čaká ešte mnoho ďalších možných úprav. Náš tím sa zameriaval najmä na úpravu kódu do konvencií, odstráneniu nepotrebných častí a teda sme sa snažili kód sprehľadniť pre ďalšiu prácu. Okrem toho sme zakomponovali ZMP pre stabilnejšiu rýchlu chôdzu, upravili detekciu pádu, upravili a vylepšili logovanie, analyzovali, opravovali a dopĺňali nové funkcie do TestFrameworku a mnoho ďalších vecí. Je tu však niekoľko častí, ktorým sa náš tím nestihol venovať a ktoré by bolo vhodné do budúcnosti doplniť a opraviť.

Pri analýze TestFrameworku sme si všimli niekoľko nedostatkov, z ktorých náš tím opravil len niektoré. Jednou z neopravených chýb sú nefunkčné anotácie. Nefunkčnosť anotácií náš tím zanalyzoval. Pri tejto analýze sme zistili, že pravdepodobnou príčinou nefunkčnosti anotácií je, že využívali časti z ruby, no tie už sú odstránené. Podrobnejšiu analýzu môžete nájsť v inžinierskom diele nášho tímu.

Ďalším nedostatkom TestFrameworku je prepínanie medzi testovacími prípadmi, ktoré nefunguje. Tomuto problému sa náš tím nevenoval.

V TestFrameworku tiež nebolo možné pridávanie hráčov, no tento problém sa nášmu tímu podarilo odstrániť. Vznikol však ďalší problém a to že prvý pridaný hráč je zaseknutý a nič nevykonáva. Ďalší pridaní hráči sú však už v poriadku.

Venovali sme sa aj príprave na fakultný turnaj RoboCupu. V TestFrameworku sme vytvorili niekoľko meraní pre turnaj, no niektoré ešte stále chýbajú a to – freeride, obchádzanie prekážok a kop na bránu s orientáciou. Okrem toho chýbajú pre spomenuté merania chýbajú aj taktiky.

Náš tím doplnil ZMP medzi pohyby – jedná sa o stabilnejší rýchly pohyb. Tento pohyb však nie je zakomponovaný v žiadnej z taktík a agent ho teda nevyberá. Je teda potrebné upraviť, prípadne vytvoriť novú taktiku ktorá tento pohyb bude využívať.

Jedným z problémov je aj to, že ak hráč nemá naplánované žiadne akcie a stojí na mieste, začne strácať rovnováhu a pomaly sa sklápať. Okrem toho

hráč má problémy, keď má loptu pod sebou a nevidí na ňu. Do budúcna by teda tiež mohlo byť vhodné otestovať hráča v predklone, prípadne pohyb hlavy.

Príloha A: Inštalačná príručka

Posledná úprava	Metod Rybár
Platné od	24. apríl 2015
Poznámky	

Úvod

Túto inštalačnú príručku vytvoril tím Infinity a má slúžiť pre tých, ktorý budú náš projekt testovať a pre budúce tímy, ktoré sa budú podieľať na tomto projekte. Príručka spája niektoré ostatné staršie návody na inštaláciu projektu a upravuje ich, aby korešpondovali so súčasným stavom projektu.

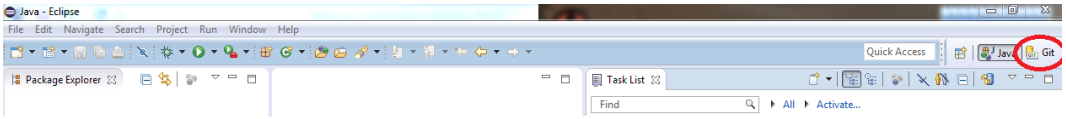
Príručka obsahuje importovanie projektu do nástroja eclipse, inštaláciu potrebných súčastí, spúšťanie samotného projektu a jeho častí a otestovanie niektorých konkrétnych funkcionalít a situácií. Okrem toho bola stručne opísaná práca v TestFrameworku, práca nášho tímu na projekte a niektoré problémy, ktoré náš tím nestihol odstrániť. V prípade akýchkoľvek nejasností, prípadne vzniknutých problémov sa ozvite na mail tímu team8@centrum.sk.

Importovanie projektu

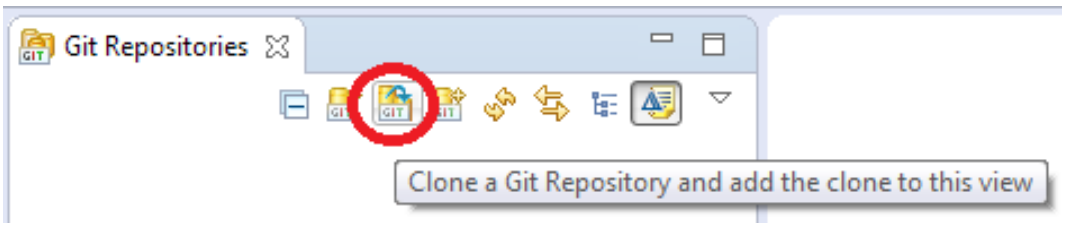
V tejto kapitole je v krokoch opísané importovanie projektu. Náš tím pracoval na projekte prostredníctvom Eclipse Luna a preto bude v návode opísaný postup práve pre toto vývojové prostredie. Projekt funguje aj s inými verziami Eclipse, len je potrebné mať nainštalovaný plugin pre Git, ktorý posledná verzia Eclipse Luna už obsahuje. Projekt by mal fungovať aj s prostredím Netbeans, no s tým mal náš tím problémy a preto odporúčame použitie Eclipse Luna.

V nasledujúcich krokoch je opísaný postup pre importovanie projektu do Eclipse:

1. Spusťte Eclipse
2. Vytvorte si pre projekt workspace
3. V pravom hornom rohu sa prepnite do Git (musíte mať nainštalovaný plugin pre Git, alebo používať Eclipse Luna)



4. Následne je potrebné si naklonovať repozitár z nášho Bitbucketu na váš lokálny disk. Na toto potrebujete prístup do nášho Bitbucketu. Ak ho nemáte ozvite sa nám na tímový mail team8@centrum.sk. Pre klonovanie stlačte vľavo hore ikonku „Clone a Git repository...“ podľa obrázka:



5. Do poľa URI skopírujte nasledovný odkaz na branch nášho projektu (`https://bitbucket.org/robocup_tp09/agent/branch`), polia Host a Repository path sa vyplnia automaticky. Zadajte ešte do polí User a Password vaše meno a heslo, ktoré používate pri prihlasovaní na účet v Bitbucket-e, v ktorom máte prístup k nášmu projektu a kliknite na tlačidlo Next.

6. V nasledovnom kroku stlačte tlačidlo Deselect All a následne zaškrtnite iba branch Master a stlačte tlačidlo Next.

7. Zvoľte si cestu, kam sa vám branch má naklonovať. Stlačte tlačidlo Finish a počkajte kým sa branch naklonuje.

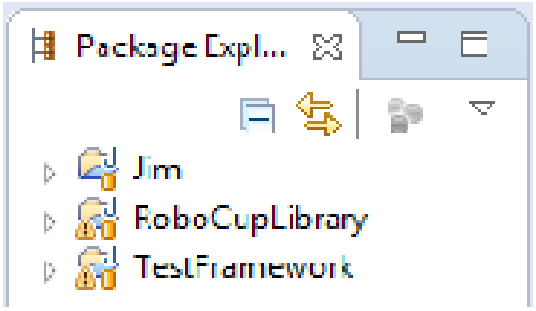
8. Teraz by ste mali vľavo hore vidieť naklonovaný branch. Otvorte si zložku kam ste si naklonovali branch (defaultne - C:\Users\[username]\git\master). Zo zložky TestFramework skopírujte súbor s názvom „.project“ do zložky Jim, skopírovaný súbor otvorte v textovom dokumente a prepíšte tretí riadok z <name>TestFramework</name> na <name>Jim</name> a súbor uložte.

9. Vráťte sa do Eclipse, kde kliknete pravým tlačidlom na myši na naklonovaný branch vľavo hore a zvolíte Import Projects...

10. Nechajte zvolené Import Existing Projecťs a stlačte tlačidlo Next.

11. Mali by ste vidieť tri projekty a to Jim, RoboCupLibrary a TestFramework. Nechajte pri nich políčka zaškrtnuté a stlačte tlačidlo Finish.

12. Projekty by sa vám mali naimportovať do eclipse. Teraz sa už len vpravo hore prepnite naspäť z Git na Java a mali by ste vidieť naimportované tri projekty.



Inštalácia potrebných súčastí projektu

V tejto kapitole je v krokoch opísaná inštalácia súčastí projektu, ako aj odkazy z ktorých si súčasti nainštalujete. Inštalácia je opísaná pre Windows, Linux a Mac, no inštaláciu na Linux a Mac náš tím neskúšal a teda v tomto návode je len prepísaný návod z wiki, ktorý vytvoril niektorý z minuloročných tímov.

Inštalácia na Windows

V nasledujúcich bodoch je opísaná inštalácia potrebných súčastí na Windows. Inštalácia funguje pre 32 aj 64 bitovú verziu Windowsu.

1. Ako prvé si stiahnite potrebné inštalačné súbory:

   MS Visual C++ 2008 Redistributable Package (x86): `http://www.microsoft.com/en-us/download/details.aspx?id=29` (nepovinné)

   Simspark 0.2.4: `http://sourceforge.net/projects/simspark/files/simspark/`

   rcssserver3d 0.6.7: `http://sourceforge.net/projects/simspark/files/rcssserver3d/`

   Ruby - `http://rubyinstaller.org/downloads/`

2. V poradí v akom sú uvedené ich nainštalujte.

3. Reštartujte počítač.

4. Choďte do zložky bin, ktorá sa nachádza v zložke kde ste nainštalovali rcsserver (C:\Program Files (x86)\rcssserver3d 0.6.7\bin) a spustite rcssserver3d.cmd a rcssmonitor3d.cmd v tomto poradí. Malo by sa vám zobraziť okno so simulačným prostredím - futbalovým ihriskom. Potom spustite rcssagent3d.cmd a mal by sa na ihrisko pridať hráč, ktorý však slúži iba na otestovanie že všetko funguje správne.

Oficiálny návod nájdete tu: `http://simspark.sourceforge.net/wiki/index.php/Installation_on_Windows`

Inštalácia na Linux a Mac

Pre inštaláciu na Ubuntu 14.04 alebo 14.10 si najskôr nainštalujte nasledovné balíčky

```
sudo apt-get install build-essential xorg-dev libudev-dev
libts-dev libgl1-mesa-dev libglu1-mesa-dev libasound2-dev
libpulse-dev libopenal-dev libogg-dev libvorbis-dev
libaudiofile-dev libpng12-dev libfreetype6-dev libusb-dev
libdbus-1-dev zlib1g-dev libdirectfb-dev

sudo apt-get install doxygen

sudo apt-get install freeglut3 freeglut3-dev

sudo apt-get install libode-dev libode-sp-dev libode1
libode1sp

sudo apt-get install libdevil1c2 libdevil-dev

sudo apt-get install libboost-all-dev

sudo apt-get install php5-gd
```

Následne si nainštalujte Ruby. Testovaná je verzia 1.9 a 2.0 napríklad pomocou RVM
`https://www.digitalocean.com/community/tutorials/how-to-install-ruby-on-rails-on-ubuntu-14-04-using-rvm`

Stiahnite si najnovšie zdrojové kódy podľa inštrukcií na `http://simspark.sourceforge.net/wiki/index.php/Installation_on_Linux#Ubuntu`

Podľa inštrukcii na stránke si ich aj skompilujte a nainštalujte. Ak natrafíte na problém s chýbajúcou referenciou na libruby, musíte si ručne vytvoriť symlinku do /usr/lib/ z vášho aktuálneho libruby.so.x.x podľa verzie ktoré máte nainštalovanú a názvu súboru ktoré vám compiler hlási ako chýbajúcu.

Inštaláciu na Mac náš tím neskúšal, no je opísaná na wiki jedným z minuloročných tímov spolu s ďalšími návodmi na inštaláciu: `http://labss2.fiit.stuba.sk/TeamProject/2014/team08is-si/wiki/index.php/N%C3%A1vody_a_in%C5%A1tal%C3%A1cie`

Spúšťanie projektu

Táto kapitola obsahuje v jednotlivých krokoch spúšťanie a nastavenie projektu. Pred spustením projektu musíte mať nainštalované všetky potrebné súčasti popísané v tomto dokumente a importovaný projekt vo vývojovom prostredí.

Spúšťanie agenta Jim

Pred spustením samotného agenta, je potrebné spustiť najskôr server a monitor. Otvorte priečinok bin, v zložke kde máte nainštalovaný rcsserver (C:\Program Files (x86)\rcssserver3d 0.6.7\bin) a spustite najskôr rcssserver3d.cmd a potom rcssmonitor3d. Potom by ste mali vidieť simulačné prostredie – ihrisko. Teraz spustite projekt Jim (spustite triedu Main.java z balíčka sk.fiit.jim.init). Na ihrisko by sa mal pridať hráč.

V simulačnom prostredí viete pohybovať kamerou pomocou W, S, A, D a simuláciu spustíte stlačením R. Stlačením Q simuláciu ukončíte.

Spúšťanie TestFrameworku

Spúšťanie TestFrameworku prebieha rovnako ako spúšťanie agenta Jima popísané vyššie a teda najskôr musíte spustiť server a potom monitor. TestFramework spustíte z triedy Init.java v balíčku sk.fiit.testframework.init.

Po spustení TestFrameworku by sa vám malo zapnúť rozhranie, v ktorom môžete nastavovať polohu lopty a agentov, pridávať agentov, simulovať streľbu, sledovať informácie a rôzne ďalšie veci. Agentov môžete pridať z tabu Manage agents, vybratím tímu a stlačením Add. Následne by sa mal hráč pridať na ihrisko.

V simulačnom prostredí viete pohybovať kamerou pomocou W, S, A, D a simuláciu spustíte stlačením R. Stlačením Q simuláciu ukončíte.

Testovanie funkčnosti projektu

V tejto kapitole sa budeme venovať otestovaniu niektorých konkrétnych situácií, či už ide o funkcionalitu, ktorá bola pridaná našim tímom, či o otestovanie rôznych taktík a správania agenta. Okrem toho tu budú opísané aj problémy spojené s TestFrameworkom, ktoré zatiaľ neboli vyriešené a opravené.

Testovanie rôznych taktík

Rôzne taktiky môžete nastavovať v textovom súbore, ktorý sa nachádza v projekte Jim, s názvom settings.properties. V tomto súbore je potrebné prestaviť DEBUG_TACIC_ENABLE = false na true. Taktiky môžete meniť zmenou hodnoty DEBUG_TACTIC_NAME = DefaultTactic, na požadovanú taktiku. Taktiky sa nachádzajú v balíčkoch v projekte Jim:

- sk.fiit.jim.decision.tactic – predvolená taktika (DefaultTactic)
- sk.fiit.jim.decision.tactic.attack – taktiky pre útok
- sk.fiit.jim.decision.tactic.defense – taktiky pre obranu
- sk.fiit.jim.decision.tactic.oldgoalie – taktika pre brankára, no táto taktika nefunguje
- sk.fiit.jim.decision.tactic.tournament – taktiky pre turnaj (rôzne disciplíny)

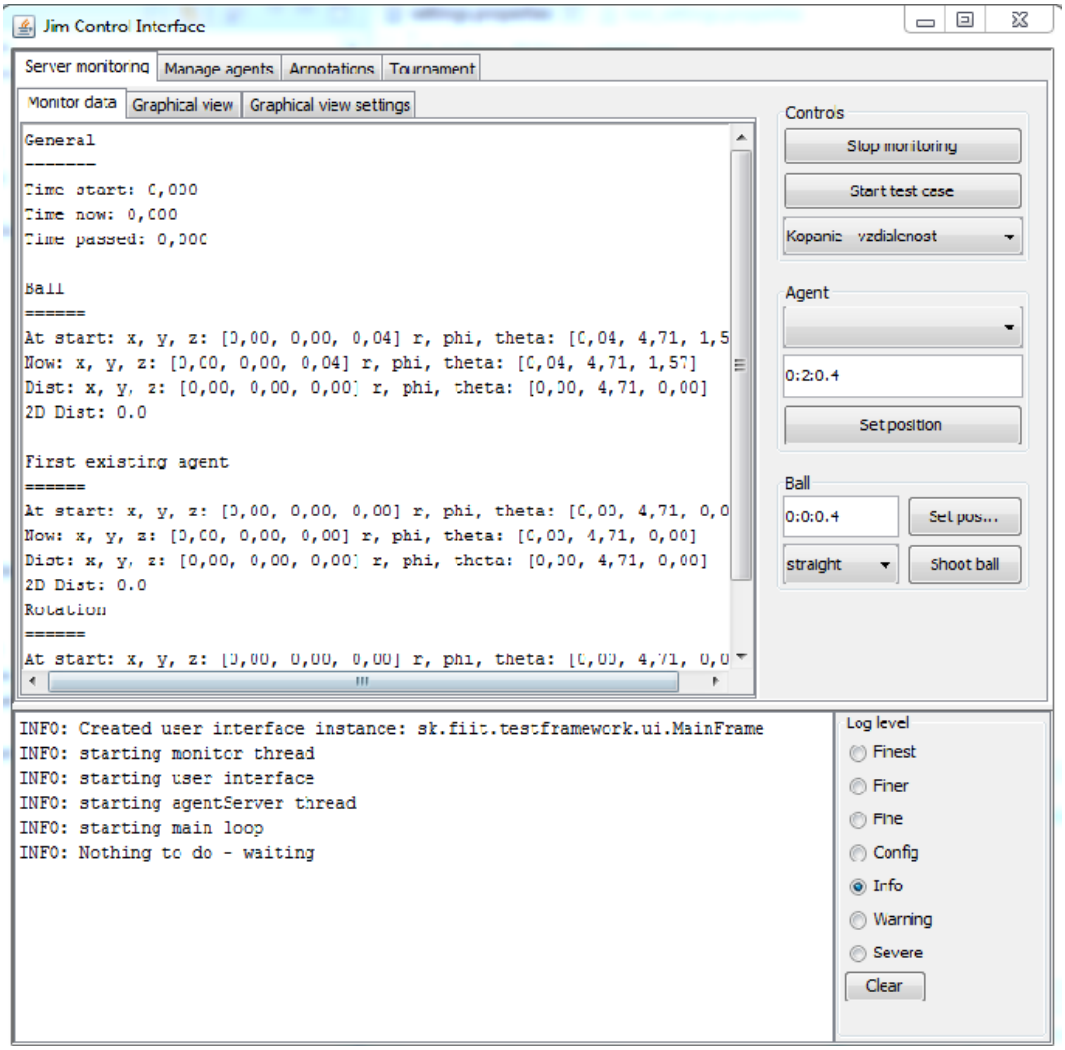
Testovanie TestFrameworku

Pri spúšťaní TestFrameworku sa držte pokynov uvedenými vyššie v príručke. TestFramework slúži pre rôzne merania, simulovanie striel, testovanie, anotácie, pridávanie hráčov, nastavenie pozície lopty, nastavenie pozície hráčov zisťovanie informácií a celkové nastavovanie simulačného prostredia. V nasledujúcich bodoch bude opísané grafické rozhranie TestFrameworku a spúšťanie jednotlivých funkcií TestFrameworku.

Po spustení TestFrameworku vidíme v grafickom rozhraní v karte „Server monitoring“ nastavovanie testovacích prípadov, prepínanie medzi jednotlivými agentmi so zmenou ich pozície (agentov je nutné najskôr pridať), nastavenie polohy lopty a simuláciu streľby s nastavením strany a vzdialenosti, z ktorej bude lopta vystrelená. Okrem toho si v pravom dolnom rohu môžete nastaviť 7 rôznych úrovni logovania, pričom logy vidíte v dolnej časti obrazovky. Najväčšiu plochu zaberajú dáta monitoru, v ktorom sú rôzne informácie o polohách agentov, lopty a ich natočení. V karte „Server monitoring“

je okrem iného možné prepnúť sa do karty pre 2D pohľad z hora ihriska a jeho nastavenia.

V karte „Manage agents“ je možné do jednotlivých tímov pridávať hráčov a ďalšia karta slúži pre anotovanie. V poslednej karte Tournament, ktorú do TestFrameworku pridal náš tím, je možné sa prepínať medzi jednotlivými disciplínami FIIT turnaja a merať jednotlivé disciplíny. V momentálnom stave sa medzi týmito disciplínami nachádza merania rýchlosti, stability chôdze, kopu lopty na presnosť a kopu lopty na určený bod. Mali by však pribudnúť aj ostatné merania, aby boli pokryté všetky disciplíny z turnaja. Použitie týchto meraní je vysvetlené v samotnom grafickom rozhraní. Treba však myslieť na to, že funkcie v týchto kartách nezabezpečujú zmenu taktiky potrebnej na vykonanie disciplíny, slúžia iba na meranie a agenta je teda potrebné nastaviť na prislúchajúcu taktiku podľa disciplíny, čo je v tejto príručke opísané vyššie.



Obr. 17: Ukážka grafického rozhrania TestFrameworku

Nedostatky v TestFrameworku

Náš tím sa zaoberal aj odstraňovaním niektorých nedostatkov TestFrameworku. Bohužiaľ sa nám všetky chyby nepodarilo odstrániť.

Opravené bolo napríklad pridávanie hráčov, kde sa však vyskytla nová chyba – prvý pridaný hráč je zaseknutý. Ďalší pridaný hráči sa však už správajú korektne.

Jednou z ďalších nedostatkov je nefunkčné prepínanie testovacích prí-

padov. Tento nedostatok sa nepodarilo odstrániť a nie je jasné či niekedy fungoval.

Dalším nedostatkom sú nefunkčné anotácie. Anotácie kedysi fungovali správne, no v momentálnom stave nefungujú. Náš tím tento nedostatok nestihol odstrániť, no pracovali sme na analýze nefunkčnosti týchto anotácií, ktorú môžete nájsť v inžinierskom diele nášho tímu.

Príloha B: Štandard pre kód

Posledná úprava	Juraj Šimek
Platné od	19. november 2014
Poznámky	Pridané príklady logovacích typov

Tento dokument opisuje to, aké konvencie je nutné dodržiavať pri dopĺňaní kódu projektu agenta hrajúceho robotický futbal a s ním súvisiacich projektov. Konvencia je určená pre všetkých členov tímu, ktorí zasahujú do zdrojového kódu projektu. Konvencie písania kódu uvedené v tomto dokumente vychádzajú najmä z dokumentu *Java Code Conventions* [15].

Základné konvencia písania kódu v jazyku Java

Názvy

Názvy *tried, rozhraní a výčtových typov* začínajte veľkým písmenom, pričom je dôležité, aby ste im vyberali výstižné, ale pokiaľ možno stručné názvy. Pri viacslovných pomenovaniach používajte štýl camel-case. V prípade rozhraní uveďte v rámci názvu predponu `i`.

```
class KickHighSkill { }
interface IHighSKill { }
```

Názvy *premenných a metód* začínajte malým písmenom. Pri viacslovných pomenovaniach znova použite camel-case. Opäť je dôležité dbať na to, aby bol názov zrozumiteľný, ale pritom stručný.

```
void getPrescribedSituations() { ... }
int currentHighSkill = 40;
```

Názvy *konštánt* uvádzajte veľkým písmom. Ich jednotlivé slová oddeľujte podtržníkom (_).

```
public final static int THIS_IS_CONSTANT = 30;
```

Názvy *balíkov* píšte malými písmenami.

```
sk.fiit.jim.agent
```

Formátovanie zdrojového kódu

Každý kód vo vnútri akéhokoľvek bloku odsaďte *tabulátorom*:

```
{
    int x = 4;
}
```

Otváraciu zátvorku označujúcu blok kódu píšte hneď *za* príkazom, ku ktorému prislúcha:

```
if (x == 3) {
    ...
}
```

Telo každého cyklu a podmieneného príkazu uvádzajte v zátvorkách `{` a `}` *aj v prípade, že predstavuje len jeden príkaz!* Toto je nesmierne dôležité kvôli predchádzaniu chybám pri rozširovaní kódu v budúcnosti.

```
if (x == 3) {
    return false;
} else {
    return true;
}
```

Každý príkaz píšte do osobitného riadku.

```
Xcoord++;
Ycoord++;
kick();
```

Každú *premennú* (aj lokálnu) vhodne definujte na novom riadku.

```
int myVariableA = 4;
int anotherVariable;
int thisVariableIsAlsoNeeded;
```

Dĺžka riadku by nemala presiahnuť 80 znakov. Ak je príkaz príliš dlhý rozdeľte ho, pričom rozdelenú časť vhodne odsaďte tak, aby sa operátory nachádzali na novom riadku. Bližšie informácie nájdete v kapitole *Wrapping Lines* dokumentu [15]. V tomto prípade treba pamätať hlavne na to, aby bolo pri pohľade na kód jasné, čo patrí spolu, a čo je už iný príkaz.

Medzi definíciami metód v triede uveďte jeden prázdny riadok.

```
public String toString() {
    ...
}

/**
 * Comment
 */
public int getPlaygroundSize() {
    ...
}
```

Serializácia

Ak trieda požaduje implementovanie rozhrania *Serializable* a v rámci toho definíciu *serialVersionUID*, neuvádzajte *@SupressWarning("serialization")*, ale do vnútra triedy uveďte definíciu, napríklad *static final long serialVersionUID = 42L;* Číslo *serialVersionUID* môže byť samozrejme ľubovoľné. Viac o jeho význame a o serializácii objektov je v [17] a [3].

Písanie komentárov v jazyku Java

Jednoriadkové komentáre píšte výlučne pomocou `//` a neuzatvárajte ich medzi `/*` a `*/`

```
// single line comment
```

Ak má komentár viac riadkov, napíšte ho medzi `/*` a `*/`, pričom je vhodné aby ste na každom riadku uviedli znak `*`. Medzi znakom `*` a prvým písmenom komentára uveďte jednu medzeru.

```
/*
* This is
* Multiline
* Comment
*/
```

Dokumentačné komentáre

Pred každou triedou a metódou, ktorá je verejná sa uvádzajte dokumentačný komentár medzi `/**` a `*/` generujúci JavaDoc. Viac informácií o písaní JavaDocu je v [14]. Nepoužívajte žiadne vymyslené tagy, ktoré nedefinuje JavaDoc. JavaDoc pripúšťa tieto tagy:

```
@author
@version
@param
@return
@exception
@see
@since
@serial
@deprecated
```

Význam jednotlivých tagov je uvedený v [14].

Každý *autor* uvedie do dokumentačného komentára kódu, ktorému prislúcha svoje meno v nasledovnom tvare:

```
@author <meno_autora> (<nazov_timu> - <rok>)
```

Pričom:
`<meno_autora>` - autorovo celé meno
`<nazov_timu>` - meno tímu, ak v tíme nepracuje, uvedie len rok
`<rok>` - rok v ktorom je kód upravovaný

Príklad:

```
@author Juraj Simek (Infinity - 2014)
```

Pred každou public metódou do komentára uveďte pomocou `@param` význam jednotlivých parametrov, pomocou `@exception` opíšte kedy vyhadzuje akú výnimku a pomocou `@return` napíšte, čo metóda vracia:

```
/**
 * Returns position of the goal according to player's side.
 *
 * @param side player's team side
 * @return absolute position of the goal
```

```
 */
private Vector3D getGoalAbsolutePosition(Side side) {
    ...
}
```

Dokumentačné komentáre musia byť stručné, ale predovšetkým zrozumiteľné, aby sa v nich vyznali aj tí, čo daný kód neprogramovali.

Výnimky

Ak zachytávate výnimku, je potrebné ju zalogovať. Nikdy možné výnimky neignorujte pomocou `catch(Exception e) {}` ! Vhodné je buď výnimku zachytiť a zalogovať a potom ošetriť výnimočný stav, alebo ju znova vyhodiť a ošetriť prípadný výnimočný stav v inej časti kódu, ak to v tejto časti nie je možné. Nasledovný kód by sa preto nemal vyskytovať často:

```
catch(NoSuchMethodException e) {
    LOG.error("Situation␣description", e);
    throw e;
}
```

Niekedy je však vhodné zalogovať, že výnimka niekde nastala a znova ju vyhodiť. Preto je vyššie uvedený kód povolený v situáciách, keď je to naozaj opodstatnené. Ak uznáte za vhodné zalogovať výnimku a opäť ju vyhodiť bez jej ošetrenia, uveďte komentár vysvetľujúci vaše rozhodnutie.

Logovanie

V metóde, kde sa loguje získate *loger* pomocou `JLog.getLogger()`. Ideálne je, aby ste vytvorili členskú premennú pre celú triedu, ktorá bude loger používať:

```
Private Logger LOG = JLog.getLogger();
```

Logujte potom pomocou štandardného API triedy Logger [7]. Pre úplnosť informácií v krátkosti uvádzame to, ako sa loguje pomocou tohto rozhrania. Štandardne loger vytvoríte ako privátnu statickú konštantu v triede, ktorá ho používa. Vďaka tomu budete môcť zachytávať, v ktorej triede vznikla udalosť logovania. Logeru dáte identifikátor, ktorý najčastejšie predstavuje meno danej triedy:

```
private final static Logger LOG =
   Logger.getLogger(LoggingExamples.class.getName());
```

Na logovanie v Jimovi však použite predkonfigurovaný globálny loger pomocou `JLog.getLogger()` tak, ako to bolo opísané vyššie.

Keď ste získali loger, v akejkoľvek metóde danej triedy potom možno tento loger použiť na logovanie rôznych udalostí. Java API predpisuje nasledovné najčastejšie používané úrovne logovania *Level.FINEST, Level.FINER, Level.FINE, Level.INFO, Level.WARNING a Level.SEVERE*, kde *FINEST – FINE* sú na najnižšej úrovne logovania a logujú sa nimi najmenej podstatné informácie. Nimi sa logujú najmä informácie o úspešnom vykonaní určitého bloku kódu. Level *FINEST* používajte na logovanie pomocných ladiacich výpisov v čase vývoja novej funkcionality. Pomocou *INFO* logujte rôzne informácie, ktoré chcete vložiť do logu, majú informačný charakter a neviete im priradiť vhodnejší level. *WARNING* slúži na logovanie varovaní (napríklad informácie o tom, že nastala výnimka a bola ošetrená) a *SEVERE* na logovanie závažných chýb, z ktorých sa program nevie zotaviť. Logovať môžete pomocou metódy `log()`:

```
LOG.log(Level.INFO, "Information");
```

Pre základné typy logovania definované Java API logerom môžete použiť aj logovanie pomocou predpísaných metód:

```
LOG.finest("Finest");
LOG.finer("Finer");
LOG.fine("Fine");
LOG.info("Information");
LOG.warning("Warning");
LOG.severe("Severe");
```

Loger JLog používaný v projekte Jim navyše, oproti úrovniam v triede Level, definuje tieto typy logov:

```
LogType.INIT
LogType.AGENT_MODEL
LogType.WORLD_MODEL
LogType.INCOMING_MESSAGE
LogType.OUTCOMING_MESSAGE
```

```
LogType.LOW_SKILL
LogType.HIGH_SKILL
LogType.GUI
LogType.TACTIC
```

Tieto typy umožňujú logovať informácie pre špeciálne udalosti. Pomocou týchto typov logujte len informácie do úrovne *Level.INFO* štandardného Java API logera, pričom ak môžete použiť jeden z týchto rozšírených typov, použite ho. Ak chcete logovať nejakú udalosť, ktorú LogType nepredpisuje, buď ju do LogType pridajte, alebo použite Level.INFO. Nový typ logu môžete pridať do triedy LogType pridaním nasledovného riadku v zozname definícií konštánt:

```
public static LogType <NEW_TYPE_NAME> =
  new LogType("<NEW_TYPE_NAME>",
    BASE_LEVEL_NUMBER + <NEXT_NUMBER>);
```

<NEW_TYPE_NAME> predstavuje názov nového logovacieho typu a píšte ho veľkými písmenami, nakoľko ide o konštantu. *<NEXT_NUMBER>* udáva poradové číslo nového levelu a zapíšte ho ako celé číslo o jednotku väčšie od najväčšieho čísla, ktoré používa nejaký typ logu definovaný v triede *LogType*.

Význam jednotlivých typov dodefinovaných v triede LogType je nasledovný:

- *INIT*: Inicializačné správy napríklad vo funkcii main(), nadviazanie spojenia so serverom, načítanie XML súborov a všetky akcie, ktoré konfigurujú agenta skôr, ako začne odosielať nejaké dáta na sever. Príklad: sk.fiit.jim.init.SkillsFromXmlLoader metóda laod()

  ```
  long xmlEnd = System.currentTimeMillis();
  double seconds = (xmlEnd - xmlStart) / 1000.0;
  LOG.log(LogType.INIT, "Moves␣loaded␣from␣files␣in␣"
              + seconds + "␣seconds");
  ```

- *AGENT_MODEL*: Logy týkajúce sa modelu agenta. Najčastejšie v balíku jim.agent. Tento level možno použiť všade, kde sa spracovávajú informácie o reprezentácii agenta, ako jeho poloha. Príklad: sk.fiit.jim.agent.AgentInfo metóda isWayFree()

  ```
  LOG.log(LogType.AGENT_MODEL,
  ```

```
        "opponentPosition = ["
        + opponentPlayerPosition.getX()
        + ","
        + opponentPlayerPosition.getY() + "]");
```

- *WORLD_MODEL*: Logy týkajúce sa modelu sveta. Používa sa napríklad vo funkciách, ktoré prepočítavajú polohu objektov na ihrisku. Príklad: sk.fiit.jim.agent.models.WorldModel metóda calculateBallPosition()

```
LOG.log(LogType.WORLD_MODEL,
        "Player IDs are not implemented yet!!!");
```

- *INCOMING_MESSAGE*: Logovanie správy prichádzajúcej zo servera alebo logovanie informácií z funkcií, ktoré takúto správu spracúvajú. Príkad: sk.fiit.jim.agent.communication.Communication metóda receive()

```
String incoming = new String(buffer, 0,
                             messageLength);
LOG.log(LogType.INCOMING_MESSAGE, incoming);
```

- *OUTCOMING_MESSAGE*: Logovanie správy odosielanej na server alebo logovanie informácií z funkcií, ktoré takúto správu spracúvajú a odosielajú. Príklad: sk.fiit.jim.agent.communication.Communication metóda transmit()

```
private void transmit(String what) throws
IOException{
   LOG.log(LogType.OUTCOMING_MESSAGE, what);
   ...
```

- *LOW_SKILL*: Informačné logy týkajúce sa low skillov. Príklad: sk.fiit.jim.agent.moves.LowSkill metóda step()

```
LOG.log(LogType.LOW_SKILL,
       "Currently active phase: "
       + activePhase.name);
```

- *HIGH_SKILL*: Informačné logy týkajúce sa high skillov. Príklad: sk.fiit.jim.agent.highskill.GoTo metóda pickLowSkill()

```
else if (this.tacticalInfo.isOnPosition()) {
   LOG.log(LogType.HIGH_SKILL, "close");
   return null;
}
```

- *GUI*: Informačné logy týkajúce sa grafického rozhrania robota. Príklad: sk.fiit.jim.gui.ReplanWindow metóda reloadScripts()

```
LOG.log(LogType.GUI,
        "Script reload request from GUI");
```

- *TACTIC*: Informačné logy týkajúce sa vrstvy taktík. Príklad: sk.fiit.jim.decision.tactic.MatchStarterTactic metóda runBeam()

```
this.beamExec
   .BeamAgent(Vector3D.cartesian(-4, 7, 0.1));
LOG.log(LogType.TACTIC, "BEAM - LEFT 7");
```

Ak logujete iba pomocné výpisy (najmä pre seba), ktoré nie sú potrebné pre finálne logovanie, používajte *Level.FINEST*, ako už bolo spomenuté vyššie. Loger môžete potom v main() funkcii nastaviť tak, aby sa logovali len vyššie úrovne. Ak potrebujete vypisovať počas vývoja nejaké informácie pomocou *System.out.println()*, môžete tak urobiť, ale pri finalizovaní a nasadení vášho kódu nezabudnite tieto časti skutočne zmazať, nie len zakomentovať.

Loger môžete konfigurovať vo funkcii main() projektu. Pred jeho prvým spustením je potrebné, aby ste zavolali metódu *JLog.setup()*, ktorá prijíma 2 boolean hodnoty. Prvá určuje, či sa bude logovať na konzolu, druhá, či sa bude logovať do súboru. Súbor sa nachádza v koreňovom adresári projektu a má názov *jim_logs.html*. Logy sú uložené v prehľadnej HTML tabuľke.

Bez ďalších nastavení sa loger obalený triedou JLog správa ako štandardný Logger z Java API. Ak chcete logovať aj dodatočné typy logov menované vyššie, musíte v main() tieto typy pridať pomocou metódy addLogType():

```
JLog.addLogType(LogType.INCOMING_MESSAGE);
```

Ak chcete pridať všetky typy, ktoré definuje LogType, zavoláte metódu addAllLogTypes() triedy JLog. Potom možno logovať tieto nové typy pomocou metódy log triedy Logger:

```
LOG.log(LogType.INCOMING_MESSAGE, "message");
```

Logy by mali byť zrozumiteľné. Každý zápis do logu má mať opodstatnenie a má byť zmysluplný a formulovaný tak, aby sa dalo pochopiť, čo sa udialo a prečo. Logovať treba len pomocou tu opísaného API. Na logovanie sa nepoužívajú vlastné metódy a hlavne nie System.out.println(). Nedefinujte ďalšie logery ale používajte globálny loger, ktorý je nakonfigurovaný pomocou JLog. Vhodné je obmedziť sa len na používanie metód *log(), finest(), finer(), fine(), info(), warning(), severe().*

Ďalšie konvencie

Metódy, ktoré vracajú kolekcie alebo polia nesmú vracať null. Namiesto null je vráťte prázdnu kolekciu alebo prázdne pole.

Pri pomenovávaní premenných a písaní komentárov sa vyhnite skratkám. Všeobecne známe skratky (napr. HTML, TCP/IP) sú v poriadku.

Ak potrebujete použiť nejakú funkcionalitu, najprv sa dôkladne pozrite do dokumentácie, či na to neexistuje trieda alebo nejaká metóda a až vtedy ak neexistuje vytvorte odpovedajúci kód. Nikdy neprogramujte triedy ako hešovacie tabuľky, ktoré má Java v knižniciach.

Používajte výhody generického programovania pri práci s kolekciami. Vždy uvádzajte do zátvoriek < a >, aké typy sa v danej kolekcii budú používať.

Metódy, ktoré vracajú kolekcie alebo polia nesmú vracať *null*. Namiesto *null* je potrebné vrátiť prázdnu kolekciu alebo prázdne pole.

Vyhnite sa skratkám. Všeobecne známe skratky sú v poriadku.

Ak chcete niečo vytvoriť, najprv sa pozrite do dokumentácie, či na to neexistuje trieda a až vtedy ak neexistuje ju vytvorte. Neprogramujte veci ako hešovacie tabuľky, ktoré má Java v knižniciach.

Používajte výhody generického programovania pri práci s kolekciami. Vždy uvádzajte do zátvoriek < a >, aké typy sa v danej kolekcii budú používať.

References

[1] Tomáš Boleček. *Hráč simulovaného robotického futbalu*. Slovenská technická univerzita v Bratislave, 2014.

[2] Mike Depinet, Patrick MacAlpine, and Peter Stone. "Keyframe Sampling, Optimization, and Behavior Integration: Towards Long-Distance Kicking in the RoboCup 3D Simulation League". In: *Robocup Proceedings*. 2014.

[3] Journal Dev. *Java Serialization Example Tutorial, Serializable, serialVersionUID*. `http://www.journaldev.com/2452/java-serialization-example-tutorial-serializable-serialversionuid`. [Online; accessed 19-October-2014]. 2013.

[4] Jake Fountain et al. "Motivated reinforcement learning for improved head actuation of humanoid robots". In: *RoboCup 2013: Robot Soccer World Cup XVII* (2013).

[5] Jaroslav Grega. *Robocup 3D - Nižšie schopnosti hráča*. Slovenská technická univerzita v Bratislave, 2014.

[6] Ján Hudec. *Motorika hráča simulovaného futbalu*. Slovenská technická univerzita v Bratislave, 2012.

[7] Jakob Jenkov. *Java Logging*. `http://tutorials.jenkov.com/java-logging/index.html`. [Online; accessed 19-October-2014].

[8] Adailton J.C. Junior et al. "Bahia3D Team Description Paper". In: *Robocup Proceedings*. 2011.

[9] N. Kofinas, E. Orfanoudakis, and M.G. Lagoudakis. "Complete analytical inverse kinematics for NAO". In: *Autonomous Robot Systems (Robotica), 2013 13th International Conference on*. 2013, pp. 1–6. DOI: `10.1109/Robotica.2013.6623524`.

[10] Nikolaos Kofinas, Emmanouil Orfanoudakis, and MichailG. Lagoudakis. "Complete Analytical Forward and Inverse Kinematics for the NAO Humanoid Robot". English. In: *Journal of Intelligent & Robotic Systems* (2014), pp. 1–14. ISSN: 0921-0296. DOI: `10.1007/s10846-013-0015-4`. URL: `http://dx.doi.org/10.1007/s10846-013-0015-4`.

[11] Martn Košický. *Robocup 3D - Nižšie schopnosti hráča*. Slovenská technická univerzita v Bratislave, 2014.

[12] P. MacAlpine et al. “Design and Optimization of an Omnidirectional Humanoid Walk: A Winning Approach at the RoboCup 2011 3D Simulation Competition”. In: *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence*. 2012.

[13] Pavol Mešťaník. *Robocup 3D simulovaná liga - Nižšie schopnosti hráča*. Slovenská technická univerzita v Bratislave, 2014.

[14] Oracle. *How to Write Doc Comments for the Javadoc Tool*. `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`. [Online; accessed 19-October-2014].

[15] Oracle. *Java Code Conventions*. `http://www.oracle.com/technetwork/java/codeconventions-150003.pdf`. [Online; accessed 19-October-2014]. 1997.

[16] Peter Paššák. *Optimalizovanie pohybov robota pomocou evolučných algoritmov v 3D simulovanom robotickom futbale*. Slovenská technická univerzita v Bratislave, 2014.

[17] Java Practices. *Implementing Serializable*. `http://www.javapractices.com/topic/TopicAction.do?Id=45`. [Online; accessed 19-October-2014].

[18] Alan Santos et al. “Bahia RT 2014 - Team Description Paper”. In: *Robocup Proceedings*. 2014.

[19] Adriano Veiga. “Bahia3D - A Team of 3D Simulation for Robocup”. In: *Robocup Proceedings*. 2009.