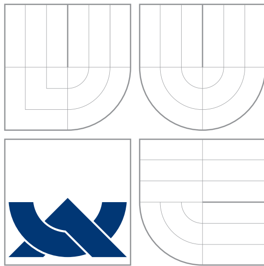
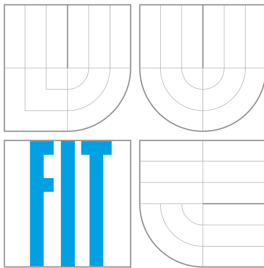


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ADVENTURE HRA S INTELIGENTNÍMI SPOLUPRACUJÍCÍMI POSTAVAMI

ADVENTURE GAME WITH INTELLIGENT COOPERATING ACTORS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE LUKÁŠ VACEK
AUTHOR

VEDOUCÍ PRÁCE Ing. FRANTIŠEK ZBOŘIL, Ph.D.
SUPERVISOR

BRNO 2016

Zadání diplomové práce/18328/2015/xvacek02

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav inteligentních systémů Akademický rok 2015/2016

Zadání diplomové práce

Řešitel: **Vacek Lukáš, Bc.**

Obor: Inteligentní systémy

Téma: **Adventure hra s inteligentními spolupracujícími postavami**
Adventure Game with Intelligent Cooperating Actors

Kategorie: Umělá inteligence

Pokyny:

1. Nastudujte teorie spolupracujících inteligentních systémů, norem a závazků v multiagntních systémech.
2. Vytvořte osnovu scénářů pro adventure hru, kde kromě hráče vystupuje několik postav s několika rolemi, tj. definovanými povinnostmi, oprávněními používat prostředky a schopnostmi používat interakční protokoly pro sjednávání závazků a řešení konfliktů.
3. Vytvořte části příběhu tak, aby schopnost interakce mezi hráčem a postavami v jednotlivých rolích, a to zejména ovlivňováním jejich záměrů ve vhodných stavech hry, umožnila hráči dosažení cílů hry.
4. Jednotlivé postavy navrhněte jako BDI agenty a celý systém navrhněte pomocí nástroje Prometheus.
5. Daný scénář implementujte jako adventure hru (textovou, případně grafickou).

Literatura:

- Wooldridge, M.: Introduction to multiagent systems, John Wiley & Sons, 2009

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci dřívějších projektů (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Zbořil František, doc. Ing., Ph.D.**, UITS FIT VUT

Datum zadání: 1. listopadu 2015

Datum odevzdání: 25. května 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav inteligentních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Petr Hanáček
vedoucí ústavu

Abstrakt
Cílem této diplomové práce je navrhnout a implementovat framework, který lze využít pro vývoj agentních systémů. Framework je nadstavbou nad knihovnou JADE a je realizován v jazyce Java. Framework je využit pro implementaci adventure hry. Ve hře bude několik postav (agentů), které budou mít určité role, budou spolupracovat a snažit se dosáhnout svých cílů.

Abstract
The goal of this master's thesis is to design and implement framework that can be used for development of agent systems. Framework is implemented in Java and encapsulates JADE library. Framework is used for implementation of adventure game. There are several characters (agents) with specific roles who cooperate and try to achieve their goals.

Klíčová slova
Agentní systém, BDI, JADE, adventura, důvěra, spolupráce.

Keywords
Agent system, BDI, JADE, adventure game, trust, cooperation.

Citace
VACEK, Lukáš. *Adventure hra s inteligentními spolupracujícími postavami.* Brno, 2016. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Zbořil František.

Adventure hra s inteligentními spolupracujícími postavami

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Františka Zbořila, Ph.D.

. . . .
Lukáš Vacek
23. května 2016

Poděkování

Chtěl bych poděkovat vedoucímu mé diplomové práce Ing. Františku Zbořilovi, Ph.D, za poskytnutou pomoc při konzultacích.

© Lukáš Vacek, 2016.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod **4**

2 Agentní systémy **5**
2.1 Historie umělé inteligence 5
2.2 Inteligentní agenti 7
2.3 Anticipující agenti 9
2.4 Agenti jako systémy řízené záměrem 10
2.5 Mobilní agenti 10
2.6 Komunikace a spolupráce agentů 10
2.6.1 Kooperativní distribuované řešení problému 11
2.6.2 Agenti s vlastním zájmem 11
2.6.3 Částečné komplexní plánování 12
2.6.4 Normy a zákony 12
2.6.5 Nekonzistence mentálních stavů agentů 12

3 FIPA **14**
3.1 Abstraktní architektura agentního systému 14
3.2 Komunikační akty agentů 15
3.2.1 Inform 15
3.2.2 Request 16
3.2.3 Subscribe 16
3.2.4 Agree 16
3.2.5 Propose 16
3.2.6 Cancel 16
3.2.7 Confirm 16
3.2.8 Failure 16

4 Návrh frameworku **18**
4.1 JADE 18
4.2 Specifikace požadavků pro využívání frameworku 18
4.2.1 Požadavky na rozhraní frameworku 18
4.2.2 Požadavky na aplikaci 19
4.3 Komunikace 19
4.3.1 Přijímání zpráv 19
4.3.2 Odesílání zpráv 19
4.3.3 Hlasování 19
4.3.4 Historie komunikace 20
4.4 Správa BDI stavů 20

4.5 Správa skupin agentů . . . 20
4.5.1 Komunikace agentů ve skupině . . . 21
4.5.2 Vytvoření a zrušení skupiny . . . 21
4.5.3 Přidávání nových agentů do skupiny . . . 22
4.5.4 Opuštění skupiny . . . 23
4.5.5 Vyloučení člena ze skupiny . . . 23
4.6 Služby . . . 24

5 Implementace frameworku **26**
5.1 Základní třídy frameworku . . . 26
5.1.1 Třída IntelligentAgent . . . 26
5.1.2 Třída Belief . . . 27
5.1.3 Třída SpecialMsg . . . 27
5.1.4 Třída AgentGroup . . . 27
5.1.5 Služby . . . 28
5.1.6 Ostatní třídy . . . 28
5.2 Rozhraní frameworku . . . 29

6 Návrh adventure hry **30**
6.1 Představení reality show . . . 30
6.2 Pravidla hry . . . 30
6.3 Dostupné akce . . . 31
6.4 Agent arbitr . . . 32
6.4.1 Hlavní cyklus arbitra . . . 32
6.5 Agent hráč . . . 35
6.5.1 Důvěra k hráčům . . . 35
6.5.2 Historie hlasování . . . 35
6.5.3 Vlastník imunity a indicie . . . 35
6.6 Požadavek na vykonání akce . . . 36
6.7 Požadavek na reakci . . . 40
6.8 Vnímání prostředí . . . 42
6.9 Zpracování informací . . . 43
6.10 Proces hlasování . . . 43
6.11 Návrh grafického uživatelského rozhraní . . . 44

7 Implementace adventure hry **45**
7.1 Implementace arbitra . . . 45
7.2 Implementace počítačových hráčů . . . 46
7.3 Umělá inteligence hráčů . . . 46
7.4 Implementace grafického uživatelského rozhraní . . . 47
7.5 Testování adventure hry . . . 49

8 Závěr **50**
8.1 Možné rozšíření adventure hry . . . 50

Literatura **51**

Přílohy **52**
Seznam příloh . . . 53

A Obsah CD **54**

B Rozhraní frameworku **55**

C Příklady použití frameworku **57**

C.1 Správa představ 57
C.2 Vlastní filtrování představ 58
C.3 Rozhraní IAgentGroup 60
C.4 Třída Group 60
C.5 Správa skupin 60

D Ukázky ze hry **63**

Kapitola 1

Úvod

Cílem této diplomové práce je navrhnout a implementovat framework, který lze využít pro vývoj agentních systémů. Framework bude nadstavbou nad knihovnou JADE, proto jsem pro jeho realizaci zvolil jazyk Java. Framework, jehož návrh je popsán v kapitole 4, bude využitý pro implementaci adventure hry. Ve hře bude několik postav (agentů), kteří budou mít určité role, budou spolupracovat a snažit se dosáhnout svých cílů. Postavy mezi sebou budou spolupracovat pomocí komunikačních aktů, norem a závazků. Spolupracovat a rozhodovat se mohou agenti jak sami, tak ve skupinách, které mohou vytvářet.

Kapitola 2 popisuje agentní systémy, architekturu a chování jednotlivých inteligentních agentů a jejich komunikaci a spolupráci. Jsou zde také uvedeny způsoby rozhodování nebo uvažování agentů. Problém, který má být řešen multiagentním systémem, bývá často dost složitý, proto je nutné udělat dekompozici problému na menší, snadněji řešitelné podproblémy. Tyto podproblémy se mohou dělit dále na menší a menší až dokud nebudou řešitelné agentem. Pro co nejvyšší efektivitu je potřeba, aby agenti spolupracovali, sdíleli své znalosti a řešení. K této kapitole jsem využil knihu An Introduction to MultiAgent Systems [12] a přednášek z předmětu AGS - Agentní a multiagentní systémy [13].

Multiagentní systémy jsou popsány standardem od organizace FIPA [1]. Architektura agentního systému, způsob a příklady komunikace, jakým se spolu agenti dorozumívají podle tohoto standardu, je shrnuta v kapitole 3. U jednotlivých komunikačních aktů je popsáno, v jakých příležitostech jsou vhodné, jaké jsou jejich podmínky pro použití a co obsahují za parametry. Vhodné používání komunikačních aktů přispívá k lepšímu řešení dané úlohy.

Další kapitola 4 obsahuje návrh frameworku, který by měl usnadňovat vývoj multiagentních aplikací. Jsou zde navrženy základní třídy frameworku s protokoly pro komunikaci mezi agenty, pro správu skupin, představ a služeb. Kapitola 5 ukazuje způsob implementace a popisuje strukturu a princip základních tříd pro funkčnost frameworku. V této kapitole je uvedeno i kompletní rozhraní frameworku. Díky navrženému frameworku se implementuje adventure hra, jejíž pravidla a herní činnosti jsou popsány v kapitole 6. Jsou zde také navrženi agenti, kteří ve hře budou vystupovat. Jejich jednotlivé role jsou popsány v podkapitolách. Návrh adventure hry obsahuje také kompletní protokol pro veškerou komunikaci mezi agenty ve hře. Podle návrhu se hra bude implementovat, což popisuje kapitola 7. Zde jsou zmíněny implementační detaily o jednotlivých agentech, použitých algoritmech a o implementaci grafického uživatelského rozhraní. V závěru jsou sepsány možné rozšíření této diplomové práce.

Kapitola 2

Agentní systémy

Myšlenka agentních systémů je velmi jednoduchá. Agentem je počítačový systém, který je schopný nezávisle provádět akce v prostředí v zájmu svého klienta. Místo toho, aby se agentovi muselo výslovně říkat, v jaký čas a jakou akci má provést, tak se agent může sám rozhodnout pro to, co je potřeba vykonat, aby uspokojil své stanovené cíle. Multiagentní systém se skládá z několika agentů, kteří mezi sebou spolupracují. Nejčastější typ spolupráce je pomocí zasílání zpráv. Ve většině případech budou mít agenti v multiagentním systému rozdílné cíle, a proto spolu musí komunikovat, vyjednávat a spolupracovat, aby celkový multiagentní systém plnil stanovenou funkci. V našich každodenních životech a denních rutinách si můžeme všimnout mnoha procesů, které připomínají multiagentní systém.

2.1 Historie umělé inteligence

Výraz „umělá inteligence“ poprvé použil John McCarhy, podle jehož definice umělá inteligence zahrnuje hraní her, expertní systémy, zpracování hlasu, neuronové sítě a robotiku. Umělá inteligence je ale velmi široký pojem, pro který existuje obrovské množství definic, kniha [15] uvádí tyto tři definice:

- Umělá inteligence je označení uměle vytvořeného jevu, který dostatečně přesvědčivě připomíná přirozený fenomén lidské inteligence.
- Umělá inteligence označuje tu oblast poznávání skutečnosti, která se zaobírá hledáním hranic a možnosti symbolické, znakové reprezentace poznatku a procesu jejich nabývání, udržování a využívání.
- Umělá inteligence se zabývá problematikou postupu zpracování poznatku - osvojováním a způsobem použití poznatku při řešení problému.

Tento odstavec, který čerpá z knihy [9], shrnuje historii umělé inteligence od raných počátků až do dnešní doby. Za počátek umělé inteligence je považováno rozmezí mezi lety 1943-1955, kdy Warren McCulloch a Walter Pitts ve svém díle čerpali ze znalostí fyziologie, funkcí mozkových neuronů, formální analýzy výrokové logiky a Turingova modelu výpočetního stroje. Spojením těchto oblastí navrhli model umělých neuronů, kde každý neuron je buď aktivní nebo neaktivní. Neuron se stává aktivním při stimulaci od okolních neuronů. Aktivace neuronu má tedy vliv na okolní neurony. Později dokázali, že každá výpočetní funkce může být vyřešena použitím sítě spojených neuronů. Důležitým faktem bylo, že logické spojky se dali snadno implementovat do této síťové struktury neuronů. Také prohlásili, že tato síť je schopna se sama učit. Postupem času byl model vylepšován.

V roce 1955 Newell a Simon vyvinuli *The Logic Theorist*, mnohými považovaný za první program s umělou inteligencí. Program reprezentoval danou úlohu jako stromový model, který se pokoušel vyřešit nalezením větve, jejíž výsledek je s největší pravděpodobností ten správný [2]. V roce 1959 byl představen *Geometry Theorem Prover*, který dokazoval i obtížné matematické věty. Ve stejném roce byl navržen programovací jazyk **Lisp**, který se stal nedílnou součástí při programování umělé inteligence. V roce 1963 se umělá inteligence používala pro vyřešení úloh jako jsou například IQ testy, matematické úlohy nebo rébusy. Populární úlohou byla hra **Strips** [14], při které na stole leželo několik krychlí a cílem bylo přeskládat tyto krychle podle specifikovaného cíle. Počítačový program měl k dispozici robotí ruku, pomocí které mohl v jeden okamžik přenášet pouze jednu krychli.

Začátkem 60.let byly představeny pojmy jako jsou **adalines sítě** a **perceptrony**. Největším problémem pro programy této doby se staly složitější úlohy. Programy fungovaly na jednodušších příkladech, pro rozsáhlejší a komplexnější úlohy ale selhávaly. Dalším problémem byl nedostatek nebo absence počátečních znalostí o zkoumaném tématu. Příkladem může být vývoj rusko-anglického slovníku, který prováděl pouze překlady mezi slovy a už nerespektoval gramatická pravidla jazyků. V roce 1965 publikoval Lotfi A. Zadeh článek o fuzzy množinách, ve kterém představuje základy teorie fuzzy množin, kterými lze vhodně popsat komplikované systémy, na jejichž popis klasická matematika nestačila [15]. Na rozdíl od logiky, která pracuje pouze s ostrými hodnotami typu pravda-nepravda, fuzzy logika zavádí pravděpodobnostní příslušnost daného prvku do množiny. Míry příslušnosti do množiny ovlivňuje, jak moc se na daný prvek budou aplikovat definovaná pravidla. Na konci 60.let se prováděly experimenty, které jsou dnes známé jako evoluční algoritmy. Evoluční algoritmy jsou odvozeny podle evolučních procesů probíhajících v přírodě již po miliony let. Princip evolučních algoritmů spočívá v tom, že se periodicky tvoří tzv. generace jedinců, z kterých přežívají při tvorbě nových potomků jen ti nejlepší. Ohodnocení kvality řešení reprezentované jedincem je spočtena tzv. *fitness funkcí.* Dlouhou dobu existoval jen jeden typ evolučních algoritmů - genetické algoritmy. Ty kopírovaly genetické procesy, které probíhají při tvorbě nových potomků v biologickém světě. Až v poslední době se objevil další typ evolučního algoritmu, který má také evoluční charakter (např. diferenciální evoluce), a který se daleko jednodušeji zpracovává na počítačích.

Sedmdesátá léta znamenala příchod **expertních systémů** a rozvoje rozpoznávání obrazů. Expertní systémy se liší především rozsáhlými databázemi, které plní úlohu encyklopedií a obsahují znalosti, bez kterých se při řešení úlohy z dané oblasti nelze obejít. Přestože vznikalo mnoho nových výzkumů a modelů, trend umělé inteligence se postupně odkláněl od neuronových sítí. Až v roce 1986 po úpravách původních modelů dosáhli neuronové sítě skvělých výsledků. Sítě se ukázali ideálními i pro rozpoznávání řeči a psaného textu. Nyní se často setkáváme s pojmem **data mining**, který představuje dolování skrytých a netriviálních informací z rozsáhlých dat.

Většině algoritmům chybí takzvaná samostatnost, čímž se rozumí potřeba lidské síly pro jejich spuštění. Často také člověk musí sestavit trénovací množinu, stanovit topologii pro neuronové sítě nebo nastavit pravidla pro fuzzy logiku. Proto se vědci snaží sestavit stroj, který by pro svoji činnost člověka potřeboval málo či vůbec. Řešením je implementovat schopnost přizpůsobit se měnícím okolním podmínkám. Takovéto stroje jsou charakterizovány pojmem *autonomní.* V dnešním světě jsou již vyvíjeny samostatné systémy, pod čímž si můžeme představit robota, který se umí samostatně chovat v jakémkoliv prostředí bez pomoci vzdáleného supervizora (člověk nebo jiný nadřízený systém). Tito roboti jsou používáni například v armádě pro průzkum nepřátelského prostředí ze vzduchu nebo země. Snaha o samostatnost systému je zřejmá, protože když dojde ke ztrátě spojení mezi robotem

a lidskou obsluhou, pak musí robot být schopen samostatné činnosti.

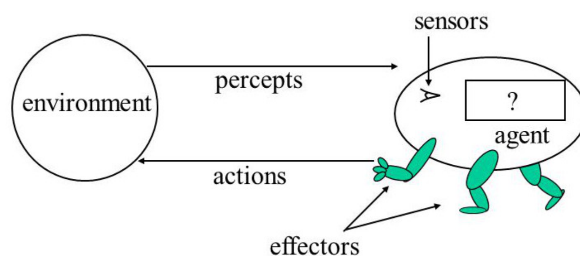
V současnosti je umělá inteligence dost využívána v lékařství. Ve špičkových laboratořích vědci zdokonalují svalové buňky, které jsou pak využívány jako ovladače a akční členy v jednoduchých umělých mechanismech. Kniha [8], vydaná v roce 2007, predikovala, že se za několik let budou na stejném principu sériově vyrábět protézy, které budou zcela organickou součástí poškozených lidských těl a v dnešní době již existuje mnoho takových případů. Již dlouho lidé využívají umělých kyčlí, srdcí, ledvin nebo plic. Velké očekávání byla vkládána také do výzkumu kmenových buněk z kostní dřeně. Myšlenkou je dopravit kmenové buňky na poškozené místo v těle, kde se z nich mohou vyvinout například buňky chrupavky, jater, plic nebo jiného orgánu. Umělá inteligence přináší naději, že se lidstvu podaří vyvinout řešení pro pacienty s roztroušenou sklerózou nebo Parkinsonovou či Alzheimerovou chorobou.

V kapitole *Roboty včera, dnes a zítra* v knize [15] je nastíněno, jakým směrem se vývoj umělé inteligence bude ubírat. Budoucnost zcela jistě patří umělé inteligenci, která se bude neustále zlepšovat. Pravděpodobně vzniknou i některá dnes neznámá odvětví. Použití umělé inteligence se bude nejspíš vyvíjet dvěma směry, a to v samostatných systémech a systémech buď s centrálně umístěnou inteligencí nebo v systémech s distribuovanou umělou inteligencí. V budoucnosti lze také očekávat mnoho legislativních úprav pravidel komunikace s umělým světem. Závažný problém představuje vznik umělého života v informačních systémech, který je schopný manipulovat s elektronickým podpisem nebo zbraňovými systémy. Řešení problému je možné pomocí antiinteligentních programů, které tyto systémy monitorují a snaží se detekovat podezřelé chování. V důsledku těchto hrozeb proběhlo několik jednání o návrhu právní regulace, která zakazuje výrobu samoreprodukujících se programů či vývoj aplikací s implementovaným pudem sebezáchovy [3]. Scénáře, kdy roboti ovládnou celý svět ale patří spíše do sci-fi filmů. Velmi zajímavé čtení na toto téma o rozdílech ve fungování lidského mozku oproti počítačům a problémech umělé inteligence nabízí kniha *On Intelligence* [5]. Autor zde popisuje, že lidská mysl není tvořena pouze mozkovou kůrou, ale celým lidským těle. Podle něho je reálné sestavit inteligentního robota, který bude mít ekvivalentní mozkovou kůru a lidské smysly, ale zakomponování emočních systému a lidských zkušeností považuje za nereálný a zcela nesmyslný úkol.

2.2 Inteligentní agenti

Agent je počítačový systém, který je situován do určitého prostředí a je schopen provádět autonomní akce tak, aby splnil své dané cíle. Agent prováděním svých akcí ovlivňuje prostředí, ve kterém se nachází. To, jak se prostředí ovlivní, záleží vždy na typu a aktuálním stavu prostředí a prováděné akci. Je tedy možné, že vykonání stejné akce může mít v různých stavech prostředí jiné výsledky. Výběr akce může být realizován přes funkci užitku, která každému z množiny stavů prostředí přiděluje reálné číslo, které udává užitek agenta v tomto stavu. Agent má pak znalost o užitku stavů a volí si takovou akci, díky které ovlivní prostředí do takového stavu, že hodnota užitku bude maximální. Stav prostředí agent zjišťuje pomocí svých senzorů. Abstraktní pohled na agenta a prostředí znázorňuje obrázek 2.1.

Každý agent má množinu svých akcí, kterými může prostředí ovlivňovat. V daném prostředí a v dané situaci však agent nemusí být schopen provést některé akce. Například akce *zvednout objekt ze země* může být vykonána pouze v případě, že tento objekt se bude nacházet v blízkosti agenta a bude vážit méně, než je agent schopný unést. Každá akce tedy může obsahovat předpoklady, jejichž splnění je nutné k tomu, aby tyto akce byly



Obrázek 2.1: Agent situován do prostředí

proveditelné. Prostředí, ve kterém se agent nachází, můžeme klasifikovat do několika typů:

- Spojité nebo Diskrétní

  Stavy prostředí tvoří buď spojitou nebo diskrétní množinu. V diskrétním prostředí je přesný a konečný počet akcí a vjemů.

- Přístupné nebo Nepřístupné

  Určuje, zda agent svými senzory může vnímat celé prostředí nebo pouze jeho část. Příkladem přístupného prostředí může být šachovnice, u které má agent veškeré informace o všech políčkách. Opakem je agent situovaný v bludišti, který postupným procházením získává znalosti o bludišti.

- Statické nebo Dynamické

  Statické prostředí se mění pouze v případě, že ho agent svou akcí ovlivnil. U dynamického prostředí se změny mohou projevit bez jakéhokoli přičinění agenta.

- Deterministické nebo Nedeterministické

  V deterministickém prostředí je nový stav dán aktuálním stavem a provedenou akcí. Pokud agent vykoná určitou akci v daném stavu nedeterministického prostředí, pak nové prostředí nemusí být vždy stejné.

- Epizodní nebo Neepizodní

  Epizody rozdělují běh systému na části, které jsou na sobě navzájem nezávislé.

- Strategické

  Více agentů jedná v prostředí současně.

Na mnoho řídících systémů lze pohlížet jako na agentní systémy. Například termostat kontroluje svými senzory teplotu prostředí a na základě jednoduchých pravidel se rozhodne, zda nechá topení zapnuté nebo ho vypne. Těžko ale o tomto systému můžeme prohlásit, že je inteligentní. Inteligentní agenti by měly mít tyto vlastnosti:

- Autonomní

  Agent jedná v prostředí bez přímého vlivu z okolí a má úplnou kontrolu nad svým chováním.

- Reaktivní

  Agenti, kteří vnímají prostředí a jsou schopni včas a neustále reagovat na měnící se prostředí tak, aby dosáhli svých cílů.

- Proaktivní

  Agent se ujímá iniciativy a sám ovlivňuje prostředí za účelem dosažení svých cílů.

- Sociální

  Agenti interagují s ostatními agenty pro dosažení svých cílů.

Vytvoření reaktivních či proaktivních systémů není složité, ale vytvoření systému, který dosahuje efektivní rovnováhy mezi proaktivním a reaktivním chování, už tak jednoduché není. Snahou je, aby agenti dosáli svých cílů pomocí akcí, které tvoří nějaký vzorec. Nevyžádané je ale chování, kdy se agent snaží slepě vykonávat tyto akce i když je jasné, že to fungovat nebude nebo že daný cíl není z nějakého důvodu dosažitelný. V takových případech chceme, aby agent byl schopný reagovat na tuto situaci. Samozřejmě také nechceme, aby agent neustále reagoval na své okolí a nikdy se nezaměřil na cíle, kterých chce dosáhnout. Úspěšně integrovat reaktivní a cílem řízené chování je jedním z velkých problémů při návrhu agenta.

Mnoho programátorů pracující s objektově orientovanými jazyky mají tendenci vzhlížet na agenty jako na objekty. Existuje ale několik zásadních rozdílů mezi agenty a objekty. Stav agenta je dán jeho mentálními stavy a agent jedná samostatně a aktivně vzhledem ke svým cílům. Má svobodnou vůli, zda vyhoví požadavkům ve zprávě a na rozdíl od objektu to nemusí dělat zadarmo. Agent může požadavek odmítnout s tím, že nemá čas nebo ho neumí zpracovat. Na druhé straně objekt je výpočetní entita, která zapouzdřuje nějaký stav a je schopná vykonávat akce a komunikovat pomocí předávání zpráv. Velkým rozdílem mezi objektem a agentem je stupeň jejich autonomie. Objekt může mít kontrolu nad svým vnitřním stavem a pomocí modifikátoru přístupu (public, private, protected) určíme, odkud můžeme přistupovat k proměnným nebo metodám. Například metodu typu public je možné vyvolat kdykoliv a metoda se musí provést. Objekt nemá možnost odmítnout toto volání tak, jak to může udělat agent. Na rozdíl od objektu je agent schopný reaktivního, proaktivního a sociálního chování. Agent má také plnou kontrolu nad svým chováním a je realizován jedním vláknem, kdežto objektově orientovaný model řídí jeden proces.

2.3 Anticipující agenti

Technika reaktivního plánování určuje v každém kroku následující akci, v závislosti na aktuálním kontextu a popisu chování. Na základě svých vnitřních stavů a vjemů o prostředí si agent volí akci, kterou provede. Rozhodování při výběru může ovlivnit také existence předem daných plánů z knihovny plánů. Opakem reaktivního agenta je agent, který využívá anticipaci a pokouší se predikovat budoucí stav prostředí a využívá predikci pro své rozhodnutí. Příkladem může být rozhodnutí o tom, zda si má agent vzít deštník, když jde ven. Reaktivní agent si deštník vezme pouze tehdy, když venku prší. Anticipující agent se rozhodne na základě toho, zda je nebe zamračené a jaký je tlak vzduchu. Může tedy předpokládat, že za několik minut začne pršet, proto si vezme deštník. Podle definice uvedené v [8] anticipující systém obsahuje model sebe sama a popřípadě svého okolí, na kterém predikuje své budoucí možné stavy. Na základě této predikce ovlivňuje své chování tak, aby budoucí stav odpovídal jejich okamžitým záměrům. Později byly také zavedeny termíny anticipující

systém v silném nebo slabém slova smyslu. Rozdílem je to, že silný systém si nepotřebuje vytvářet model, protože ten je explicitně součástí jejich struktury.

2.4 Agenti jako systémy řízené záměrem

Agenti řízeni záměrem se rozhodují na základě svých mentálních stavů. Jedním z modelů řízený záměrem je model **BDI**, kde jednotlivá písmena popisují používané mentální stavy - představa (*belief*), přání (*desire*) a záměr (*intention*). Z hodnot těchto mentálních stavů si už lze dovolit předpovědět, co má agent v plánu. Představy jsou informace o prostředí, kterým agent věří, ale nemusí být vždy pravdivé. Přání ukazují, čeho chce agent dosáhnout. Jednotlivá přání mohou být v rozporu, proto agent nemusí nutně splnit všechny. Záměry jsou možnosti, které si agent vybral pro dosažení. Záměrem také může být částečný cíl, aby se po jeho splnění přiblížil k hlavnímu cíli.

2.5 Mobilní agenti

Definice a funkce mobilních agentů jsou popsány v knize *Agent Technology* [6]. Jsou to softwarové procesy schopné pohybu v sítích a interakcí s cizími agenty. Cíle mobilních agentů jsou různorodé, může se jednat o rezervaci letenek na internetu nebo o řízení telekomunikační sítě. Jedná se o agenty, protože splňují vlastnosti autonomního chování a spolupráce. Mobilních agentů lze využít i při běžných operací. Pokud chce uživatel stáhnout obrovské množství obrázku a z nich použít pouze jediný, pak je časově jednodušší poslat agenta na danou adresu, kde provede lokální hledání popsaného souboru, který pak bude vrácen uživateli jako výsledek akce.

2.6 Komunikace a spolupráce agentů

Pokud spolu chtějí dva agenti komunikovat, je nutné, aby si stanovili takzvanou **ontologii**, která specifikuje, k jaké doméně se bude jejich komunikace vztahovat. Příkladem ontologie může být situace, kdy si dva agenti vyměňují informace o pravidlech fotbalu. Problém ale nastává, pokud jeden agent si pod pojmem fotbal představí ragby (anglicky football). Proto je nezbytné v ontologii určit přesně, k jakému sportu se zasílané zprávy vztahují (například *Soccer-rules*). Agenti spolu komunikují přes zasílání zpráv. Formát zpráv je definován formátem **KQML**, který popisuje typ řečového aktu, obsah zprávy, ontologii, příjemce, odesílatele a další. Parametry KQML zprávy závisí na typu řečového aktu. Příklad KQML zprávy je uveden v kapitole 3.2.1. Typy zpráv s jejich parametry jsou popsány standardem **FIPA** a je jim věnována kapitola 3.

V knize [10] jsou popsány základní doporučení a pravidla pro komunikaci mezi agenty. Pravidla jsou známa pod pojmem *Gricean maxims* a obsahují čtyři hlavní body : kvantita, kvalita, vztah a způsob. Pravidlo kvantity doporučuje poskytnout příjemci zprávy přesně ty informace, které momentálně potřebuje. Neměl by nastat případ, kdy mu agent něco zatají nebo ho zahltí příliš mnoha detaily. Pravidlo kvality klade důraz na to, aby si agenti sdělovali pouze ty informace, o jejichž pravdivosti jsou přesvědčeni. Veškerá komunikace, kterou mezi sebou agenti provádějí, by se měla vztahovat k hlavnímu předmětu projednávané věci. Agenti by tak měli odpovídat přesně na to, na co se jich jiný agent tázal. Poslední pravidlo klade důraz na to, aby agenti odpovídali čistě a jasně, čímž se vyhnou nejasnostem, dvojznačnostem nebo chaosu.

2.6.1 Kooperativní distribuované řešení problému

Pro efektivní běh systému je nutné, aby spolu agenti vzájemně spolupracovali. Postup lze použít v situaci, kdy se skupina autonomních agentů rozhodne spolupracovat pro dosažení společného cíle. Pokud má navržený systém vyřešit nějaký složitější problém, je dobré si úlohu rozdělit do několika snadněji řešitelných podproblémů, které mohou být vyřešeny jednotlivými agenty. Tento proces se anglicky nazývá *Cooperative distributed problem solving*. Dekompozice problému je typicky hierarchická, takže podproblémy jsou dále rozdělovány na menší a menší. Úroveň dekompozice většinou reprezentuje úroveň řešeného problému. Dekompozici úlohy na jednodušší podproblémy může mít na starosti individuální agent. Agenti mezi sebou v systému sdílejí cíle a jsou navrženi tak, že pokud umí vyřešit nějakou úlohu, která je sdílena mezi agenty, tak ji vyřeší. V případě, že agent vyřeší svou část úlohy, může výsledek zaslat ostatním agentům (lze využít komunikační akty *request*, *response* nebo *inform* popsané v kapitole 3.2). Vyřešené podproblémy se pak integrují do celkového řešení. Této spolupráce se využívá v případě, že všichni agenti jsou navrženy stejnou organizací a cíle jednotlivých agentů nejsou v rozporu.

Tento princip využívá protokol kontraktační sítě (*Contract net*), kde každý agent v síti může vystupovat v jedné ze dvou rolí - jako manažer úkolu nebo jako jeho řešitel. V případě, že je manažer, tak informuje ostatní o novém úkolu. Pokud se chce agent stát řešitelem úkolu, pak zašle manažerovi nabídku, která obsahuje jeho schopnosti a podmínky potřebné k vyřešení úlohy. Manažer pak z obdržených nabídek vybere tu, která je nejlepší a agenta kontaktuje. Tento agent je pak zodpovědný za vyřešení úkolu.

Kniha [11] popisuje čtyři fáze při kooperativním řešení problému. Celý proces začíná v okamžiku, kdy si některý agent uvědomí potenciál skupinového provedení některého úkolu. Agent následně začne budovat tým. Někteří agenti nemají zájem o dosažení vyřešení dané úlohy, proto nechtějí být součástí týmu. Druhá část končí v ten moment, kdy je sestaven tým pro řešení úlohy a každý její člen má závazek k provedení společného úkolu. V třetí fázi agenti jednají o plánech, kterými lze dosáhnout cíle, a poslední fáze je už vykonávání zvoleného plánu.

2.6.2 Agenti s vlastním zájmem

Běžnější než pracování agentů na vyřešení distribuovaného problému je společnost agentů s vlastním zájmem (*self-interested agents*). Tito agenti nesdílejí své cíle, jsou navrženi tak, aby reprezentovali své zájmy. Zájmy jednoho agenta mohou být v konfliktu se zájmy jiného, stejně jako to bývá v lidské společnosti. Agent pro dosažení svých cílů musí spolupracovat s ostatními.

Existují dva parametry, kterými se dá ohodnotit úspěšnost implementovaného mulitagentního systému. Prvním parametrem je **soudržnost**, která ukazuje, jak dobře se multiagentní systém chová jako celek. Zaměřuje se na kvalitu řešení, efektivitu využívání zdrojů nebo na to, jak se výkon systému snižuje při selhání. Druhým parametrem je **koordinace**. V perfektně koordinovaném systému budou schopnosti agentů popsané nějakým vnitřním modelem. Agenti budou moci vzájemně předvídat schopnosti ostatních agentů a v systému bude co nejméně konfliktů, při kterých se agenti zbytečně narušují. Tímto se lze vyhnout konfliktům, jejichž řešení stojí čas a námahu.

2.6.3 Částečné komplexní plánování

Spolupráce agentů může být řešena pomocí takzvaného *Partial global planning*, který zahrnuje tři opakující se kroky. V prvním kroku si každý agent rozhodne o svém cíli a vygeneruje si plán pro jeho dosažení. V dalším kroku si agenti vymění informace o svých plánech a určí se, kde plány a cíle interagují. Nakonec agenti změní své lokální plány tak, aby lépe spolupracovali. Tyto opakované akce jsou zahrnuty do struktury částečného globálního plánu (*partial global plan*), která se postupně generuje s tím, jak si agenti mění informace o svých plánech.

Struktura obsahuje cíl, kterého se celý systém snaží dosáhnout, dále mapu aktivit, která reprezentuje, co agent dělá a jakých výsledků dosáhne. Ve struktuře je také graf konstrukce řešení, který popisuje, jak by spolu agenti měli komunikovat a jaké informace by si měli vyměňovat, aby celý systém dosáhl požadovaného výsledku. Protože agenti vnímají pouze svojí aktivitu a své okolí, sdílení informace o výsledcích ostatních agentů jim může pomoci. Takovou informaci může agent buď poslat všem a nebo pouze těm, kteří se o výsledek zajímají (využívá se komunikační akt *subscribe*). Když se detekuje, že více agentů pracuje na stejné úloze, pak se z nich vybere pouze jeden, který tuto úlohu dořeší. Ostatní agenti práci zruší z důvodu šetření zdrojů. Pokud akce nějakého agenta negativně ovlivňuje ostatní agenty nebo jim brání ve vykonání svých plánů, pak se provede přeplánování této konfliktní akce.

2.6.4 Normy a zákony

Norma je stanovený a očekávaný vzor chování. Zákon (*social law*) je podobný normě, ale navíc obsahuje nějakou autoritu. Příkladem normy může být chování u autobusové zastávky. Lidé čekající na autobus vytvářejí frontu v pořadím, v jakém přišli. Po příjezdu autobusu se nejprve počká, až všichni cestující vystoupí a až poté začnou lidé z fronty postupně nastupovat. Tato norma není prosazována, je to pouze očekávané chování a nedodržování této normy způsobí pouze nepochopení a odsouzení přítomných lidí na zastávce. Při nedodržení zákonů je postih daleko větší (například zákon o silničním provozu). Zákony mohou být stanoveny offline nebo za běhu systému skupinou agentů. Offline navrhování zákonů je jednodušší pro implementaci, bohužel ale ne vždy jsou všechny možné stavy systému dopředu známy při jeho návrhu.

2.6.5 Nekonzistence mentálních stavů agentů

Jedním z největších problémů při spolupráci agentů je nekonzistence mezi agenty. Různí agenti mohou mít nekonzistentní mentální stavy. Například nekonzistence mezi představami agentů může vzniknout při poruše senzorů u agenta nebo se prostředí může nějak změnit, ale agent nemá přehled o celém prostředí a tuto změnu nezaregistruje. Ve větších systémech je problém nekonzistence nevyhnutelný, existují však přístupy pro jeho řešení. Jedním z řešením může být ignorování. Příkladem je kontraktační síť, kdy pouze manažer úlohy má správnou představu o okolí a představy ostatních agentů jsou ignorovány a považovány za nepravdivé. Další možností může být vyřešení nekonzistence pomocí vyjednávání. Při vyjednávání jsou používány komunikační akty *propose*, *accept proposal* nebo *refuse* (viz kapitola 3.2).

Problém spolupráce úzce souvisí s řízením vzájemné závislosti mezi akcemi více agentů. Příkladem může být, když dva agenti chtějí vstoupit do místnosti ve stejný čas, ale dveřmi může projít pouze jeden agent. Pro agenty je tedy nezbytné se domluvit, kdo půjde první.

Od některého agenta se očekává, že zašle druhému zprávu buď s výzvou ať projde dveřmi nebo s oznámením, že v daném čase vstoupí do místnosti sám. Jak již bylo zmíněno, od agenta se očekává, že bude sociální. Pokud agent vykonal nějaký plán a ví, že jiný agent chce vykonat stejný plán, pak ho informuje o výsledku, aby mu ušetřil práci i čas.

Kapitola 3

FIPA

FIPA[1], což je zkratka pro název **Foundation for Intelligent Physical Agents**, je organizace zabývající se standardizací pro vývoj agentních systémů. FIPA obsahuje celkem 25 norem, které popisují architekturu agentního systému, řízení agentů, komunikaci mezi jednotlivými agenty v rámci jedné nebo více platforem a další oblasti související s agentními systémy.

3.1 Abstraktní architektura agentního systému

Základním prvkem je **agentní platforma**, kde se spouštějí jednotliví agenti. Agentní platforma zapouzdřuje všechny níže popsané komponenty a vytváří jeden agentní systém.

Hlavním prvkem v agentním systému je autonomní agent, který má svůj jedinečný identifikátor **AID** v rámci jedné platformy. Agent obsahuje atributy, které ho popisují a tzv. **Agent locator**, který umožňuje ostatním agentům získat informace o tom, jak a na které adrese s daným agentem komunikovat. Příkladem agent-locatoru může být typ *SMTP* a adresa *agent7@mail.com*.

Pokud chce agent vyhledat jiného agenta, použije k tomu službu, kterou nabízí **Agent-directory-service (ADS)**. ADS obsahuje různé **Agent-directory-entry (ADE)**, které odkazují na jednotlivé agenty. ADE obsahuje agentovo jméno, atributy a agent-locator. Nepovinné atributy v ADE mohou obsahovat dodatečné informace o agentovi (např. kolik stojí jeho služby atd.) a slouží k popisu agenta, kterého může chtít jiný agent vyhledat. Agent může kontaktovat ADS s požadavkem na:

- **Register**

  registraci své ADE, aby ho ostatní agenti mohli na platformě vyhledat

- **Deregister**

  zrušení registrace své ADE, kterou si v minulosti již registroval

- **Modify**

  ke změně parametrů své ADE, která už je registrovaná u ADS

- **Search**

  k vyhledání ADE jiného agenta (po získání ADE ho může agent kontaktovat)

Neúspěšnost zvolené akce lze zjistit pomocí návratové hodnoty z ADS, která je ještě doplněna o vysvětlení, proč daná akce nemůže být provedena. Příčinou selhání akce může být nenalezení popisované ADE nebo její neplatnost či nedostatečná práva agenta. V opačném případě ADS vrací potvrzení o úspěšnosti akce. U akce *Search* jsou vrácena také ADE, která splňují vyhledávací kritéria.

Na jedné agentní platformě se také nachází **Service-directory-service (SDS)**, jehož nabízené služby mohou agenti využívat. Hierarchie SDS se podobá hierarchii ADS. Služby jsou popsány pomocí **Service-directory-entry (SDE)**, který obsahuje jméno a typ služby, její atributy a **service-locator**, který popisuje agentům, jak přistupovat ke službě. Agenti stejně jako u ADS mohou posílat požadavky na vyhledání služby jiných agentů nebo na registraci, úpravu či zrušení své služby pro ostatní agenty.

Při startu agenta na platformě je agentovi vždy přístupná minimálně jedna služba, označovaná jako **Service root**, která obsahuje množinu SDE záznamů, čímž umožňuje agentovi nabootovat služby jako jsou ADS, SDS či MTS.

Velmi důležitou částí platformy je **Message-transport-service (MTS)**, který zajišťuje veškerou komunikaci mezi agenty. Pro komunikaci dvou agentů na různých platformách se využije právě MTS. MTS odesílá a přijímá **Transport-Message (TM)** s obsahem i formátem předávané zprávy a informacemi o odesílateli a příjemci. TM zapouzdřuje další možnosti jako jsou jazyk zprávy, zakódování nebo ontologie, která popisuje, co obsah zprávy reprezentuje. MTS může využívat agent-locator k vybrání komunikace s agentem.

3.2 Komunikační akty agentů

Zprávy, které si agenti mezi sebou zasílají, mohou být různých typů. Někdy agent potřebuje pouze informovat agenta, v některých případech se agent táže jiného agenta a potřebuje od něho získat odpověď atd. Jediný povinný komunikační akt pro FIPA je **not-understood**, pomocí kterého agent sděluje příjemci, že si je vědom jeho požadavku, ale neumí na něj reagovat. U většiny agentních systémů je ale potřeba, aby agenti uměli používat více komunikačních aktů, což ve výsledku přispívá k lepší spolupráci. Nejpoužívanější akty jsou popsány níže.

3.2.1 Inform

Pokud si agent zvolí jako komunikační akt *Inform*, pak odesílatele informuje, že odeslané tvrzení je pravdivé na základě agentova přesvědčení. Akt *Inform* je zvolen v případě, že odesílatel chce, aby příjemce věděl o pravdivosti tvrzení, které je v obsahu zprávy. Souvisejícím aktem je *Inform if*, kterým agent chce požádat příjemce o odeslání pravdivosti tvrzení podle jeho přesvědčení. Příklad zprávy *Inform* je demonstrována níže.

```
(inform
   :sender (agent-identifier :name i)
   :receiver (set (agent-identifier :name j))
   :content "weather (today, raining)"
   :language Prolog)
```

3.2.2 Request

Aktem *Request* odesílatel žádá příjemce o vykonání akce, která je popsána v obsahu zprávy jazykem, kterému příjemce rozumí. Agent tímto aktem může požádat jiného agenta například o otevření souboru. Pokud odesílatel chce, aby příjemce vykonal akci pouze za určitých podmínek, pošle akt *Request When* a v obsahu zprávy popíše podmínky akce a samotnou akci, kterou má příjemce provést. V případě, že odesílatel chce, aby příjemce provedl danou akci vždy, když bude splněna daná podmínka, využije akt *Request Whenever*.

3.2.3 Subscribe

Agent, který přijal zprávu typu *Subscribe*, bude informovat odesílatele o hodnotě objektu při každé jeho změně. Agent může tento akt využít například k získání aktuálního kurzu české koruny. Objekt, který chce agent sledovat, je popsán v obsahu zprávy.

3.2.4 Agree

Pokud agent chce potvrdit, že někdy v budoucnu danou akci provede, odešle zprávu typu *Agree*, jejíž obsahem budou podmínky a akce, která se při splnění podmínek provede. Tento komunikační akt agent často zasílá po obdržení zprávy *Request*.

3.2.5 Propose

Agent využije komunikační akt *Propose*, pokud chce předložit návrh k provedení akce jinému agentovi za splnění určitých podmínek. Akce a podmínky jsou popsány v obsahu zprávy. Pokud příjemce této zprávy má v úmyslu akci provést, informuje odesílatele aktem *Accept Proposal*. V jiném případě pošle zprávu typu *Reject Proposal*, kterým dává najevo, že za daných podmínek nemá zájem o provedení této akce (důvodem může být například vysoká cena akce). Akt *Call for Proposal* může agent využít v situaci, kdy chce od odesílatele získat některé parametry pro akci, kterou chce v budoucnu provést. Tento akt je často využíván při vyjednávání mezi agenty.

3.2.6 Cancel

Zprávou *Cancel* agent dává najevo, že nemá zájem, aby příjemce provedl předtím požadovanou akci. Tento akt může agent použít, pokud v minulosti poslal zprávu typu *Subscribe* a již nemá zájem o to, aby mu byly zasílány zprávy při každé změně sledovaného objektu.

3.2.7 Confirm

Když odesílatel chce informovat příjemce o pravdivosti tvrzení, které je popsáno v obsahu zprávy, zvolí akt *Confirm* nebo *Disconfirm* na základě jeho vlastního přesvědčení o tvrzení.

3.2.8 Failure

Když agent selhal při provádění požadované akce, tak tuto akci a důvod selhání pošle přes komunikační akt *Failure*.

Agenti, kteří mezi sebou komunikují a zasílají si tzv. *ACL zprávy* určují komunikační akt parametrem *Performative*. Zpráva obsahuje ale ještě několik dalších parametrů, které mohou být využity k efektivní spolupráci agentů. Mezi další parametry patří:

- Sender, Receiver : Odesílatel a příjemce zprávy.
- Reply-to : Odpověď na tuto zprávu nebude zaslána odesílateli (*Sender*), ale agentovi popsaným v *Reply-to*.
- Content : Obsah zprávy, který je závislý na komunikační aktu. Nejčastěji je zde popsaná akce a podmínky pro její vykonání.
- Language : Popisuje použitý jazyk v obsahu zprávy.
- Encoding : Určuje kódování zprávy.
- Ontology : Popisuje, co obsah zprávy reprezentuje, což pomůže agentovi ke správné interpretaci obsahu zprávy.
- Protocol : Definuje interakční protokol mezi zprávami, který řídí konverzaci agentů. Parametr je volitelný, pokud je ale specifikován, pak parametry *Conversation-id* a *Reply-by* jsou povinné.
- Conversation-id : Parametr, který zapouzdřuje množinu zpráv dohromady, čímž tvoří konverzaci.
- Reply-with, In-reply-to : Hodnota parametru *Reply-with* se použije při odpovědi jako parametr *In-reply-to*.
- Reply-by : Popisuje čas, do kterého chce agent obdržet odpověď na odeslanou zprávu.

Agenti samozřejmě mohou využívat mnoho dalších komunikačních aktů, které jsou popsány na stránce FIPA[1].

[1] http://www.fipa.org/specs/fipa00037/index.html

Kapitola 4

Návrh frameworku

V kapitole se seznámíme s návrhem frameworku pro vývoj agentních systémů. Framework je nadstavbou nad knihovnou JADE, která je popsána v podkapitole 4.1. Kromě funkčnosti, kterou JADE nabízí, jsou ve frameworku navrženy metody pro lepší práci se skupinami agentů, s komunikací mezi agenty, se službami nebo s BDI stavy agentů. Některé složitější implementace chování v JADE budou zapouzdřeny do jednodušších metod. Návrh frameworku je rozdělen podle funkčnosti do jednotlivých podkapitol, které jsou detailně popsány níže. Při návrhu byl kladen důraz na dodržování standardu FIPA.

4.1 JADE

JADE (Java Agent Development Framework) je framework usnadňující vývoj inteligentních agentů a agentních systémů v jazyce Java. JADE podporuje FIPA standardy, poskytuje prostředí, kde jsou agenti spouštěni a nabízí nástroj pro řízení a monitorování agentní platformy s grafickým výstupem. Agenti v JADE jsou implementováni jako vlákna a žijí v takzvaných kontejnerech. Za běhu si JADE vytvoří několik kontejnerů, pouze jeden je ale označován jako hlavní. Hlavní kontejner slouží jako bootovací a všechny ostatní kontejnery jsou u něho registrovány. Hlavní kontejner obsahuje seznam všech kontejnerů v systému s informacemi o tom, jak je kontaktovat, dále seznam agentů v systému i s jejich vnitřními stavy a pozicí. Ostatní funkcionalita, praktické příklady a popis, jak JADE spouští agenty je popsán v [4].

4.2 Specifikace požadavků pro využívání frameworku

Pro správné využití frameworku musí být splněny vyjmenované požadavky.

4.2.1 Požadavky na rozhraní frameworku

Rozhraní frameworku obsahuje následující funkčnost:

- Vytváření a správu skupin agentů,
- vytváření a správa BDI stavů agenta,
- vytváření a správa služeb,

- různé způsoby komunikace mezi agenty:
  přijímání zpráv zvoleného typu,
  komunikace ve skupině,
  komunikace s čekáním na odpověď zvoleného typu,
  hlasování mezi agenty,
  historie obdržených i odeslaných zpráv.

4.2.2 Požadavky na aplikaci

Třída, která bude framework využívat, musí:

- Mít importovanou knihovnu JADE,
- být implementovaná v jazyce Java,
- dědit třídu inteligentního agenta z frameworku.

4.3 Komunikace

Jedna z nejdůležitějších funkcí frameworku jsou metody pro komunikaci mezi agenty.

4.3.1 Přijímání zpráv

Agent při svém spuštění zavolá metodu, která běží po celou dobu jeho života. Metoda neustále cyklí ve smyčce, kde zpracovává přijaté zprávy pomocí funkce `receive()` z JADE třídy `Agent.java`. V případě přijetí zprávy, se na základě jejích parametrů (jméno odesílatele, typ zprávy, id konverzace apod.) rozhodne, jaké metodě se zpráva přepošle. Agent má k dispozici metodu, která ho informuje o příjetí zprávy v rámci skupiny, ve které je sám agent jejím členem. Dále si agent může popsat, jaký typ zpráv má metoda přijímat. Tímto si agent může registrovat na odběr zprávy s určitým odesílatelem, obsahem, typem, protokolem a s dalšími parametry. Třetí metoda přijímá zprávy, které agent nemá zaregistrovány k odběru a nejedná se o skupinovou zprávu.

4.3.2 Odesílání zpráv

Agent může odesílat zprávy několika způsoby. Vedle klasického poslání zprávy může agent zvolit variantu, kde zprávu pošle a čeká na přijetí zprávy určitého typu. Tím se dá docílit jednoduchého potvrzení přijetí zprávy (ACK) mezi agenty. Tyto varianty se dají použít i pro komunikaci ve skupině, což je detailněji popsáno v podkapitole 4.5.

4.3.3 Hlasování

V případě, že se agent chce dotázat několika jiných agentů a rozhodnout se podle jejich odpovědí, může využít metody pro hlasování. Pokud stačí, aby agenti vyjádřili svůj souhlas či nesouhlas pomocí typu zprávy (*AGREE* nebo *REFUSE*), pak metoda vrací rovnou výsledek hlasování. Je stanovený maximální čas, do kterého agent musí odpovědět na výzvu o hlasování. Pokud tak neučiní, počítá se to jako projevení nesouhlasu.

4.3.4 Historie komunikace

Agent si veškerou proběhlou komunikaci uchovává, aby mohl umožnit filtrování zpráv podle zadaných požadavků. Framework nabízí následující funkce pro rychlé vyhledávání zpráv:

- Filtrování zpráv od zvoleného agenta,
- filtrování zpráv, které zaslal zvolený agent,
- filtrování veškeré komunikace se zvoleným agentem (sloučení prvních dvou metod),
- filtrování zpráv, které jsou ve specifikovaném formátu.

4.4 Správa BDI stavů

Agent volí své akce na základě vjemů prostředí a svých představ. V představách má uloženo vše podstatného pro jeho budoucí chování. Framework bude umožňovat vytváření, upravování, filtrování a mazání představ. Představy jsou uloženy ve formě termů [7]. Pojmenování představy odpovídá funkčnímu symbolu f četnosti n, který obsahuje $t_1, \ldots, t_n$ termů, které vyjadřují parametry představy. Každý z t_1 až t_n termů je reprezentován jménem a hodnotou. Filtrování je poté prováděno na základě jména funkčního symbolu a jmen jeho parametrů.

Při filtrování lze využívat několika základních metod, uživatel si ale může nadefinovat vlastní složitější metody. Podle datového typu parametru představy je možné využít některých ze základních metod. Pro číselné parametry lze použít metody pro filtrování větších, menších nebo stejných hodnot. Při filtrování řetězců se může využít operací rovnosti nebo testování výskytu podřetězce. U parametrů s datovým typem seznam je možné filtrovat pouze ty představy, jejichž seznam obsahuje specifikovanou hodnotu. Pro jedno filtrování lze využít i více než jednoho filtru. Použití metod pro správu BDI stavů agenta i s jeho výsledky může vypadat například takto:

```
NovaPredstava("osoba", [ ["Jmeno","Petr"],["Vek",31],["Mesto","Praha"] ]);
NovaPredstava("osoba", [ ["Jmeno","Lukas"],["Vek",25],["Mesto","Brno"] ]);
NovaPredstava("osoba", [ ["Jmeno","Michaela"],["Vek",23],["Mesto","Policka"] ]);

Predstava("osoba");
Výsledek: [ ["Petr",31,"Praha"], ["Lukas",25,"Brno"], ["Michaela",23,"Policka"] ]

Filtr mladsiNez30("Vek", ZakladniMetody.CisloMensi, 30);
Filtr jmenoLukas("Jmeno", ZakladniMetody.RetezecRovno,"Lukas");

Filtruj("osoba", mladsiNez30);
Výsledek: [ ["Lukas",25,"Brno"], ["Michaela",23,"Policka"] ]

Filtruj("osoba", [mladsiNez30, jmenoLukas]);
Výsledek: [ ["Lukas",25,"Brno"] ]
```

4.5 Správa skupin agentů

Framework umožňuje agentům sdružovat se do skupin a vykonávat různé typu operací. Agent může vytvářet skupiny a přidávat do nich další členy, komunikovat v rámci skupiny,

zrušit nebo opustit skupinu, vyhledávat skupiny podle zadaných parametrů nebo vyloučit člena ze skupiny. Správu všech skupin bude mít na starosti pouze jeden inteligentní agent - **správce**. Ostatní agenti budou zasílat správci požadavky na provádění skupinových operací a správce na základě pravidel rozhodne, zda je operace proveditelná a případně ji zrealizuje. Požadavek agenta může být zamítnut, například když chce vyloučit člena ze skupiny, do které sám nepatří. Správce si uchovává všechny informace o založených skupinách a jejích členech. Protokol komunikace agenta se správcem je popsán v dalších podkapitolách. Pokud správce nerozumí požadavku od agenta, pak odpovídá zprávou s typem *NOT-UNDERSTOOD*.

Každá vytvořená skupina obsahuje unikátní jméno a může být buď veřejná nebo tajná. V případě tajné skupiny není pro agenty, kteří do ní nepatří, možné ji vyhledat i přes zadání správných parametrů. Agent, který skupinu vytvořil, je označen jako její zakladatel a také určuje, zda skupina bude demokratická nebo o všem bude rozhodovat právě on. Pokud je skupina demokratická, pak se při operacích přijímání nového člena nebo vyloučení člena rozhoduje podle hlasování mezi členy. Členové skupin mají k dispozici základní informace o skupině a jejích členech a jsou také informováni v situacích, kdy byl přijat nebo vyloučen jiný člen.

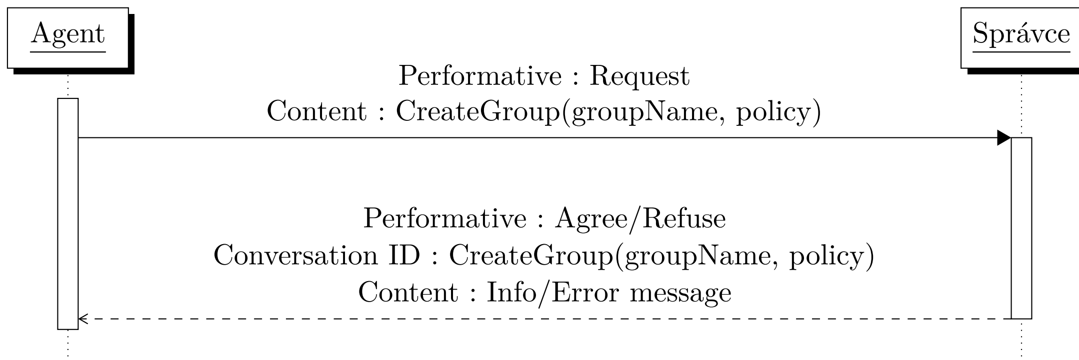
4.5.1 Komunikace agentů ve skupině

Komunikaci agentů ve skupině má na starosti také správce. Agent, který chce komunikovat s ostatními členy skupiny, zašle správci požadavek s obsahem *SendMessageToGroup(group,content)*, který zprávu přepošle ostatním členům. Agenti také mohou pomocí frameworku zaslat zprávu skupině a čekat, dokud na ní všichni příjemci neodpoví určitým typem zpráv. Typ zpráv je specifikován díky třídě `MessageTemplate` z knihovny JADE.

4.5.2 Vytvoření a zrušení skupiny

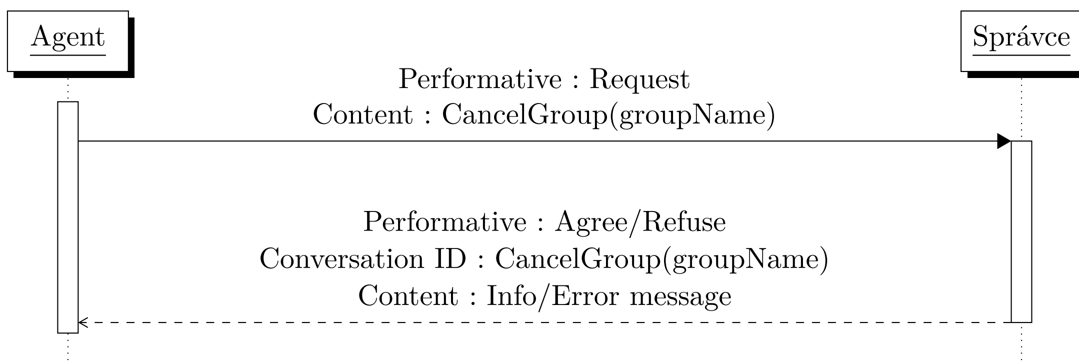
Při vytváření skupiny agent posílá správci zprávu typu *REQUEST* s obsahem *CreateGroup(groupName,policy)*, kde *groupName* je unikátní jméno skupiny a hodnota *policy* určuje, zda se jedná o demokratickou skupinu. Vytvoření nové skupiny potvrdí správce agentovi zasláním zprávy typu *AGREE*. Pokud skupina nebyla vytvořena, správce odpovídá agentovi zaslání zprávy typu *REFUSE* s obsahem, který vysvětluje, proč agentův požadavek nemůže být splněn. Jediným důvodem pro zamítnutí požadavku na vytvoření nové skupiny je situace, kdy už existuje skupina se stejným názvem.

Zakladatel může do skupiny přidat členy už při jejím vytvoření. Docílí toho zasláním zprávy typu *REQUEST* s obsahem *AddAgentsByOwner(agent1, agent2, agent3)*. Správce těmto členům zašle uvítací zprávu typu *INFORM* s textem *WelcomeMessage(groupName)*, díky které agent zjistí, že je členem nové skupiny



Obrázek 4.1: Vytvoření skupiny

Zrušit se může jen nedemokratická skupina a to pouze jejím zakladatelem. *REQUEST* požadavek na zrušení skupiny se zasílá správci s obsahem *CancelGroup(groupName)*. Pokud se skupina zrušila, pak jsou její členové informování zprávou typu *INFORM* s identifikací *CancelGroupInfo(groupName)* a agentovi, který zasílal požadavek je odpovězeno typem *AGREE* s předmětem konverzace stejným jako byl obsah jeho požadavku - *CancelGroup(groupName)*.



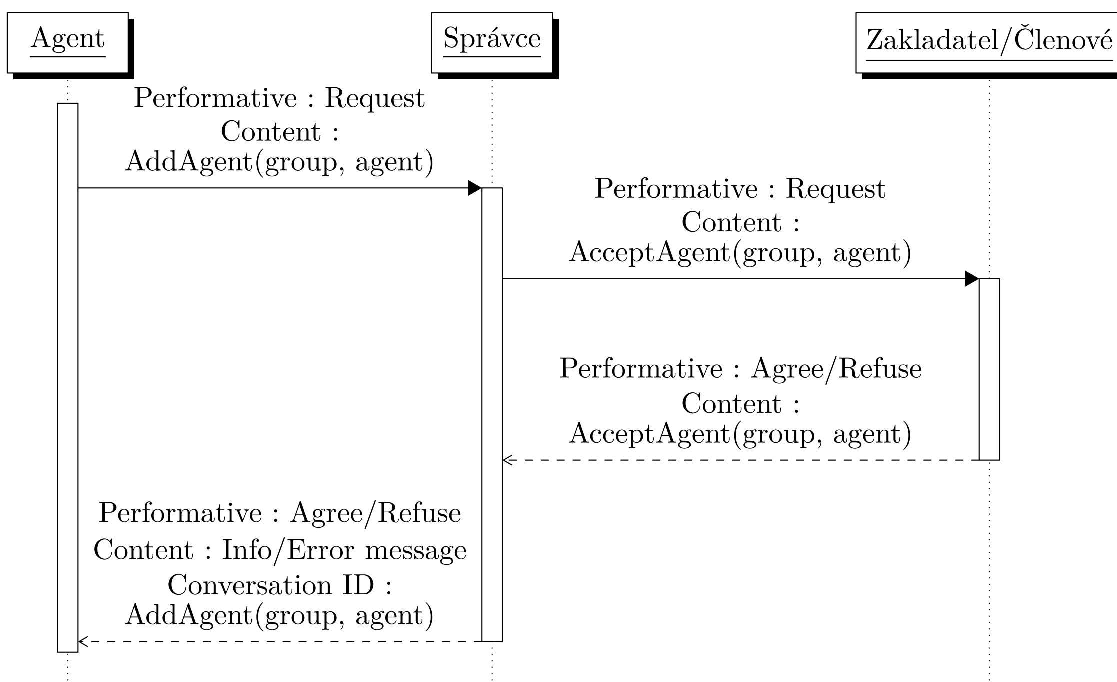
Obrázek 4.2: Zrušení skupiny

4.5.3 Přidávání nových agentů do skupiny

Agenti mohou být přidáni do skupiny zakladatelem při vytváření nebo později některým členem skupiny. Přidání nového agenta se navrhuje zasláním zprávy *REQUEST* s obsahem *AddAgent(groupName, agentName)* správci. Tento typ zprávy používá i agent, který se chce ke skupině připojit. Pokud je skupina demokratická, pak správce vyzve ostatní členy skupiny, aby se vyjádřili ohledně přijmutí nového člena. Tato zpráva má formát *REQUEST* s obsahem *AcceptAgent(groupName, agentName)*. Dotázaní agenti odpovědí zprávou se stejným obsahem a s typem *AGREE* nebo *REFUSE*. Pokud skupina není demokratická, pak je kontaktován pouze její zakladatel, který svým hlasem rozhodne o přijetí agenta. Agent, který zasílal žádost o přijetí nového agenta správci, není zahrnut v procesu hlasování a automaticky se předpokládá s jeho souhlasem. Zda byl agent přidán do skupiny je všem členům sděleno zprávou *INFORM* od správce s identifikací *AddAgentResult(groupName, agentName)*, kde v obsahu je výsledek operace. Agentovi, který správci zasílal požadavek,

je odpovězeno s type *AGREE* nebo *REFUSE*. Zamítnutí požadavku nastává v následujících případech:

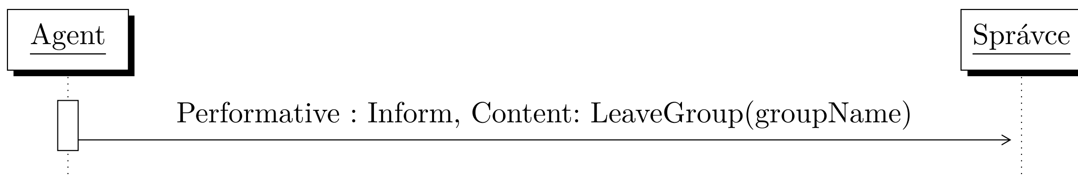
- Skupina *groupName* neexistuje,
- agent *agentName* již je členem skupiny *groupName*,
- agent *agentName* nebyl přijat do skupiny.



Obrázek 4.3: Přidání člena do skupiny

4.5.4 Opuštění skupiny

Každý člen má možnost opustit skupinu. Pokud chce odstoupit zakladatel skupiny, pak je skupina zrušena. Agent informuje správce o svém opuštění zprávou *INFORM* s obsahem *LeaveGroup(groupName)*. Správce oznámí tuto skutečnost členům skupiny zprávou *AgentLeftGroup(groupName, agentName)* typu *INFORM*.



Obrázek 4.4: Opuštění skupiny

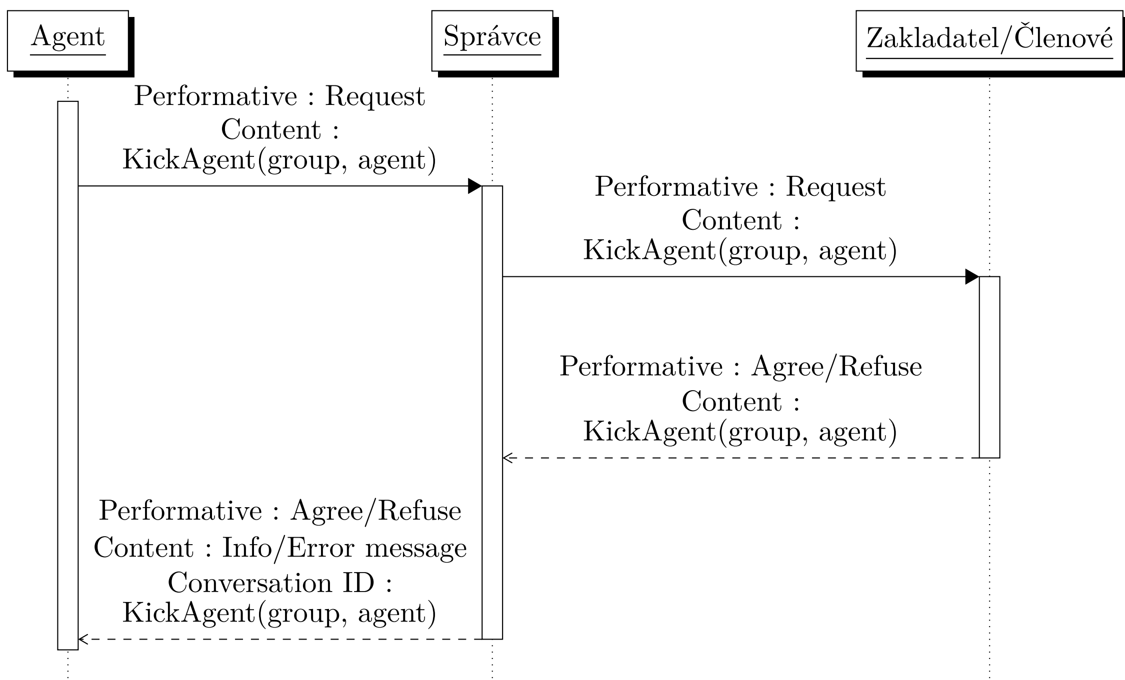
4.5.5 Vyloučení člena ze skupiny

Ze skupiny může být člen také vyloučen. Tato varianta se může použít například, když agent poruší některá ze stanovených pravidel skupiny nebo když nemá dostatečnou reputaci.

Agent1 zasílá správci žádost o vyloučení *Agent2* s obsahem *KickAgentFromGroup(groupName, Agent2)*. Správce po obdržení této zprávy ověří následující fakta:

- Skupina *groupName* existuje,
- agenti *Agent1* a *Agent2* jsou členy skupiny *groupName*,
- agent *Agent2* není zakladatelem skupiny *groupName*,
- *Agent1* a *Agent2* jsou dva rozdílní agenti.

Pokud je některá z těchto podmínek porušena, pak správce pošle zprávu agentovi *Agent1* typu *REFUSE* s obsahem, který popisuje důvod zamítnutí požadavku. V opačném případě proběhne hlasování ohledně vyloučení člena. Princip hlasování je stejný jako v případě schválení nového člena (viz 4.5.3), s tím rozdílem, že obsah posílané zprávy je *KickAgentFromGroup(groupName, Agent2)*. *Agent1* obdrží výsledek operace ve zprávě typu *AGREE* nebo *REFUSE*. U demokratické skupiny zasílá správce všem zúčastněným členům výsledek hlasování. Pokud došlo k vyloučení člena *Agent2* ze skupiny, pak ho o tom informuje správce zprávou typu *INFORM* s obsahem *YouWasKicked(groupName)*.



Obrázek 4.5: Vyloučení člena ze skupiny

4.6 Služby

Při spolupráci a komunikaci agentů se může využívat vyhledávání služeb. Agent vyhledá jiného agenta, kterého pak může požádat zprávou typu *REQUEST* o provedení nabízené služby. Při registraci služby si agent může určit *cenu služby*, která vyjadřuje, za jaké situace agent bude nabízenou službu realizovat. Framework obsahuje následující metody pro správu služeb:

- Zaregistrování a odregistrování služby,
- úprava registrované služby,
- vyhledávání služeb podle parametrů.

Kapitola 5

Implementace frameworku

Framework je implementovaný v jazyce Java. Při vývoji jsem používal integrované vývojové prostředí *NetBeans 8.1* na operačním systému *Windows 10* a knihovnu *JADE 4.3.3*. V kapitole jsou popsány důležité třídy frameworku a příklady jeho použití jsou ukázány v příloze C. Výsledné rozhraní je zobrazeno na konci této kapitoly.

5.1 Základní třídy frameworku

V této podkapitole je popsána třída `IntelligentAgent`, která je nejdůležitější částí frameworku. Dále jsou zde popsány další třídy, díky kterým je implementováno chování z návrhu - například správa představ či skupin.

5.1.1 Třída IntelligentAgent

Když uživatel chce využívat metod rozhraní frameworku, které jsou vypsány v kapitole 5.2, musí jeho třída být potomkem `IntelligentAgent`. Tato třída je základním kamenem frameworku a dědí třídu `Agent` knihovny JADE. `IntelligentAgent` implementuje rozhraní `ICommunication`:

```
public interface ICommunication {
  void ReceiveMsg(ACLMessage msg);
  void ReceiveMsgFromGroup(ACLMessage msg);
  void ReceiveSpecialMsg(ACLMessage msg);
}
```

Toto rozhraní zajišťuje, že třída bude umět reagovat a zpracovávat přijaté zprávy různých typů. `IntelligentAgent` obsahuje následující privátní proměnné:

- `List<Belief> beliefs` - seznam agentových představ,
- `History history` - obsahuje historii komunikace s ostatními agenty,
- `DFAgentDescription dfd` - JADE třída pro registrování, modifikaci a odregistrování služeb,
- `List<SpecialMsg> specialMsgs` - seznam typů zpráv, které si agent registroval k odběru.

5.1.2 Třída Belief

Aktuální stav agenta je popsán množinou jeho představ, kterou využívá při reagování na vjemy z prostředí nebo při výběru akce, kterou provede. Framework nabízí metody pro vytváření, aktualizování, filtrování a mazání představ. Představy jsou implementovány třídou `Belief`, která obsahuje proměnné `String beliefName` a `List<Variable> variables`, jenž popisují jméno a parametry představy. Parametr je vyjádřen dvojicí proměnných jméno a hodnota : `String variableName` a `Object variableValue`. Dvě třídy typu `Variable` jsou stejné, pokud se rovnají jejich jména i hodnoty. Když se rovnají jejich jména a mají obě hodnoty shodného datového typu, pak jsou třídy `Variable` stejného typu. Řetězec `variableName` je využit z důvodu uplatnění metod pro filtrování nebo upravování představ. Představu *bel1* lze aktualizovat na *bel2* jen tehdy, když jména představ (`beliefName`) se rovnají a všechny jejich parametry jsou stejného typu. Příklady na používání metod pro správu představ jsou popsány v příloze C.1.

Filtrování představ

Pomocí třídy `FilterOption`, která implementuje rozhraní `IFilterTerm`, lze filtrovat různé typy představy se zadanými parametry. Rozhraní `IFilterTerm` obsahuje předdefinované základní operace pro filtrování (například porovnávání číselných hodnot, práce s řetězci nebo seznamy apod.), se kterými třída `FilterOption` umí pracovat. Třída také musí definovat metodu `Boolean TermMatches(Belief bel)` z rozhraní, která rozhodne, jestli je filtr pro představu `bel` splněn. Při inicializaci třídy `FilterOption` se specifikuje, pro jaké jméno představy a pro jaký její parametr se bude filtr používat, dále pak operace a parametr pro zvolenou operaci. Konstruktor třídy má tento tvar: `public FilterOption(String beliefName, String variableName, int operation, Object params)`.

Uživatel si jednoduše může vytvořit svojí třídu pro filtrování představ, ve které si může sám nadefinovat složitější operace. Třída pouze musí implementovat rozhraní `IFilterTerm`. Příklad vytvoření vlastní filtrovací třídy je ukázán v příloze C.2, kde třída `MyFilter` obsahuje operaci `int IS_CUSTOMER_RELIABLE`, která pracuje s třídou `Customer` a filtruje spolehlivé zákazníky.

5.1.3 Třída SpecialMsg

Agenti si mohou rezervovat různé typy zpráv, které pak obdrží v metodě `void ReceiveSpecialMsg(ACLMessage msg)`. Agent může specifikovat všechny parametry zprávy v třídě `ACLMessage`, kterou pak předá jako parametr konstruktoru `public SpecialMsg( String agentName, ACLMessage message)`. Když agent obdrží zprávu ve třídě `WaitingForMessages`, která dědí JADE třídu `CyclicBehaviour`, tak je zpráva porovnána se všemi zaregistrovanými typy a podle toho se pak určí, jakou metodou rozhraní `ICommunication` se agentovi zpráva doručí.

5.1.4 Třída AgentGroup

Jak je již popsáno v návrhu správy skupin v 4.5, pro veškerou činnost agentů je zapotřebí *správce* - třída `AgentGroup`. Zprávy pro správce jsou požadavky agentů na nějakou skupinovou akci a na základě jejího obsahu provede správce akci podle některého z protokolů, které jsou popsány v kapitole návrh. Použitím rozhraní `IAgentGroup`, které je znázorněno

v příloze C.3, se třída zavazuje, že bude umět reagovat na všechny navržené požadavky od členů skupiny.

Při využívání metod frameworku může správce během prováděného protokolu komunikovat s ostatními členy skupiny. Výsledky akce se ale předávají jako návratová hodnota metody. Nedochází tedy k přímé komunikaci mezi správcem a agentem, který mu zaslal požadavek. Správce během odpovědí na požadavky zajišťuje, že prováděné operace jsou konzistentní vůči pravidlům pro skupiny. Příklad práce se skupinou je znázorněn v příloze C.5.

5.1.5 Služby

Agenti mohou nabízet ostatním své služby. Framework zajišťuje metody pro zaregistrování a odregistrování služby, její úpravu nebo vyhledávání služby podle jejího jména či vyhledávání všech služeb, které nabízí některý agent. Pro tyto operace není implementovaná žádná nová třída, využívá se metod knihovny JADE se snahou zapouzdřit je tak, aby jejich používání bylo co nejjednodušší.

5.1.6 Ostatní třídy

Následující třídy jsou také využity při práci s frameworkem. Jedná se ale spíše o pomocné nebo méně složitější třídy.

Třída MethodResult

U velké části metod frameworku je jejich výsledek vrácen ve třídě `MetohdResult`, pomocí které informujeme agenta o tom, zda se operace zdařila a případně o jejím výsledku. Vlastnost `bool result` určuje, zda operace proběhla úspěšně. Pří selhání akce má agent k dispozici popis chyby v proměnné `errorMessage`.

```
MethodResult {
  Boolean result;
  public String errorMessage;
  public String infoMessage;
}
```

Třída History

Pro uchovávání všech zpráv, s kterými agent pracoval, se používá třída `History`. Při odesílání a přijetí každé zprávy je zpráva přídána do seznamu `List<ACLMessage> messages`. Framework nabízí funkce pro filtorvání zpráv podle příjemce či odesílatele nebo také podle parametrů zasílané zprávy (například identifikátor komunikace).

Třída Constants

Všechny používané konstanty (například časové limity pro odpovědi nebo parametry využívané při posílání zpráv) jsou definované ve třídě `Constants`.

Třída Helper

Třída `Helper` obsahuje statické pomocné metody, které se využívají při běžných činostech jako například parsování textu, převod datových typů, práce se seznamy, regulární výrazy nebo pomocné výpisy.

Třída Group

Inteligentní agent `AgentGroup` (viz podkapitola 5.1.4) si musí uchovávat přehled o všech vytvořených třídách a členech. V privátní proměnné má proto uloženy všechny existující skupiny - `List<Group> groups` a po ověření pravomocí agenta k provedení operace pracuje pouze s metodami třídy `Group`, které jsou vypsány v příloze C.4.

5.2 Rozhraní frameworku

Kompletní rozhraní frameworku je vypsáno v příloze B.

Kapitola 6

Návrh adventure hry

V kapitole se seznámíme s návrhem adventury, pro jejíž vývoj se použije implementovaný framework. Důležitý byl výběr hry, na které by se dala dobře prokázat funkčnost a použitelnost frameworku. Snažil jsem se zvolit takovou hru, ve které by vystupovalo několik inteligentních postav, které by spolu komunikovaly, spolupracovaly a sdružovaly se do skupin, ve kterých by sdílely znalosti. Tyto postavy budou reprezentovány agenty, kteří budou využívat funkcí frameworku. Z těchto důvodů jsem se rozhodl inspirovat americkou reality show **Survivor**. Pravidla této hry jsou popsány na začátku této kapitoly. Následující kapitoly popisují význam jednotlivých akcí, které může hráč během hry vykonávat. Je zde i popsána klíčová role arbitra, který celou hru řídí, a navrženy protokoly pro zprávy, které bude arbitr používat při komunikaci s hráči. Spolu s ukázkou některých protokolů je i znázorněn arbitrův životní cyklus. Na konci kapitoly je hrubě navrženo grafické uživatelské rozhraní, přes které se bude hra ovládat.

6.1 Představení reality show

Survivor, v české verzi známá pod názvem **Kdo přežije**, je velmi oblíbená reality show. Nejznámější verze je vysílaná v USA, kde se již odehrálo 31 řad. Vítěz vždy kromě velké popularity získává i finanční odměnu v hodnotě milionu dolarů. Cílem soutěže je přežít 39 dní na zvoleném ostrově. Soutěžící si však sami musejí postavit přístřešek, rozdělat oheň nebo si nějakým způsobem obstarat jídlo. Běžnými prvky této hry jsou hádky, zrady, dohody, zranění, hlad či dehydratace.

V soutěži Survivor je mnoho prvků, které mohou jistým způsobem připomínat chování multiagentního systému. Už od začátku se hráči shromažďují do menších skupin, kde se domlouvají a provádějí skupinová rozhodnutí. Často řeší, koho celý tým zvolí na hlasování. Jednotliví soutěžící samozřejmě mohou skupinu zradit nebo spolupracovat s jinou, to už záleží pouze na nich a na jejich zvolené strategii. Detailnější popis hry a její pravidla jsou k nalezení na webových stránkách soutěže[1].

6.2 Pravidla hry

Pravidla pro implementovanou adventure hru jsou lehce zjednodušena oproti originálu, ale základní kostra je zachována. Ve hře začíná 10 soutěžících, z nichž jeden bude hlavní hrdina, za kterého se bude hrát. Každý den budou hráči plnit nějakou soutěž a výherce

[1] http://www.kdoprezije.cz/o_soutezi.php

získá imunitu. Imunita chrání hráče před vyloučením ze soutěže při večerním hlasování. Každý večer se totiž pořádá schůze všech soutěžících, na které každý hráč hlasuje pro jednoho protihráče, který by měl hru opustit. Každý hráč hlasuje pouze jednou a právě pro jednoho hráče. Nesmí však hlasovat pro soutěžícího, který získal imunitu. Hráč s nejvyšším počtem hlasů musí hru opustit. Hlasování probíhá anonymně, hráči tedy vědí, kdo dostal kolik hlasů, ale netuší od koho. V případě remízy při hlasování si výherce imunity vybere mezi soutěžícími s nejvyšším počtem hlasů toho, kdo soutěž opustí. V reality show se o imunitu soutěží v dovednostních soutěžích, v implementované hře se rozhodne o výherci náhodně mezi všemi hráči. Všichni hráči jsou o výherci informováni a vědí tak dopředu, koho nebudou moci při hlasování volit. Součástí hry je také skrytý symbol imunity, který lze využít pouze bezprostředně po hlasování. Tento symbol způsobí, že hlasy pro hráče, který použil skrytý symbol, nebudou započítány. Používání této imunity patří vždy k největším zápletkám a zvratům reality show. Pokud se všichni hráči dohodnou na vyřazení soutěžícího, který vlastní skrytý symbol imunity a použije ho, pak o soutěžícím, který vypadne může rozhodnout pouze jeden hlas - od vlastníka imunity. Tento symbol může být použit v kterémkoliv hlasování a je reprezentován malým přívěskem, který hráč může klidně někomu darovat (například svému příteli, který je před hlasováním v ohrožení). Skrytý symbol imunity je důkladně schován někde na ostrově. K jeho nalezení slouží takzvaná indicie, která upřesňuje jeho polohu. Indicie si může hráč všimnout při společných hostinách, většinou bývá ukryta někde mezi nabízenými potravinami. Získaná indicie může hráčovi výrazně pomoci při nalezení skrytého symbolu imunity. Bez ní má hráč velmi malou šanci na nalezení imunity. Pokud hráč symbol imunity při hlasování použije, pak může být symbol během následujících dnů ukryt někde na ostrově a může být znovu využit libovolným hráčem.

Na závěr každého dne se provádí již zmíněné hlasování a soutěž opouští vždy jeden hráč - ten, který dostal nejvíce hlasů. V momentě, kdy zůstanou pouze tři soutěžící, proběhne hlasování o tom, který z nich soutěž vyhraje. Hlasování se účastní všech sedm vypadlých hráčů, proto je strategicky výhodné si při hře neudělat moc nepřátel. V případě remízy probíhá hlasování znovu mezi dvěma nejlepšími.

6.3 Dostupné akce

Každý hráč má k dispozici několik druhů akcí, kterými může komunikovat, informovat nebo spolupracovat s ostatními agenty:

- Informovat

  Hráč může oznámit zvolenému soutěžícímu koho bude volit nebo volil v hlasování, kdo je jeho přítel či nepřítel, kdo má imunitu nebo indicii k imunitě nebo mu může ukázat (pokud ho vlastní) symbol skryté imunity či indicii k jejímu nalezení.

- Dotázat se

  Hráč se může vybraného soutěžícího zeptat, koho bude volit nebo volil v hlasování, kdo je jeho přítel či nepřítel, co si myslí o konkrétním hráči (zda je jeho přítel nebo nepřítel), kdo má imunitu nebo indicii k imunitě nebo ho může vyzvat k ukázání symbolu skryté imunity či indicie.

- Skupinové akce

  Hráč může založit skupinu s vybranými soutěžícími.

Pokud je hráč členem skupiny, může všechny členy skupiny informovat nebo se dotázat (popsáno výše), opustit skupinu nebo přidat či vyřadit člena ze skupiny.

- Hledat symbol skryté imunity

  Hráč se pokusí o nalezení symbolu skryté imunity.

- Akce se symbolem imunity

  Hráč může požádat vybraného soutěžícího o darování symbolu skryté imunity nebo může symbol, pokud ho vlastní, darovat vybranému hráči.

- Komunikovat

  Hráč si jde nezávazně popovídat s vybraným soutěžícím.

6.4 Agent arbitr

Arbitr je nezbytným agentem v systému, má na starosti veškerou organizační komunikaci se všemi soutěžícími. Soutěžící vyzývá k provedení akce, k zareagování na danou událost nebo je informuje o výsledcích hlasování, zvolené operace nebo o dalších herních událostech. Jako jediný má přehled o všech zvolených akcích soutěžících, zná volby jednotlivých hráčů při hlasováním a udržuje si přehled o tom, kdo vlastní imunitu nebo indicii k jejímu nalezení. Jeho hlavním úkolem je zajištění plynulosti hry, vyzývání soutěžících k akcím a reakcím či informování hráčů o událostech, které se jich týkají.

6.4.1 Hlavní cyklus arbitra

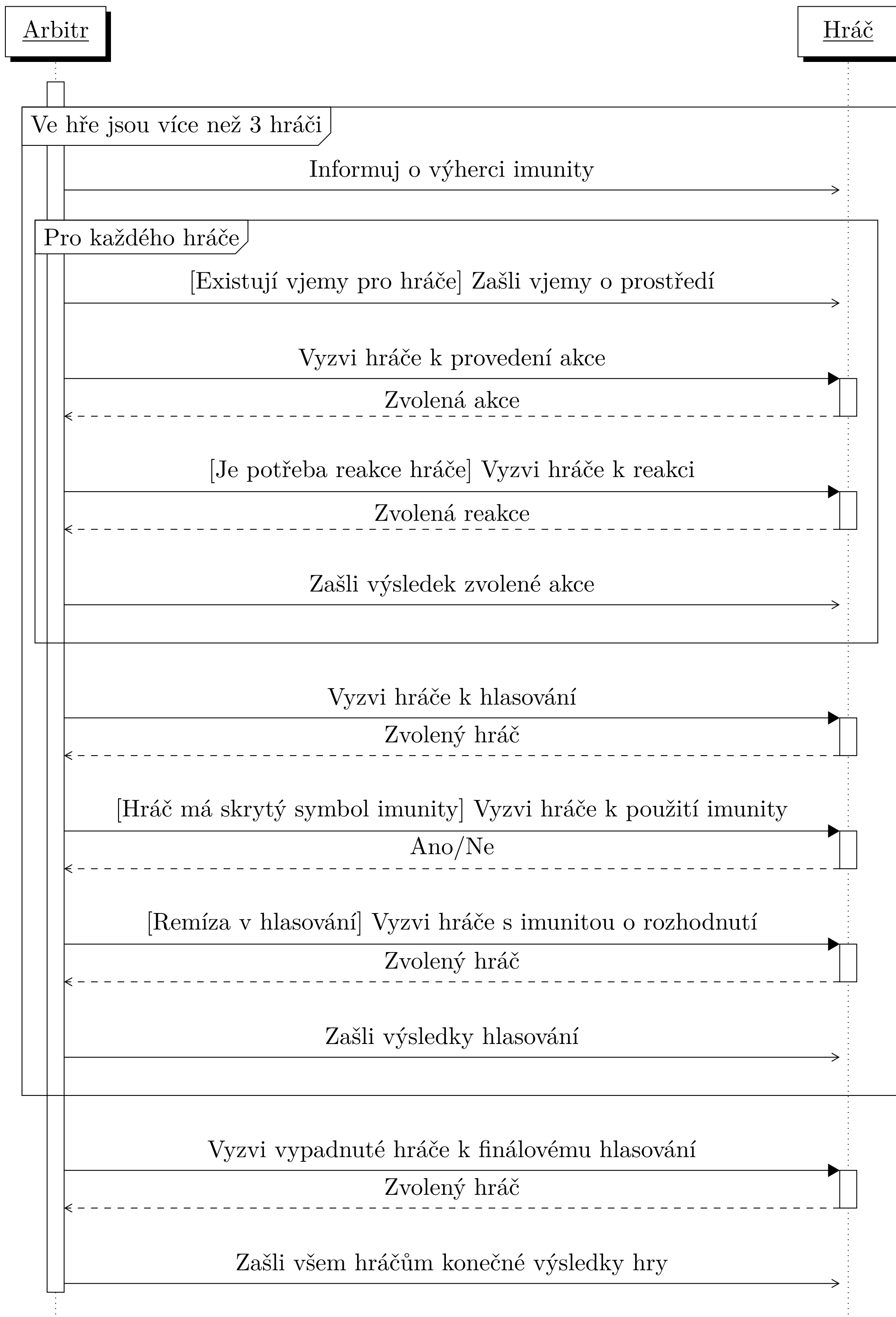
Životní cyklus agenta znázorňuje sekvenční diagram 6.1, detailnější popisy komunikačních protokolů mezi arbitrem a soutěžícími jsou popsány v následujících podkapitolách. Arbitr řídí průběh hry posíláním zpráv jednotlivým agentům podle popsaných protokolů níže. Při komunikaci mezi arbitrem a hráči je kladen důraz na dodržování standardu FIPA. Nejdůležitějšími prvky zprávy je její identifikátor komunikace (*conversation ID*), komunikační akt (viz 3.2) a samotný obsah zasílané zprávy. Podle identifikátoru komunikace lze zjistit, čeho se zpráva týká a v jaké fázi hry se nacházíme. Každá zpráva podle standardu FIPA musí mít zvolený komunikační akt, který určuje typ zprávy. V samotném obsahu zprávy je funkce s parametry, na kterou má agent zareagovat nebo ji jiným způsobem zpracovat.

Po spuštění hry a vytvoření všech deseti soutěžících zašle arbitr všem soutěžícím zprávu, která zahajuje hru. Zpráva má identifikátor *GameStarted* a v jejím obsahu jsou jména všech soutěžících. Po přijetí této zprávy zná každý agent své protihráče. Na začátku každého dne arbitr zašle zprávu všem hráčům o tom, kdo vyhrál imunitu a nebude moci být na večerním hlasování volen. Poté arbitr postupně vyzývá všechny hráče ke zvolení akce, díky které mohou hru nějakým způsobem ovlivnit. Ještě před požadavkem na výběr akce zasílá arbitr hráčovi vjemy (detailněji popsáno v 6.8), které odpovídají agentovu vnímání prostředí svými senzory. V rámci této hry to znamená situaci, kdy hráč vidí nebo slyší aktivity ostatních hráčů. Tyto vjemy jsou zasílány arbitrem z toho důvodu, že jako jediný zná akce ostatních hráčů a nedochází tak k porušení konzistence. O tom, které dostupné vjemy se hráči zašlou, se rozhoduje pomocí pravděpodobnostního rozložení a náhodného generování mezi existujícími vjemy. Díky získaným vjemům (například kdo se s kým přátelí) může hráč přehodnocovat představy a ovlivnit své budoucí chování.

Poté arbitr vyzývá hráče ke zvolení některé z dostupných akcí. Ve zprávě, kterou hráč zaslal arbitrovi, je specifikováno jakou akci a s jakými parametry chce hráč provést. Po obdržení vybrané akce arbitr ověří hráčovo oprávnění. Například hráč si nemůže zvolit akci ukázání symbolu skryté imunity, když tento symbol nevlastní. V takovém případě arbitr informuje hráče o důvodu neplatnosti této akce a hráč přichází o možnost provedení akce v tomto kole. Tento případ by ale neměl nastat, protože hráč zná podmínky pro vykonání jednotlivých akcí a v rámci jeho vlastního zájmu by si měl vybírat pouze platné akce. U některých typů akcí (například dotazovací akce) je potřeba vyjádření ostatních hráčů. Z parametrů zprávy arbitr odvodí, od jakých hráčů se vyžaduje reakce a zašle jim zprávu s požadavkem na reakci. Po obdržení všech potřebných odpovědí zasílá původnímu hráči výsledek této akce. Detailnější popis protokolu pro výběr akce, její provedení a vyhodnocení je popsán v podkapitole 6.6. Až všichni soutěžící provedou určitý počet akcí během jednoho dne, pak se přejde k hlasování o tom, kdo soutěž opustí.

Návrh protokolu o hlasování je diskutován v podkapitole 6.10. Všem hrajícím soutěžícím se zašle zpráva s výzvou ke hlasování, v jejímž obsahu budou specifikována jména soutěžících, kteří se mohou volit. Již vypadlí hráči nebo hráč s imunitou v této nabídce nejsou a nikdo pro ně tak nemůže hlasovat. Každý hráč zvolí toho z nabídky, kterého chce ze hry vyřadit. Arbitr vyhodnotí obdržené hlasy a v případě, že nějaký hráč má symbol skryté imunity, dotáže se ho, zda chce symbol na toto hlasování využít. Poté co arbitr obdrží hlasy od všech soutěžících, tak výsledky vyhodnotí a v případě, že více hráčů dostalo nejvyšší počet hlasů, tak o poraženém rozhodne výherce imunity. Následně arbitr hráčům zašle informace o tom, kdo byl vyřazen ze hry a jaký byl celkový výsledek hlasování, tj. kolik hlasů obdržel každý hráč. Tento proces volby akce a hlasování o tom, kdo hru opustí, se opakuje dokud ve hře nezůstanou tři hráči.

O vítězi celé hry se rozhodne ve finálovém hlasování, kterého se účastní již vypadlí soutěžící. Arbitr tyto hráče vyzve k hlasování a každý z nich si zvolí jednoho ze tří finalistů, který by měl soutěž vyhrát. Arbitr pak sečte hlasy všech sedmi hráčů a v případě remízy je vyzve znovu o hlasování, tentokrát pouze mezi dvěma nejlepšími hráči, kteří v první fázi obdrželi nejvyšší počet hlasů. Pak již je výherce znám a arbitr informuje všechny zúčastněné hráče soutěže o výsledcích finálového hlasování. Nakonec ještě arbitr zašle všem hráčům zprávu o celkovém pořadí v soutěži a poté hra končí.



Obrázek 6.1: Hlavní cyklus arbitra

6.5 Agent hráč

Akce a reakce, které hráč vykonává, rozhodují o jeho úspěchu či neúspěchu v soutěži. Každý hráč si vytváří, uchovává a modifikuje svoji množinu představ, na základě které provádí veškeré akce. Pro správu představ se využije implementovaný framework, který je detailněji popsán v kapitole 4.4. Nejdůležitější představy jsou popsány níže v této kapitole. Každý hráč musí ve hře reagovat na tyto hlavní typy zpráv :

- Výzva k vykonání akce,
- výzva k zareagování na určitou situaci,
- výzva k hlasování,
- přijetí nových informací od ostatních hráčů,
- přijetí nových informací o prostředí.

6.5.1 Důvěra k hráčům

Nejdůležitější představou je představa *Trust*, která obsahuje jako parametry jméno protihráče a hodnotu důvěry, kterou k němu hráč má. Při začátku hry si hráč pro každého soutěžícího vytvoří tuto představu s jeho jménem a hodnotu důvěry inicializuje na 50. Hodnota důvěry se může pohybovat mezi hodnotami 0-100, kde 100 odpovídá maximální důvěře k danému hráči. Hráč se pak zejména podle důvěry rozhoduje o tom, jaké akce si bude volit a jakých hráčů se budou týkat, jak bude reagovat na výzvy jiného hráče nebo koho bude volit při hlasování. Podle akcích nebo reakcích ostatních hráčů si průběžně tyto představy modifikuje.

6.5.2 Historie hlasování

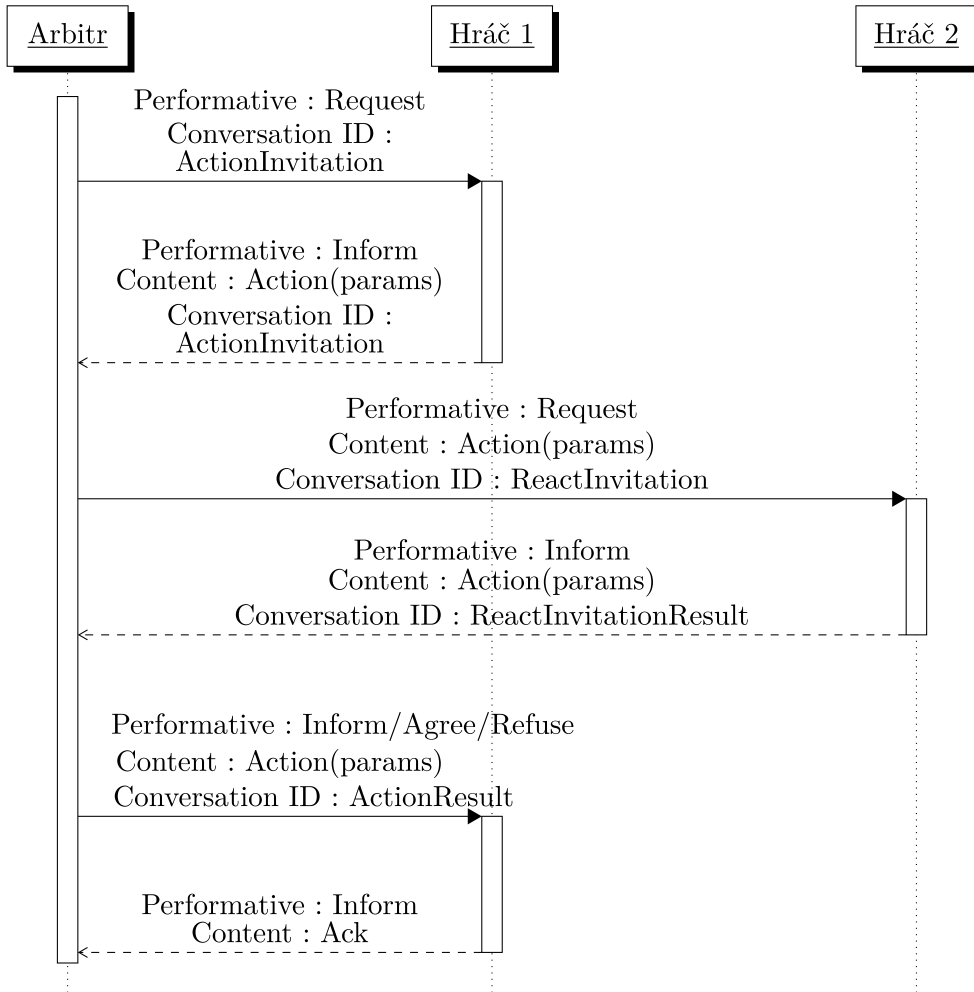
Po každém hlasování arbitr zasílá všem zúčastněným výsledky. Hráč si pro každý výsledek vytvoří představu *Voting*, kde si zaznamená o kolikáté hlasování se jednalo, kolik kdo dostal hlasů a pro koho sám hráč volil. Tato představa je využívána, když se hráče někdo ptá, pro koho hlasoval. Hráč také na základě jednotlivých výsledků hlasování může odvodit podle počtu obdržených hlasů svoji oblíbenost či neoblíbenost. Toto vyhodnocení může sehrát klíčovou roli při rozhodování, v jakém hlasování má hráč použít symbol skryté imunity.

6.5.3 Vlastník imunity a indicie

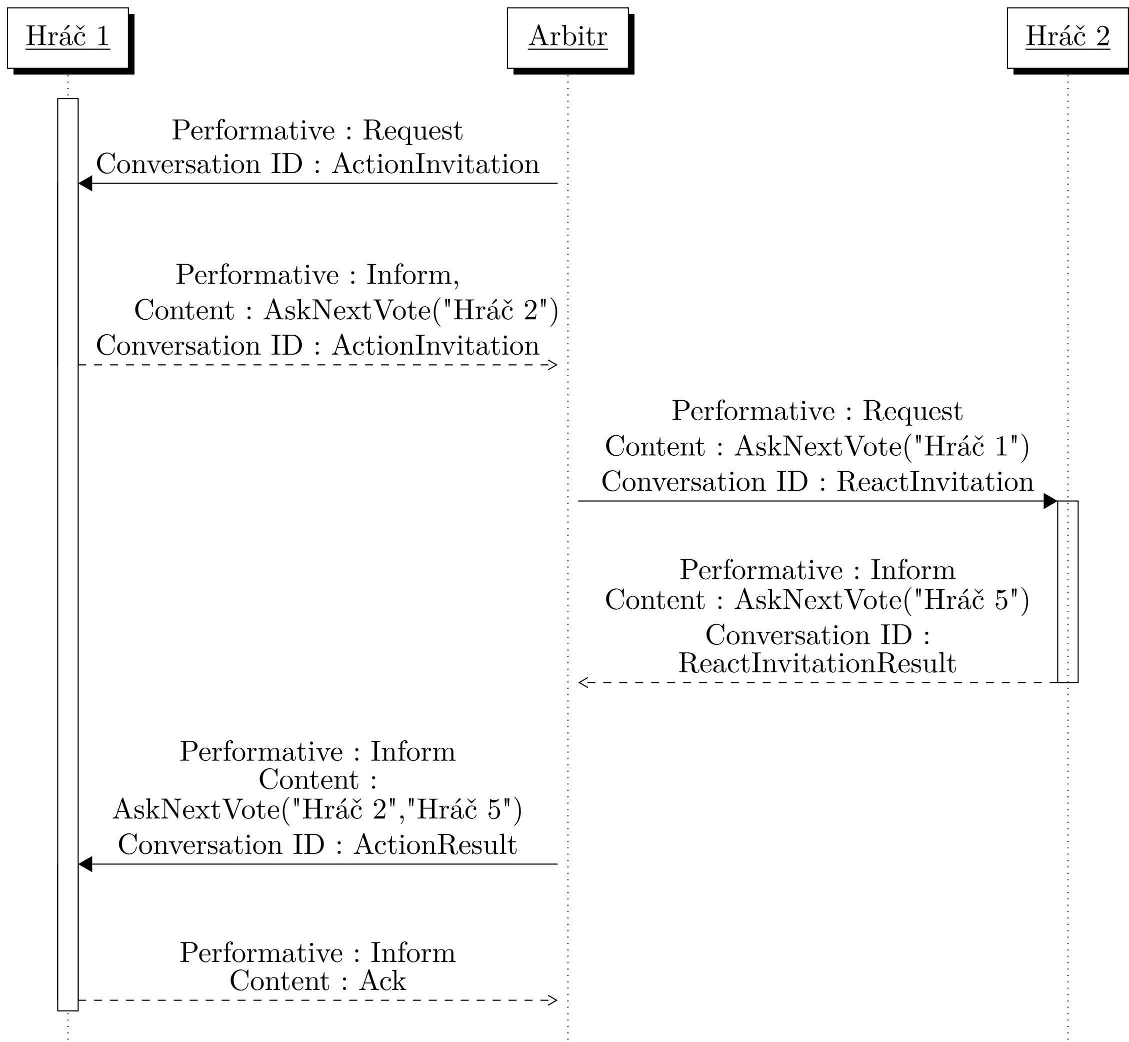
Protože hráč může buď vidět, jak si někdo bere indicii k nalezení symbolu skryté imunity, nebo ho ostatní spoluhráči mohou informovat, kdo je vlastníkem imunity či indicii, tak je vhodné si tyto informace udržovat mezi svými představami. Představy *ImmunityOwner* a *ClueOwner* obsahují jména soutěžících, kteří hráči ukázali symbol imunity nebo indicii. Pokud je hráč informován od jiného o tom, kdo je vlastník indicie nebo imunity, pak si tento fakt uloží do představ *HasClue* nebo *HasImmunity*. Tyto představy hráč využívá buď při dotazech od jiných hráčů nebo v případě, že chce od někoho získat indicii nebo symbol imunity.

6.6 Požadavek na vykonání akce

Agent svými zvolenými akcemi ovlivňuje představy ostatních agentů ve hře. Volba akcí soutěžícího má nejvyšší vliv na jeho celkové umístění ve hře. Zprávu typu *Request* s identifikátorem komunikace *ActionInvitation* zašle arbitr hráči, od kterého je požadován výběr dostupné akce z 6.3 . Hráč na požadavek zareaguje posláním zprávy se stejným identifikátorem a s obsahem, ve kterém je popsána akce i s jejími parametry. Pokud je pro získání výsledku akce potřeba spolupráce s jiným hráčem, pak mu arbitr zašle zprávu s identifikátorem *ReactInvitation* s popisem, na jakou situaci má hráč reagovat. Reakci pošle hráč arbitrovi s identifikátorem *ReactInvitationResult* a arbitr z této odpovědi vytvoří zprávu s výsledkem pro hráče, který si vybíral, jakou akci provede. Tato zpráva má identifikátor *ActionResult* a v jejím obsahu je popsáno, o jakou akci se jedná a jaký je její výsledek. U některých akcí je výsledek popsán v typu zprávy - *Agree* nebo *Refuse*. Jedná se zejména o akce, které mohou být buď přijaty nebo odmítnuty (například akce komunikace s vybraným hráčem může být buď přijata nebo odmítnuta). Sekvenční diagram výzvy pro hráče k vykonání akce je ukázán na obrázku 6.2, obrázek 6.3 pak ukazuje konkrétní případ, kdy si *Hráč 1* zvolí jako svou akci dotázání se *Hráče 2* na to, koho bude volit v příštím hlasování. Na příkladě je zobrazeno, že *Hráč 2* odpověděl, že v dalším kole bude volit *Hráče 5*. Soutěžící není nijak zavázán odpovídat na veškeré otázky a může hráči sdělit, že mu na otázku neodpoví - v příkladě by to znamenalo zaslání zprávy s obsahem *AskNextVote()* na vyžádání reakce od arbitra.



Obrázek 6.2: Sekvenční diagram pro volbu akce



Obrázek 6.3: Sekvenční diagram pro akci dotázání na následující hlasování

Tabulka 6.1 pak obsahuje výčet všech zpráv, kterými hráč může zareagovat na zprávu s identifikátorem *ActionInvitation*. V prvním sloupci je obsah zprávy, kterou hráč odpovídá arbitrovi. V druhém a třetím sloupci je typ a obsah zprávy, ve kterém arbitr zasílá hráčovi výsledek operace. Jsou zde znázorněny všechny možné výsledky akcí. Tato zpráva má vždy identifikátor *ActionResult* a výsledek provedené akce se dá vyčíst buď z jejího obsahu nebo z typu zprávy. Výsledek akcí, které lze buď přijmout nebo odmítnout, se předává jako typ zprávy *Agree* nebo *Refuse* a její obsah je stejný, jako obsah zprávy, kterou odeslal hráč arbitrovi. U ostatních zpráv je výsledek provedené akce popsán v jejich obsahu.

U akce *Talk("player2")* si hráč chce promluvit s *player2*. Arbitr mu zašle požadavek typu *ReactInvitation* a podle toho, zda si *player2* chce popovídat nebo ne, nastaví typ zprávy na *Agree* nebo *Refuse*. Provedením *ShowImmunity* nebo *ShowClue* chce hráč ukázat svůj symbol skryté imunity nebo indicii k jejímu nalezení hráči *player2*. V případě, že hráč imunitu nebo indicii nemá, tak mu arbitr tuto akci neumožní a vrací mu jako výsledek akce *Refuse*. Jinak je *player2* informován o tom, že mu hráč symbol nebo indicii ukázal. Podobně to je i u akce *GiveImmunity*, kdy hráč daruje symbol skryté imunity. Akce *INFORM* může být nahrazena některou z akcí : *MyNextVote*, *MyLastVote*, *MyFriend*, *MyEnemy*, *HasClue*

nebo *HasImmunity*. Všechny tyto akce mají stejnou syntaxi, liší se pouze významem. Cílem akce je oznámit hráči *player2*, že *player5* bude mojí další nebo byl mojí poslední volbou při hlasování, je můj přítel či nepřítel nebo si myslím, že má indicii nebo skrytý symbol imunity. Tento typ akcí nepřináší hráčovi žádné výsledky nebo nové znalosti, je vykonávána pouze z důvodu informovat jiného hráče. Pokud hráč chce získat na některou z těchto otázek odpověď, může využít akce *ASK("player2")*, kde *ASK* může nabývat hodnot *AskNextVote*, *AskLastVote*, *AskFriend*, *AskEnemy*, *AskImmunityOwner* nebo *AskClueOwner*. Odpověď od hráče *player2* je obsažena ve výsledku v druhém parametru. Pokud tento parametr v obsahu výsledné zprávy není, pak to znamená, že *player2* nechce na otázku odpovědět nebo nezná její odpověď. Podobně lze také vyzvat hráče *player2* k ukázání imunity nebo indicii - *AskShowImmunity* a *AskShowClue*, případně ho můžeme požádat k darování imunity - *RequestForImmunity*. Když chce hráč zjistit, co si *p2* myslí o hráči *p5*, pak v odpovědi na *AskAbout("p2","p5")* lze vyčíst, zda je to jeho přítel nebo nepřítel. Zvolenou akcí *FindImmunity* se hráč pokouší hledat symbol skryté imunity. Arbitr ho pak pomocí typu zprávy informuje o tom, zda symbol našel. Pokud hráč vlastní indicii, pak je daleko větší pravděpodobnost nalezení symbolu.

U skupinových akcí je protokol výběru akce odlišný. Celá akce se provede nejprve v rámci skupiny a až poté se informuje arbitr, který celou akci pouze potvrdí. Pokud si hráč chce vybrat akci vytvoření skupiny, pak skupinu založí a arbitrovi pošle pouze zprávu *CreatedGroup(groupName,"p2;p3;p5")*, která ho informuje o tom, jaká skupina a s jakými členy byla založena. Hráč je poté zakladatel skupiny a rozhoduje o případném přijetí nebo odebrání členů ze skupiny. Každý člen skupiny může akcí *GroupAddPlayer* nebo *GroupKickPlayer* zaslat žádost o rozhodnutí na zakladatele skupiny, nebo může skupinu akcí *GroupLeave* opustit. Arbitr není členem skupiny, ale má informace o členech skupiny a o všech akcích, které se ve skupině provádí. Akce *INFORM* i *ASK*, které jsou popsány v tabulce 6.1 lze využít i ve skupině a informovat nebo se dotázat více členů současně. Když se hráč chce zeptat celé skupiny, pak zašle všem členům zprávu s identifikátorem *ReactInvitation* a s obsahem, který popisuje zvolenou otázku. Každý člen mu odpoví a hráč pak zašle všem členům skupiny všechny přijaté odpovědi včetně své volby ve zprávě s identifikátorem *GroupActionResult*. Tímto je zajištěno, že všechny odpovědi jsou všemi členy sdíleny. Hráč pak pouze informuje arbitra, že provedl akci *GroupAsk* a v parametrech upřesní o jakou skupinu se jednalo a jak členové odpověděli. U akcí *GroupAskShowImmunity* nebo *GroupAskShowClue* získá hráč odpovědi od všech členů, které zašle arbitrovi a ten ověří, že pokud někdo odpověděl kladně, pak symbol nebo indicii musí vlastnit. Příklad zvolení akce *GroupAskNextVote* u skupiny *myGroup*, která obsahuje členy *Hráč 1*, *Hráč 2* a *Hráč 4* je znázorněna na obrázku 6.4. V zobrazeném příkladě *Hráč 2* neodpověděl na otázku, koho bude volit. Ostatní budou hlasovat pro soutěžícího *Hráč 5*.

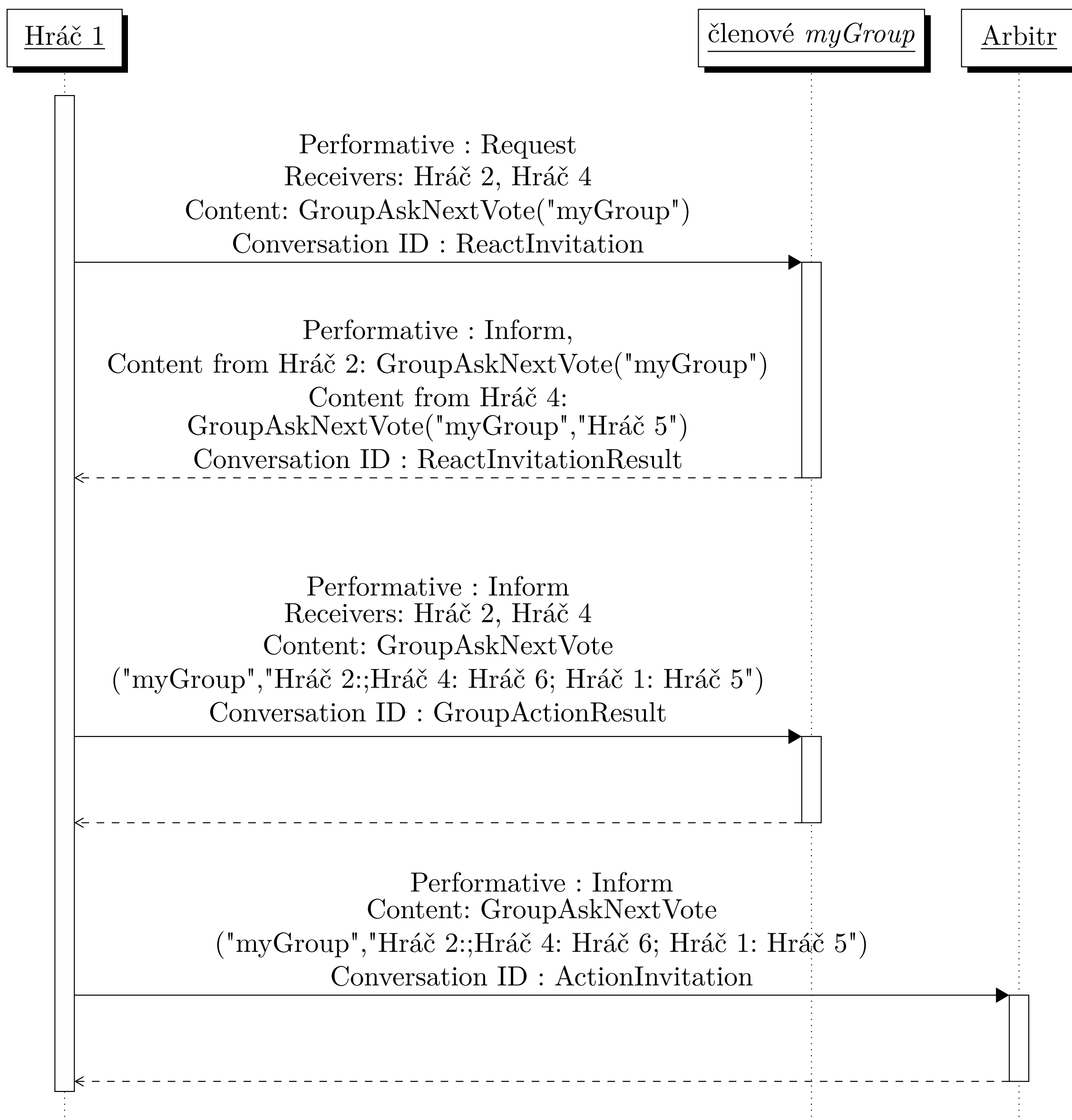
Akce	Typ zprávy	Obsah zprávy
Talk("player2")	Agree/Refuse	Talk("player2")
ShowImmunity("player2")	Agree/Refuse	ShowImmunity("player2")
ShowClue("player2")	Agree/Refuse	ShowClue("player2")
INFORM("player2","player5")	Inform	INFORM("player2","player5")
AskAbout("p2","p5")	Inform	AskAbout("p2","p5","Enemy") AskAbout("p2","p5","Friend") AskAbout("p2","p5","Neutral")
ASK("player2")	Inform	ASK("player2","player5") ASK("player2")
AskShowImmunity("player2")	Agree/Refuse	AskShowImmunity("player2")
AskShowClue("player2")	Agree/Refuse	AskShowClue("player2")
FindImmunity()	Agree/Refuse	FindImmunity()
GiveImmunity("player2")	Agree/Refuse	GiveImmunity("player2")
RequestForImmunity("player2")	Agree/Refuse	RequestForImmunity("player2")

Tabulka 6.1: Možnosti výběru akce

6.7 Požadavek na reakci

Zprávy, u kterých se očekává reakce od hráče, mají identifikátor *ReactInvitation*. Hráč na tento typ zpráv musí vždy odpovědět s identifikátorem *ReactInvitationResult*. Reakce souhlasu či nesouhlasu hráč vyjádří nastavením typu zprávy (*Agree* nebo *Refuse*) u své odpovědi. Pokud se po hráči chce detailnější odpověď, pak jí předá v obsahu zprávy. Reakce od hráče se očekává při :

- Jiný hráč si zvolil akci, která se týká hráče a má na ní nějakým způsobem zareagovat,
- arbitr se dotazuje hráče, který má skrytý symbol imunity, zda ho chce využít po proběhlém hlasování,
- arbitr oznamuje hráči, že uviděl indicii k symbolu skryté imunity a žádá ho o reakci, zda si ho chce tajně vzít.



Obrázek 6.4: Sekvenční diagram pro akci skupinové dotázání na následující hlasování

Tabulka 6.2 ukazuje všechny typy zpráv s identifikátorem *ReactInvitation*, na které hráč může reagovat. Jejich obsah je v prvním sloupci, druhý a třetí sloupec znázorňuje typ a obsah hráčovi reakce. Zpráva s odpovědí má vždy identifikátor *ReactInvitationResult* a je odesílána arbitrovi. Pouze v případech, kdy se jedná o skupinové akce, je zpráva zaslána hráči, který požadavek na reakci zaslal. Skupinové zprávy nejsou v tabulce znázorněny, reakce jsou ale stejné jako u klasických akcí. Příklad skupinové akce je znázorněn na obrázku 6.4. Na zprávy, které obsahují *Talk*, *AskShowImmunity*, *AskShowClue*, *GroupAskShowImmunity*, *GroupAskShowClue*, *RequestForImmunity*, *NoticedClue* nebo *UseImmunityRequest* hráč reaguje pouze nastavením typu zprávy s odpovědí na *Agree* nebo *Refuse*. U akce *NoticedClue* se hráč musí rozhodnout, zda chce vzít indicii k nalezení skrytého symbolu imunity. Při této akci je riziko, že ho ostatní hráči uvidí, což se pak v nějakých případech může obrátit proti němu. Například když si hráč všimne, jak někdo bere indicii, tak se ho může

zeptat na to, kdo je jejím vlastníkem. Hráč pak z jeho odpovědi může poznat, že lže a změnit k němu svoji důvěru. Stejně jako v tabulce 6.1, tak i tady akce *ASK* může nabývat stejných hodnot. Například na výzvu *AskFriend("player2")*, kdy se hráč *player2* dotazuje, kdo je hráčův přítel, může odpovědět *AskFriend("player5")*, čímž mu sdělí, s kým se nejvíce přátelí. Pokud nechce hráči *player2* odpovědět, pak mu zašle zprávu s obsahem *AskFriend()*. Pokud má hráč symbol skryté imunity a chce ho při hlasování využít, pak odsouhlasí výzvu *UseImmunityRequest*. Zprávou s obsahem *AskAbout("player2","player5")* se *player2* dotazuje, co si hráč myslí o *player5*. Hráč ve své odpovědi sdělí, zda je s *player5* přítel či nepřítel.

Obsah	**Typ zprávy reakce**	**Obsah zprávy reakce**
Talk("player2")	Agree/Refuse	Talk("player2")
ASK("player2")	Inform	ASK("player5")/ASK()
AskShowImmunity("p2")	Agree/Refuse	AskShowImmunity("p2")
RequestForImmunity("p2)	Agree/Refuse	RequestForImmunity("p2")
NoticedClue()	Agree/Refuse	NoticedClue()
UseImmunityRequest()	Agree/Refuse	UseImmunityRequest()
AskAbout("player2","player5")	Inform	Enemy/Friend/Neutral

Tabulka 6.2: Možnosti zareagování

6.8 Vnímání prostředí

Jak již bylo zmíněno, agenti svými akcemi působí na své prostředí a ovlivňují ho. Každý agent pak svými senzory vnímá stav prostředí a volí podle toho i své akce. Vnímání hráčova prostředí je zajištěno zasíláním zpráv s identifikátorem *PerceptInvitation*. Tyto zprávy zasílá hráčovi arbitr, který zná akce ostatních hráčů a proto může agenta informovat o tom, co se kolem něho děje. Samozřejmě arbitr nebude zasílat hráči všechny informace o tom, co se v jeho prostředí děje, ale tato volba se bude provádět náhodným výběrem s určitou pravděpodobností. Hráč arbitrovi vždy potvrdí přijetí informace o prostředí zaslání zprávy typu *Inform* s obsahem *Ack*. Situace, o kterých arbitr informuje hráče jsou:

- Hráč uviděl jiného soutěžícího, jak si bere skrytý symbol imunity,
- hráč obdržel informaci o tom, kdo vyhrál imunitu a nebude moci při dalším hlasování vypadnout ze soutěže,
- hráč uviděl, jak se spolu baví dva protihráči,
- hráč uviděl, jak se spolu baví několik protihráčů ve skupině.

6.9 Zpracování informací

Kromě vnímání prostředí je hráč také informován podle akcí, kteří si ostatní hráči zvolili. Tyto zprávy jsou typu *Inform* a obsahují identifikátor *InformationReceived*. V tabulce možnosti výběru z dostupných akcí (viz 6.1) tomu například odpovídá akce *INFORM("player2","player5")*. V případě, že se protihráč rozhodne provést tento typ akce, tak může hráč získat informace o tom, koho jeho protihráč volil nebo bude volit, s kým se přátelí nebo kdo je jeho nepřítel nebo o kom si myslí, že má symbol skryté imunity nebo indicii k jejímu nalezení. Informována také může být celá skupina, v níž je hráč členem. Pokud se někdo rozhodne hráči či skupině ukázat symbol skryté imunity nebo indicii, pak zpracování a použití této informace patří mezi velmi důležité aspekty hráčovy strategie. Rozdíl mezi ukázáním symbolu imunity (*ShowImmunity*) a obdržením informace, že někdo má imunitu (*HasImmunity*) je v tom, že v druhém případě to nemusí být pravda a může se jednat pouze o blafování nebo zmatení hráče. Pokud je symbol hráči ukázán, pak si může být jistý, kdo opravdu symbol má. Když hráči někdo ukáže indicii k nalezení imunity, pak i on má daleko větší šanci na nalezení symbolu imunity a může také ukazovat indicii ostatním členům.

Hráč je také informován o tom, co se děje ve skupinách, do kterých patří. Hráč odebírá informace o přijetí nebo nepřijetí nových členů do skupiny nebo vyřazení člena ze skupiny nebo o tom, že někdo dobrovolně skupinu opustil. Upravování představ po získání těchto typů informací může sehrát velkou roli v celé hře.

6.10 Proces hlasování

Poslední variantou zprávy, které agent při hře dostává, jsou výzvy ke hlasování. Každý den se hlasuje o tom, kdo soutěž opustí. Hráči jsou vyzváni arbitrem zprávou s identifikátorem *VotingRequest*, kde jména hráčů, pro které lze hlasovat jsou zmíněny v obsahu zprávy. Hráč arbitrovi odpovídá zprávou s identifikátorem *Voting*, v jejímž obsahu je jméno hráče, který by měl být ze soutěže vyřazen. Pokud některý z hráčů má skrytý symbol imunity, pak se ho arbitr dotáže zprávou s identifikátorem *UseImmunityRequest*. Výsledky hlasování arbitr rozešle všem hráčům, kteří se účastnili hlasování. Z výsledků je patrné, kdo kolik hlasů obdržel a kdo soutěž musí opustit. Hráče s nejvyšším počtem hlasů arbitr vyřadí ze skupiny aktivních hráčů a tím pro něj soutěž končí. V případě, že při hlasování obdrželo více hráčů nejvyšší počet hlasů, pak o vyřazení jednoho z nich rozhodne výherce imunity. Arbitr ho opět vyzývá, tentokrát zprávou s identifikátorem *ImmunityVotingRequest* a v obsahu jsou vypsáni ti hráči, kteří v první fázi obdrželi nejvíce hlasů. Výherce imunity odpovídá opět zprávou s identifikátorem *Voting*. V soutěži tímto způsobem proběhne sedm hlasování, až zůstanou 3 nejlepší, mezi kterými se rozhodne o celkovém vítězi soutěže. Arbitr vyzve všech sedm vypadnutých hráčů zprávou s identifikátorem *WinnerVotingRequest*, aby hlasovali pro toho hráče, který by měl soutěž vyhrát. Všichni tři hráči jsou popsáni v obsahu zprávy, proto vypadnutí vědí, pro koho mohou hlasovat. Poté, co všichni dotázaní odpoví zprávou s identifikátorem *Voting*, arbitr pošle všem hráčům, včetně třech finalistů, výsledky hlasování. Protože hlasujících je sedm, pak může nastat remíza pouze mezi dvěma hráči. Pokud se tak stane, pak arbitr znovu vyzve všech sedm hlasujících k novému hlasování mezi těmito dvěma finalisty. Po zaslání výsledků hlasování ještě zašle arbitr všem hráčům konečné pořadí ve zprávě s identifikátorem *GameResult* a tímto hra končí.

6.11 Návrh grafického uživatelského rozhraní

Pro adventure hru bude implementováno grafické uživatelské rozhraní, u kterého je důraz především na intuitivnost a jednoduchost ovládání. Hry se účastní deset hráčů, devět z nich budou počítačoví hráči, kteří budou volit své akce na základě svých představ. Desátým hráčem bude člověk, který bude své akce vykonávat skrz grafické prostředí. Grafické uživatelské rozhraní musí zajistit, aby veškerá komunikace, tj. výběr akce, reakce nebo hlasování, probíhala přesně podle navrženého protokolu. Hráč si tedy pomocí grafického rozhraní zvolí akci s parametry, kterou chce provést a akce se musí pak transformovat do zprávy pro arbitra s odpovídající syntaxí. Podobně to je i se zprávami, ve kterých arbitr žádá hráče o reakci. Při přijetí takového typu zprávy budou hráči pomocí grafického rozhraní nabídnuty možné reakce na danou situaci. Z hráčovi odpovědi se poté opět musí vytvořit zpráva s odpovědí pro arbitra.

Při spuštění hry má hráč možnost zadat jména a pohlaví všech deseti soutěžících. Každému soutěžícímu bude přiřazena ikona se vzhledem, kterou může hráč nastavit podle vlastního výběru. Ikony by měly sloužit pro zvýšení jednoduchosti a hratelnosti. Poté zbývá výběr hráčů potvrdit a hra začíná. Grafické uživatelské rozhraní bude obsahovat několik záložek. Nejdůležitější bude záložka pro výběr akcí a reakcí. Na této záložce si hráč bude vybírat jakou provede akci, koho si vybere při hlasování nebo jakým způsobem bude reagovat na výzvy od arbitra či ostatních hráčů. U každé akce bude mít hráč na výběr pouze ty volby, které jsou realizovatelné. Například nebude zde možnost volit při hlasování hráče, který ma imunitu či nebude možné se bavit s již vypadlými hráči atd. Při výběru akcí, u kterých se bude volit mezi jednotlivými hráči, budou zobrazeny pro přehlednost i jejich ikony.

Dále bude aplikace obsahovat záložku pro vytváření poznámek. Počítačoví hráči si veškeré znalosti ukládají do svých představ a podle toho jednají. Hráč, který hraje hru a ovládá soutěžícího pomocí grafického rozhraní, si musí nějakým způsobem pamatovat a zaznamenávat jednotlivé akce a jejich výsledky, aby dosáhl ve hře úspěchu. Z tohoto důvodu si může na záložce vytvářet a modifikovat poznámky o svých nebo soupeřových akcích a reakcích nebo o obdržených informacích. K udržení přehledu ve hře mu slouží také další záložky. Všechny výsledky proběhlých hlasování jsou zobrazeny na záložce s historií hlasování. Další záložka slouží k udržení přehledu o tom, v jakých skupinách je hráč zapojen. Jsou zde informace o jméně skupiny a jejích členech. Seznam všech hráčů, kteří se hry zúčastnili se nachází na poslední záložce. U všech hráčů je jejich ikona a je zde také informace o tom, jestli hráč stále hraje nebo ze hry už vypadl.

Kapitola 7

Implementace adventure hry

Adventure hra byla implementována v jazyce Java a vývoj probíhal ve stejném prostředí jako implementace frameworku. V této kapitole jsou popsáni všichni agenti, kteří se ve hře vyskytují a jsou její nedílnou součástí. Agent `Arbiter` řídí celou hru a postupně vyzývá k akcím počítačové hráče, které jsou reprezentovány třídou `PlayerPC` a také hráče `Player`, který komunikuj s arbitrem přes uživatelské rozhraní. U počítačových hráčů jsou popsány základní algoritmy pro výběr akcí a pro uvažování hráče. Detailněji je také popsáno GUI, které má společně s umělou inteligencí počítačových hráčů velký vliv na hratelnost celé hry.

Po spuštění aplikace se zavolá třída `SurvivorGame` a ta vytvoří instanci `SurvivorGUI`, která zobrazí úvodní menu hry v grafickém okně. Implementace této třídy, která vytváří a obsluhuje grafické uživatelské rozhraní je popsána níže v kapitole 7.4. Až hráč nastaví jména, pohlaví a vzhled všech soutěžících, tak může spustit hru. Po stisknutí tlačítka pro start hry se zavolá metoda `StartGame` třídy `SurvivorGUI`, kde se vytvoří agenti pomocí metod třídy JADE, kteří budou reprezentovat vybrané hráče ve hře. Implementace adventury se skládala ze dvou částí - vývoj aplikační vrstvy a implementace grafického uživatelského rozhraní.

7.1 Implementace arbitra

Agent arbitr byl implementován podle návrhu z kapitoly 6.4. Třída `Arbiter` dědí od `IntelligentAgent` a může tak využívat všech dostupných metod implementovaného frameworku. Po vytvoření všech hráčů, kteří se budou hry účastnit, se vytvoří i agent arbitr, kterému se v parametrech předá seznam všech jmen soutěžících. Jako první arbitr využívá metody pro vytváření skupiny, ve které budou aktivní hráči. Do skupiny přidá všechny soutěžící a po každém hlasování z této skupiny odebere hráče, pro kterého hra skončila. Během provádění akcí arbitr často potřebuje přístup ke konstantám, jako jsou třeba jména dostupných akcí ve hře nebo nastavení pravděpodobnostních hodnot. Všechny potřebné konstanty jsou ve třídě `Constants`, kterou využívájí i ostatní třídy. Jsou zde definovány cesty ke grafickým souborům nebo také hodnoty, které jsou potřebné pro správu představ.

Arbitr si ve svých privátních proměnných uchovává informace o výherci imunity a vlastníkovi skryté imunity. Proměnná třídy `ImmunityIdol` obsahuje vše podstatné, co arbitr potřebuje znát ohledně skrytého symbolu imunity, například zda je symbol imunity dostupný nebo ho vlastní některý z hráčů. Jsou zde také informace o tom, kolikrát se jednotliví hráči pokusili imunitu hledat. Arbitr využívá také třídy `Helper`, která nabízí metody pro vytváření zpráv pro hráče, zpracovávání výsledků hlasování nebo pro práci s pravděpodobností. Ve hře je zakomponováno několik událostí, které se mohou vyskytnou s určitou pravdě-

podobností. Náhodně je takto rozhodnuto o vítězi imunity nebo o tom, v jakém kole si některý z hráčů všimne indicie k nalezení skrytého symbolu imunity. Náhodně jsou také pro hráče vybírány vjemy z prostředí, které se mu budou zasílat zprávou s identifikátorem *PerceptInvitation*.

7.2 Implementace počítačových hráčů

Počítačoví hráči jsou realizováni třídou `PlayerPC`, která dědí `IntelligentAgent` třídu. Hráč si při spuštění zaregistruje metodou `SetSpecialMsg` odběr zpráv od arbitra. Každá zpráva od arbitra bude přijate v metodě `ReceiveSpecialMsg(ACLMessage msg)`, kde bude hráč reagovat na všechny výzvy a zpracovávat informace, které mu bude arbitr zasílat. Ve funkci `ReceiveMsgFromGroup(ACLMessage msg)` bude přijímat zprávy, které hráče informují o přijetí nebo vyloučení člena ze skupiny nebo zprávy, které souvisí s dotazy nebo sdílení informací uvnitř skupiny.

7.3 Umělá inteligence hráčů

Nejdůležitější částí je pro hráče vytváření a modifikování představ na základě toho, jak se hra vyvíjí. Metoda `CreateBelief(String beliefName, Object[] params)` vytváří představu, která je pak uložena do báze znalostí. Tímto způsobem jsou vytvářeny představy *Player*, *Voting* a další. Funkci jsou v parametrech předány pořebné hodnoty pro inicializaci představy.

S každou nově přijatou informací si hráč může upravovat své představy. Metody pro úpravu znalostí obsahují prefix `UpdateBeliefAfter`, které jsou předány veškeré parametry z nově získané informace. Například když hráč byl informován od jiného hráče, tak si upraví představy zavoláním metody `UpdateBeliefsAfterInformationReceived`, které předá parametry o tom, kdo ho informoval, o jaké akci a s jakým výsledkem. Většinou hráč upravuje hodnoty důvěry u představy *Player*, k čemuž slouží metoda `UpdatePlayerBeliefTrust( String playerName, int changeInTrust)`, která k důvěře hráče přičte hodnotu druhého parametru (hodnota může být i záporná). Zvýšení důvěry může být vyvoláno některou z následujících akcí, kterou protihráč vůči hráči vykonal:

- Protihráč byl přidán do stejné skupiny,
- protihráč ukázal hráči imunitu nebo indicii,
- protihráč daroval hráči imunitu,
- protihráč si nezávazně promluvil s hráčem,
- protihráč sdílel informaci ve skupině,
- protihráč odpověděl na otázku ve skupině,
- protihráč odpověděl na otázku od hráče,
- hráč se dozvěděl pozitivní informaci o protihráči.

Důvěra se ale může i snižovat a to v následujících případech :

- Hráč byl vyřazen ze skupiny,

- protihráč dobrovolně opustil skupinu,
- protihráč odmítl daroval hráči imunitu,
- protihráč odmítl ukázat imunitu nebo indicii,
- protihráč si odmítl nezávazně promluvil s hráčem,
- protihráč odmítl odpovědět na otázku ve skupině,
- protihráč odmítl odpovědět na otázku hráče,
- hráč se dozvěděl negativní informaci o protihráči.

Podle hodnot důvěry můžeme hráčův vztah k ostatním soutěžícím rozdělit na pozitivní, negativní a neutrální. Pokud některý hráč nabývá důvěry větší jak 66, pak je považován za přítele. Pokud je důvěra ke hráči nižší než 33, pak je označen za nepřítele. Prahové hodnoty, které vymezují tyto vztahy jsou definovány ve třídě `Constants`. Strategie hráče je taková, že volí své akce a reakce tak, aby si udržoval důvěru u svých přátel. Hráč s přáteli sdílí důvěrné informace (kdo má imunitu nebo indicii), odpovídá jim na dotazy, pokud zná odpověď a nemá zájem konat akce, které by na přátele měly negativní dopad. Hráč může také důvěru k jinému protihráči upravovat na základě toho, co si o něm myslí jeho přítel. Jedním z cílů hráče je vytvořit komunitu, ve které si budou všichni důvěřovat. Hráč chce ze hry vyřadit svého největšího nepřítele, proto může nabádat ostatní své přátele k tomu, aby ho při hlasování zvolili. Metody `GetEnemyFromPlayers` a `GetBestFriendFromPlayers` vrací z představ hráče jeho největšího přítele či nepřítele, vůči kterým hráč provádí většinu svých akcí, případně je využívá i při běžném nebo finálovém hlasování. S mírou důvěry se pracuje i v rámci skupinových akcí. Důvěra ke skupině se dá vyjádřit jako průměrná hodnota důvěry ke všem členům skupiny. Na základě důvěry ke skupině se hráč může rozhodnout, jestli chce skupinu opustit nebo zda chce uvnitř skupiny sdílet určité znalosti. Funkce `int GetGroupTrust(String groupName)` vypočítá důvěru ke skupině.

Proces výběru akce závisí na několika parametrech. Ze začátku hry má hráč snahu vytvořit nebo se zařadit do nějaké existující skupiny a co nejrychleji vytvořit přátelství s některými hráči. V průběhu hry pak koná pozitivní akce vůči svým přátelům. Na základě představ *Voting* hráč může rozpoznat pomocí metody `boolean IsInDanger(String player)`, zda on nebo nějaký jiný soutěžící je v ohrožení při dalším hlasování. Záleží to na celkové pozici a počtu obdržených hlasů při posledním hlasování. Pokud hráči hrozí vyřazení ze soutěže, pak se svými akcemi snaží buď nalést nebo získat od jiného soutěžícího skrytý symbol imunity. Pokud má hráč symbol skryté imunity, pak může vyhovět prosbě od svého přítele a symbol mu darovat. Samozřejmě vlastnit symbol skryté imunity je obrovská výhoda, proto ho daruje pouze velmi dobrému příteli (hodnota důvěry musí být větší než konstanta `BELIEF_GOOD_FRIEND_TRUST`), který je v ohrožení při následujícím hlasování. Pokud ale hrozí vyřazení i hráči, který symbol vlastní, pak ho sám při hlasování použije.

7.4 Implementace grafického uživatelského rozhraní

O veškeré grafické rozhraní se stará třída `SurvivorGUI`, pomocí které hráč vybírá své akce a reaguje na vzniklé situace. Hráč ale musí umět přijímat a odesílat zprávy arbitrovi a ostatním členům. Proto byla implementovaná třída `Player`, která také dědí třídu `IntelligentAgent`, která tuto funkčnost zajišťuje. Stejně jako třída `PlayerPC` umí i `Player`

reagovat a přijímat všechny navržené zprávy z kapitoly 6. Rozdíl spočívá v tom, že se hráč nerozhoduje podle svých představ a umělé inteligence, ale pomocí komunikace s třídou `SurvivorGUI` zobrazí informaci nebo výzvu na grafické uživatelské rozhraní. Uživateli je vždy zobrazeno, o jakou událost se jedná a jaké jsou možnosti na zareagování. `SurvivorGUI` obsahuje metody pro:

- Výběr akce (zpráva od arbitra s identifikátorem *ActionInvitation*),
- zareagování na akci od arbitra (*ReactInvitation*) ,
- vybrání hráče pro hlasování (*VotingRequest*),
- zobrazení vjemů z prostředí (*PerceptInvitation*),
- zobrazení obdržené informace (*InformationReceived*).

Těmto metodám jsou předány parametry získané ze zprávy od arbitra nebo jiného hráče a podle toho jsou nastaveny prvky GUI. V prvních třech případech očekává arbitr od hráče odpověď na zaslanou zprávu. Hráč si zvolí některou z nabízených možností v GUI a tato volba je poté převedena do odpovídající odpovědi typu `ACLMessage`, která je vrácena třídě `Player` a ta jí už odešle arbitrovi nebo jinému hráči jako konečnou odpověď.

Grafické uživatelské rozhraní se skládá z pěti hlavních záložek: výběr akcí, správa poznámek, historie hlasování, seznam skupin a záložka se všemi hráči, kteří se hry účastní. Záložka pro výběr akcí také zobrazuje veškeré informace, které jsou hráči zaslány. Je proto nejdůležitější záložkou a hráč ji využívá ke všem svým volbám, které v soutěži musí provést. Ostatní záložky jsou pouze pro udržování přehledu o tom, co se ve hře stalo. Pro výběr a zobrazování informací byly navrženy třídy `ActionPanel`, `ReactionPanel`, `InformationPanel`, `ResultsPanel` a `GameResultsPanel`. Všechny třídy dědí `JPanel` z Javy a jsou umístěny na první záložku. Podle obdržené zprávy pak třída `SurvivorGUI` nastaví viditelnost pouze tomu panelu, ke kterému se zpráva vztahuje. Tento panel pak podle parametrů obdržené zprávy zobrazí vše podstatné a bude čekat na hráčovu odezvu. `ActionPanel` slouží pro výběr akce a pro výběr hráče při hlasováním o tom, kdo by soutěž měl opustit. Během volby akce se nejprve provede výběr typu akce, kterou chce hráč povést a poté se vygenerují nové tlačítka s jinými popisy, pomocí kterých si hráč zvolí parametry pro vybranou akci v prvním kroku. Bude vygenerováno i tlačítko s popisem zpět, kterým hráč vrátí svůj poslední výběr. Při každé volbě akce nebo některého jejího parametru bude na zásobník uložena instance třídy `ButtonsSetting`. Tato třída si uchovává informaci o všech viditelných prvcích GUI a jejich popisech v aktuální fázy výběru akce. V případě stisknutí tlačítka zpět jsou všechny prvky GUI nastaveny na hodnoty z vrcholu zásobníku.

Pokud arbitr zašle výzvu na reakci, tak si hráč vybere některou z nabízených možností prostřednictvím panelu `ReactionPanel`. Informační zpráva se zobrazí na `InformationPanel` a hráč ji musí potvrdit stisknutím tlačítka. Znázornění výsledků hlasování a celkového umístění ve hře umožňují `ResultsPanel` a `GameResultsPanel`. Rozmístění grafických prvků v celém GUI bylo vytvořeno pomocí vestavěného nástroje vývojového prostředí *NetBeans*. Hlavním důrazem při tvorbě GUI byla jednoduchost a srozumitelnost, ukázky ze hry jsou k nahlédnutí v příloze D. Ikony pro výběr hráčů jsou dostupné na webu[1].

[1] http://www.iconarchive.com/show/face-avatars-icons-by-hopstarter.1.html

7.5 Testování adventure hry

Adventure hra byla při vývoji průběžně testována s každou nově implementovanou funkčností. Pro usnadnění práce při hledání chyb byla použita třída pro logování. Protože ve hře probíhá obrovské množství komunikace, bylo by dost těžké odhalovat místo vzniku chyby jiným způsobem. Z logovacího výstupu se dají také vyvodit reakce jednotlivých počítačových hráčů a pozorovat změny v bázi znalostí po provedených akcích. Hra byla testovaná na operačním systému *Windows 10*.

Klíčovým prvkem se ukázal začátek hry, kdy je důležité se s někým spřátelit nebo být v početnější skupině, ve které si hráči budou důvěřovat. Tento typ vývoje průběhu hry byl očekávaný již při návrhu. Při implementaci jsem používal výhradně metod třídy `IntelligentAgent`, což prokázalo vhodnost použití frameworku pro aplikace s multiagentními systémy.

Kapitola 8

Závěr

Tato diplomová práce navázala na semestrální projekt, ve kterém byl navržen framework pro vývoj agentních systémů. Tento framework se podařilo implementovat a jeho funkčnost byla prokázána na adventure hře. Hra byla zvolena tak, aby se dalo využít co nejvíce nabízených metod implementovaného frameworku. Ve hře vystupuje několik postav, které spolu komunikují, sdružují se do skupin a sdílejí mezi sebou znalosti. Každá postava si také individuálně spravuje vlastní bázi představ, na základě kterých volí své akce. Při implementaci hry jsem využíval výhradně metod frameworku, což dokázalo širokou možnost využití v multiagentních aplikací.

V diplomové práci jsou postupně popsány všechny důležité principy z oblasti agentních systémů a také standard FIPA, jenž se snaží framework dodržovat. Jednotlivé prvky multiagentního chování pak byly zakomponovány do návrhu tak, aby byla pokryta co nejširší vrstva možných akcí, které může provádět agent v systému. Metody frameworku byly navrženy tak, aby doplňovaly funkčnost robustní knihovny JADE. Framework zapouzdřuje složitější akce v JADU do jednodušších metod, přináší možnost práce agentů ve skupinách nebo umožňuje spravovat množinu představ agentů.

8.1 Možné rozšíření adventure hry

Implementovaná adventure hra by mohla být dále rozšířena o několik vhodných doplňků, které by přispěly k větší hratelnosti a zábavnosti hry. Téměř neomezeným prostorem ke zlepšení je oblast umělé inteligence počítačových hráčů. Lze navrhnout nové představy a pravidla, podle kterých si bude hráč volit akce nebo reagovat na vzniklé události. Přestože hra obsahuje hlavní prvky, které se nachází v reality show Survivor, mohou být navrženy některé nové akce, které hráč bude moci používat. Dalším prvkem pro zvýšení atraktivity hry je přidělení vlastností jednotlivým hráčům. Vlastnosti jako jsou například chytrost, upřímnost, bystrost, ale také i sklony k lhaní, podrazům nebo blafování by mohli hru zajímavě zkomplikovat. Posledním rozšířením by mohla být implementace hry po síti s více hráči.

Literatura

[1] FIPA Standard Status Specification [online]. http://www.fipa.org/repository/standardspecs.html, 2002 [cit. 2016-04-11].

[2] Historie a současnost umělé inteligence [online]. http://www.fi.muni.cz/usr/jkucera/pv109/2000/xvacek.htm, [cit. 2016-04-13].

[3] Budoucnost umělé inteligence [online]. http://www.scienceworld.cz/technologie/nazor-budoucnost-umele-inteligence-hadani-z-kristalove-koule-1695/, [cit. 2016-05-05].

[4] Caire, G.: JADE programming for beginners. http://jade.tilab.com/doc/tutorials/JADEProgramming-Tutorial-for-beginners.pdf, 2009.

[5] Hawkins, J.: *On Intelligence*. Times Books, 2004, iSBN 0-8050-7853-3.

[6] Jennings, N. R.; Wooldridge, M. J.: *Agent technology*. Springer Verlag, 2002, iSBN 3-540-63591-2.

[7] Šlapal Josef: Základy matematické logiky. Vysoké učení technické v Brně, 2016-04-09.

[8] Mařík, V.; Štěpánková, O.; Lažanský, J.: *Umělá inteligence*. Academia, 2007, iSBN 978-80-200-1470-2.

[9] Russell, S.; Norvig, P.: *Artificial Intelligence, a Modern Approach*. Pearson Education Inc., 2003, iSBN 0-13-790395-2.

[10] Shoham, Y.; Leyton-Brown, K.: *Multiagent Systems*. Cambridge University Press, 2009, iSBN 978-0-521-89943-7.

[11] Wooldridge, M.: *Reasoning about rational agents*. The MIT Press, 2000, iSBN 0-262-23213-8.

[12] Wooldridge, M.: *An Introduction to MultiAgent Systems*. John Wiley & Sons Ltd, 2009, iSBN 9780470519462.

[13] Zbořil, F.: Podklady k přednáškám kurzu AGS. Fakulta informačních technologií, Vysoké učení technické v Brně, 2016-04-06.

[14] Zbořil, F.: IZU - Základy umělé inteligence : Logika a UIN, resoluční metoda a její využití při řešení úloh. Fakulta informačních technologií, Vysoké učení technické v Brně, 2016-04-13.

[15] Zelinka, I.: *Umělá inteligence - Hrozba nebo naděje*. BEN, 2003, iSBN 80-7300-068-7.

Přílohy

Seznam příloh

A Obsah CD **54**

B Rozhraní frameworku **55**

C Příklady použití frameworku **57**

C.1 Správa představ 57
C.2 Vlastní filtrování představ 58
C.3 Rozhraní IAgentGroup 60
C.4 Třída Group 60
C.5 Správa skupin 60

D Ukázky ze hry **63**

Příloha A

Obsah CD

Přiloženého CD obsahuje:

- **src** - zdrojové soubory
  - **IntelligentAgentFramework** - zdrojové soubory frameworku
  - **Survivor** - zdrojové soubory hry
- **tex** - soubory se zdrojovým kódem pro LaTeX obsahující text této práce
- **README.txt**

Příloha B

Rozhraní frameworku

```
IntelligentAgent();

/**** COMMUNICATION ****/
void SetSpecialMsg(String agentName, ACLMessage msg);
void SetSpecialMsg(String agentName);
void SetSpecialMsg(ACLMessage msg);
void RemoveSpecialMsg(String agentName, ACLMessage msg);
void RemoveSpecialMsg(String agentName);
void RemoveSpecialMsg(ACLMessage msg);
void ReceiveMsg(ACLMessage msg);
void ReceiveSpecialMsg(ACLMessage msg);
void ReceiveMsgFromGroup(ACLMessage msg);
void SendMessage(ACLMessage msg);
void SendMessageToGroup(String groupName, ACLMessage msg);
void SendMessageToGroupAndWaitForAnswer(String groupName, ACLMessage msg, MessageTemplate
    answerTemplate);
List<ACLMessage> SendAndWaitForAnswer(ACLMessage msg, MessageTemplate answerTemplate);
List<ACLMessage> VotingAmonsAgents(ACLMessage msg, String initiatorOfVoting,
    MessageTemplate template);
Boolean WasVotingAccepted(ACLMessage msg,MessageTemplate template);

/**** COMUNICATION HISTORY ****/
public List<ACLMessage> GetHistoryMsgs(SpecialMsg template);
public List<ACLMessage> GetHistoryMsgsFromAgent(String sender);
public List<ACLMessage> GetHistoryMsgsSendedToAgent(String receiver);
public List<ACLMessage> GetHistoryMsgsCommunicationWithAgent(String agentName);

/**** BELIEFS ****/
Boolean AddBelief(Belief belief);
List<Belief> FilterBeliefs(IFilterTerm filter);
List<Belief> FilterBeliefs(List<IFilterTerm> filters);
List<Belief> GetBeliefs(String termName);
List<Belief> GetBeliefs();
Boolean ExistBeliefs(Belief bel);
MethodResult UpdateBelief(Belief oldBelief, Belief newBelief);
List<Belief> GetSameTypeBeliefs(Belief t);
Boolean RemoveBeliefs(String termName);
Boolean RemoveBelief(Belief belief);
Boolean RemoveBeliefs(List<IFilterTerm> filters);
Boolean RemoveBeliefs(IFilterTerm filter);
void RemoveAllBeliefs();

/**** GROUPS ****/
```

```
MethodResult CreateGroup(String groupName, GroupPolicy policy);
MethodResult CreateGroup(String groupName, List<String> agents, GroupPolicy policy);
MethodResult CreateGroup(String name, List<String> agents);
MethodResult CreateGroup(String name);
MethodResult AddAgentToGroup(String groupName, String agentName);
MethodResult JoinGroup(String groupName);
void LeaveGroup(String groupName);
MethodResult CancelGroup(String groupName);
MethodResult KickAgentFromGroup(String groupName, String agentName);
List<String> SearchGroupsWithAgent(String agentName);
List<String> GetGroupMembers(String groupName);
List<String> GetMyGroups();

/**** SERVICES ****/
void RegisterService(ServiceDescription sd);
void ModifyService(ServiceDescription sd);
void DeregisterService(ServiceDescription s);
void DeregisterService(String serviceName);
ServiceDescription SearchServiceByName(String serviceName);
List<ServiceDescription> SearchServiceByOwner(String agentName);
```

Příloha C

Příklady použití frameworku

C.1 Správa představ

```
@Override
protected void setup() {

  /***** ADDING BELIEFS *****/
  List<Variable> osobaPetr = new ArrayList<>();
  osobaPetr.add(new Variable("jmeno", "Petr"));
  osobaPetr.add(new Variable("vek", 31));
  osobaPetr.add(new Variable("mesto", "Praha"));
  Belief belPetr = new Belief("osoba",osobaPetr);
  AddBelief(belPetr);

  List<Variable> osobaLukas = new ArrayList<>();
  osobaLukas.add(new Variable("jmeno", "Lukas"));
  osobaLukas.add(new Variable("vek", 25));
  osobaLukas.add(new Variable("mesto", "Brno"));
  Belief belLukas = new Belief("osoba",osobaLukas);
  AddBelief(belLukas);

  List<Variable> osobaMisa = new ArrayList<>();
  osobaMisa.add(new Variable("jmeno", "Michaela"));
  osobaMisa.add(new Variable("vek", 23));
  osobaMisa.add(new Variable("mesto", "Policka"));
  Belief belMichaela = new Belief("osoba",osobaMisa);
  AddBelief(belMichaela);

  Helper.PrintBeliefs(GetBeliefs());
  //Beliefs:
  //osoba: (jmeno, Petr), (vek, 31), (mesto, Praha)
  //osoba: (jmeno, Lukas), (vek, 25), (mesto, Brno)
  //osoba: (jmeno, Michaela), (vek, 23), (mesto, Policka)

  /***** UPDATE BELIEF *****/
  List<Variable> osobaMisa2 = new ArrayList<>();
  osobaMisa2.add(new Variable("jmeno", "Michaela"));
  osobaMisa2.add(new Variable("vek", 24));
  osobaMisa2.add(new Variable("mesto", "Policka"));
  Belief belMichaela2 = new Belief("osoba",osobaMisa2);

  MethodResult m = UpdateBelief(belMichaela, belMichaela2);
  if(!m.result) {
```

```
    System.err.println(m.errorMessage);
  }

  Helper.PrintBeliefs(GetBeliefs());
  //Updated beliefs:
  //osoba: (jmeno, Petr), (vek, 31), (mesto, Praha)
  //osoba: (jmeno, Lukas), (vek, 25), (mesto, Brno)
  //osoba: (jmeno, Michaela), (vek, 24), (mesto, Policka)

  /***** FILTER BELIEFS *****/
  FilterOption fOsoba_VekPod30 = new FilterOption("osoba",
  "vek", FilterOption.NUMBER_LESS, 30);
  Helper.PrintBeliefs(FilterBeliefs(fOsoba_VekPod30));
  //Filtered beliefs:
  //osoba: (jmeno, Lukas), (vek, 25), (mesto, Brno)
  //osoba: (jmeno, Michaela), (vek, 24), (mesto, Policka)

  FilterOption fOsoba_JmenoObsahujeS = new FilterOption("osoba",
  "jmeno", FilterOption.STRING_CONTAINS, "s");
  List<IFilterTerm> filters = new ArrayList<IFilterTerm>();
  filters.add(fOsoba_JmenoObsahujeS);
  filters.add(fOsoba_VekPod30);
  Helper.PrintBeliefs(FilterBeliefs(filters));
  //Filtered beliefs:
  //osoba: (jmeno, Lukas), (vek, 25), (mesto, Brno)

  /***** REMOVE BELIEFS *****/
  List<Variable> zvirePes = new ArrayList<>();
  zvirePes.add(new Variable("druh", "pes"));
  Belief belZvire = new Belief("zvire",zvirePes);
  AddBelief(belZvire);
  RemoveBeliefs(filters);
  Helper.PrintBeliefs(GetBeliefs());
  //Actual beliefs:
  //osoba: (jmeno, Petr), (vek, 31), (mesto, Praha)
  //osoba: (jmeno, Michaela), (vek, 24), (mesto, Policka)
  //zvire: (druh, pes)

  RemoveBeliefs("osoba");
  Helper.PrintBeliefs(GetBeliefs());
  //Actual beliefs:
  //zvire: (druh, pes)

  RemoveBelief(belZvire);
  Helper.PrintBeliefs(GetBeliefs());
  //Actual beliefs: []
}
```

C.2 Vlastní filtrování představ

```
class MyFilter extends FilterOption{
  public static final int IS_CUSTOMER_RELIABLE = OPERATION_COUNT+1;
  public MyFilter(String termName, String variableName, int operation, Object params) {
    super(termName,variableName,operation,params);
  }
  @Override
  public Boolean TermMatches(Belief term) {
    Object variableValue = GetVariableValue(term);
```

```java
    if(variableValue == null) {
      return false;
    }
    if(operation >= 0 && operation <= OPERATION_COUNT) {
      return GetOperationResult(variableValue);
    }
    else if(operation == IS_CUSTOMER_RELIABLE) {
      if((params==null) && (variableValue instanceof Customer)) {
        Customer c = (Customer)variableValue;
        if(!c.badReputation && !c.latePayment && c.trust > 50 && c.agreedContract > 5) {
          return true;
        }
      }
    }
    return false;
  }
}

class Customer {
  public String customerName;
  public boolean badReputation;
  public boolean latePayment;
  public int trust;
  public int agreedContract;

  public Customer(String customerName, int trust, boolean badReputation, boolean
      latePayment, int contract) {
    this.customerName = customerName;
    this.trust = trust;
    this.badReputation = badReputation;
    this.latePayment = latePayment;
    this.agreedContract = contract;
  }

@Override
protected void setup() {
  Customer c1 = new Customer("customer1", 95, false, false, 20);
  Customer c2 = new Customer("customer2", 25, false, false, 10);
  Customer c3 = new Customer("customer3", 60, true, false, 10);
  Customer c4 = new Customer("customer4", 80, false, false, 7);
  List<Variable> c1var = new ArrayList<>();
  c1var.add(new Variable("customerClass", c1));
  Belief belc1 = new Belief("customer",c1var);
  AddBelief(belc1);

  List<Variable> c2var = new ArrayList<>();
  c2var.add(new Variable("customerClass", c2));
  Belief belc2 = new Belief("customer",c2var);
  AddBelief(belc2);

  List<Variable> c3var = new ArrayList<>();
  c3var.add(new Variable("customerClass", c3));
  Belief belc3 = new Belief("customer",c3var);
  AddBelief(belc3);

  List<Variable> c4var = new ArrayList<>();
  c4var.add(new Variable("customerClass", c4));
  Belief belc4 = new Belief("customer",c4var);
  AddBelief(belc4);
```

```
  FilterOption duveryhodnyZakaznik = new MyFilter("customer", "customerClass",
      MyFilter.IS_CUSTOMER_RELIABLE, null);
  Helper.PrintBeliefs(FilterBeliefs(duveryhodnyZakaznik));
  //Filtered beliefs:
  //customer: customerClass:["customer1", 95, 20, false, false]
  //customer: customerClass:["customer4", 80, 7, false, false]
}
```

C.3 Rozhraní IAgentGroup

```
public class AgentGroup extends IntelligentAgent implements IAgentGroup;

public interface IAgentGroup {
  MethodResult CreateGroup(String agent, String groupName, Group.GroupPolicy p);
  MethodResult AddAgentToGroup(String group, String agent, String requestAgent);
  MethodResult AddAgentsToGroupByCreator(String groupName, List<String> agents, String
      groupCreator);
  void LeaveGroup(String groupName, String agentName);
  MethodResult CancelGroup(String groupName, String agentName);
  MethodResult KickAgentFromGroup(String groupName, String unwantedAgentName, String
      initiatorOfKicking);
  List<String> SearchGroupWithAgent(String agentName);
  void SendMessageToGroup(String groupName, ACLMessage msg);
  void SendMessageToGroupAndWaitForAck(String groupName, ACLMessage msg);
  List<String> GetMyGroups(String agentName);
  List<String> GetGroupMembers(String groupName);
}
```

C.4 Třída Group

```
public class Group {

  private String name;
  private String creator;
  private List<String> agents;
  private GroupPolicy policy;
  private Boolean secret;

  public enum GroupPolicy {
    DEMOCRATIC, DICTATOR
  }

  public Group(String groupName, String creator, GroupPolicy policy);
  public Boolean AddAgent(String agentName);
  public void LeaveGroup(String agentName);
  public String GetName();
  public void SetName(String name);
  public String GetCreator();
  public List<String> GetAgents();
  public int GetAgentsNumber();
  public GroupPolicy GetPolicy();
  public void SetSecret(Boolean s);
  public Boolean IsSecret();
}
```

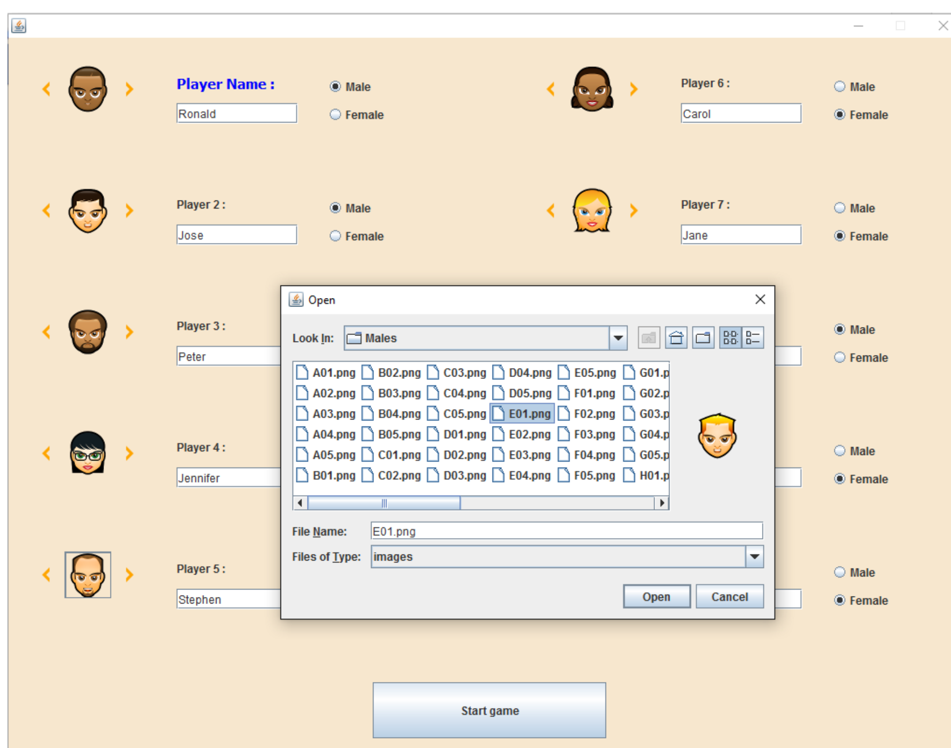
C.5 Správa skupin

```java
//SPUŠTĚNÍ AGENTŮ Alice, Bob, Charlie a GroupAgent
public class IAFrameworkExample extends IntelligentAgent{

  @Override
  protected void setup() {
    jade.core.Runtime rt = jade.core.Runtime.instance();
    Profile p = new ProfileImpl();
    jade.wrapper.AgentContainer mainContainer = rt.createMainContainer(p);
    ContainerController cc = getContainerController();
    try {
      AgentController agent1 = cc.createNewAgent("Alice",
          Agent1.class.getCanonicalName(),null);
      AgentController agent2 = cc.createNewAgent("Bob",
          Agent2.class.getCanonicalName(),null);
      AgentController agent3 = cc.createNewAgent("Charlie",
          Agent3.class.getCanonicalName(),null);
      AgentController groupAgent = cc.createNewAgent(Constants.GROUP_AGENT_NAME,
          group.AgentGroup.class.getCanonicalName(),null);
      agent1.start(); agent2.start(); agent3.start(); groupAgent.start();
    } catch (StaleProxyException e) {
      e.printStackTrace();
    }
  }
}

//AGENT: ALICE
@Override
protected void setup() {
  List<String> members = new ArrayList<>();
  members.add("Bob");
  MethodResult m = CreateGroup("skupina1", members, Group.GroupPolicy.DEMOCRATIC);
  if(!m.result)
    System.out.println(m.errorMessage);

  String membersNames = Helper.ListToString(GetGroupMembers("skupina1"));
  //Members: Alice, Bob

  //za 5 s zkontroluje členy
  addBehaviour(new WakerBehaviour(this, 5000) {
    @Override
      protected void handleElapsedTimeout() {
        String membersNames = Helper.ListToString(GetGroupMembers("skupina1"));
        // Members: Alice, Bob, Charlie
      }
  });

@Override
public void ReceiveMsgFromGroup(ACLMessage msg) {
  if(msg.getSender().getLocalName().equals(Constants.GROUP_AGENT_NAME) &&
       msg.getContent().contains(Constants.ACCEPT_AGENT)) {
    //Souhlasí s přijetí nového člena
    ACLMessage m = new ACLMessage(ACLMessage.AGREE);
    m.setContent(msg.getContent());
    m.addReceiver(new AID(Constants.GROUP_AGENT_NAME, AID.ISLOCALNAME));
    SendMessage(m);
  }
}
//PŘIJATÉ ZPRÁVY:
```

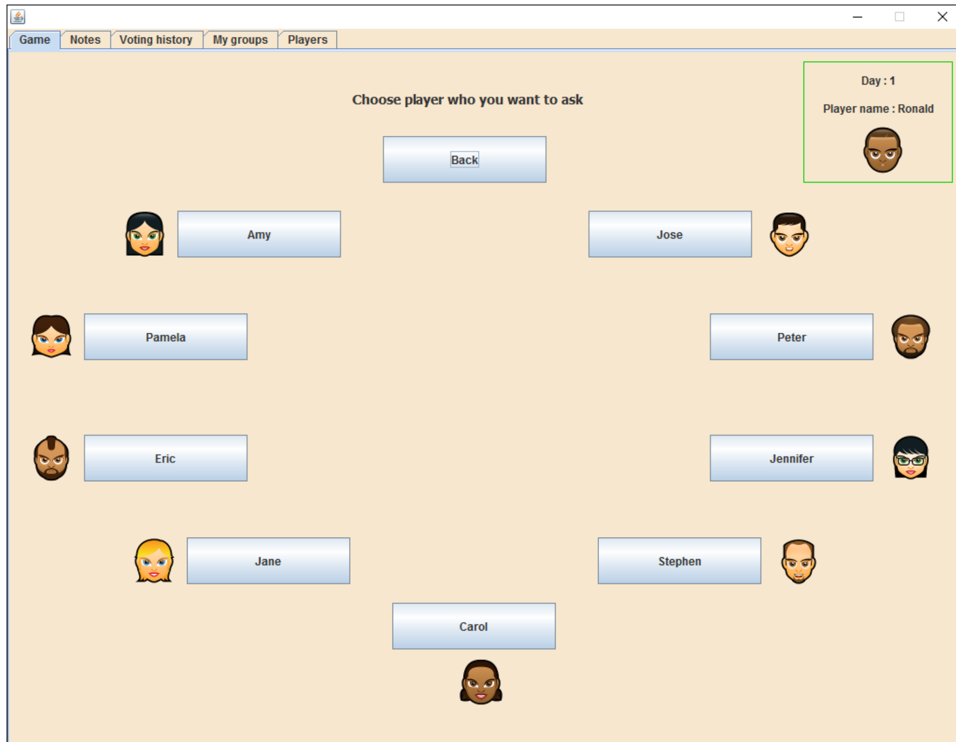
```
1.Od: GroupAgent, Obsah: "AcceptAgent(skupina1,Charlie)", Typ: Request
2.Od: GroupAgent, Obsah: "Agent Charlie was added to group skupina1.", conversation ID:
    "AddAgentResult(skupina1,Charlie)", Typ: Inform



//AGENT: Bob
@Override
public void ReceiveMsgFromGroup(ACLMessage msg) { .. Jako Alice .. }
//PŘIJATÉ ZPRÁVY:
1.Od: GroupAgent, Obsah: "WelcomeMessage(skupina1)", Typ: Inform
2.Od: GroupAgent, Obsah: "AcceptAgent(skupina1,Charlie)", Typ: Request
3.Od: GroupAgent, Obsah: "Agent Charlie was added to group skupina1.", conversation ID:
    "AddAgentResult(skupina1,Charlie)", Typ: Inform

//AGENT: Charlie
@Override
protected void setup() {
  //za 2 s se zkusí připojit do skupiny
  addBehaviour(new WakerBehaviour(this, 2000) {
    @Override
    protected void handleElapsedTimeout() {
      List<String> groups = SearchGroupsWithAgent("Alice");
      if(!groups.isEmpty()) {
        MethodResult m = JoinGroup(groups.get(0));
        if(!m.result)
          System.out.println(m.errorMessage);
      }
      List<String> myGroups = GetMyGroups();
      //myGroups: [skupina1]
    }
  });
}

@Override
public void ReceiveMsgFromGroup(ACLMessage msg) { ... }
//PŘIJATÉ ZPRÁVY:
1.Od: GroupAgent, Obsah: "WelcomeMessage(skupina1)", Typ: Inform
```

Příloha D

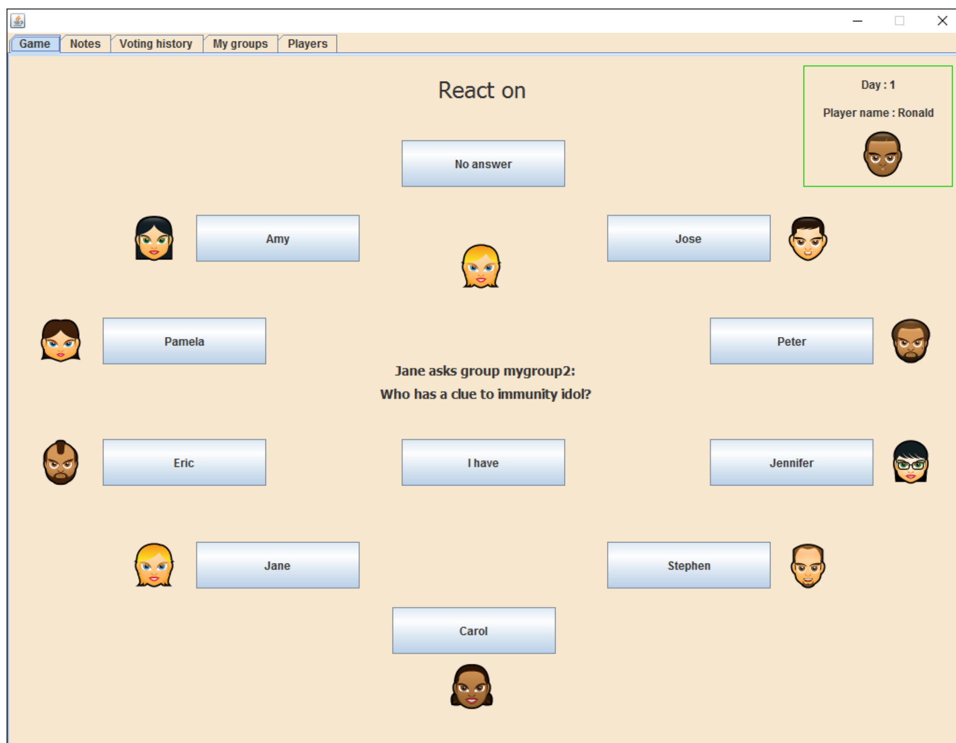
Ukázky ze hry



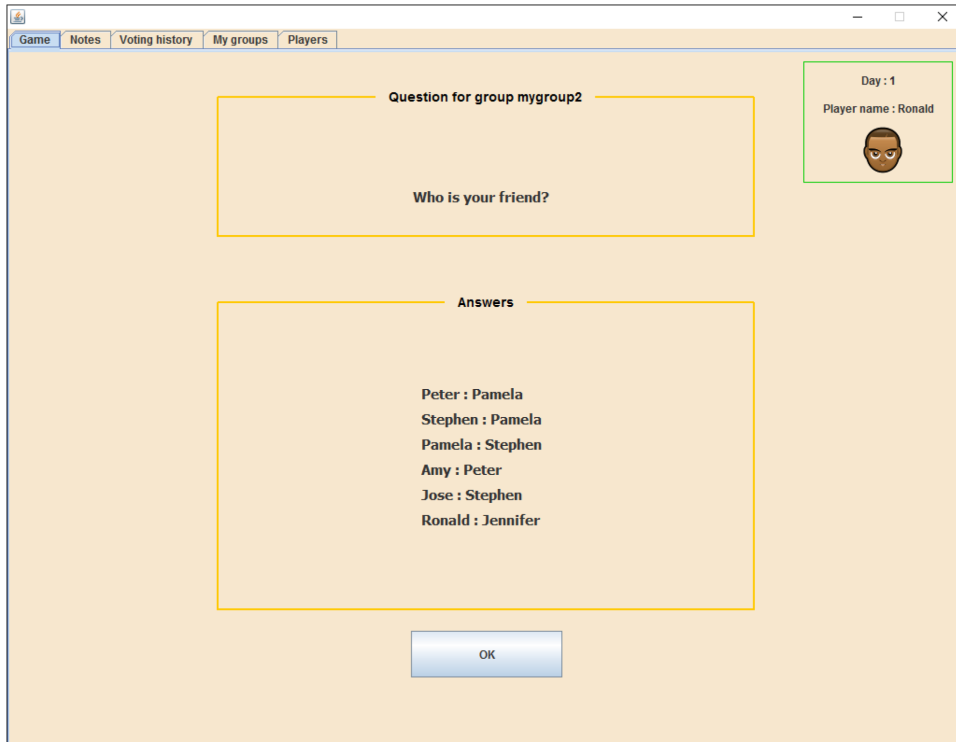
Obrázek D.1: Nastavení hráčů před spuštěním hry



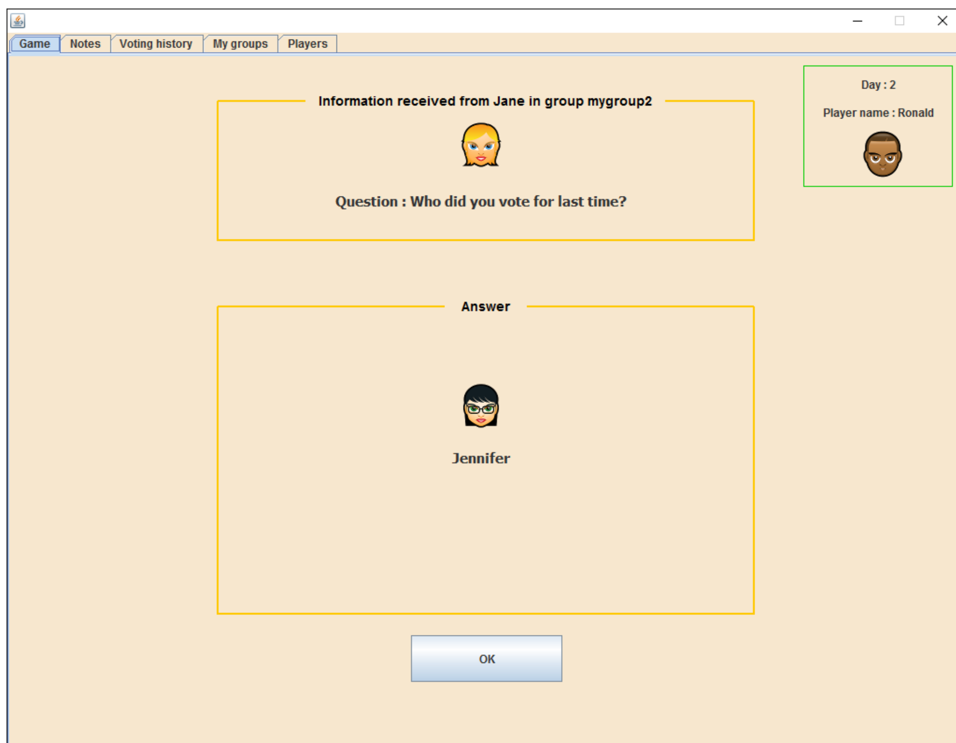
Obrázek D.2: Výběr akce



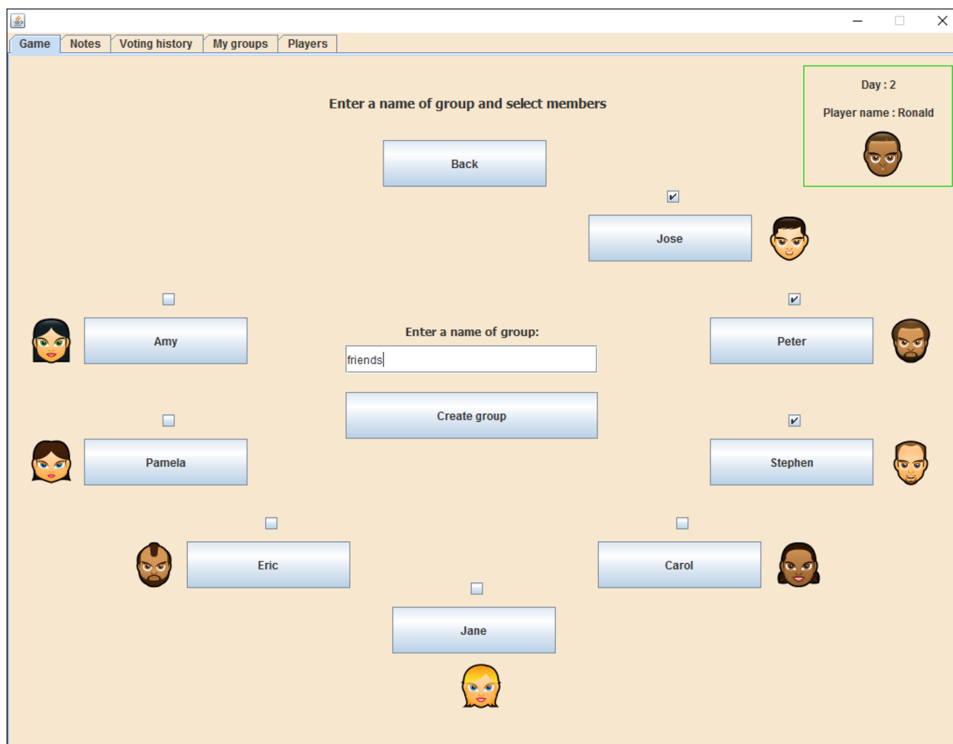
Obrázek D.3: Výběr reakce



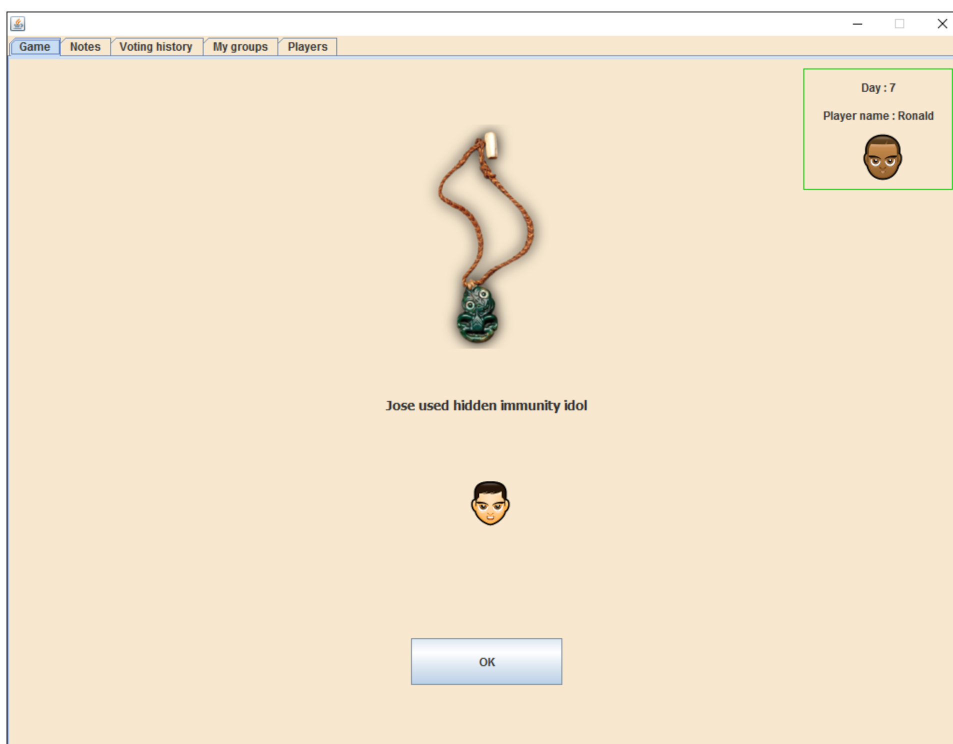
Obrázek D.4: Výsledek skupinové akce



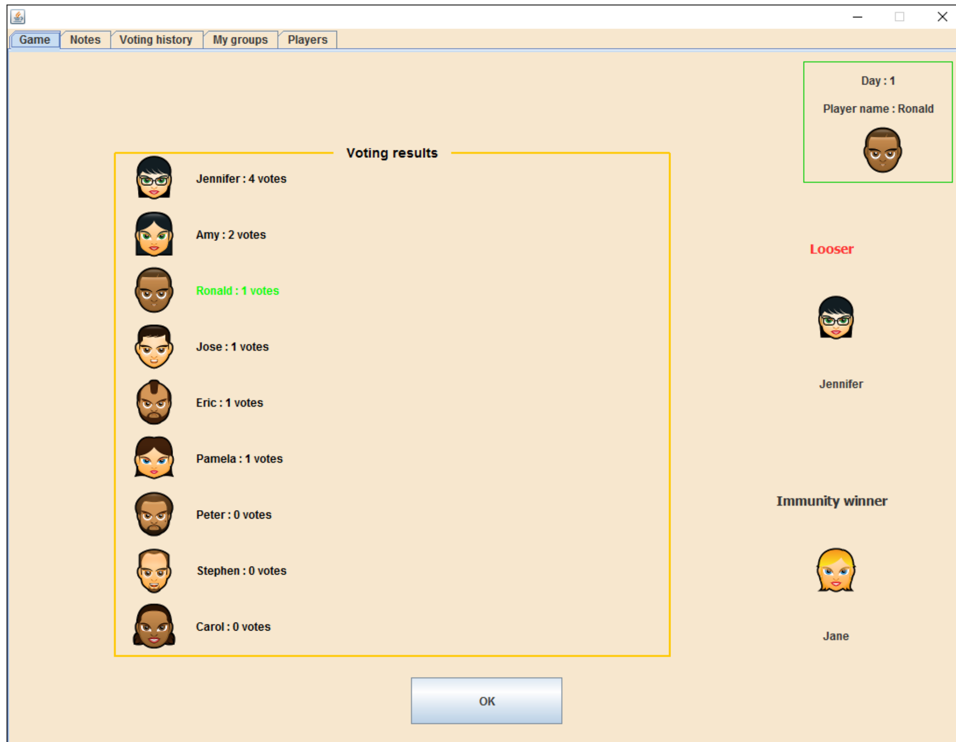
Obrázek D.5: Přijetí nové informace



Obrázek D.6: Založení nové skupiny

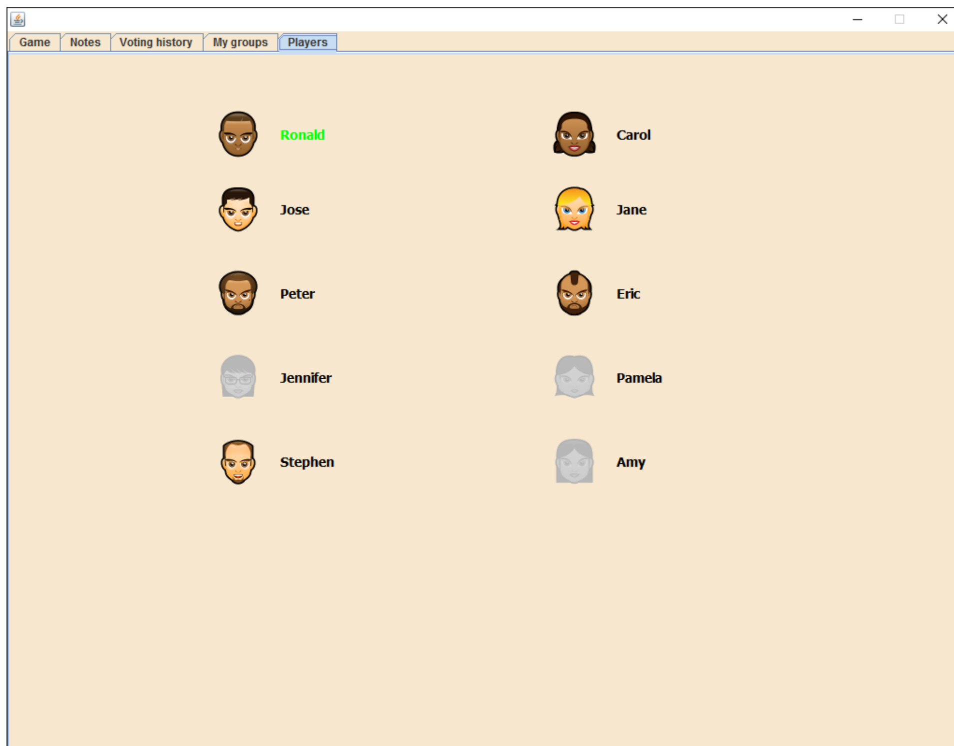


Obrázek D.7: Informace o použití symbolu skryté imunity



Obrázek D.8: Výsledky hlasování

Obrázek D.9: Záložka skupin



Obrázek D.10: Záložka hráčů

Game | Notes | Voting history | My groups | Players

Voting number	Immunity owner	Looser	My vote	Voting result	Used immunity idol
1	Margaret	Steven	Steven	Steven : 6 Ronald : 3 Dorothy : 1 Sarah : 0 Donald : 0 Laura : 0 Jennifer : 0 Daniel : 0 Karen : 0	
2	Ronald	Daniel	Margaret	Daniel : 5 Dorothy : 3 Margaret : 1 Sarah : 0 Donald : 0 Laura : 0 Jennifer : 0 Karen : 0	
3	Ronald	Dorothy	Margaret	Dorothy : 4 Sarah : 1 Laura : 1 Jennifer : 1 Margaret : 1 Donald : 0 Karen : 0	
4	Ronald	Laura	Laura	Laura : 3 Jennifer : 3 Sarah : 1 Donald : 0 Karen : 0 Margaret : 0	
5	Ronald	Donald	Jennifer	Sarah : 2 Donald : 2 Jennifer : 2 Karen : 0 Margaret : 0	

Obrázek D.11: Záložka historie hlasováni