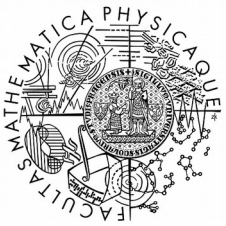


MATEMATICKO-FYZIKÁLNÍ
FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Martin Strupek

Hearthstone simulátor

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot

Studijní program: Informatika

Studijní obor: Programování a softwarové systémy

Praha 2018

Děkuji vedoucímu mé bakalářské práce Mgr. Jakubu Gemrotovi a také Pavlu Šmejkalovi za cenné rady při jejím vypracování.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V …..... dne..... podpis

Název práce: Hearthstone simulátor

Autor: Martin Strupek

Katedra / Ústav: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Katedra softwaru a výuky informatiky

Abstrakt: Cílem této práce bylo vytvoření simulátoru karetní hry Hearthstone v jazyce C#, který by otevřel možnosti analyzování kvality balíčků a taktik pomocí umělé inteligence a snadnou implementaci nových karet a pravidel. Námi vytvořený simulátor implementuje mechaniky a karty základního setu. Dále implementuje tři hrdiny včetně jejich základních karet a schopností. Simulátor umožňuje efektivně provádět velké množství simulací her dvou umělých inteligencí proti sobě. Také umožňuje uživateli otestovat mechaniky a chování umělé inteligence v grafickém prostředí. Pro demonstraci jsou k dispozici dvě jednoduché inteligence, které jsou schopny hru hrát a zvítězit. Simulátor podporuje doplnění všech zbývajících mechanik. Podporuje i snadnou tvorbu a implementaci vlastních mechanik a karet včetně jejich následného testování, čímž umožňuje vytváření dokonalejších umělých inteligencí, balíčků a karet a jejich následnou optimalizaci. Analýza rychlosti simulace náhodných her ukázala, že náš simulátor umožňuje simulovat jednotlivé hry dostatečně rychle a zároveň nejrychleji z námi testovaných konkurenčních simulátorů, což z něj činí vhodnou volbu pro zájemce o aplikaci umělé inteligence v rámci hry Hearthstone.

Klíčová slova: simulátor, karetní hra, Hearthstone, umělá inteligence

Title: Hearthstone simulator

Author: Martin Strupek

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Department of Software and Computer Science Education

Abstract: The goal of this work was creation of a Hearthstone simulator in C# which would open possibilities for analyzing quality of decks and tactics with the use of artificial intelligence as well as easy implementation of new cards and game rules. Our simulator implements mechanics and cards of the base set. It also implements three heroes including their base cards and special ability. The simulator is capable of effectively simulating large amount of games played by two artificial intelligence programs against each other. It also allows the user to test game mechanics and behaviour of his artificial intelligence in a graphically visualised environment. There are two artificial intelligences, which are capable of playing and winning the game, already implemented for demonstration purposes. The simulator supports implementation of all remaining mechanics. It also enables easy creation and implementation of custom mechanics and cards and their testing, which makes it possible to create better artificial intelligence, decks and cards and their optimisation. Analysis of the speed of our simulator has shown that our simulator can simulate games fast enough. In fact, it is faster than any of the other simulators we tested for comparison. This makes our simulator a valid choice for anyone interested in application of artificial intelligence in Hearthstone.

Keywords: simulator, card game, Hearthstone, artificial intellignece

Obsah

1 Úvod

Předmětem této bakalářské práce je tvorba simulátoru karetní hry Hearthstone[1] - populární online sběratelské karetní hry. Tento simulátor je napsán v jazyce C# a umožňuje uživateli efektivně simulovat velké množství her a tvorbu umělé inteligence určené k hraní hry Hearthstone. V době zadání práce byl na internetu volně dostupný pouze jeden udržovaný a funkční simulátor – Fireplace[2]. Fireplace je napsán v jazyce Python a jeho implementace není dostatečně rychlá a uživatelsky přívětivá, co se jeho využití pro tvorbu umělé inteligence a testování herních balíčků týče. Proto jsme se rozhodli vytvořit vlastní simulátor, který bude poskytovat lepší uživatelský komfort i další možnosti oproti simulátoru Fireplace. Tento záměr se ukázal jako správný, neboť v lednu 2017 došlo k zahájení vývoje nového simulátoru týmem lidí, kteří se zabývají simulátory této karetní hry, protože Fireplace nebyl postačující. Jejich nový simulátor - Sabberstone[3], který je nyní ve vývoji, je stejně jako náš psaný v jazyce C# a snaží se rozšířit možnosti pro uživatele. S oběma těmito simulátory se budeme v rámci této práce srovnávat a závěrem by mělo být zjištění, zda a v jakých oblastech je náš simulátor lepší, nežli tyto dva konkurenční.

Analyzovali jsme nedostatky simulátoru Fireplace a problémy, kterým musí jeho potenciální uživatel čelit. Jedná se o simulátor v jazyce Python, který funguje v konzolové verzi a nabízí uživateli možnost vytvořit si vlastní umělou inteligenci a tu nechat hrát proti jiné. Uživatel si však při tvorbě inteligence musí hlídat pravidla hry a dávat si pozor na to, aby omylem nezměnil stav hry nedovoleným způsobem. Výstup z programu ani rozhraní pro uživatele nejsou, dle našeho názoru, dostatečně přívětivé. Také očekáváme, že rychlost simulace tímto simulátorem nebude postačující pro využití umělé inteligence využívající různé prohledávací algoritmy – například Monte Carlo Tree Search. Tato domněnka bude ověřena v rámci experimentální části této práce.

Naše práce má proto pro tvorbu simulátoru následující tři cíle:

1. Vytvořit uživatelsky přívětivé UI, které uživateli umožní, aby se zaměřil na tvorbu umělé inteligence nebo vlastních karet.
2. Umožnit snadnou implementaci nových karet a mechanik.
3. Umožnit snadné simulování velkého množství her.

1.1 První cíl

Rozhodli jsme se usnadnit uživatelské rozhraní tak, aby se případný uživatel mohl pouze věnovat tvorbě vlastní umělé inteligence (Russel & Norwig, 2009). K tomu slouží jednak vizualizační část programu, která umožňuje uživateli hrát proti umělé inteligenci a sledovat její chování, ale také třída s metodami určenými k získávání informací o herním stavu a k vykonávání jednotlivých akcí, která tak uživateli usnadňuje tvorbu umělé inteligence. Simulátor si sám hlídá pravidla a uživateli, respektive agentovi umělé inteligence, dává přístup k výše zmíněné sadě funkcí, a dokonce umožňuje simulovat hru o několik tahů dopředu. Tím se výrazně liší od simulátoru Fireplace, kde si například validní cíle pro svou kartu musí hráč hledat sám. Toto usnadnění umožní i začínajícím uživatelům se zájmem o umělou inteligenci naprogramovat si vlastního, rychlého a dobře fungujícího agenta. Konzolová část aplikace cílí na optimalizaci již otestovaného agenta, kdy uživatel může nechat provést libovolné množství simulací her a získat o nich záznamy a statistiky. Cílem je simulátor dostatečně rychlý na to, aby agentovi umožňoval využít algoritmů umělé inteligence, např. Monte Carlo Tree Search, pro predikci nejvhodnějšího tahu. Rychlost simulace bude otestována v experimentální části (kapitola 7). V neposlední řadě byl kladen důraz na jednoduchost přidávání nových karet. Karty jsme se rozhodli reprezentovat přehlednou strukturou ve formě XML dokumentu. Uživatel si tak snadno může vytvořit vlastní kartu, kterou by chtěl otestovat.

Tímto cílem tedy ve zkratce klademe důraz na rychlost simulace a jednoduchost uživatelského rozhraní.

1.2 Druhý cíl

Oblíbeným cílem hráčů hry Hearthstone je tvorba vlastních karet. Není však snadné určit, zda nová karta není příliš silná a i samotná tvorba nových mechanik může být při použití konkurenčních simulátorů problematická. Náš simulátor hry Hearthstone by měl umožnit uživateli snadno vytvořit vlastní kartu a otestovat a ohodnotit její sílu a použitelnost. Tento cíl se dělí na dvě podkategorie – editování existujících (i hráčem vytvořených) karet a tvorba kompletně nových karet a mechanik. První podproblém v případě naší implementace cílí na usnadnění editace karet tím, že jednotlivé karty jsou umístěny v separátním XML souboru, jehož editací lze karty upravovat i vytvářet. Uživatel díky tomu nemusí rekompilovat celý projekt.

To je důležité zejména pokud by se uživatel rozhodl dynamicky testovat jednu či více nových karet a chtěl při jejich tvorbě zajistit, aby jejich síla nebyla příliš velká a nenarušovala tak rovnováhu hry. Druhý podproblém souvisí se způsobem jakým jsou karty implementovány za pomocí výše zmiňovaném XML souboru a je dále vysvětlena v kapitole 4 – Analýza implementace cílů. Ve zkratce: ukázalo se, že drtivá většina mechanik se dá popsat pomocí aplikace několika základních a jednoduchých mechanik typu „dober kartu“ nebo „uděl X zranění.“

1.3 Třetí cíl

Příkladem uživatele, který využije náš simulátor, budiž hráč, který si chce před turnajem ověřit teorii, zda by výměna třech klíčových karet v jeho balíčku zvýšila šance na výhru proti balíčku jeho soupeře. Tento hráč by tedy využil našeho simulátoru tak, že by vytvořil umělou inteligenci, která by uměla hrát s jeho balíčkem a nechal ji odehrát dostatečné množství her proti umělé inteligenci hrající s balíčkem jeho protivníka. Následně by na základě analýzy statistiky výher mohl určit, kterou kartu by mohl ve svém balíčku vyměnit.

Naším cílem tedy je uživateli usnadnit otestování této modelové situace. Důrazem na rychlost simulace bychom chtěli, aby simulování průběhu jedné hry trvalo co nejkratší dobu a uživatel byl tak schopen získat relevantní vzorek odehraných her v rozumném čase.

1.4 Struktura práce

Struktura práce je následující. Kapitola 2 se zabývá pravidly hry Hearthstone a terminologií, aby se čtenář mohl orientovat v dalších pasážích využívajících specifické termíny hry. V kapitole 3 jsou popsány související práce a programy zabývající se tematikou simulování hry Hearthstone. Kapitola 4 analyzuje problematiku tvorby tohoto simulátoru a zdůvodňuje volby jednotlivých postupů. Kapitola 5 je uživatelská dokumentace, popisující, jak program správně používat. Kapitola 6 je programátorská dokumentace. V kapitole 7 je popsán experiment, ve kterém jsme porovnali rychlost našeho simulátoru s ostatními podobnými programy. Kapitola 8 analyzuje, zda se nám podařilo splnit námi vytyčené cíle práce. Závěrečná kapitola 9 shrnuje výsledky porovnání s ostatními simulátory a úspěšnost naší implementace.

2 Pravidla hry Hearthstone a základní termíny

Hearthstone je online karetní sběratelská hra, vytvořená společností Blizzard entertainment. Jedná se o hru zasazenou do světa hry Warcraft. Hráči představují hrdiny, kteří si po náročném dni zašli do taverny, aby si spolu u piva zahráli karty. Jde o karetní hru pro dva hráče, kdy si každý hráč na začátku nejprve vybere hrdinu, který jej bude ve hře reprezentovat avatarem a k němu odpovídající balíček 30 karet. Hráč vyhraje, pokud se mu povede zničit oponentova hrdinu. Toho se dá docílit tím, že sníží jeho životy na 0. Hráč tak může učinit třemi způsoby. Jednak pomocí hraní kouzel, která poškodí oponenta nebo pomocí monster, kterými může na oponenta útočit. Poslední možností je počkat, až oponentovi dojdou karty a on zemře na lineárně se zvyšující poškození, které obdrží pokaždé, kdy se pokusí dobrat si z prázdného balíčku kartu.

2.1 Grafické rozhraní hry Hearthstone a základní herní entity

V této podkapitole budou popsány jednotlivé prvky karetní hry Hearthstone. Jedná se o popis herní plochy, na které se hraje. Karet a jejich jednotlivých komponent. Pro každou entitu je krátce vysvětlena její relevance v rámci pravidel a cíle hry Hearthstone.

Herní plán:

Obr. 1: Popis grafického rozhraní hry Hearthstone

1 – Avatar hrdiny hráče (a) a oponenta (b); 2 – Schopnost hrdiny hráče (a) a oponenta (b); 3 – Mana krystaly – plné (a) a prázdné (b) ; 4 – Hráčova monstra (a) a oponentova monstra (b); 5 – Tlačítko konec kola; 6 – Signalizace časového limitu kola; 7 – Historie zahraných karet; 8 – karty na ruce hráče (a) a oponenta (b)

Hrací plocha - Hrací plocha je prostor, kam hráči vyvolávají monstra. Každý hráč může na svou stranu herního plánu vyvolat až 7 monster. Aby mohl vyvolat další, musí některé z již vyvolaných zemřít.

Hrdina - Hrdina reprezentuje hráče v průběhu hry. V základu má 30 životů, o které když přijde, zemře a hráč prohraje. Volba hrdiny ovlivňuje jednak jeho schopnost, ale také jaké karty smí hráč použít při stavbě svého herního balíčku.

Mana krystal - Poskytuje hráči zásobu many. Mana je platidlo, které hráč využívá k tomu, aby mohl zahrát karty ze své ruky. Hráč začíná hru bez mana krystalů a na začátku každého kola se jejich počet zvýší o jeden, až do maxima 10 krystalů, a všechny se doplní manou. Mana krystal má dva stavy – prázdný a plný. Plný krystal obsahuje jednu manu. Například, pokud hráč zahraje kartu v ceně 2 many a má 3

plné mana krystaly ,tak zahrání karty vyčerpá 2 ze 3 jeho mana krystalů a hráči zbydou 3 mana krystaly – 1 plný a 2 prázdné.

Schopnost hrdiny - Schopnost hrdiny je reprezentována speciálním tlačítkem vedle hráčova hrdiny. Její použití stojí 2 many a lze tak standardně učinit jednou za hráčovo kolo. Schopnost hrdiny je vázána na hrdinu, tedy mág bude mít vždy „uděl 1 zranění.“

Karta - Existují karty tří typů - monstra, kouzla a zbraně. Každý typ má specifické vlastnosti a pravidla, nicméně existuje několik společných charakteristik pro všechny karty. Každá karta má vlastní cenu, 'manacost', tedy kolik many stojí zahrání dané karty. Dále má pole pro popis speciálních vlastností a efektů karty. Také má svou raritu, neboli vzácnost. Z herně nedůležitých prvků mají karty také společný design, vlastní obrázek a vlastní jméno. Nyní rozebereme jednotlivé typy.

Obr. 3: Karta monstra

Obr. 4: Karta kouzla

Obr. 2: Karta zbraně

Monstrum - Jinak také řečeno minion. Jedná se o nejčetněji zastoupenou skupinu karet. Monstra hráč zahraje z ruky, případně je vyvolá pomocí jiných karet. Monstrum má určitý maximální počet životů. Když je monstrum zraněno, sníží se jeho aktuální počet životů a pokud se dostane na nulu, je monstrum zničeno. Monstrum má také hodnotu určující sílu jeho útoku. Pokud hráč zaútočí daným monstrem, cíl typicky dostane počet zranění odpovídající síle útoku útočníka. (Mechaniky budou popsány v separátní podkapitole.) Posledním specifikem monster je příslušnost k určité rase. Některá monstra mají kromě typu "monster" navíc ještě podtyp, například "murloc". Tyto podtypy jsou dalším rozlišujícím faktorem, na který se mohou odvolávat jiné karty ve svých efektech. Monstrum po zahrání zůstane

vyloženo na stole a hráč s ním může od následujícího kola útočit jedenkrát za kolo do validního cíle. Zničená monstra jsou odstraněna ze hry.

Kouzlo - Kouzla neboli „spells“ jsou karty, které hráč může zahrát, a které mají jednorázový efekt, po jehož provedení jsou odstraněna ze hry. Kouzla jsou ovlivňována takzvaným "spell damage +X" - což je schopnost některých monster, která zvyšuje poškození, která způsobí hráčem zahraná kouzla o X.

Zbraň - Zbraň neboli „weapon“ je karta, která po zahrání umožní hráči útočit s jeho hrdinou, jako by to bylo monstrum. Zbraň má stejně jako monstrum útok, ale na rozdíl od něj má místo životů takzvanou výdrž - „durability“. Každý útok se zbraní sníží její výdrž o jedna. Když se její výdrž sníží na nula, zbraň se rozpadne a je odstraněna ze hry. Hrdina smí mít vedle sebe vyloženou pouze jednu zbraň a pokud hráč zahraje novou v okamžiku, kdy jeho hrdina nějakou zbraní stále disponuje, dojde ke zničení starší zbraně.

2.2 Základní termíny používané ve hře

Vyvolat / summon - Vyvolání monstra způsobí, že se na příslušné herní ploše objeví dané monstrum. Tento efekt spouští efekty vázané na vyvolání monstra.

Zahrát / play - Zahrání se vztahuje na karty, které má hráč na ruce. Když hráč zahraje kartu z ruky, provede se její battlecry, pokud jde o monstrum, a dojde ke spuštění efektů vázaných na událost, kdy hráč zahraje kartu. Následně dojde k vyvolání monstra.

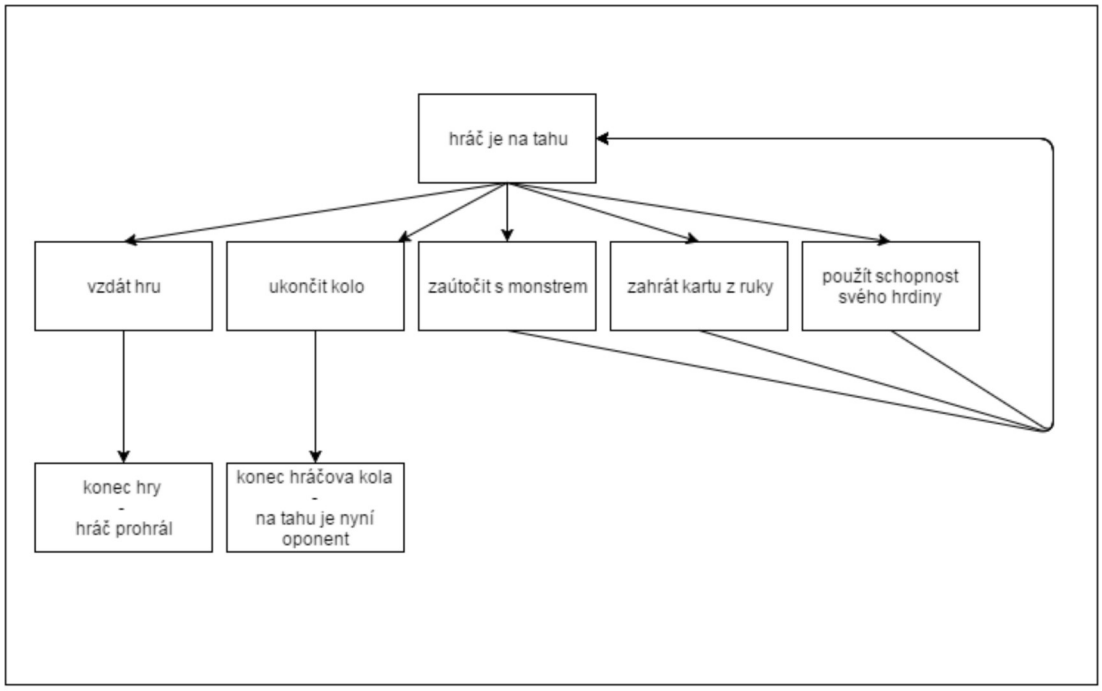
Nepřítel / enemy - Jedná se o označení všech entit na oponentově polovině hrací plochy. Jde tedy o monstra, hrdinu a zbraň oponenta.

Charakter / character - Charakter je instance monstra nebo hrdiny. Jedná se tedy případně i o hráčovo monstrum či hrdinu.

Buff – Některé karty umožní změnit počet životů či sílu útoku ostatních monster nebo jim přiřadit nějakou schopnost. Tento počin budeme v následujícím textu označovat jako „udělení buffu.“

2.3 Průběh herního kola

Hra se dělí na jednotlivá kola. Hráč odehraje své kolo a poté následuje kolo protivníka. Hráči spolu navzájem nemohou interagovat a i jednotlivé akce hráče spolu nijak neinteragují; avšak vlastnosti oponentových nestvůr mohou interagovat s akcemi a efekty prováděnými v kole hráče. Před tím, než hráčovo kolo začne, dojde k automatickému provedení efektů vázaných na začátek kola. Stejně tak i na konci kola dojde k provedení efektů vázaných na jeho konec.



Obr. 5: Průběh hráčova kola a akce, které v jeho průběhu může dělat

Hráč má ve svém kole několik možností co učinit za tah. Tyto možnosti jsou přehledně shrnuty v grafu (Obr. 5). Vždy může ukončit kolo stisknutím příslušného tlačítka. Také může vybrat monstrum na své straně hracího plánu, pokud toto monstrum smí zaútočit. V tom okamžiku však jeho následující akce musí být buď zrušení výběru daného monstra, čímž se vrátí do základního stavu nebo zvolení validního cíle. Pokud zvolí cíl útoku, hra provede daný útok, aplikuje jeho výsledek a vrátí se do základního stavu, kdy čeká na další akci hráče. Průběh útoku bude popsán v další části (2.5). Pokud má hráč na ruce kartu, jejíž cena je menší nebo rovna množství many, kterou hráč v současné době disponuje, tato karta se zvýrazní a nabídne se jako další možná akce výběru. Pokud ji hráč vybere, má opět možnost výběr zrušit nebo zvolit validní cíl pro danou kartu. Monstra bez schopností pouze zahraje. Pokud má monstrum schopnost typu „battlecry," která vyžaduje, aby hráč

vybral její cíl, hráč tak musí v průběhu zahrání karty učinit – pokud existuje validní cíl. V opačném případě se monstrum zahraje jako monstrum bez schopnosti. Schopnosti monster budou vysvětleny v další části (2.4). Pokud se jedná o kartu typu kouzlo, hráč ji smí zahrát jen v případě, že pro ni existuje validní cíl. V případě, že se jedná o kartu, kde hráč musí vybrat cíl manuálně. Zbraně hráč pouze zahraje.

2.4 Efekty karet

Karty mají různé efekty. Efekt karty je na ní vždy popsán a efekty jsou velmi různorodé. Existuje ovšem soubor určitých klíčových slov, která se vyskytují na kartách – zejména monstrech, a která určují specifické efekty. Jedná se o následující klíčová slova:

„Battlecry: <efekt X>“ - Pokud hráč zahraje tuto kartu z ruky, tak dojde k provedení X. (Pozor, je velmi důležité rozlišovat mezi zahráním a vyvoláním. Zahrání je, když hráč zahraje kartu z ruky, vyvolání je, pokud hráč kartu vyvolá jiným způsobem, například pomocí jiné karty.)

„Deathrattle: <efekt X>“ - Pokud monstrum zemře, provede se X.

Taunt - Pokud je na oponentově straně monstum se schopností taunt, tak smí hráč jako validní cíl útoku zvolit pouze monstra s touto schopností.

Stealth – Protihráč nemůže vybrat monstrum jako cíl žádné svojí akce. Hráč, který má kontrolu nad monstrem se schopností stealth jej jako cíl zvolit může. Pokud monstrum zaútočí, ztratí stealth.

Windfury - Monstrum smí zaútočit dvakrát za kolo.

Charge - Monstrum smí zaútočit v kole kdy se objevilo na hrací ploše.

Frozen - Monstrum přijde o následující instanci útoku. Tedy, když zmrazím oponentovo monstrum, tak nebude útočit v příštím oponentově kole. Pokud bych si zmrazil své vlastní monstrum, tak pokud již zaútočilo, bude zmrzlé celé mé následující kolo. Pokud ne, bude zmrzlé jen v tomto kole.

2.5 Průběh útoku

Pokud hráč deklaruje útok nějakým svým monstrem, dojde automaticky k provedení následující sekvence.

Monstrum se deklaruje jako útočící s určitým cílem. Dojde ke zkontrolování efektů spouštěných pokud monstrum zaútočí. Pak následuje fáze, kterou jsme nazvali „clash.“ Při fázi clash dojde k tomu, že aktuální síla útoku útočníka se odečte od aktuálních životů obránce a naopak. Pak dojde ke zkontrolování efektů spouštěných, když monstrum ztratí životy. Následuje kontrola, zda-li některé z monster má 0 nebo méně životů. Pokud k tomu dojde, tak se na přeživších monstrech zkontroluje, jestli mají schopnosti spouštěné smrtí jiného monstra, která se následně provede. Na mrtvém monstru se případně spustí schopnost deathrattle. Tím skončí sekvence útoku a hra se opět přesune do základního stavu.

Existují i speciální případy, kdy se útok může provést jinak, než bylo původně zamýšleno. Může to být způsobeno některými schopnostmi monster nebo kouzel. Tyto případy však nemohou nastat s ohledem na množinu karet, které tato práce implementuje a tedy je zbytečné je nyní rozebírat a vysvětlovat.

2.6 Obecná pravidla hry a mechanik

Cílem hry je snížit počet životů oponenta na 0. Hra může skončit výhrou jednoho z hráčů, ale i remízou, pokud dojde k snížení životů obou hrdinů v důsledku zahrání jedné karty. Tedy například pokud by oba hráči měli 2 životy a hráč na tahu by zahrál kartu „Hellfire,“ která udělí 3 zranění všem charakterům, tak oba hrdinové zemřou a nikdo nevyhraje – hra skončí remízou.

Hráči se na začátku každého kola naplní manakrystaly manou a zvýší se jejich množství o jeden. Hráč získá každé kolo tímto způsobem jeden manakrystal, až do maxima 10 manakrystalů. Také dojde k odemčení schopnosti hráčova hrdiny. Hráč si také musí dobrat jednu kartu z herního balíčku. Pokud jeho herní balíček neobsahuje karty, dostane poškození podle hodnoty jeho fatigue. Tato hodnota roste lineárně o 1 zranění za každou neúspěšně dobranou kartu. Hráč smí mít v ruce nejvýše 10 karet a všechny další dobírané karty jsou bez náhrady odstraněny ze hry.

Hráč smí zahrát kartu z ruky, pokud jsou splněny základní podmínky. Musí mít dostatek many, který odpovídá ceně karty, kterou chce zahrát. Dále musí mít v případě kouzel validní cíl. Pokud tedy bude hráč chtít zahrát kartu „Sacrificial Pact,“ která má za efekt „znič démona a obnov 5 zranění svému hrdinovi“ a na herní ploše

nebude vyložena žádná karta s podtypem „démon,“ hráč tuto kartu nemůže zahrát. Ovšem v případě kouzel, která mají efekt aplikovaný na všechny validní cíle, smí hráč kouzlo zahrát a ono se provede bez efektu. Tedy například zahrání karty „Flamestrike,“ která „udělí 4 poškození všem nepřátelským monstrům,“ v momentě, kdy oponent nemá žádná monstra na stole, nebude mít žádný efekt, ale hráč ji smí zahrát. Monstra jsou omezena pouze velikostí hracího pole, které má kapacitu 7 monster pro každého z hráčů. Jejich battlecry se neprovede, pokud nemá validní cíl, což ale není překážka pro zahrání monstra.

S monstry smí hráč zaútočit jednou za kolo do libovolného validního cíle. Validní cíl je buď oponentův hrdina nebo jeho monstra. Dále pak jsou zde omezení logicky plynoucí z monster se schopnostmi taunt a stealth (viz. předchozí sekce 2.4). Monstrum nesmí zaútočit v kole, kdy bylo vyvoláno, pokud nemá schopnost charge. Poškození, která monstra i hráči obdrží, na nich zůstávají až do konce hry nebo do vyléčení. Hrdinu tedy lze vyléčit pouze do aktuálního maxima životů (typicky 30). Případné trvalé změny ve formě buffů životů monstra zvyšují aktuální i maximální počet životů o příslušnou hodnotu. Buffy se odstraní pokud je monstrum odstraněno z hrací plochy nebo pomocí efektu silence. Při odstranění buffu, který přidával monstru životy, dojde nejprve ke snížení maximálního počtu životů. Pokud by pak mělo monstrum mít více aktuálních životů než maximálních, dojde ke snížení jeho aktuálních životů na příslušnou hodnotu.

Pokud některá z hráčových akcí spustí řetězec reakcí, hra vyhodnotí tento řetězec a až poté umožní hráči zvolit další akci. Některé efekty karet pracují s náhodnými cíli. Příkladem budiž mágovo kouzlo „Arcane Missiles,“ které rozdělí 3 zranění mezi náhodné nepřátelské cíle. V tomto případě se po každém uděleném zranění kontroluje, jestli by daný cíl neměl zemřít. Pokud by došlo k tomu, že by první zranění snížilo životy monstra na 0, tak zbylé 2 budou rozděleny mezi zbývající cíle. Tedy nedojde k zbytečnému plýtvání na již zabitá monstra.

Hráč smí také jedenkrát za kolo využít schopnost svého hrdiny. Ta stojí standardně 2 many a liší se dle zvoleného hrdiny. Pro balíčky platí v podstatě 3 pravidla. Balíček smí obsahovat pouze karty příslušící odpovídajícímu hrdinovi a karty neutrální. Dále se v něm nesmí jedna karta vyskytovat více než dvakrát a pokud se jedná o kartu legendární kvality, tak ta se smí v balíčku vyskytovat pouze jedenkrát. Balíček musí mít 30 karet.

3 Related work

V této kapitole se podrobněji seznámíme s existujícími implementacemi simulátorů hry Hearthstone a dalšími programy, které se zabývají touto tematikou. Hlavním producentem těchto programů je skupina HearthSim[4]. Tato skupina se zabývá tvorbou a použitím simulátorů hry Hearthstone. Až na dvě vyjímky – Fireplace a SabberStone se nám nepodařilo najít žádný jiný funkční simulátor hry Hearthstone, který by byl udržován a aktualizován podle přidávaných karet.

3.1 Fireplace

Fireplace je první funkční simulátor hry Hearthstone, který byl naprogramován a uvolněn k širšímu využití. Jeho vývoj začal v srpnu 2014 a byl ukončen koncem roku 2016. Nemá naimplementovány všechny karty ani mechaniky. Jedná se o simulátor v jazyce Python, který nemá grafické rozhraní a s uživatelem komunikuje pouze pomocí konzole. Tento simulátor má řadu nedostatků, které se náš simulátor snaží odstranit. Jedním ze základních problémů je rychlost tohoto simulátoru. Konkrétní měření a data budou dále popsána v kapitole 7 – Experiment. Dalším z problémů je nízký uživatelský komfort a zejména absence grafického rozhraní. Fireplace je nejstarší z testovaných simulátorů. Jeho výstup je čistě konzolový. Ačkoliv byl vyvíjen dlouho, přesto má tento simulátor jen relativně malou uživatelskou příručku. Uživatel je odkázán na několik krátkých článků na internetu, které popisují jednotlivé úkony a fungování simulátoru. Ke svému správnému fungování potřebuje, aby uživatelův Python obsahoval knihovnu Hearthstone. V té je dostupná databáze karet ve formátu XML dokumentu, jak byla extrahovaná z oficiálního klienta hry Hearthstone. Pozitivní aspekt tohoto přístupu je, že uživatel má k dispozici velké množství karet. Nevýhodou je, že přidávání vlastních karet není příliš intuitivní a je relativně náročné a zdlouhavé. Zásadním problémem je, že přestal být dále vyvíjen a udržován.

3.2 Hearthbreaker

Hearthbreaker[5] je další simulátor v jazyce Python 3. Jedná se o jeden z původních projektů HearthSim. Tento simulátor byl využit DeepMind týmem společnosti Google pro tvorbu nových karet. (Ling et al, 2015) Cílem jejich experimentu bylo zjistit, zda je systém využívající strojové učení schopen

vygenerovat kód karty, jejíž parametry mu byly zadány. Ukázalo se, že ačkoliv některé karty byly vygenerovány správně, ve většině případů byla karta generována chybně. Je důležité zmínit, že karty, které byly implementovány správně, byly v podstatě editované kopie již existujících karet, na kterých se neuronová síť učila. Chyby byly obecně dělány při generování kódu efektů karet, zatímco název a případné atributy jako například útok nebo cena seslání byly generovány správně. Hearthbreaker obsahuje implementaci většiny karet až do edice Black Rock Mountain. Tento projekt však není nadále vyvíjen a byl opuštěn.

3.3 HearthShroud

HearthShroud[6] je simulátor hry Hearthstone napsaný v Haskellu. Nabízí konzolové rozhraní. Jeho projekt není již dále udržován.

3.4 SabberStone

SabberStone je simulátor, jehož vývoj začal na přelomu 2016/2017, tedy již po zadání naší práce. Jeho vznik byl zapříčiněn absencí rychlého simulátoru, který by umožnil implementaci rychlých AI. Je napsán v C# Net Core a celkově je velmi podobný našemu programu, co se záměru jeho tvorby týče. Jelikož je to jediný další funkční simulátor, budeme se i s ním porovnávat, co se výkonnosti týče. Sabberstone je nejmladší simulátor hry Hearthstone, který byl vyvinut jako náhrada nepostačujícího simulátoru Fireplace skupinou HearthSim. Jeho hlavními přednostmi jsou rozšířené možnosti pro uživatele a vyšší rychlost simulace. Uživateli umožňuje klonovat herní stav v libovolném momentu simulace. Jeho grafické rozhraní je v současné době ve vývoji, ale simulace her v konzolovém režimu je plně funkční. Stejně jako Fireplace využívá databázi karet extrahovanou z herního klienta Hearthstone. Jelikož se jedná o stejný XML dokument jako v případě simulátoru Fireplace, jsou benefity i negativa tohoto přístupu stejné. Detailnější analýza tohoto simulátoru bude zmíněna v kapitole 7 – Experiment..

4 Analýza problému a cílů

V této kapitole budeme popisovat, pro jaké postupy a návrhy jsme se při tvorbě našeho simulátoru rozhodli a proč. V rámci analýzy námi vytyčených cílů projdeme návrhem zdrojového XML dokumentu i návrhem vlastního simulátoru.

4.1 Vytvořit uživatelsky přívětivé UI, které uživateli umožní, aby se zaměřil na tvorbu umělé inteligence nebo vlastních karet.

Přívětivé uživatelské rozhraní se v našem případě skládalo ze dvou hlavních komponent. První je vlastní okno grafické aplikace, druhé je rozhraní, přes které uživatel komunikuje s programem. Tyto dva podcíle jsme zvolili z důvodu, že je zapotřebí poskytnout komfort jak uživateli obsluhujícímu grafickou aplikaci, tak uživateli, který se rozhodne vytvářet vlastní umělou inteligenci a bude tedy potřebovat komunikovat s uživatelským rozhraním simulátoru.

Za účelem poskytnutí přívětivého uživatelského rozhraní jsme se rozhodli umožnit uživateli, aby mohl využívat funkce, kterými simulátor manipuluje s herním stavem. Také návrh rozhraní pro umělou inteligenci byl určen co nejjednodušeji – uživatelova inteligence musí pouze implementovat rozhraní, které po ni vyžaduje, aby na otázku (při zavolání příslušné metody) „Tady máš současný herní stav, co chceš dělat?“ odpověděla pomocí jedné z předdefinovaných odpovědí – akcí.

Aby byla vizuální aplikace uživatelsky přívětivá pro hráče hry Hearthstone, rozhodli jsme se co nejvíce přiblížit designu původní hry. V původní hře jsou ovšem elementy, které by značně ztížily, či zhola znemožnily provedení některých zamýšlených funkcí našeho programu. Jedná se převážně o fakt, že v originále hráč hraje karty způsobem „drag and drop.“ Tento přístup je sice pro hráče pohodlný, ale v našem případě by znemožnil rozdělení akce zahrání karty na dvě akce – vybrání karty a vybrání cíle. Proto jsme se rozhodli v našem návrhu grafické aplikace přistoupit k řešení, ve kterém hráč volí své akce postupně.

V ostatních ohledech napodobujeme rozložení z hry Hearthstone, aby se i nový uživatel mohl snadno zorientovat v ovládacích prvcích našeho programu. Jsou zde některé rozdíly, způsobené naším minimalistickým návrhem. Primárně se jedná o zobrazování detailů karet, které hráč může vybrat. Ty se zobrazují ve speciálních polích pod hrací plochou. Stejně tak se karty, které hráč může vybrat, zobrazují ve speciálních seznamech. Tento přístup jsme zvolili z následujícího důvodu. Jedním z důležitých benefitů vlastního simulátoru je, jak bude v následujícím textu detailněji vysvětleno, že hráči nabízí seznam karet, které může zahrát a po zvolení jedné z těchto karet i karty, které mohou být validními cíli této vybrané karty. Tato feature se promítla do designu vizuální aplikace ve formě výše zmíněných dvou polí, které obsahují karty, které hráč může zahrát a karty, které může zvolit jako cíl konkrétní zahrané karty. Svou volbu pak musí potvrdit stisknutím tlačítka „Confirm.“ Pro

usnadnění ovládání jsme zvolili i možnost vybírat karty přímo v polích odpovídajících hráčově ruce a herní ploše. V důsledku tohoto rozhodnutí jsme však museli zvolit vhodný přístup pro rozlišení volby cíle dané karty. Existuje totiž situace, kdy hráč může jednu kartu zvolit zároveň jako kartu, se kterou chce provést akci, ale také jako cíl jiné své karty. Jedná se typicky o situaci, kdy hráč má na stole monstrum, které může zaútočit a on si zvolí kartu z ruky – například kouzlo, které umožňuje udělit zranění monstru. V tomto okamžiku má hráč stále možnost vybrat libovolnou jinou kartu ze seznamu hratelných karet – tedy i monstrum na stole. Ovšem jak rozlišit, zda chce své monstrum na stole zvolit jako cíl svého kouzla nebo zda změnil názor a místo kouzla z ruky chce zaútočit tímto monstrem? Proto jsme se rozhodli upustit od volby cíle pomocí vybírání daných karet na herní ploše a místo toho jsme se rozhodli využít následující konvenci: uživatel si může vybrat kartu k zahrání jak ze seznamu hratelných karet, tak přímo z herní plochy. Případný cíl dané karty ovšem musí zvolit pouze v seznamu validních cílů. Ve hře Hearthstone je tento problém eliminován tím, že pokud hráč vybere jednu ze svých karet, přepne se do režimu vybírání cíle a jeho zvolením okamžitě dochází k zahrání karty. My jsme se rozhodli přesunout pomyslné potvrzení volby z „vyberu kartu co chci zahrát“ za „vyberu cíl své aktuálně zvolené karty.“ K tomuto jsme dospěli úvahou, že hráč má tendenci častěji měnit kartu, kterou chce zahrát, tedy by mačkání potvrzení po zvolení karty bylo nepraktické, zejména proto, že by v případě, že změní názor, musel zmáčknout jiné tlačítko, kterým by svou volbu zrušil. Náš přístup tedy kompletně eliminuje nutnost existence tlačítka „zruš volbu.“

4.2 Umožnit snadnou implementaci nových karet a mechanik.

V případě obou konkurenčních simulátorů byl jeden velký nedostatek v oblasti implementace nových karet a modifikace existujících.. Oba měli karty i jejich efekty natvrdo zabudované ve svém kódu. Případná tvorba nové karty nebo editace stávající karty by vyžadovala od uživatele aby upravil kód programu. To není žádoucí, neboť cílovým uživatelem může být i hráč, který nemá zájem/schopnost programovat, ale pouze by chtěl s využitím stávajících mechanik vytvořit či editovat stávající kartu. Alternativně, uživatel by mohl chtít simulovat velké množství her, kde by testoval chování jedné konkrétní karty s ohledem na to, jak by se měnily její parametry. Pokud by využil konkurenční simulátory, musel by danou kartu implementovat ve všech možných kombinacích konfigurací a následně využít v simulacích jen tu, kterou by zrovna testoval. V našem případě by ovšem bylo možné mít jediný template dané karty v databázi a pak již jen v rámci testovacího skriptu měnit parametry této karty pro jednotlivá testování. Rozhodli jsme se tedy tento požadavek reflektovat v našem návrhu karetní databáze a efektů karet. Uživatel vytvoří kartu tak, že ji popíše ve zdrojovém XML dokumentu. Simulátor pak při spuštění programu načte všechny karty. Odpadá tak nutnost rekompilovat a upravovat celý program pro úpravu každé jednotlivé karty.

4.2.1 Databáze a návrh karet

Prvním úkolem bylo rozhodnout, jak chceme reprezentovat karetní databázi. Rozhodli jsme se zvolit uživatelsky přívětivou formu XML dokumentu, která bude z velké části nezávislá na vlastním kódu programu. Důvodem pro tuto volbu byl důraz na to, aby uživatel měl možnost editovat a tvořit karty, aniž by přitom musel opětovně kompilovat simulátor. Analyzovali jsme jednotlivé karty, které se vyskytují ve hře Hearthstone. Zvláštní důraz jsme kladli na karty, které jsme v rámci rozsahu naší práce měli implementovat. Takto jsme zjistili, že karty mají určité společné rysy. Prakticky všechny entity ve hře jsou určitá obdoba objektu karta. Jedná se zejména o typ karty – monstrum, kouzlo, zbraň, hrdina – a následně tedy o případný počet životů, sílu útoku a cenu za zahrání. Druhá významná spojitost je schopnost dané karty. Ty jsme rozdělili na schopnosti určené klíčovými slovy a na schopnosti určené textem. Schopnosti určené klíčovými slovy byly popsány v druhé kapitole, která pojednává o pravidlech. Jedná se o schopnosti jako například taunt, charge atd. Tyto schopnosti jsme se rozhodli reprezentovat pomocí takzvaných tagů. Tag je označení, které umožňuje třídit karty. Jednotlivé schopnosti se na tyto tagy mohou odvolávat. Schopnosti určené textem jsou o poznání složitější na implmentaci. Analýza karet ukázala, že drtivá většina těchto schopností využívá konečnou a rozumně velkou množinu mechanik. Jedná se například o dobrání karty, udělení zranění či zničení monstra. Tyto mechaniky jsme se proto rozhodli implementovat v rámci kódu programu a v rámci XML dokumentu se na ně pouze odkazovat. V rámci všech karet, které jsou ve hře Hearthstone k dispozici, je malé množství karet, které mají unikátní schopnosti. Tyto karty by musely mít tyto schopnosti naprogramované v kódu jen pro sebe. Nicméně žádná z těchto karet není implementována v rámci množiny karet, kterou jsme implementovali v rámci našeho simulátoru.[1] Pokud jde o schopnosti, které jsou určeny textem, bylo nutno určit, jak tyto schopnosti budou určovat svůj cíl. Tento problém je zjednodušen faktem, že jediná možnost, kdy hráč může vybrat cíl pro kartu, je okamžik, kdy danou kartu hraje. Pokud tedy hraná karta vyžaduje cíl zvolený hráčem, první z případných mnoha schopností karty určuje pomocí svých tagů, jaké cíle jsou přípustné. Ostatní schopnosti jsou již vázány buď na cíl určený hráčem nebo na cíle určené pomocí svých vlastních tagů. Příkladem budiž karta

1 Nicméně díky způsobu, jakým náš simulátor implementuje karty je dodatečné přidání těchto unikátních schopností možné a o nic více obtížné než přidání kterékoliv jiné zcela nové mechaniky – viz. Implementace Totemic Call.

„Sacrificial Pact“ (Kód 1). Ve zbytku této podkapitoly budeme odkazovat na řádky výňatku z kódu formou (ř. 1).

```
0<Card>
1    <Name>Sacrificial Pact</Name>
2    <Attack/>
3    <HP/>
4    <Manacost>0</Manacost>
5    <Tags>
6      <Spell/>
7    </Tags>
8    <Skills>
9      <Skill>
10       <Type>generic</Type>
11       <Target>
12         <Tags>
13           <Demon/>
14         </Tags>
15       </Target>
16       <Effect>
17                          <Function   targets="1"   selector="PLAYER"   value
="1">Destroy</Function>
18       </Effect>
19     </Skill>
20     <Skill>
21       <Type>generic</Type>
22       <Target>
23         <Tags>
24           <Own/>
25           <Player/>
26         </Tags>
27       </Target>
28       <Effect>
29         <Function targets="1" selector="ALL" value="5">Heal</Function>
30       </Effect>
31     </Skill>
32   </Skills>
33   <TargetNeeded>YES</TargetNeeded>
34 <Targetable>YES</Targetable>
35 </Card>
```

Kód 1 – reprezentace karty Sacrificial Pact v XML databázi

Jedná se o kouzlo, které má následující efekty: znič démona (ř. 17) a vyleč svého hrdinu o 5 životů (ř. 29). Schopnost karty využívá dvou efektů – znič a vyleč. Každý z těchto efektů ovšem cílí na jiný objekt. Zničen má být démon, ale vyléčen hráčův hrdina. Proto má tato karta dvě schopnosti, každá z nich volající jinou z implementovaných funkcí a mající jiné cíle (ř. 9 a ř. 20). První schopnost ničí démona. Aby hráč mohl kartu zahrát, musí mít zvolený validní cíl. Tím je libovolná karta, která obsahuje tag „demon“ (ř. 13). Pokud tento požadavek splní, kartu lze zahrát a první schopnost zničí démona, zatímco druhá si jako svůj cíl zvolí vlastního hrdinu (ř. 24 a ř. 25) a vyléčí jej o 5 životů. Naším přístupem jsme tedy umožnili uživateli, aby si tyto schopnosti modifikoval (změnou počtu a typu funkcí, jejich parametrů a atributů), dle svých přání a představ, aniž by kvůli tomu musel, jako je tomu v ostatních simulátorech, zasahovat do kódu programu. Tvorba nových karet je také velmi jednoduchá, neboť se na karty program odkazuje pomocí jejich jména (ř. 1), které slouží, jako unikátní identifikátor.

4.2.2 Návrh a implementace nových efektů karet

Jak již bylo nastíněno výše v podkapitole o návrhu karet (4.2.1), efekty karet jsme navrhli pomocí několika základních funkcí. Ty jsou určeny k manipulaci s herním stavem. Nyní ovšem zbývá rozebrat situaci, kdy uživatel chce implementovat naprosto odlišnou a novou mechaniku, která v současné implementaci není podporována. Někdy je také žádoucí sloučit více mechanik do jedné konkrétní, pokud by daný efekt získal četnější zastoupení v rámci karetní databáze. Tento postup praktikuje i samotný Hearthstone. V poslední expanzi došlo k zavedení nového klíčového slova „Lifesteal“ a celkové revizi této mechaniky. (Tato mechanika není v naší množině implementovaných karet zastoupena.) Nicméně uživatel by tedy v podobném případě musel sáhnout k editaci vlastního kódu. Technické detaily jsou popsány v následující kapitole „Uživatelská dokumentace.“ Editace by spočívala v napsání konkrétního kódu daného efektu v souboru „CardFunctions.cs“ a doplnění tohoto efektu do metody „ParseFunction“ v souboru „CardLoader.cs.“ V tomto okamžiku by daný efekt mohl být použit jako funkce v rámci XML struktury karty. Jelikož v rámci implementace efektu má uživatel již přístup k funkcím modifikujícím herní stav, je tento krok již relativně snadný a pohodlný.

4.3 Umožnit snadné simulování velkého množství her.

Tento cíl jsme si vytyčili s ohledem na to, jak dlouho trvá simulace jedné náhodné hry v konkurenčních simulátorech. Opět se jedná o problematiku, která se dá rozdělit na několik drobnějších úkolů. Jednak se jedná o návrh vlastního simulátoru a jednak jde o design rozhraní pro spouštění vlastních simulací. V tomto pořadí tyto problémy nyní rozebereme.

4.3.1 Simulátor a herní stav

Herní stav

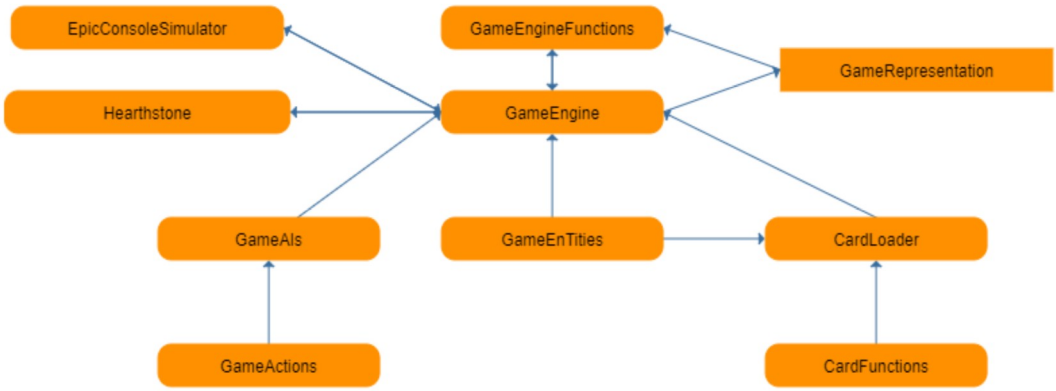
Základním prvkem našeho simulátoru je herní stav. Jedná se o třídu, která popisuje aktuální stav hry. Obsahuje všechny informace, které jsou o hře k dispozici. Umožňuje být klonován, díky čemuž si umělá inteligence může simulovat vlastní průběhy hry a výpočty, aniž by přitom ovlivnila skutečný herní stav. Herní stav je stav hry v okamžiku, kdy se čeká na akci hráče neboi umělou inteligenci (dále AI). Po zvolení akce dojde k řetězovému volání funkcí, které vyústí buď opět v návrat do stavu čekání na akci nebo v konec hry. Protože jsou všechny informace o hře obsaženy v jednom objektu, je snadné předávat a modifikovat tyto data.

Simulátor

Abychom oddělili řízení simulace od vlastních funkcí modifikujících herní stav, rozhodli jsme se tuto myšlenku reflektovat v návrhu simulátoru. Ten tvoří dvě hlavní třídy. Jedná se o `GameEngine` a `GameEngineFunctions`. První z nich řídí vlastní simulaci. Má jen několik primárních úkolů. Jedním z nich je zajištění funkce grafického rozhraní. V původním návrhu jsme v rámci této třídy měli implementovány i metody, které operovaly s herním stavem. Dospěli jsme však k závěru, že je výhodnější mít oddělenou třídu, která bude implementovat veškeré operace prováděné na herním stavu. Takto každá instance AI dostane předán odkaz na instanci herního stavu. Sama si pak vytvoří vlastní instanci třídy `GameEngineFunctions`, kterou následně může využívat pro efektivní získávání informací o herním stavu a jeho případném vývoji. Třída `GameEngineFunctions` obsahuje všechny nástroje pro manipulaci s herním stavem i pro simulaci průběhu jedné hry.

Závislosti jednotlivých modulů jsou znázorněny v následujícím diagramu (Obr. 6). `GameEngine` řídí simulaci a využívá k tomu `GameEngineFunctions`.

Pro načtení karetní databáze využívá `CardLoader`. Ten využívá `CardFunctions` pro správné načítání karet. Umělé inteligence načte `GameEngine` ze souboru `GameAIs`. Ty využívají pro komunikaci s programem instance tříd obsažených v `GameActions`. Veškeré herní entity jsou obsaženy v souboru `GameEntities`. Ten využívá jak `CardLoader` tak `GameEngine`. `GameEngine` vytvoří instanci třídy `GameRepresentation`, se kterou prak pracuje `GameEngineFunctions`.



Obr. 6: Závislosti jednotlivých komponent programu

Při návrhu simulátoru zvažovali různé možnosti přístupu k simulaci průběhu hry. Zvažovali jsme přístup podobný tomu, který využívá Sabberstone. Tedy jakýsi zásobník, do kterého se přidávají jednotlivé události a ty se postupně vyhodnocují. Po analýze průběhu hry jsme se rozhodli implementovat současný systém. Namísto zásobníku jsou jednotlivé metody volány řetězově a samy si kontrolují důležité události. Jsme tedy limitováni velikostí zásobníku volání, která je ovšem s ohledem na potenciální množství zřetězených akcí více než dostatečná.[2] Tento přístup ideově vychází z přístupu použitého při návrhu implementace karet. Existuje jen malé omezené množství akcí, které interagují s herním stavem. Příkladem budiž dobrání karty. V našem případě by samotná metoda, která dobírá kartu, zkontrolovala zda mezi monstry na hrací ploše není některé, které by interagovalo s dobíráním karet. V námi implementované množině karet ovšem nejsou karty, které by interagovaly se hrou mimo okamžiku jejich zahrání. Udržování jednoho herního stavu usnadňuje

2 Ve hře Hearthstone existují situace, které by dle popisu karet měly způsobit nekonečné zacyklení akcí, ovšem tento problém je řešen nastavením neviditelných limitací. Příkladem budiž karta Bouncing Blade, která uděluje zranění do náhodného monstra, dokud nějaké nezemře. Ve skutečnosti ovšem udělí maximálně 80 zranění.

jeho klonování. Herní stav je vždy po skončení řetězu funkcí v základním stavu, kdy čeká na další hráčovu akci. Případné odebírání zrušených akcí ze zásobníku je v našem řešení provedeno kontrolou validity prováděné akce v průběhu řetězu jejího provádění.

4.3.2 Rozhraní pro simulace

Aby uživatel mohl pohodlně simulovat hry v rámci konzolové aplikace, je GameEngine navržen tak, že uživatel vytvoří jeho instanci, s konkrétním nastavením aktuálního herního stavu, a ta pak uživateli nabízí možnost se resetovat do tohoto základního stavu. Dále pak umožní uživateli, v případě, že proti sobě nechá hrát dvě umělé inteligence, odsimulovat danou hru a vrátí mu výsledek této hry. Uživatel tak může provést požadované množství volání metody „`SimulateOneGame`“ a tak docílit odehrání požadovaného množství simulací. Technické detaily provádění simulace jsou popsány v následující kapitole - „Uživatelská dokumentace.“

5 Uživatelská dokumentace

Tato kapitola popisuje uživatelskou dokumentaci a způsoby, jakými uživatel může pracovat s tímto simulátorem. Je rozdělena do několika segmentů, které odpovídají jednotlivým logickým blokům programu. První blok popisuje konzolovou aplikaci a způsob práce s jejím vstupem a výstupem. V druhém segmentu se čtenář seznámí s grafickým rozhraním určeným pro ladění uživatelem vytvořených umělých inteligencí. Třetí segment popisuje jak může uživatel vytvořit vlastní karty a mechaniky, které by chtěl otestovat. Závěrečný čtvrtý segment popisuje postupy nutné pro tvorbu a implementaci vlastního agenta umělé inteligence.

5.1 Požadavky ke spuštění

Pro spuštění konzolové aplikace nejsou žádné speciální požadavky. Pokud chce uživatel programovat vlastní umělou inteligenci, bude potřebovat Microsoft Visual Studio 2015 nebo vyšší. Pro implementaci nových karet je zapotřebí pouze libovolný editor XML dokumentů.

5.2 Konzolová aplikace

Konzolová aplikace (dále jen aplikace/program) slouží k analyzování velkého počtu her a je primárně určena jako nástroj pro tvorbu optimálních herních balíčků pro danou umělou inteligenci.

5.2.1 Obsluha aplikace

Uživatel má dvě možnosti, jak aplikaci obsluhovat. Buď může zadat přímo parametry při spuštění nebo může informace poskytnout po spuštění aplikace.Když uživatel spustí aplikaci, ta s ním začne komunikovat pomocí standardního výstupu na konzoli. Příklad vstupních parametrů:

```
EpicConsoleSimulator.exe 10000 basic_mage.txt
basic_mage.txt RandomAI RandomAI Yes
```

obecně

```
EpicConsoleSimulator.exe #her balíček1 balíček2 AI1 AI2
Log
```

Pokud by hráč zadal neplatný počet parametrů nebo nezadal žádné bude na ně aplikací dotázán. Nejprve aby poskytl počet her (#her), které chce nechat odsimulovat, následně musí zadat název souboru obsahující seznam karet pro prvního hráče (balíček1), název souboru obsahující seznam karet pro druhého hráče (balíček 2), název AI hrající za prvního hráče (AI1), název AI hrající za druhého hráče (AI2) a závěrem jestli chce, aby se průběh hry vypisoval na konzoli (Log) ve formátu Yes/No. Tyto údaje může uživatel zadávat i formou argumentů při spouštění aplikace. Jako je tomu v případě výše.

5.2.2 Interpretace výstupu

Aplikace v každém případě poskytne výstup v následujícím formátu (Kód 2). Opět se budeme odkazovat na jednotlivé řádky formou (ř. 1) atd.

```
1 Simulator started in Console Mode
2 Performance measurement started.
3 Total Time Elapsed: 00:00:03.7212880
4 Final score after 10000 simulations:
5 6909/3091/0
6 basic_mage.txt + RandomAI vs basic_mage.txt + RandomAI
7
8 00:00:03.7475693 total
9 00:00:00.8736308 preparation
10 00:00:02.8669029 gametime
11 00:00:00.0070360 difference
12
13 155371 total turns elapsed
```

Kód 2

konzolová aplikace nejprve oznámí začátek simulace (ř 2). Po jejím skončení oznámí celkový čas, který simulace zabrala (ř. 3). Následuje rekapitulace, kolik her bylo odehráno (ř. 4). Na následujícím řádku (ř. 5) se vypíše počet výher prvního hráče (p1w), počet výher druhého hráče (p2w) a počet her, které skončily nerozhodně remízou. Formát tohoto řádku je p1w/p2w/remízy. Další řádek (ř. 6) obsahuje

rekapitulaci kombinací balíčků a agentů, kteří hráli proti sobě v této simulaci. Poté je vypsán detailnější rozbor času simulace. Celkový čas (ř. 8), čas strávený přípravou simulace (ř. 9) – tedy načítáním karet z databáze, iniciací herního stavu před každou hrou a úklidem po hře. Čas strávený čistě v herní smyčce je na následujícím řádku (ř. 10) označen jako gametime. Poslední řádek (ř. 11) spíše pro kontrolu udává časový rozdíl mezi celkovou dobou simulace a součtu přípravy a herního času. Je to v podstatě čas, který zabrala vlastní obsluha kódu simulace. Poslední řádek (ř. 13) informuje uživatele, kolik kol celkem simulace trvala. Toto číslo udává počet kol obou hráčů, takže pokud bychom chtěli mít tradiční pojem herního kola jakožto dvojice hráčova a oponentova tahu, bylo by nutné toto číslo vydělit dvěma.

5.2.3 Formát vstupních souborů

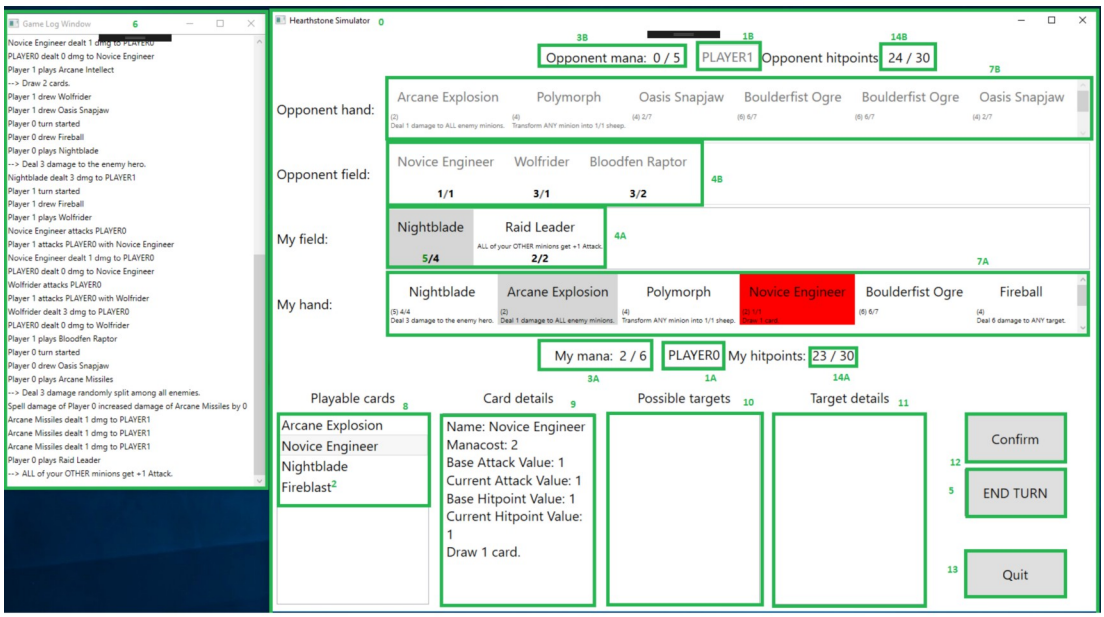
Hráč musí poskytnout seznam karet pro balíčky, jejichž hry má program simulovat. Tyto seznamy musí hráč poskytnout ve formě textového souboru. Soubor musí splňovat předepsaný formát. První řádek musí obsahovat jméno hrdiny (v současné době je podporován „Mage," „Shaman" a „Warlock"). Následující řádky musí obsahovat po jednom jménu karty, kterou chce hráč mít obsaženou v balíčku. Hráč může zvolit libovolné celkové množství karet. Není tedy v tomto případě limitován pravidly, podle kterých by balíček měl obsahovat přesně 30 karet.

Název AI musí korespondovat s AI, která je v programu implementována. V základu jsou implementovány dvě jednoduché AI - „RandomAI" a „FaceHunterAI." Tyto slouží k demonstraci toho, jak může AI vypadat, aby si začínající uživatel mohl snáze implementovat svou vlastní. Pokud by tak chtěl učinit, musí do souboru `GameAIs` připsat vlastní třídu – potomka `GenericAI`, která by implementovala vlastní metodu „`getAction`." Poté do metody třídy „`AISelector`" v témže souboru připsat rozpoznání právě vytvořené AI do metody „`GetAI`."

5.3 Vizuální aplikace pro ladění

Vizuální aplikace (dále jen aplikace/program) slouží k ladění umělé inteligence. Uživatel zde může hrát proti umělé inteligenci (ale i sám proti sobě) a sledovat její reakce v závislosti na jeho tazích. Obrázek (Obr. 7) zobrazuje uživatelské rozhraní v průběhu hry. Obrázek je v plném rozlišení k dispozici v

příloze. Následující podkapitola (5.3.1) se bude odkazovat na popisky obrázku (Obr. 7) tímto způsobem: „(1A)“.



Obr. 7 Zobrazení vizuální aplikace našeho simulátoru

0 – Okno vizuální aplikace; 1 – Avatar hrdiny hráče (a) a oponenta (b); 2 – Schopnost hrdiny hráče; 3 – Mana krystaly – hráče (a) a oponenta (b) ; 4 – Hráčova monstra (a) a oponentova monstra (b); 5 – Tlačítko konec kola; 6 – Historie zahraných karet a událostí; 7 – karty na ruce hráče (a) a oponenta (b); 8 – Seznam hratelných karet a monster, která mohou útočit; 9 – Detaily vybrané karty; 10 – Seznam validních cílů zvolené karty; 11 – Detaily zvoleného cíle; 12 – Potvrzovací tlačítko; 13 Tlačítko pro vypnutí hry; 14 – životy hráče (a) a oponenta (b)

5.3.1 Obsluha aplikace

V základu je nastavena hra uživatele proti AI FaceHunterAI s náhodnými balíčky, kdy bude uživatel hrát proti AI celou hru od začátku do konce. K tomu má k dispozici jednoduchou vizualizaci herního plánu. Ta kopíruje předlohu originálního herního plánu hry Hearthstone. K dispozici má několik ovládacích prvků. Tlačítko pro ukončení tahu (5), kterým může ukončit svůj tah. Také má k dispozici pole, kde mu hra vypíše všechny karty, které může vybrat a zahrát (8), případně i monstra, se kterými může zaútočit. Karty, které lze zahrát jsou zvýrazněny i v rámci herního plánu. Pokud uživatel jednu takovou kartu vybere, ať již kliknutím na kartu na herním plánu či vybráním položky seznamu hratelných karet, ve vedlejším listu se

mu zobrazí všechny validní cíle (10). Existuje i možnost, že toto pole bude prázdné a uživatel tak nebude mít co vybrat. V obou případech se po vybrání cíle, který však již musí zvolit pouze ze seznamu validních cílů, případně po vybrání karty, která nevyžaduje cíl, odemkne tlačítko „Confirm“ (12), jehož stisknutím hráč potvrdí svou volbu a hra zahraje příslušnou kartu. V průběhu těchto akcí se všechny vybrané karty zvýrazňují v seznamech karet, které znázorňují karty na hráčově ruce(7A,7B) a monstra na herní ploše (4A,4B). Toto je podstatné, protože karty jsou zobrazeny jen pomocí svého názvu a pokud by hráč měl na ruce i na stole kartu monstra se stejným názvem, která by měla schopnost „battlecry – uděl 1 zranění do nepřátelského cíle,“ tak by bez tohoto zvýraznění hráč nemohl vědět, zda vybírá kartu z ruky, jehož schopností chce udělit 1 zranění do oponenta nebo monstrum na hrací ploše, kterým chce zaútočit. Poslední aspekt grafického rozhraní jsou rozšířené informace o kartách. Pokud hráč vybere kartu, její rozšířené informace se vypíší pod listem, ze kterého byla vybrána. Tyto informace zobrazují současný i maximální počet životů a sílu útoku. Veškeré informace o průběhu hry jsou dále poskytovány v okně záznamu hry (6), které uživateli poskytuje přehledné informace o jednotlivých akcích, které byly v průběhu hry prováděny. Hráč může kdykoliv ukončit hru stisknutím tlačítka „Quit“ (13).

Uživatel má možnost modifikovat počáteční herní stav pomocí souboru „GameState.xml“ (Kód 3)

```
<GameStates>
  <GameState>
    <Use>False</Use>
    <P1AI>Random</P1AI>
    <Player0>
      <HPMax>30</HPMax>
      <HPCurrent>20</HPCurrent>
      <ManaTotal>5</ManaTotal>
      <ManaCurrent>3</ManaCurrent>
       <Fatigue>0</Fatigue>
       <Deck>
         <CardName>Arcane Missiles</CardName>
         <CardName>Arcane Missiles</CardName>
       </Deck>
       <Hand>
         <CardName>Fireball</CardName>
         <CardName>Nightblade</CardName>
```

```
18        <CardName>Wolfrider</CardName>
19      </Hand>
20      <Field>
21        <Card>
22          <CardName>Murloc Raider</CardName>
23          <CurrHP>3</CurrHP>
24          <CurrAT>6</CurrAT>
25          <MaxHP>7</MaxHP>
26          <MaxAT>9</MaxAT>
27        </Card>
28      </Field>
29    </Player0>
30    <Player1>
31      ...<!-- stejný formát jako Player0 -->
32    </Player1>
33    <Actions>
34      <Play>
35        <CardName>Fireball</CardName>
36        <TargetPosition>-1</TargetPosition>
37        <Owner>0</Owner>
38      </Play>
39    </Actions>
40  </GameState>
41</GameStates>
```

Kód 3 – Formát souboru GameState.xml popisující stav hry k načtení

V něm lze popsat situaci, ve které se bude hra nacházet po spuštění programu. Nastavení tohoto souboru je relativně přímočaré. To, zda se má hra pokusit načíst daný stav určuje element „Use“ (ř. 3), kde hodnota „True“ způsobí, že se hra o načtení pokusí, zatímco ostatní hodnoty způsobí standardní start hry. Element „GameState“ (ř. 2) obsahuje dva Elementy popisující herní plochu a karty na ruce obou hráčů (ř. 5 a ř. 30). Hráč musí nastavit maximální (ř. 6) a aktuální životy (a ř.7) a manakrystaly hráče (ř. 8 a ř. 9). Také jeho úroveň fatigue (ř. 10). Element „Deck“ (ř. 11) obsahuje seznam karet (ř. 12 a ř. 13), které se mají odebrat z balíčku karet, který je určen v textovém souboru, jak bylo popsáno výše (kapitola 5.2.3). Element „Hand“ (ř. 15) obsahuje jména (ř. 16 – ř. 19) karet, které mají být umístěny do ruky daného hráče. Poslední element se jmenuje „Field“ (ř. 20) a může obsahovat elementy „Card.“ (ř. 21) ty se svým formátem liší od elementu „Card“ v souboru „XMLCardBase.xml.“ V tomto případě obsahuje element „Card“ pouze název karty,

kterou chceme, aby měl hráč na své herní ploše (ř. 22) a její aktuální a maximální hodnoty útoku (ř. 23 a ř. 24) a životů (ř. 25 a ř. 26). Tímto jsme vyčerpali všechny elementy elementu „Player0“ a „Player1.“ Zbývá nám poslední element „Actions“ (ř. 33). Ten popisuje, jaké karty chceme zahrát, abychom dále modifikovali herní plochu. V současné době podporuje karty typu „Spell.“ Uživatel tedy v našem případě chce, aby se před začátkem hry seslalo kouzlo „Fireball.“ Element „Actions“ obsahuje elementy „Play“ (ř. 34) .Každý z těchto elementů reprezentuje jednu zahranou kartu. Ta je určena pomocí svého názvu (ř. 35). Dále určuje, který hráč ji hraje, což je důležité pro karty, které ovlivňují pouze například oponentovy karty, pomocí elementu „Owner“ (ř. 35). Hodnota 0 znamená Player0 a 1 znamená Player1. Poslední element se jmenuje „TargetPosition“ (ř. 36). jeho hodnotou je číslo, které určuje pozici cíle. Protože máme k dispozici 2 pole, 2 hráče a možnost žádného cíle, zvolili jsme následující postup. Hodnota 0 znamená žádný cíl, hodnota 1 znamená Player0 a další hodnoty znamenají pořadí monster na hrací ploše hráče Player0. Záporné hodnoty mají stejný význam, jen pro hráče Player1.

5.4 Tvorba vlastních karet[3]

Klíčovým aspektem simulátoru je i snadná implementace a tvorba vlastních karet. Uživatel k tomu potřebuje pouze modifikovat soubor `XMLCardBase`.xml. Tento soubor obsahuje všechny karty, se kterými program pracuje. Jeho uspořádání je následující. Kořenový element „Cards“ má pod sebou jen libovolné množství jednotlivých karet – elementů „card.“ Karta je určena pomocí několika základních elementů. Jméno, útok, životy, cena za seslání a v případě, že se jedná o kouzlo, zda vyžaduje cíl k zahrání. Také obsahuje seznam takzvaných tagů. Ty slouží k snadné identifikaci a třídění karet. Vezměme příklad následující karty (Kód 4):

```
1<Card>
2    <Name>Arcane Missiles</Name>
3    <Attack/>
4    <HP/>
5    <Manacost>1</Manacost>
6    <Tags>
```

3V tomto podbodu (5.4) budeme referencovat příslušné řádky jednotlivých kusů kódu (Kód X) formou řádek 1 ~ (ř. 1).

```
7       <Spell/>
8     </Tags>
9     <Skills>
10      <Skill>
11        <Type>Generic</Type>
12        <Target>
13          <Tags>
14            <Enemy/>
15          </Tags>
16        </Target>
17        <Effect>
18                        <Function  targets="3"  selector="RANDOM_SPLIT"  value
="3">Deal_Damage</Function>
19        </Effect>
20      </Skill>
21    </Skills>
22    <TargetNeeded>NO</TargetNeeded>
23    <Targetable>NO</Targetable>
24  <Description>Deal 3 damage randomly split among all enemies.</Description>
25</Card>
```

Kód 4 – reprezentace karty Arcane Missiles v XML databázi

Jedná se o kartu s názvem „Arcane Missiles," která stojí 1 manu (ř. 5) na zahrání. Z jejích tagů je patrné, že se jedná o kartu typu kouzlo (ř. 7). Nemá proto žádný útok ani životy. Také není třeba, aby jí hráč ručně cíloval, protože se její cíle vyhodnotí samy. Poslední část, která nám zbývá vysvětlit je element „skills" (ř. 9). Ten tvoří seznam jednotlivých schopností - „skill" (ř. 10) Každá schopnost je v podstatě konkrétní efekt (ř. 18), který má karta provést. Některá monstra nemají efekty žádné a tedy budou mít tento seznam prázdný. Některá monstra mají však komplikovanější efekty, než výše zmíněný (kapitola 2.4) například Taunt nebo Charge. Tyto efekty je nutno popsat pomocí jednotlivých částí, které provádějí. V našem případě se jedná o efekt udělení zranění. Ten je dále určen několika parametry. Jedná se o množství cílů, způsob vybrání cílů a celková hodnota zranění. Konkrétně tedy dojde k udělení 3 zranění, která budou po 1 rozdělena mezi 3 náhodné cíle. Tato schopnost má vlastní seznam tagů (ř. 13), které ovlivňují, jaké cíle jsou validní, pro tuto konkrétní část schopnosti. Některé karty mají více než jeden jednoduchý efekt. Řekněme, že by hráč chtěl vytvořit vlastní kouzlo, které by udělilo 4 zranění do 4 různých nepřátel a zároveň by hráče vyléčilo o 5 životů. Musel by tedy využít dvou

rozdílných efektů – udělení zranění a léčení. Tyto efekty by navíc měly rozdílné cíle. Uživatel tedy může využít výše zmíněný kód karty Arcane Missiles jako vzor a upravit název a cenu za seslání. Nyní zbývá jen implementovat požadované efekty. Efekt udělení zranění můžeme opět použít jako vzor, ale je třeba změnit parametry. Požadovaná funkce by vypadala následovně (Kód 5).

```
<Function targets="4" selector="RANDOM" value ="4">Deal_Damage</Function>
```

Kód 5 – způsob zaznamenání funkce karty v XML databázi.

Uživatel musí změnit počet cílů i hodnotu zranění. Ovšem také musí změnit způsob výběru cíle z „RANDOM_SPLIT“ na „RANDOM.“ Tím docílí toho, aby se 4 zranění nedělila po jednom, ale aby došlo k udělení 4 zranění do každého náhodného cíle. Zbývá implementovat schopnost léčení. Protože se jedná o schopnost, která bude mít jiné cíle, než schopnost udělení zranění, musíme vytvořit novou schopnost celou a ne jen vložit další funkci. Uživatel tedy vytvoří následující blok kódu (Kód 6):

```
1       <Skill>
2         <Type>Generic</Type>
3         <Target>
4           <Tags>
5             <Own/>
6             <Player/>
7           </Tags>
8         </Target>
9         <Effect>
10           <Function targets="1" selector="ALL" value="5">Heal</Function>
11         </Effect>
12       </Skill>
```

Kód 6 – způsob popisu schopnosti karty v XML databázi

Rozdílné tagy (ř. 5 a ř. 6) nyní určují hráče, který hraje toto kouzlo jako validní cíl. Funkce léčení (ř. 10) nyní podle zadaných parametrů vybere všechny validní cíle (tedy hráčovu postavu) a vyléčí jí 5 zranění. Hodnota parametru počtu cílů je pro způsob výběru všech validních cílů nepodstatná – simulátor automaticky vybere všechny cíle, které splňují podmínky určené tagy této schopnosti.

Obdobně může uživatel vytvořit kartu monstra. Zde bude muset vyplnit hodnoty útoku a životů a v základním seznamu tagů karty nahradit tag „Spell“ tagem

„Minion.“

5.5 Tvorba vlastní umělé inteligence

Tvorba vlastní umělé inteligence je jednou z úloh, kterou se náš simulátor snaží uživateli co nejvíce usnadnit. Uživatel musí již napsat vlastní kód příslušné inteligence. Umělá inteligence má jednoduché rozhraní, přes které s ní simulátor komunikuje. Vždy, když simulátor potřebuje získat akci k provedení, zavolá na AI metodu `GetAction`, která má dva parametry. Referenci na instanci třídy `GameEngineFunctions` a referenci na instanci `GameRepresentation`, tedy herního stavu. Simulátor v základu předává odkaz na aktuální herní stav a nikoliv na jeho kopii a uživatelova AI by tento stav neměla měnit. Tento přístup byl zvolen pro ušetření času, který by zabralo klonování herního stavu pro každé rozhodnutí AI. AI jako taková si může v případě nutnosti herní stav naklonovat sama. Dalším důležitým faktorem je, že při startu simulace dojde k vytvoření jediné instance AI, a tedy je možné si uchovávat informace mezi jednotlivými tahy či akcemi. Po zavolání `GetAction` musí uživatelova AI vrátit instanci potomka třídy GenericAction. Jedná se o jednu ze čtyř následujících možností, které reprezentují jednotlivé druhy akcí, které může hráč, potažmo AI ve hře udělat – `EndTurnAction`, `AttackWithMonsterAction`, `PlayCardFromHandAction`, `SelectCardAction`, `SelectTargetAction` a `UseHeroPowerAction`. Jejich konkrétní popis je k dispozici v programátorské dokumentaci.

5.5.1 Příklad implementace jednoduché AI

V této sekci názorně ukážeme, jak by mohla vypadat jednoduchá reflexivní AI. Tedy AI, která bude reagovat pouze na stav hry, na jehož základu bude volit vhodnou akci.

Nejprve si uživatel musí připravit ideu, jak by tato AI měla reagovat. Pro jednoduchost nechť zvolí modifikaci agresivní AI, která je již v základu implementována, a pojmenuje ji třeba „`SmartFaceMageAI`.“ Její chování se nebude příliš lišit od svého vzoru, jen bude chytře vyhodnocovat možnost využití schopnosti hrdiny a určitých kouzel. Řekněme, že klíčové vylepšení bude spočívat v tom, že se AI bude snažit ničit oponentova monstra, pomocí kouzel a schopnosti hrdiny, pokud k tomu bude mít příležitost. Nejedná se o složitou úpravu a můžeme

tedy popsat pouze změny oproti inteligenci `FaceHunterAI`. `FaceHunterAI` se snaží nejprve zahrát co nejvíce monster. Tento postup je třeba upravit následujícím způsobem. `SmartFaceHunterAI` bude mít nové pravidlo s následujícím zněním: „Pokud má oponent monstrum, kterému zbývá jeden život, použije AI schopnost hrdiny ke zničení tohoto monstra.“ Před sekci, kdy AI hraje karty je tedy nutno přidat následující kus kódu (Kód 7):

```
1 int me = Game.CurrentPlayer;
2 int opponent = engine.GetOtherPlayer(Game.CurrentPlayer);
3 if (Game.HeroPowerUsages[me] == 0){
4 List<Card> weaklings = new List<Card>();
5 foreach (Card monster in Game.Fields[opponent]){
6     if (monster.currenthitpoints == 1){
7         weaklings.Add(monster);
8     }
9 }
10 if (weaklings.Count >= 1){
11     return new UseHeroPower(weaklings[0]);
12     }

13}
```

Kód 7 – ukázka části rozhodovacího procesu AI

V tomto kusu kódu uživatel nejprve získá informace o tom, který index odpovídá jeho AI a který oponentovi (ř. 1 a ř. 2). Následně si ověří, zda má možnost využít schopnost svého hrdiny (ř. 3). V případě, že tuto možnost má, proiteruje přes všechna oponentova monstra a vybere z nich ta, která mají aktuálně jeden zbývající život (ř. 4 – ř 9). Pokud mu po této operaci vyšel neprázdný list (ř. 10), vybere první monstrum z něj (ř. 11) a vrátí akci „`UseHeroPower`“ s cílem prvního monstra. Nyní by uživateli zbývalo otestovat tuto změnu proti `FaceHunterAI` a dle výsledku by zjistil, zda tato úprava měla smysl a zda zlepšila šance na výhru.

6 Programátorská dokumentace

V této kapitole detailně popíšeme strukturu simulátoru a jednotlivé funkce a metody, se kterými se uživatel může při práci s kódem programu setkat. Kapitola je rozdělena na dvě základní části – část popisující strukturu XML dokumentu karetní databáze (6.1). Druhá část (6.2) popisuje jednotlivé moduly vlastního simulátoru. Nejprve se věnuje projektu konzolové aplikace `EpicConsoleSimulator` (6.2.1). Následuje podkapitola popisující projekt vizuální aplikace `Hearthstone` (6.2.2). Poslední podkapitola (6.2.3) se zabývá projektem `GameIntestines`, který obsahuje zdrojové soubory vlastního simulátoru, které se kompilují do stejnojmenné knihovny.

GameIntestines obsahuje následující soubory:

`GameEngine.cs` zodpovídající za řízení průběhu simulace hry. `GameEngineFunctions.cs` obsahující metody modifikující herní stav. `CardFunctions.cs`, soubor obsahující jednotlivé implemenace funkcí podporovaných programem (viz. Kapitola 5.4). `CardLoader.cs` je zodpovědný za veškeré načítání externích souborů. Ať již se jedná o seznam karet v balíčku hráče či databázi karet. `GameActions` obsahuje jednotlivé akce, které může uživatel/AI v průběhu svého tahu vyvolat. Tímto přístupem jsme umožnili vznik rozhraní, které přijímá potomky `GenericAction` a pomocí volání jejich metody `Perform` provádí jednotlivé akce, aniž bychom museli rozlišovat, o kterou akci se jedná. `GameAIs` je soubor, ve kterém jsou uchovávány všechny umělé inteligence. Pokud bude chtít uživatel tvořit vlastní umělou inteligenci, bude editovat tento soubor. a `GameEntities` obsahuje všechny třídy entit, jejichž instance jsou používány v rámci simulace. Hlavními jsou třída Card a GameRepresentation, které reprezentují kartu, respektive herní stav.

6.1 Databáze karet – XMLCardBase

Karty, které se budou do hry načítat jsou uloženy v souboru XMLCardBase, který musí být validní podle schématu XMLCardBase.xsd. Struktura karetní databáze je následující. Kořen se jmenuje `Cards` a pod ním jsou uloženy přímo všechny karty – element `Card`. Karta je určena pomocí několika povinných elementů. Jedná se o jméno karty – `Name`, které musí být unikátní, protože slouží simulátoru pro identifikaci katy. Dále jde o sílu útoku – `Attack`, množství životů – `HP`, cenu za seslání – `Manacost`, seznam tagů – `Tags`, indikátor, zda daná karta vyžaduje cíl – `TargetNeeded` a případně seznam schopností a efektů karty – `Skills`. `Name`, `Attack`, `HP`, `Manacost`, `Tags` a `Target Needed` jsou povinné elementy. Element `Description` je nepovinný a obsahuje textový popis efektů

karty.V případě, že se jedná o monstrum, musí být vyplněn `Attack` a `HP`, u kouzel tyto elementy zůstávají prázdné. `Tags` obsahuje seznam jednotlivých tagů, které slouží k identifikaci karty.

Schopnosti karty jsou popsány pomocí jednoho či více elementů `Skill`, které jsou podelementy elementu `Skills`. Každý jednotlivý `Skill` musí obsahovat elementy `Type`, `Target` a `Effect`. Type určuje co za typ schopnosti se jedná. V současné implementaci je rozpoznáván pouze typ „`Battlecry`," „`Aura`" a „`Triggered`." Ostatní karty mají jako dočasný typ standardního efektu „`Generic`." V případě, že se jedná o typ „`Triggered`," musí následovat element `Trigger`, jehož hodnota určuje, která akce spouští danou schopnost. Validní triggery jsou „`Self_Takes_Damage`," který se spustí v okamžiku kdy karta obdrží zranění, „`Self_Deals_Damage`," který je spouštěn pokud karta udělí zranění, „`KillTarget`," spouštěný pokud karta způsobí zabití monstra a „`End_Of_Turn`," který je spouštěn na konci kola. Element `Target` obsahuje podelement `Tags`, který stejně jako vlastní karta obsahuje jednotlivé tagy. Tyto tagy slouží k určení validních cílů. Validní cíl musí obsahovat stejné tagy a případně se musí nacházet na určité straně herního plánu v případě tagů `Enemy` a `Own`. Tag `Any` znamená, že libovolný cíl je validní. Element `Effect` musí obsahovat jeden a více elementů `Function`. Element `Function` má atributy určené podle své hodnoty. Simulátor rozpoznává následující možnosti – `Heal`, `Draw_Card`, `Deal_Damage`, `Give_Buff`, `Summon`, `Discard`, `Destroy`, `CountAndReplace`, `Freeze`, `Transform`, `GiveTag`, `TotemicCall`. Tyto hodnoty reprezentují funkce, které simulátor umí provádět. Analýzou karet, které jsme chtěli implementovat jsme zjistili, že efekty karet se velmi často opakují a i relativně složité efekty se dají případně poskládat z těchto základních. Případné rozšiřování karetní báze by případně mohlo vyžadovat rozšíření této množiny funkcí. Jednotlivé atributy elementu `Function` tedy odpovídají parametrům, které vyžadují konstruktory příslušných tříd reprezentujících jednotlivé karetní funkce. Karta může mít několik různých schopností. Monstra mohou mít více než jednu schopnost. Pokud hráč může respektive musí vybrat cíl pro schopnost, její validní cíle jsou určeny pomocí první schopnosti. Ostatní schopnosti berou si cíl vybírají buď na základě této volby nebo berou všechny nebo náhodné cíle podle tagů.

6.2 Simulátor

Vlastní projekt simulátoru je rozdělen do tří částí. `Hearthstone`, `GameIntestines` a `EpicConsoleSimulator`. `Hearthstone` je zodpovědná za vizuální aplikaci. `GameIntestines` je jádro vlastního simulátoru. `EpicConsoleSimulator` je konzolová aplikace.

6.2.1 EpicConsoleSimulator

Tento projekt obsahuje jediný zdrojový soubor `program.cs`. Jediná metoda `Main` je zodpovědná za nastavení simulace a její následné spuštění a vyhodnocení. Nejprve se načtou všechny vstupní informace. Buď formou argumentů při spuštění nebo formou čtení ze standardního vstupu. Následně dojde k iniciaci herního jádra a pak opakovanému simulování hry. Výsledky jsou pokaždé zaznamenány a po skončení simulace vypsány na standardní výstup. Případné úpravy nastavení simulace v konzolovém režimu tedy uživatel musí provádět v tomto souboru.

6.2.2 Hearthstone

Tento projekt obsahuje dvě důležité komponenty. Jedná se o `ViewModel` a `SimulationWindow`. `SimulationWindow` je vlastní okno vizualizační aplikace, které využívá `ViewModel` k tomu, aby získávalo notifikace o změnách a následně podle nich aktualizoval uživatelské rozhraní. `ViewModel` sleduje změny na položkách instance třídy `GameEngine`.

6.2.3 GameIntestines

Obsahuje všechny části simulátoru. Ty jsou rozděleny do několika souborů – `CardFunctions`, `CardLoader`, `GameActions`, `GameAIs`, `GameEngine`, `GameEngineFunctions` a `GameEntities`.

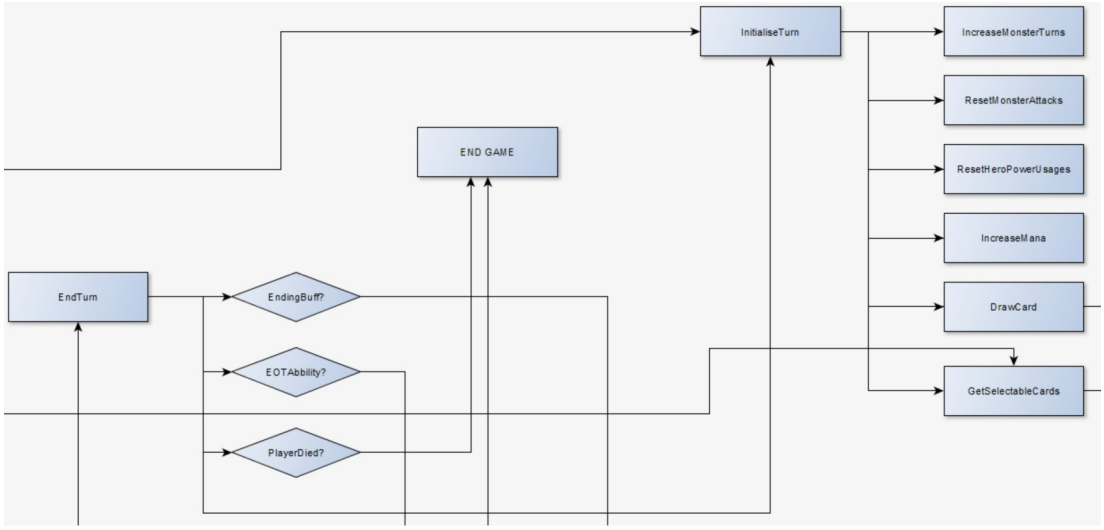
GameEngine

Obsahuje několik proměnných a metod určených pro předávání důležitých informací pro komunikaci s `ViewModelem`. Dále je zde několik metod určených k řízení vlastních simulací. `PrepareGame` slouží k přípravě vlastní simulace. Dojde k základní inicializaci karetní databáze, vytvoření instance třídy herního stavu `GameRepresentation` a instance třídy `GameEngineFunctions`, která

obsahuje veškeré metody pro manipulaci s herním stavem. Následně dojde k zavolání metody `SetStuff`, která resetuje herní stav a nastaví jej do základní podoby. Pokud tedy uživatel potřebuje připravit konkrétní herní stav, může upravit tělo metody `SetStuff` nebo upravit herní stav po jejím zavolání. Metoda Load`Game` pouze volá `PrepareGame` a `SetStuff` a slouží pro inicializaci hry v grafickém režimu. Navíc v případě, že se ze souboru GameState.xml podařilo načíst herní stav a soubor indikuje, že tento stav má být použit jako základní stav pro následující hru pomocí elementu Use s hodnotou „True“ načte hru do stavu požadovaného hráčem. V případě, že načtení stavu selže ve fázi načítání, začne hru od začátku, pokud dojde k načtení neplatných dat, program se ukončí. Metody `getWinner` a `getTurns` slouží k extrahování statisticky relevantních informací o herním stavu a v současné době jsou využívány konzolovou verzí aplikace. Metody `EndTurn`, `SelectSecondaryTarget` a `SelectCardFromHand` jsou volány z grafického prostředí a slouží ke komunikaci hráče se simulátorem. ProcessInputFromGUI slouží k vytvoření instance akce, kterou chce hráč udělat a jejímu následnému provedení. V případě, že následuje tah AI, bude cyklicky volat metodu GetAction na instanci AI, dokud nedojde k ukončení kola a na tah se opět nedostane hráč nebo dokud nedojde k vítězství jednoho z hráčů.

GameEngineFunctions

Následující obrázek (Obr. 8) znázorňuje výřez z diagramu popisujícího závislosti volání jednotlivých metod třídy GameEngineFunctions. Celý diagram je k dispozici v příloze. Námi zvolený výřez znázorňuje metody EndTurn a InitialiseTurn a metody na nich závislé. Tyto metody jsou na sobě závislé a, jak je z diagramu patrné, metoda EndTurn volá metodu InitialiseTurn.



Obr. 8: Výřez diagramu třídy GameEngineFunctions

Jedná se o jádro celého simulátoru. `GameEngineFunctions` obsahuje jedinou třídu stejného jména. Ta obsahuje všechny metody, které umožňují modifikovat herní stav a získávat informace o něm. Také obsahuje jednu binární veřejnou proměnnou `writeDebugTexts`, která určuje, zda má docházet k vypisování textů na konzoli. Tento přístup byl zvolen z toho důvodu, že pokud by uživatel chtěl, aby umělá inteligence nechala klonovat herní stav a simulovat hru o několik tahů dopředu, ne nutně by přitom vyžadoval, aby každá instance simulované hry vypisovala na konzoli hlášení o tom, co se právě v dané prohledávané větvi stromu děje. Obecné informace ohledně argumentů jednotlivých metod. List stringů `TargetTags` reprezentuje list tagů, které budou v metodě kontrolovány, string `Selector` určuje, jak budou vybírány cíle. Simulátor rozpoznává následující možnosti – PLAYER, pokud má být volba závislá na volbě hráče a tedy cíle, který vybral při hraní karty, ALL pokud má být výsledek aplikován na všechny validní cíle, RANDOM pokud má být cíl zvolen náhodně z množiny validních cílů a RANDOM_SPLIT pokud má být určité množství rozdistribuováno přes určené množství náhodných cílů. Card `Origin` a `Target` jsou reference na instance karet, které jsou zdrojem a cílem dané akce. `GameRepresentation` Game je reference na instanci herního stavu.

Následuje popis jednotlivých metod v pořadí v jakém se objevují ve zdrojovém souboru. `TransformMinions` slouží k transformaci jednoho monstra na jiné, určené pomocí stringu `TransformedInto`. `GiveStatBuffToCards` a

`GiveStatBuffToCard` slouží k úpravě útoku a životů karty respektive karet monstra pomocí takzvaného buffu. Tato úprava je shrnuta v instanci třídy `StatBuffWrapper`, která reprezentuje daný buff. `CountAndReplace` je relativně složitá metoda. Slouží ke změně parametrů jiné funkce karty. V současné době je implementována změna a podoba buffu. Nejprve dojde k vybrání dané funkce karty a následně k spočtení validních cílů určených pomocí tagů `CountAndReplace`. Poté dojde k přepsání hodnot útoku a životů buffu v měněné funkci karty. `FreezeTargets` a `FreezeTarget` slouží k aplikování tagu Frozen na kartu respektive karty. `GiveTagToCard` a `GiveTagToCards` slouží k přidání dalšího tagu kartě či kartám. `RemoveBuffFromCard` slouží k odstraněné buffu z karty. `ActivateAufa` slouží k aktivaci aury, pokud monstrum s nějakou aurou přijde do hry. Aura je nějaká funkce, která je aplikována na každé monstrum, které přijde do hry. V současné implementaci aura modifikuje útok a obranu validních monster. Pokud je monstrum poskytující auru odstraněno ze hry, je Aura deaktivována a její efekt je odebrán ze všech monster. `Deal_Damage` slouží k udělení zranění do monster na základě toho, jaký selector je zvolen. Tato metoda pak na validní cíle volá metodu `DealDamage`, která již přímo modifikuje životy daného monstra a obsluhuje triggered schopnosti, které se na udělování zranění vážou. `Heal` slouží k odstranění zranění z dané karty. `AuraTrigger` je volána pokaždé, když dojde k události, která by měnila stav na herní ploše a slouží k aplikaci aury na všechny validní cíle. `MonsterGetsPlayed` slouží k aktivaci schopností typu battlecry, v okamžiku kdy je dané monstrum zahráno. `MonsterComesIntoPlay` slouží v současné implementaci pouze k aktivaci aur pokud přijde monstrum do hry. `PlaySpellFromHand` slouží k zahrání konkrétního kouzla z hráčovy ruky. `PlayMonsterFromHand` slouží obecně k rozpoznání, kterou akci hráč provádí. Rozliší se, zda chce zahrát kartu z ruky a případně kterou, zda chce zaútočit s monstrem nebo zda chce využít schopnosti svého hrdiny. Také dojde k ověření zda na to hráč má právo – například zda má dostatek volné many na zahrání. `SelectCardFromHand` je volána pokud hráč zvolí nějakou kartu. Simulátor pak pomocí této metody zjistí zda se jedná o kartu z ruky, monstrum na stole či schopnost jeho hrdiny a následně pro zvolenou kartu najde validní cíle které pak přidá do listu `ValidTartetsP` v reprezentaci herního stavu. `GetPlayableCards` slouží ke

zjištění, které karty z ruky může aktuálně hráč zahrát. `GetMonstersThatCanattack` slouží ke zjištění, která monstra mohou zaútočit. `GetSelectableCards` slouží ke zjištění, které všechny karty může hráč zvolit. `SelectSecondaryTarget` slouží k výběru cíle pro již zvolenou kartu. Zde je však kontrolováno, zda je cíl validní a pokud není nic se nevybere. `DebugText` slouží k výpisu na konzoli pokud je `writeDebugTexts` nastavena na true. Draw card dobere zvolenému hráči do ruky jednu kartu z balíčku. `DiscardCard` zahodí zvolenému hráči náhodnou kartu z ruky. `IncreaseMana` zvýší o jedna počet many daného hráče a doplní všechny manakrystaly manou. `UseHeroPower` využije schopnost hrdiny aktivního hráče a zvýší počet použití v daném kole o jedna. `ResetHeroPowerUsages` nastaví počet použití schopnosti hrdiny aktivního hráče na nula. `ResetMonsterAttacks` nastaví počet útoků všech monster aktivního hráče na nula. `IncreaseMonsterTurns` zvýší počítadlo kolik kol bylo monstrum ve hře o jedna pro všechna monstra aktivního hráče. `TurnManagement` slouží k řízení simulace. V případě, že hrají dvě umělé inteligence proti sobě – tedy konzolovém režimu, je spuštěna nekonečná smyčka, která inicializuje kolo dokud hra neskončí. V případě hráče proti AI – grafickém režimu je kolo iniciováno pouze jednou a další inicializace probíhá zavoláním inicializace kola na konci provedení metody ukončení kola. Tento rozdíl je způsoben tím, že v grafickém režimu čeká simulátor na akce hráče, které přímo volají jednotlivé akce typu `SelectCardFromHand`. V případě, že hrají proti sobě dvě umělé inteligence tak má simulátor vždy přístup k akci, kterou chce daná inteligence provést. `InitialiseTurn` inicializuje kolo – změní aktivního hráče, zvýší počet many, obnoví použití schopnosti hrdiny a útoky monster aktivního hráče. `EndTurn` slouží k ukončení kola. Pokud má některé monstrum triggered schopnost, která je vázána na konec kola, je provedena. Také odstraní všechny buffy, které mají trvání do konce kola -EOT. Následně zavolá inicializaci kola. `GetOtherPlayer` vrací index druhého hráče než je zadán jako argument. `CheckTagForValidity` a `CheckTagsForValidity` slouží k ověření, zda cílová karta je validním cílem pro zdrojovou kartu a její tagy. `GetSelectedTarget` vrací odkaz na kartu, kterou hráč vybral jako cíl. `GetTargets` vrací list cílů, které jsou validní pro buď určitou schopnost – ability nebo pro danou kartu a její tagy. `DestroyMonsters` ověří, zda

je vybraný cíl validní a následně dané monstrum zničí. `DeactivateAura` slouží k deaktivaci dané aury zvolené karty. Odstraní zvolenou auru ze seznamu aktivních aur a odstraní její efekt ze všech monster na hrací ploše. `KillMonster` zjistí zda karta, která způsobila smrt daného monstra nemá triggered schopnost spuštěnou pokud zabije monstrum. Dále zavolá `RemoveCardFromPlay` a následně `MonsterRemovedFromPlay`. `MonsterRemovedFromPlay` v současné implementaci pouze deaktivuje všechny aury, které odstraněné monstrum poskytovalo a následně zavolá `AuraTrigger`, protože došlo ke změně Aur potenciálně. `RemoveCardFromPlay` nejprve upraví hodnotu `FastSpellDmg` pokud dané monstrum následně odstraní kartu ze hry – ať již z ruky nebo z herní plochy. Pokud přitom odstraní kartu reprezentující hrdinu, hra se označí za skončenou a započítá se vítěz. `DealDamage` udělí zranění kartě případně více kartám monster. `Clash` je metoda, která se stará o vyhodnocení útoku jedné karty na druhou. `MonsterAttacks` je metoda připravená pro budoucí implementaci triggered schopnosti spouštěné pokud monstrum zaútočí. `MonsterAttacksTarget` je metoda připravená pro budoucí implementaci triggered schopnosti spouštěné pokud monstrum zaútočí na konkrétní cíl. `AttackWithMonster` v současné implementaci pouze zavolá `Clash` mezi útočníkem a cílem. `CanAttack` slouží k ověření zda daná karta může zaútočit. `GetValidTargets` zjišťuje validní cíle pro útok daného monstra případně hrdiny pokud má sílu útoku alespoň jedna. `SummonMonster` vyvolá pro vybraného hráče na jeho hrací plochu dané monstrum určené jeho jménem. Pokud takové monstrum v databázi nenajde, nic nevyvolá. `GetCopyOfCardFromDatabase` se pokusí podle zadaného stringu vyhledat v databázi karet danou kartu a vrátit její kopii.

GameEntities

Tento soubor obsahuje veškeré entity které se v simulátoru vyskytují. Jedná se o následující třídy – `GameRepresentation`, `Card`, `Ability`, `Decklist` a `Mana`.

`GameRepresentation` je vlastní reprezentace herního stavu. Obsahuje všechny informace o herním stavu, jako jsou monstra na hrací ploše, karty, které mají oba hráči v ruce, balíčky karet, karty reprezentující hrdiny, seznam aktivních aur, předpočítanou spell damage obou hráčů, počty použití schopnosti hrdiny obou hráčů,

zvolenou kartu a případný její cíl, umělé inteligence které hrají za jednotlivé hráče a množství many obou hráčů. Také obsahuje odkaz na karetní databázi, jejíž karty jsou vázány k této konkrétní herní reprezentaci. Tato třída se umí klonovat zavoláním metody `Clone`. Při naklonování dojde k vytvoření kopie této třídy kde všechny karty a jejich schopnosti včetně karetní databáze budou naklonovány také a proměnné předávané hodnotou se nastaví na hodnotu originálu. Tato třída se také umí resetovat do základního stavu, případně do stavu, který by uživatel chtěl docílit, pokud případnou metodu upraví.

`Card` je reprezentací karty. Karta je v podstatě každá entita ve hře. Tedy hrdina je typ karty a kouzla i monstra jsou karty. V současné implementaci má karta mnoho různých proměnných, z nichž některé nejsou v současné době využívány a byly implmentovány pro budoucí rozšíření simulátoru. Z významných proměnných a metod se jedná o následující. `Name` – jméno karty, současně i identifikátor pomocí kterého se karta vyhledává v databázi. Velikosti základních útoků a počtu životů karty – ty se nemění a slouží jako vzor původní karty – `baseattack` a `basehitpoints`. `Currentattack` a `currenthitpoints` jsou aktuální hodnoty útoku a životů karty. Obdobně funguje i určení spell damage karty. Podstatný je list stringů tags, který reprezentuje tagy karty. List abilit `Skills` reprezentuje schopnosti dané karty a `Auras` aury, které jsou na kartě aktuálně aktivní. `ListOfStatBuffs` reprezentuje buffy které aktuálně modifikují kartu. Karta se umí klonovat pomocí metody `Clone`. Lze rozlišit, zda se má karta klonovat vůči aktuálnímu hernímu stavu, ke kterému přísluší nebo zda se má herní stav nahradit jiným. Rozdíl to hraje v případě kdy hráč klonuje kartu z databáze do hry, například při vyvolání monstra pomocí schopnosti jiného nebo zda klonuje celý herní stav a chce aby v novém klonu karty neovlivňovaly stav originálu.
`Mana` reprezentuje množství many a manakrystalů které hráč vlastní.

`Decklist` reprezentuje seznam karet, které má hráč ve svém balíčku. Balíček je možné klonovat i zamíchaný, aby bylo možné docílit náhodného chování klonovaných simulací.

`Ability` reprezentuje schopnosti karty. Má referenci na kartu, ke které přísluší, trigger pokud by se jednalo o triggered schopnost, string určující o jaký typ schopnosti se jedná a odkaz na herní stav, vůči kterému se schopnosti provádějí. Také se umí klonovat a stejně jako u karty záleží vůči kterému hernímu stavu se klonuje.

Jelikož s jedná o schopnost, metoda `Perform` způsobí provedení této schopnosti karty na herní stav, na který je vázána kartou.

GameActions

Jedná se o soubor obsahující třídy potomků `GenericAction`. To jsou možné akce, které může umělá inteligence provádět, když je na ní zavolána metoda `GetAction`. Jedná se o několik základních akcí, které může hráč provádět. `EndTurnAction` ukončuje kolo, `SelectCardAction` vybere danou kartu pomocí `SelectCardFromHand`, `PlayCardFromHandAction` zahraje danou kartu pomocí `PlayMonsterFromHand`, `UseHeroPowerAction` zavolá použití schopnosti hrdiny, `SelectTargetAction` zvolí cíl pro již vybranou kartu, `AttackWithMonsterAction` ověří zda je možno zaútočit vybraným monstrem a zaútočí s ním do zvoleného cíle. `DealDamageAction` slouží jako nástroj, pomocí kterého AI může udělovat zranění libovolnému cíli mimo pravidla a proto je v současné implementaci nefunkční – slouží jako nástroj případnému uživateli pokud by chtěl testovat konkrétní scénář nebo chování.

GameAIs

Tento soubor obsahuje třídy umělých inteligencí. Ta musí být potomkem abstraktní třídy `GenericAI`. Inteligence musí implementovat metodu GetAction, která vrací instanci třídy `GenericAction`. Způsob jakým se rozhodne akci vybrat je závislý na implementaci uživatele. V současné době je k dispozici implementace náhodné AI a `FaceHunterAI` tedy náhodná a agresivní umělá inteligence. Podobně jako karta, i inteligence má vlastní jméno, které slouží jako identifikátor. Při pouštění konzolové verze simulátoru uživatel vybírá, která inteligence má s daným balíčkem hrát a jméno, které zvolí musí odpovídat jménu některé z implementovaných inteligencí.

CardLoader

Tento soubor obsahuje metody zodpovědné za načítání karet z xml dokumentu. Pokud by uživatel tedy chtěl měnit formát nebo přidávat novou základní mechaniku, musí upravit načítání v tomto souboru. Vlastní loader potřebuje při konstrukci mít k dispozici referenci na instanci herního stavu, vůči kterému bude karty načítat. Karty načítá pomocí metody `LoadCards`, která má jako atribut string obsahující jméno xml dokumentu s kartami. Je tak možné opakovaným voláním loaderu načíst karty z více zdrojových souborů LoadCards ovšem ukládá karty přímo

do listu představujícího databázi karet dané herní reprezentace. Při načítání karet volá loader své další dvě metody – `ParseAbility` a `ParseFunction`. `ParseAbility` parsuje jednotlivé schopnosti karet – tedy jednotlivé elementy Skill a pro každý efekt dané schopnosti volá `ParseFunction`. `ParseFunction` rozpoznává pouze implementované funkce, které jsou implementovány v posledním souboru – `CardFunctions`. Pokud by chtěl hráč přidat novou základní funkci, bude muset přidat záznam do metody `ParseFunction` a také její záznam do souboru `CardFunctions`. Pokud dojde při načítání karty k nějaké chybě, načítání skončí a program skončí s chybovou hláškou. Loader nekontroluje validní hodnoty, takže je možné načíst i chybnou kartu, pokud bych například vytvořil monstrum se záporným počtem životů. To by ovšem pravděpodobně způsobilo problém při hraní hry a ta by skončila s chybovou hláškou odpovídající problému, který byl ovšem způsoben špatným vzorem karty.

CardFunctions

V tomto souboru jsou implementovány jednotlivé herní efekty karet. Konkrétně jde o různé schopnosti, které mohou karty obsahovat a případně je dále kombinovat. Jedná se o efekt `DrawCard` , tedy dobrání karty, `CountAndReplace`, který volá metodu stejného jména v `GameEngineFunctions`, `GiveBuff`, který dává buff dané kartě, `Heal` a `DealDamage`, které léčí respektive udělují zranění, `Summon`, který vyvolává monstrum, `Discard` zahazuje kartu, `Destroy` monstrum ničí a `Freeze` jej zmrazí. `Transform` monstrum transformuje na jiné a `GiveTag` danému monstru přidá další tag. Příkladem specifické funkce je poslední `TotemicCall`, který slouží zatím jen jako efekt šamanovy schopnosti hrdiny. Pokud by hráč chtěl implementovat některý další set karet, který by se vyznačoval hojným výskytem nějakého specifického či zcela nového efektu, musel by daný efekt přidat jako potomka třídy `CardFunctions` do tohoto souboru a na základě jeho konstruktoru přidat záznam do metody `ParseFunction` v souboru `CardLoader`. Stejně jako karty a ability i potomci třídy `CarFunctions` se umí klonovat.

7 Experiment

V této kapitole je popsán průběh a výsledek experimentu. Jedná se o porovnání našeho simulátoru a simulátoru Fireplace a Sabberstone. Porovnávali jsme rychlost a rozšířitelnost simulátorů. Protože došlo k zahájení vývoje simulátoru Sabberstone, který se podobá našemu simulátoru, bude porovnání s tímto simulátorem také součástí této kapitoly.

7.1 Nastavení experimentu

Aby byl experiment reprezentativní, je nutné nejprve zajistit, abychom na každém simulátoru simulovali stejnou situaci. Simulace je ovlivněna několika faktory. Jedná se o obsah herního balíčku, umělou inteligenci, která s balíčkem hraje a množství her, které jsou simulovány. Rozhodli jsme se simulovat náhodné hry se základním balíčkem pro hrdinu mága. Tuto formu simulace jsme zvolili, neboť vhodně reprezentuje první hru, kterou uživatel, který se rozhodne začít hrát karetní hru Hearthstone bude hrát. Po založení herního účtu má k dispozici právě základní balíček mága a k získání ostatních základních balíčků musí porazit ostatní hrdiny ovládané primitivním AI. Roli také hrálo omezení představované implementovanou množinou karet. Náš simulátor má z porovnávaných simulátorů implementovánu nejmenší množinu karet. Toto nás výrazně omezilo ve skladbě balíčku. Základní balíček mága ovšem obsahuje pouze základní karty mága a karty z basic setu, tedy je vhodným balíčkem pro experiment. Všechny karty použité v experimentu jsou implementovány i v rámci simulátoru Fireplace a Sabberstone.

Abychom mohli změřit čas rozdílných implementací jednotlivých efektů karet, zvolili jsme náhodnou umělou inteligenci. Agent této inteligence ve svém tahu dělá náhodné tahy, tedy hraje náhodné karty z ruky, v případě nutnosti na náhodné cíle, náhodně útočí s monstry, náhodně využívá schopnosti svého hrdiny a náhodně ukončuje své kolo. Každý z testovaných simulátorů v základu obsahuje takovou umělou inteligenci. Byly mezi nimi však drobné rozdíly.

V rámci měření jsme vždy nechali simulovat 10000 náhodných her. Simulátor jsme doplnili o kód zajišťující měření času a celý projekt jsme zkompilovali v Release verzi. Simulace probíhaly na notebooku následujících specifikací: Lenovo ideapad 700-15ISK s procesorem Intel(R) Core(TM) i5-6300HQ CPU @ 2.30GHz,

2304 Mhz, 4 Core(s), 4 Logical Processor(s) a 8GB DDR4 RAM s operačním systémem Windows 10 Education N. V průběhu měření byla spuštěna vždy konzolová verze simulátoru, který byl v tu chvíli měřen a na počítači nebyly prováděny žádné úkony až do konce měření. Simulátory byly uloženy na SSD disku.

Zdrojové soubory obou zbývajících simulátorů (Fireplace a SabberStone) použité v experimentu jsou k dispozici v příloze. Pro opakování experimentu musí uživatel v případě simulátoru Fireplace musí uživatel po rozbalení archivu „FirePlace.7z“ spustit soubor `full_game.py` ve složce `fireplace-master\tests` s jediným parametrem určujícím počet simulovaných her. Tedy:

```
fireplace-master\tests>Python full_game.py 10000
```

V případě simulátoru Sabberstone je po rozbalení archivu „Ssmaster.7z“ postup následující. Ve složce `SabberStone-master\SabberStoneCoreAi\bin\Release\netcoreapp1.0` je k dispozici framework-dependent .NET Core aplikace simulátoru. Pro spuštění stačí použít příkaz:

```
..\netcoreapp1.0>dotnet SabberStoneCoreAi.dll
```

Pro nastavení počtu simulací potřebuje uživatel otevřít celý projekt „`SabberStone.sln`“ v programu VisualStudio2017 (nebo vyšší verze). Zde v projektu „`SabberStoneCoreAi`“ v metodě „`RandomGames`“ nastavit počet požadovaných náhodných her (v proměnné „`total`“). Po zkompilování program odsimuluje požadované množství náhodných her a na konzoli vypíše celkový čas simulace.

7.2 Analýza rychlosti simulace

Prvním a klíčovým experimentem jsme se rozhodli změřit celkovou rychlost simulace bez zásadních zásahů do kódu simulátorů. Všechny simulátory mají oddělenou část iniciace simulace a jejího následného provedení. Tyto části kódu jsme měřili separátně, abychom získali celkovou dobu přípravy simulace a celkovou dobu simulace hry. Rozdíl celkového času simulace a součtu těchto dvou částí je kontrolní čas, který zabrala obsluha kódu simulace. Výsledkem měření tedy byly vždy 4 časové údaje – celková doba experimentu, celková doba přípravy náhodné hry, celková doba náhodných her a čas, který zabralo provádění experimentu. Takto jsme

změřili výkon všech tří simulátorů. Jedinou zásadní změnou v kódu bylo vypnutí/odstranění výpisu průběhu jednotlivých her na konzoli, abychom měřili pouze rychlost vlastní simulace. Výsledky tohoto měření jsou znázorněny v tabulce (Tab. 1).

Simulátor	**Doba simulace**
Fireplace	816s
Sabberstone	364s
Náš Simulátor	3,8s

Tab. 1: Výsledky měření rychlosti simulátorů Fireplace, Sabberstone a Našeho Simulátoru

Je patrné, že náš simulátor svou rychlostí zdaleka předčil simulátor Fireplace a byl překvapivě rychlejší, než simulátor Sabberstone. Rozhodli jsme se proto simulátor Sabberstone podrobit důkladnější analýze. Důvodem této analýzy bylo podezření, že výrazný časový rozdíl doby simulace pomocí simulátoru Sabberstone a našeho simulátoru je způsoben neoptimálním kusem kódu. Pomocí standardního profileru Visual Studia jsme analyzovali celý projekt Sabberstone.. Jednalo se o chybný postup, při sbírání a předávání hlášení na konzoli. Provedením této optimalizace jsme byli schopni zoptimalizovat kód simulátoru Sabberstone a provést další měření s touto verzí, námi pracovně označenou jako „Sabberstone-opti.“ Výsledek tohoto měření jsme také zaznamenali do tabulky.(Tab. 2)

Simulátor	**Doba simulace**
Fireplace	816s
Sabberstone	364s
Sabberstone-opti	87s
Náš Simulátor	3,8s

Tab. 2: Výsledky měření rychlosti simulátorů Fireplace, Sabberstone, Sabberstone-opti a Našeho Simulátoru

Sabberstone-opti jme opět analyzovali. Výsledkem tohoto profilingu bylo zjištění, že po naší prvotní optimalizaci byl novým zpomalujícím faktorem návrh tohoto simulátoru. Sabberstone řeší průběh hry pomocí zásobníku, na který přidává jednotlivé události. Při následném provádění jednotlivých operací velmi často prochází jednotlivé listy, stejně tak jako přidává a odebírá události ze zásobníku. Tyto neustálé modifikace listů byly dalším faktorem, který Sabberstone zpomaloval při

simulaci jednotlivých her.

7.3 Analýza rozšiřitelnosti

Závěrečným hodnoceným aspektem byla rozšiřitelnost simulátoru. Jde o to, jak náročné je implementovat nové karty, mechaniky a pravidla. Určitým aspektem je také případná optimalizace za účelem dosažení lepšího výkonu simulátoru.

7.3.1 Fireplace

Jelikož Fireplace již není dále vyvíjen, je tato otázka jednoduší. Nyní by bylo třeba doimplementovat značné množství karet a také minimálně pravidla nové mechaniky, takzvaného úkolu neboli „questu.“ Tato nová mechanika přidává každému hrdinovi jeho speciální quest, což je kouzlo, které stojí 1 manu a pokud jej hráč hraje, vždy jej dostane v počáteční nabídce karet. Po jeho zahrání se hráči začnou počítat určité aspekty jeho úkolu. Mág musí například zahrát 6 kouzel, která nebyla v jeho balíčku na začátku hry. Jeho odměnou za splnění tohoto úkolu je kouzlo, které stojí 5 many a přidá hráči po zahrání kolo navíc. Tato mechanika by vyžadovala širší zásah do kódu, nicméně by se dal rozšířit již implementovaný systém událostí, který kontroluje vazby na konkrétní události. Přidávání nových karet je komplikovanější, protože karty se na začátku načítají z relativně komplexního XML dokumentu a i pro jednoduché karty by vytvoření odpovídajícího vzoru nebylo příliš jednoduché. Obecně tedy rozšiřitelnost tohoto simulátoru je přinejmenším problematická. Aby se s ním daly simulovat hry aktuální verze hry Hearthstone, bylo by třeba implementovat velké množství karet a mechanik.

7.3.2 Sabberstone

Sabberstone v současné době implementuje většinu doposud vydaných karet a mechanik. To je výhodné. Program tak nabízí již velmi širokou platformu funkcí, které uživatel může využít pro tvorbu vlastních karet. Tento úkon je opět komplikován tím, že program načítá karty ze stejného dokumentu jako simulátor Fireplace. Tedy složitost tohoto xml dokumentu činí drobnou překážku pro jeho rozšíření. Dalším problémem je, že jednotlivé schopnosti karet, například battlecry karty Novice engineer – dober jednu kartu, jsou pro každou kartu popisovány zvlášť v kódu programu. Tedy pokud by hráč chtěl zkusit vytvořit vlastní kartu, musí nejen upravit XML dokument, do kterého vepíše příslušný záznam karty, ale zároveň musí

také upravit soubor CoreCardsGen.cs, kde pro danou kartu musí napsat implementační část kódu, která jeho kartě přiřadí požadované schopnosti. Nevýhodou tohoto návrhu je, že pokud by uživatel chtěl testovat změny určitých karet, je nutné zasahovat do kódu programu kvůli každé kartě se speciálními schopnostmi a rekompilovat celý projekt. Také v případě, že uživatel zadá neplatný název, respektive identifikátor karty, nebude jeho karta správně načtena, ale bude mít místo toho schopnosti jiné karty, jejíž identifikátor použil.

7.3.3 Náš simulátor

Primárním problémem našeho simulátoru je malé množství implementovaných karet a mechanik. Architektura programu však umožňuje uživateli implementovat nové karty rychle a efektivně. Oproti ostatním simulátorům jsou karty reprezentovány pouze v XML dokumentu, takže uživatel může měnit a testovat karty bez zásahu do kódu simulátoru. Pokud by uživatel chtěl implementovat novou herní mechaniku, je již nutné doplnit příslušný kód do simulátoru. Implementace nových mechanik je detailněji popsána v uživatelské a programátorské dokumentaci. Stručně však jde zpravidla o využití již existujících nástrojů, které jsou poskytovány třídou GameEngineFunctions. V případě velmi specifických efektů je však vždy možnost naprogramovat daný efekt jednorázově a specificky tak, že z něj udělá další efekt rozpoznávaný parserem karet. Ve výsledku by tento přístup způsobil, že každá karta by měla svůj vlastní efekt, který by popisoval pouze její chování, což není příliš žádoucí, neboť to odstraňuje jednu z hlavních výhod našeho simulátoru – snadnou rozšiřitelnost.

8 Validace splnění cílů práce

8.1 Vytvoření uživatelsky přívětivého UI

Abychom mohli potvrdit úspěšné splnění tohoto cíle, rozhodli jsme se zeptat uživatelů na jejich pocity z práce s naším simulátorem. Cílem bylo získat zpětnou vazbu k vizuální aplikaci. Na základě této zpětné vazby pak posoudit, jak je grafické rozhraní vnímáno a zda se dá považovat za uživatelsky přívětivé.

Oslovili jme tedy skupinu testovacích uživatelů. V rámci tohoto testu dostali k dispozici složku se souborem grafické aplikace našeho simulátoru a všemi soubory potřebnými ke spuštění. Žádné další informace ani manuál jim nebyly poskytnuty a s obsluhou programu neměli předchozí zkušenosti. Snažili jsme se tak simulovat nového uživatele, který se rozhodl zkusit náš simulátor a bez zdlouhavého studování jeho obsluhy jej spustil, aby si vyzkoušel, jak funguje.Skupina se skládala z 10 uživatelů, kteří svým profilem spadají mezi cílové koncové uživatele našeho programu. Jednalo se z větší části o aktivní či bývalé hráče karetní hry Hearthstone, ovšem i jedince bez jakýchkoliv znalostí o hře Hearthstone a to v poměru 8 hráčů ku 2 nehráčům. Za zmínku stojí i fakt, že se převážně jednalo o počítačově gramotné jedince, kteří mají znalosti v oblasti programování a vývoje softwaru (tentokrát v poměru 9 ku 1 kde tento jediný programování neznalý uživatel zároveň nikdy předtím hru Hearthstone nehrál.). Poměry testovacích uživatelů jsme zvolili tak, aby reflektovali námi očekávané rozložení demografie cílových uživatelů v budoucnosti. Jejich úkolem bylo odehrát hru v grafickém režimu bez dalších doplňujících informací. K dispozici měli program puštěný v konfiguraci simulující odehrání celé hry. Po dohrání byli požádáni o poskytnutí zpětné vazby.

Analýzou zpětné vazby jsme dospěli k následujícím závěrům ohledně úspěšnosti naší implementace.

Všichni dotázaní byli schopni program spustit a úspěšně odehrát hru proti umělé inteligenci. Často se vyjadřovali kladně k faktu, že jim program nabízí a zvýrazňuje možné akce. Tento fakt zejména ocenili ti, kteří se hrou Hearthstone neměli předchozí zkušenosti.

Názory na grafické rozhraní se různily. Obecný závěr je, že by uživatelé preferovali „barevnější," či možná méně minimalistické zobrazení. Někteří také projevili přání, aby se tahy umělé inteligence odehrávaly se zpožděním, aby měli šanci sledovat, co se v průběhu tahu oponenta děje.

Tato zpětná vazba nám umožnila považovat náš cíl za splněný, neboť všichni, kteří testovali náš program zvládli splnit zadaný úkol bez obtíží. Také jsme si mohli vytyčit nové cíle v oblasti budoucího rozšíření našeho programu v oblasti grafického rozhraní.

8.2 Umožnění snadné implementace nových karet a mechanik

K ověření splnění tohoto cíle jsme využili zpětnou vazbu několika účastníků předchozího experimentu. K dispozici dostali relevantní část manuálu a zdrojový XML soubor, který obsahoval databázi karet. Jejich úkolem bylo vytvořit tři karty – jednu dle vzoru již implementované karty, druhou dle zadaných parametrů a třetí libovolnou vlastní. Tyto jsme pak ověřili ve hře. Všichni zúčastnění zvládli splnit všechny tři úkoly a v rámci zpětné vazby nepovažovali náročnost/složitost tohoto úkonu jako příliš vysokou, čímž jsme potvrdili naši hypotézu o splnění tohoto cíle práce.

Ze zpětné vazby však také vyvstalo přání uživatelů, zda by nebylo možné vyvinout editor karet, který by jim usnadnil tvorbu nových a úpravu existujících karet. Toto přání jsme reflektovali a zařadili vývoj tohoto editoru do seznamu budoucích prací a rozšíření našeho programu. Díky tomu, že náš simulátor nevyžaduje rekompilaci vlastního programu v případě editace karty, jako je tomu u konkurenčních simulátorů, má vývoj tohoto editoru smysl z hlediska budoucí rozšiřitelnosti našeho programu.

8.3 Umožnění snadného simulování velkého množství her

Tento závěrečný úkol jsme ověřili v rámci experimentu, který je popsán v předchozí kapitole. (Kapitola 7 – Experiment.) Po analýze rychlosti simulace náhodné hry v konzolovém režimu jsme potvrdili, že náš simulátor umožňuje simulovat velké množství her ve velmi krátkém čase.

9 Závěr

Naším cílem bylo vytvořit simulátor v jazyce C#, který by umožňoval rychle a efektivně simulovat velké množství her, byl snadno rozšiřitelný, umožňoval využití umělé inteligence a nabízel uživatelsky příjemné prostředí. Motivací tohoto cíle byla absence dostatečně dobrého simulátoru, který by splňoval tyto požadavky.

Po naprogramování našeho simulátoru jsme porovnali rychlost našeho simulátoru s dvěma dostupnými simulátory. Jednalo se o simulátor Fireplace napsaný v jazyce Python a simulátor Sabberstone, který byl čerstvě vyvíjen v jazyce C#. Na základě měření délky trvání 10000 náhodných her jsme dospěli k závěru, že náš simulátor je výrazně rychlejší něž Fireplace i Sabberstone a to i po provedení základních optimalizací na simulátoru Sabberstone.

Uživatelskou přívětivost grafického rozhraní a obtížnost implementace nových karet jsme otestovali na malé skupině hráčů karetní hry Hearthstone. Analyzovali jsme jejich zpětnou vazbu, která nám potvrdila, že se nám podařilo úspěšně splnit všechny naše vytyčené cíle. Připomínky k našemu programu ze strany testujících uživatelů jsme reflektovali v závěrečném odstavci, který se věnuje budoucím rozšířením a úpravám našeho programu.

V důsledku těchto zjištění jsme přesvědčeni, že náš simulátor je vhodným nástrojem pro simulaci hry Hearthstone.

Hlavním nedostatkem našeho simulátoru se ukázala být malá množina implementovaných karet. S ohledem na to, že náš simulátor byl navržen tak, aby bylo jeho rozšiřování o nové mechaniky a karty snadné, neměl by tento problém být překážkou implementace všech karet a mechanik, které se v současné době v karetní hře Hearthstone vyskytují. (http:/hearthstone.gamepedia.com/Hearthstone_Wiki)

V budoucnosti bychom chtěli náš simulátor rozšířit, aby podporoval všechny karty a mechaniky a stal se tak plnohodnotnou náhradou pomalejšího simulátoru Sabberstone. Také bychom rádi vylepšili grafické rozhraní z estetického hlediska, na základě zpětné vazby uživatelů a doplnili náš simulátor o externí editor karet, který by byl schopen editovat a tvořit karty a umožnil tak lepší komfort uživatelů při využívání této funkcionality našeho programu. Plnohodnotná verze našeho simulátoru by mohla být použita fanoušky a hráči této karetní hry k optimalizaci jejich herních balíčků a tvorbě vlastních balancovaných karet.

Seznam použité literatury

Ling, W., Blunsom, P., Grefenstette, E., Hermann, K. M., Kočiský, T., Wang, F., & Senior, A. (2016). Latent Predictor Networks for Code Generation. ***Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*****. doi:10.18653/v1/p16-1057**

Russel & Norvig: Artificial Intelligence: A Modern Approach, 3rd Prentice Hall Press Upper Saddle River, NJ, USA ©2009, ISBN:0136042597 9780136042594

http://hearthstone.gamepedia.com/Hearthstone_Wiki

Seznam použitých webových stránek

[1] - Deceptively Simple.Insanely Fun. (n.d.). Retrieved December 30, 2017, from https://playhearthstone.com/en-us/

[2] - J. (2017, November 26). Jleclanche/fireplace. Retrieved December 30, 2017, from https://github.com/jleclanche/fireplace

[3] - H. (2017, December 28). HearthSim/SabberStone. Retrieved December 30, 2017, from https://github.com/HearthSim/SabberStone

[4] - (n.d.). Retrieved December 30, 2017, from https://hearthsim.info/

[5] - D. (2016, June 21). Danielyule/hearthbreaker. Retrieved December 30, 2017, from https://github.com/danielyule/hearthbreaker

[6] - T. (n.d.). Thomaseding/hearthshroud. Retrieved December 30, 2017, from https://github.com/thomaseding/hearthshroud

Seznam Tabulek

Seznam Obrázků

10 Příloha – Obsah přiloženého CD

Spustitelný soubor EpicConsoleSimulator.exe
Spustitelný soubor Hearthstone.exe
Databáze karet XMLCardBase.xml a příslušná šablona XMLCardBase.xsd
Modelový herní stav k nahrání pozice GameState.xml
Soubor se základním balíčkem karet basic_mage.txt
Adresář HearthstoneSimulator se zdrojovými kódy a projektem programu
Soubor s pogramátorskou dokumentací ProgDoc.pdf
Adresář Experiment s archivy Fireplace.7z a SSMaster.7z
Adresář Přílohy s výběrem obrázků z textu