ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Ruják** Jméno: **Dávid** Osobní číslo: **485008**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Simulátor karetní hry Hearthstone Battlegrounds

Název diplomové práce anglicky:

Hearthstone Battlegrounds Card Game Simulator

Pokyny pro vypracování:

Umělá inteligence v odvětví her není žádnou novinkou, pro mnoho komplexních her je tvorba umělé inteligence komplikovaná záležitost. Pro takové případy je nutné vytvořit vhodný simulátor dané hry, díky kterému se může umělá inteligence jednoduše trénovat.
V rámci práce vytvořte knihovnu, která umožní simulovat karetní autobattler hru Hearthstone Battlegrounds. Analyzujte problémovou doménu a proveďte rešerši podobných simulátorů. V návrhu se zaměřte především na snadnou rozšiřitelnost, jelikož jednotlivé karty se často obměňují a tak taková činnost nesmí zabrat příliš času. Naimplementujte tuto knihovnu pro jazyk Python s ohledem na využití pro trénování umělé inteligence. Otestujte knihovnu a především její výkonost. Simulace hry musí být rychlá, aby samotné trénování umělé inteligence bylo rychlé. Na základě testování proveďte případné změny (např. převeďte pomalé části kódu do C/C++, využijte Cython).

Seznam doporučené literatury:

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. David Bernhauer, Ph.D. katedra softwarového inženýrství FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2023** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: _____

Ing. David Bernhauer, Ph.D. podpis vedoucí(ho) práce	Ing. Michal Valenta, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Datum převzetí zadání

_____ Podpis studenta

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Diplomová práca

Simulátor karetní hry Hearthstone Battlegrounds

Bc. Dávid Ruják

Katedra teoretické informatiky
Vedúci práce: Ing. David Bernhauer, Ph.D.

11. januára 2024

Poďakovanie

Chcel by som sa poďakovat' môjmu vedúcemu práce, Ing. Davidovi Bernhauerovi Ph.D. za jeho rady a pomoc pri tvorení tejto práce. Vel'ká vďaka patrí tiež mojej rodine a priateľom za ich neustálu podporu počas celého štúdia.

Prehlásenie

Prehlasujem, že som predloženú prácu vypracoval(a) samostatne a že som uviedol(uviedla) všetky informačné zdroje v súlade s Metodickým pokynom o etickej príprave vysokoškolských záverečných prác.

Beriem na vedomie, že sa na moju prácu vzťahujú práva a povinnosti vyplývajúce zo zákona č. 121/2000 Sb., autorského zákona, v znení neskorších predpisov, a skutočnosť, že České vysoké učení technické v Praze má právo na uzavrenie licenčnej zmluvy o použití tejto práce ako školského diela podľa § 60 odst. 1 autorského zákona.

V Praze 11. januára 2024

Odkaz na túto prácu

Ruják, Dávid. *Simulátor karetní hry Hearthstone Battlegrounds.* Diplomová práca. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2024. Dostupný aj z WWW: ⟨`https://projects.fit.cvut.cz/theses/4753`⟩.

Abstrakt

Táto práca je zameraná na implementovanie simulácie populárnej hry zo žánru *autobatlers* – Hearthstone Battlegrounds. Na začiatku sú stručne vysvetlené základné pojmy kartových hier, s dôrazom na Hearthstone a Hearthstone Battlegrounds. Ďalej nasleduje uvedenie vysvetlenie herných mechaník, pravidiel a priebehu hry. V ďalšej časti prebehne prebehne návrh a implementácia v jazyku C++ s použitím Cythonu. V poslednej časti prebehne testovanie výkonnosti.

Klíčová slova umelá inteligencia, zberateľské kartové hry, Hearthstone Battlegrounds, C++, Cython, Python

Abstract

This thesis focuses on the implementation of a simulation of the popular autobattler game - Hearthstone Battlegrounds. It begins with a brief explanation of the basic concepts of card games, with an emphasis on Hearthstone and Hearthstone Battlegrounds. Following that, there is an explanation of the game mechanics and rules. The subsequent section covers the design and implementation in C++ with the use of Cython. The final part involves performance testing.

Keywords artifical intelligence, collectible card games, Hearthstone Battlegrounds, C++, Cython, Python

Obsah

Úvod **1**

1 Úvod do kartových hier **3**
1.1 Začiatky kartových hier . . . 3
1.2 Zberateľské kartové hry . . . 4
1.2.1 Hearthstone . . . 7
1.3 Auto battler . . . 8
1.3.1 Hearthstone Battlegrounds . . . 8

2 Analýza herných mechaník a pravidiel **11**
2.1 Stručný prehľad . . . 11
2.2 Základné vlastnosti karty . . . 12
2.2.1 Rarita . . . 12
2.2.2 Minion . . . 12
2.2.2.1 Enchantment . . . 13
2.2.2.2 Minion type . . . 13
2.2.2.3 Efekty . . . 14
2.2.2.4 Aury . . . 15
2.2.2.5 Battlecries . . . 15
2.2.2.6 Zlatá verzia . . . 16
2.2.3 Spell . . . 17
2.2.3.1 Secret . . . 18
2.3 Hrdina . . . 18
2.3.1 Hero power . . . 19
2.3.2 Hráč a *minioni* . . . 20
2.4 *Recruit* fáza hry . . . 21
2.4.1 Možné akcie . . . 21
2.4.2 Ekonomika hry . . . 24
2.5 Bojová fáza hry . . . 24
2.6 Ďalšie mechaniky . . . 26

3 Návrh a implementácia **27**
3.1 Základné komponenty . . . 27
3.1.1 ICard . . . 27
3.1.1.1 Rarity . . . 27

3.1.2 Damage . . . 28
3.1.3 Rozhrania . . . 28
3.1.3.1 ICombative . . . 28
3.1.3.2 IArmored . . . 29
3.1.3.3 IDestructible . . . 29
3.2 Minion . . . 31
3.2.1 Minion type . . . 31
3.2.2 MinionRepository a MinionBuilder . . . 33
3.3 Task . . . 34
3.3.1 ITask . . . 34
3.3.2 DamageTask . . . 34
3.3.3 AttackTask . . . 35
3.4 Player . . . 36
3.4.1 PlayerRepository a PlayerBuilder . . . 38
3.5 Effect . . . 38
3.5.1 RandomTargets . . . 39
3.5.2 Zdrojové a cieľové kritéria . . . 39
3.5.2.1 Filtre . . . 40
3.5.2.2 Enumy . . . 40
3.5.2.3 EffectTargetCriteria . . . 41
3.5.2.4 SourceCriteria . . . 42
3.5.3 IEffect . . . 43
3.5.4 TargetBuilder . . . 44
3.5.5 Konkrétny príklad . . . 45
3.5.6 Zhrnutie . . . 47
3.6 Enchantment . . . 47
3.6.1 Zhrnutie . . . 48
3.7 Battlecry . . . 49
3.7.1 Cieľové kritéria . . . 50
3.7.2 IBattlecry . . . 50
3.7.3 Konkrétny príklad . . . 51
3.8 Aura . . . 53
3.9 Systém akcií . . . 53
3.9.1 TreeNode . . . 53
3.9.2 ActionsCenter . . . 54
3.9.3 Použitie návrhového vzoru **Observer** . . . 55
3.10 Fázy hry . . . 56
3.10.1 PhaseController . . . 57
3.10.2 *Recruit* fáza hry . . . 58
3.10.2.1 MinionPool . . . 58
3.10.2.2 UserInputController . . . 59
3.10.2.3 RecruitPhaseController . . . 59
3.10.3 Bojová fáza . . . 62
3.10.3.1 CombatPhaseController . . . 62
3.10.4 Reagovanie na zmeny . . . 64
3.11 Game Manager . . . 66
3.12 Python integrácia . . . 67
3.12.1 Cython . . . 68

4 Testovanie a budúcnosť **71**

4.1 Unit testing 71
4.2 Rýchlosť 72
4.2.1 Zlepšenia rýchlosti 74
4.3 Zmeny do budúcnosti 74
4.3.1 Testovanie 74
4.3.2 Herné mechaniky 75
4.3.2.1 Spell 75
4.3.2.2 Hero power 76
4.3.2.3 Iné 77

Záver **79**

Literatúra **81**

5 Seznam použitých zkratek **85**

6 Obsah priloženého CD **87**

Zoznam obrázkov

1.1 Príklad zápasu hry Magic: The Gathering [6] 6
1.2 Zápas Hearthstonu [13] 8

2.1 Príklad *miniona* **Imprisoner** [24] 16
2.2 Príklad *miniona* **Banana Slamma** [25] 16
2.3 Príklady *spellov* [26] 17
2.4 Príklady takzvaných *secrets* [27] 18
2.5 Príklad hrdinu *Al'akir* s pasívnou *hero power* [28] 19
2.6 Príklad hrdinu *George the Fallen* s aktívnou *hero power* [29] 20
2.7 UI *recruit* fázy hry 23
2.8 Diagram aktivity priebehu boja 25

3.1 Diagram enumu `Rarity`. 27
3.2 Diagram triedy `ICard`. Prístupové metódy sú vynechané. 27
3.3 Diagram triedy `Damage`. Prístupové metódy sú vynechané. 28
3.4 Diagram triedy `ICombative`. Nevirtuálne prístupové metódy sú vynechané. 29
3.5 Diagram triedy `IArmored`. Prístupové metódy sú vynechané. . . . 29
3.6 Diagram triedy `IDestructible`. Nevirtuálne prístupové metódy sú vynechané. 30
3.7 Diagram enumu `Version`. 31
3.8 Diagram enumu `MinionType`. 31
3.9 Diagram aktivít metódy `void takeDamage(Damage &damage)` u triedy `Minion` 33
3.10 Diagram triedy `DamageTask`. Prístupové metódy su vynechané. . . 34
3.11 Diagram triedy `AttackTask`. Prístupové metódy sú vynechané. . . 35
3.12 Diagram aktivít metódy `takeDamage(Damage &damage)` v triede `Player` 38
3.13 Diagram triedy `RandomTargets`. Prístupové metódy sú vynechané. 39
3.14 Diagram triedy `IEffect`. Prístupové metódy sú vynechané. 40
3.15 Diagram enumov `TriggerSource` a `EffectTargets` 41
3.16 Diagram triedy `EffectTargetCriteria`. Prístupové metódy sú vynechané. 41
3.17 Diagram triedy `SourceCriteria`. Prístupové metódy sú vynechané. 42
3.18 Diagram triedy `IEffect`. Prístupové metódy sú vynechané. 43

3.19 Diagram triedy `TargetBuilder`. Prístupové metódy sú vynechané. 44
3.20 Diagram triedy `DamageEffectData`. Prístupové metódy su vynechané. 46
3.21 Diagram triedy `DamageEffect` 46
3.22 Diagram triedy `Enchantment`. Prístupové metódy su vynechané. . 48
3.23 Diagram triedy `BattlecryTargetCriteria`. Prístupové metódy su vynechané. 50
3.24 Diagram enumu `BattlecryTargets`. 50
3.25 Diagram triedy `IBattlecry`. Prístupové metódy su vynechané. . . 51
3.26 Diagram triedy `EnchantmentBattlecry`. Prístupové metódy su vynechané. 52
3.27 Diagram triedy `NodeTree`. Prístupové metódy sú vynechané. . . . 54
3.28 Diagram triedy `ActionsCenter`. Prístupové metódy su vynechané. 54
3.29 Prvý vzťah *subject-observer*. 55
3.30 Druhý vzťah *subject-observer*. 56
3.31 Diagram triedy `PhaseController`. Prístupové metódy su vynechané. 57
3.32 Diagram triedy `MinionPool`. Na diagrame nie sú zobrazené konštanty. 58
3.33 Diagram triedy `RecruitPhaseView` 60
3.34 Diagram triedy `CombatPhaseView` 62

4.1 Namerané hodnoty času behu zápasov 73

Zoznam tabuliek

2.1 Počet kópií *miniona* v každom *tavern tiere* 22

4.1 Tabuľka parametrov laptopu použitému pri testovaní 72
4.2 Namerané hodnoty času behov zápasov 73

Úvod

Vývoj umelej inteligencie v oblasti hier zaznamenal v posledných rokoch výrazný pokrok, a to hlavne vďaka neustále sa rozvíjajúcim technológiám a novým prístupom k problémom v odvetví. Umelá inteligencia v hrách nie je už len luxusom, ale často kľúčovým prvkom hry. Jedným z náročných úloh v oblasti umelej inteligencie v hrách je vytvorenie inteligentného správania v kontexte komplexných hier, medzi ktoré patria aj takzvané *autobaattlers*.

Autobattlers sú nový žáner hier, ktorý kombinuje strategické rozhodovanie s prvkami náhody. V rámci tejto práce sa zameriame na *aubattler* založený na populárnej kartovej hre Hearthstone – Hearthstone Battlegrounds. Vytvorenie umelej inteligencie pre túto hru vyžaduje efektívnu simuláciu herného prostredia, aby bolo možné úspešne trénovať algoritmy, ktoré budú schopné reagovať na dynamiku hry.

Cieľom tejto práce je vytvoriť knižnicu v jazyku Python, ktorá umožní simuláciu Hearthstone Battlegrounds. Pri návrhu knižnice bude kladený dôraz na jednoduchú rozšíriteľnosť, s ohľadom na neustále zmeny a pridávanie nových kariet do hry. Flexibilita knižnice je kľúčovým prvkom pre úspešné modelovanie a trénovanie umelej inteligencie v prostredí *autobattler* hier.

V rámci tejto práce vykonáme analýzu problémovej domény, ktorá nám umožní lepšie porozumieť mechanikám a pravidlám hry. Ďalej prebehne návrh a implementácia – kód implementácie bude v jazyku C++. Na integráciu s jazykom Python bude použitý programovací jazyk Cython. Na záver práce otestujeme výkonnosť knižnice.

Kapitola 1

Úvod do kartových hier

Táto kapitola obsahuje stručnú históriu kartových hier, vrátane zberateľských kartových hier. Takisto tu je v krátkosti vysvetlený pojem *auto battler*.

Následne je vysvetlená stručná história Hearthstone a Hearthstone Battlegrounds.

1.1 Začiatky kartových hier

Hracie karty boli dovezené do Európy z Islamského sveta koncom druhej polovice 14.storočia. Balíček kariet bol praktický rovnaký ako v súčasnosti, až na ich farby a prípadne dvorné postavy. [1, s. 125]

Behom nasledujúcich storočí sa „zhouba“ v podobe hracích kariet začína rýchlo rozmáhať po nemeckých krajinách, napriek tomu, že v tamnejších mestách platil prísny zákaz kartových hier a aj napriek tomu, že proti nimi bojovala cirkev. V 15. storočí ich rozšíreniu výrazne pomohla tlač. Do českých zemí, hlavne pohraničných oblastí, začali tlačené karty z nemeckých dielní prúdiť až po polovici 15. storočia. [2, s. 118]

História hracích kariet je oveľa zložitejšia ako napríklad história stolných hier, takisto je viac spojená so samotnou históriou hracích kariet. Dôvod tejto väčšej zložitosti je ten, že balíček kariet má nespočetne viac variácií a spôsobov, ako s ním hrať nejakú kartovú hru – s jedným balíčkom sa môžu hrať viaceré hry s úplne rozdielnymi základnými pravidlami. Porovnateľnú zložitosť majú len africké hry takzvaného *wari* typu. [1, s. 125]

Táto komplexita je spôsobená tým, že až do tohto storočia bolo hranie kariet veľmi lokálnym fenoménom, narozdiel od stolných hier. Ako príklad môžeme použiť napríklad šach – až do štandardizácie pravidiel šachu počas prvého medzinárodného šachového tunaju v roku 1851 sa pravidlá šachu mierne líšili od krajiny ku krajine. Nelíšili sa však od regiónu k regiónu alebo od jedného mesta k druhému, na rozdiel od kartových kariet, kde to bolo pomerne bežné. [1, s. 125]

História hracích kariet je takisto prepojená so samotnou históriou hracích kariet – je to z toho dôvodu, že v je oveľa menej zdokumentované, s akými kartami sa hrali jednotlivé kartové hry v porovaním so stolnými hrami. Je oveľa viac typov kariet ako napríklad šachových figúrok, samotné karty sú viac

rozdielne a sú asociované so samotnými hrami, prípadne lokáciou, ktorej sa tieto hry hrali. [1, s. 125]

Vzhľadom na to, že hracie karty boli vyrábané z papiera, je ich dochovanie do dnešných čias výnimočné, ich nálezy sú datované až v novoveku. Príkladom môže byť napríklad nález troch karty z druhej polovice 16. storočia, ktoré boli nájdené Ivanem Borkovským v roku 1956 pri výskume Černé věže na Pražskom hrade. [2, s. 112]

1.2 Zberateľské kartové hry

Za zakladateľa žánru zberateľských kartových hier (po anglicky *collectible card game(CCG)* alebo *trading card game(TCG)*) sa pokladá hra Magic: The Gathering. Doktorand matematiky Richard Garfield vytvoril originálny koncept stolnej hry nazvanej *RoboRally*, s ktorým oslovil majiteľa spoločnosti Wizards of the Coast v roku 1991. Peter Adkinson však chcel viac kompaktnú hru, ktorá môže byť hraná všade. [3, str. 5]

Richard Garfield tak upravil svoj skorší koncept hry *Magic* a v 1993 sa *Magic: The Gathering* prvýkrát verejne objavila na najväčšej hernej konvencii o stolových hrách Gen Con. Keďže hra mala veľký úspech, spoločnosť Wizards of the Coast získali investíciu peňazí na vytlačenie prvej sady kariet, ktorá sa následne hneď vypredala. [3, str. 5]

Zberaním kariet sa tieto hry vyznačujú. Niektoré karty v *Magic: The Gathering* dosahujú cenu v rádovo tisícoch až desať tisícov dolárov. Podpísaná limitovaná verzia karty *Black lotus* sa predala za 540,000 dolárov. [4]

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Ruják** Jméno: **Dávid** Osobní číslo: **485008**

Fakulta/ústav: **Fakulta informačních technologií**

Zadávající katedra/ústav: **Katedra softwarového inženýrství**

Studijní program: **Informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Simulátor karetní hry Hearthstone Battlegrounds

Název diplomové práce anglicky:

Hearthstone Battlegrounds Card Game Simulator

Pokyny pro vypracování:

Umělá inteligence v odvětví her není žádnou novinkou, pro mnoho komplexních her je tvorba umělé inteligence komplikovaná záležitost. Pro takové případy je nutné vytvořit vhodný simulátor dané hry, díky kterému se může umělá inteligence jednoduše trénovat.
V rámci práce vytvořte knihovnu, která umožní simulovat karetní autobattler hru Hearthstone Battlegrounds. Analyzujte problémovou doménu a proveďte rešerši podobných simulátorů. V návrhu se zaměřte především na snadnou rozšiřitelnost, jelikož jednotlivé karty se často obměňují a tak taková činnost nesmí zabrat příliš času. Naimplementujte tuto knihovnu pro jazyk Python s ohledem na využití pro trénování umělé inteligence. Otestujte knihovnu a především její výkonost. Simulace hry musí být rychlá, aby samotné trénování umělé inteligence bylo rychlé. Na základě testování proveďte případné změny (např. převeďte pomalé části kódu do C/C++, využijte Cython).

Seznam doporučené literatury:

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. David Bernhauer, Ph.D. katedra softwarového inženýrství FIT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **05.02.2023** Termín odevzdání diplomové práce: _____

Platnost zadání diplomové práce: _____

Ing. David Bernhauer, Ph.D. podpis vedoucí(ho) práce	Ing. Michal Valenta, Ph.D. podpis vedoucí(ho) ústavu/katedry	doc. RNDr. Ing. Marcel Jiřina, Ph.D. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Datum převzetí zadání

_____ Podpis studenta

Hra je komplexná a kompletné vysvetlenie pravidiel by bolo veľmi dlhé. Pár vetami sa však dá opísať takto:

"The player fills the role of a "planeswalker," so called because as a powerful mage or wizard, he or she can "walk" between different planes of the Magic multiverse, allowing him or her to collect spells and creatures along the way, which are represented by the cards used to play the game. "Mana" is the magical energy that powers spells and is represented by five colors, and was probably the inspiration for the "energy" cards in the similar Pokémon card game. Each color of mana has its own strengths and weaknesses, its own unique environment and its own traits." [5]

Koncept kreatúr, *spells* a many je znovu využitý vo viacerých ďalších kartových hrách.

Obr. 1.1: Príklad zápasu hry Magic: The Gathering [6]

Medzi nedigitálne príklady (aj keď v dnešnej dobre majú mnohé aj digitálnu verziu) týchto hier patrí napríklad:

- Pokémon Trading Card Game, či
- Yu-Gi-Oh!

Medzi čisto digitálne príklady patrí napríklad:

- Hearthstone,
- Gwent: The Witcher Card Game,
- The Elder Scrolls: Legends alebo
- Legends of Runeterra.

1.2.1 Hearthstone

Hearthstone mal byť pôvodne malý experimentálny *niche* projekt. Známe herné štúdium Blizzard malo v tej dobe tri veľké úspešné hry: *action RPG* Diablo, *real time strategy* Starcraft II a *MMORPG* World of Warcraft. [7]

V hernom priemysle sa však začali diať zmeny, vznikali nezávislé malé projekty, ktoré aj napriek malému tímu vývojárov vedeli predať svoje hry veľkej audiencii – Blizzard chcel držať krok s dobou. [7]

„*We realised we didn't have a competency here at the company to be a little bit more experimental, a little more nimble,*“ povedal v rozhovore pre Guardian Hearthstone [7] spolutvorca Jason Chayes. „*Some of our executives got together and said, we need to start up a small team, kind of a scrappy team, that can, with few resources, find ways to do more experimental types of games. We wanted to try out different ideas and platforms – things we don't usually do because our teams are focused on those key franchises.*“

Hearthstone mal teda veľmi malý tím vývojárov v porovnaní s ostatnými projektami od Blizzardu – na svojom počiatku mal *Team 5* menej ako 17 ľudí [8], v porovnaní s Diablo či World of Warcraft, ktoré mali 60 a viac členov. [7]

Team 5 sa rozhodol zamerať na vývoj smerom ku *collectible card games*, keďže veľa vývojárov z tímu a celkovo veľa ľudí z Blizzardu malo skúsenosti s takýmto typom hier. Takisto sa rozhodli zasadiť hru do sveta World of Warcraft:

„*Warcraft is a world that an awful lot of people know about,*““ povedal Jason Chayes [9], v tom čase vedúci vývoja Hearthstonu. „*So many people have become so immersed in the Warcraft world, and there's hundreds and hundreds of characters and places to draw upon. That's very important when you're making a CCG because not only do you want lots of different characters and locations players are familiar with, but you want them to have a connection too*“

V dobe písania tejto práce(druhá polovica roku 2023) obsahuje Hearthstone takmer 4000 unikátnych kariet, väčšina kariet vychádza v takzvaných expanziách, ktorých je približne 30 [10].

Hráč si môže vybrať z 11 hrdinov. Každý tento hrdina má jemu prislúchajúce karty a spolu s jednou spoločnou sadou kariet si hráč môže poskladať vlastný balík kariet, s ktorým potom bojuje proti ostatným hráčom.

Closed beta verzia Hearthstonu vyšla v auguste 2013. [11]

Plná verzia hry vyšla 11 marca 2014. [12]

Obr. 1.2: Zápas Hearthstonu [13]

1.3 Auto battler

Auto battler je nový herný žáner (takisto sa niekedy používa pojem *auto chess*). Za jeho korene sa pokladá mód vytvorený fanúšikom na MOBA hru *Dota 2* nazvaný *Dota Auto Chess*, ktorý začal naberať na popularite v roku 2019. [14]

Základ hier tohoto typu je to, že boje prebiehajú bez hocijakej interakcie s hráčom. Hráči sú akýsi *generáli* alebo *taktici*, zbierajú hernú menu, bojové jednotky, rozhodujú o umiestnení a podobne(závisí od jednotlivých merchaník konkrétnych hier).

Príklady hier tohoto žánru:

- Teamfight Tactics
- Auto Chess
- Hearthstone Battlegrounds

1.3.1 Hearthstone Battlegrounds

Hearthstone Battlegrounds bol pôvodne len takzvaný *Tavern Brawl* [15] inšpirovaný herným žánrom *autbattler*. Tavern Brawl je herný mód pre Hearthstone – je to vlastne zápas, kde sú napríklad špeciálne pravidlá či karty a mení sa každý týždeň.

V internom Blizzard prieskume o tom, na čom by bolo vhodné pracovať na Hearthstone najbližší rok či dva, bola najčastejšia odpoveď nový herný mód. A keďže v tej dobe bol populárny nový herný žáner *autochess*, tak padlo rozhodnutie zamerať sa práve na implementáciu *autochessu* v Hearthstonu. [15]

Hearthstone Battlegrounds sa rozdielne líši od normálnej hry Hearthstonu. Niektoré aspekty hry sú podobné(niektoré karty z Battlegrounds sú napríklad takmer úplne totožné s ich verziou v Hearthstone), ale základné princípy sú zásadne odlišné. Jedna hra za skladá z viacerých zápasov, hráč nekontroluje útočenie, herné zdroje hry sú iné a podobne – ako som povedal, ide o akýsi *autochess* mód Hearthstonu. Bližšie vysvetlenie herných pravidiel a mechaník bude vysvetlené v nasledujúcej kapitole.

5. novembra 2019 bola otvorená *closed beta verzia* [16], 12 novembra bola otvorená *open beta* [17] verzia.

Kapitola 2

Analýza herných mechaník a pravidiel

Táto kapitola sa zaoberá pravidlami hry. Sú tu vysvetlené základné pojmy a termíny ako napríklad *minion* či hrdina spolu s ich konkrétnymi príkladmi. Ďalej je tu takisto ukazáný základný priebeh hry a vykonávanie herných akcií.

Vysvetľovanie pravidiel však komplikujú nasledujúce skutočnosti:

- Neexistuje žiadna oficiálna príručka o pravidlách pre Hearthstone ani Hearthstone Battlegrounds. Z toho dôvôdou bude ako primárny použitá neoficálna Hearthstone Wiki. [18]

- Aj keď väčšina pravidiel a herných mechaník je rovnaká u Hearthstone aj u Hearthstone Battlegrounds, niektoré sú odlišné a je ťažké nájsť ich znenie.

Takisto by som chcel spomenúť, že niektoré pojmy ako napríklad *minion* alebo *battlecry* nebudem prekladať z angličtiny do slovenčiny. Je to z toho dôvodu, že v slovenčine(ani v čestine) neexistuje ich preklad, ktorý by znel prirodzene a dával zmysel.

2.1 Stručný prehľad

Hearthstone Battleground je herný mód v kartovej hre Hearthstone. Jedna hra obsahuje až ôsmich hráčov, ktorí medzi sebou bojujú 1v1. Každý hráč má na začiatku možnosť si vybrať jedného hrdinu z ponúkaných štyroch. Celkovo je hrdinov v dobe písania tohto textu 99 [19], z ktorých má každý nejakú unikátnu schopnosť – či už pasívnu alebo aktívnu.

Hrá pozostáva z dvoch fáz, ktoré sa opakujú, jedno kolo hry sa skladá z práve týchto dvoch fáz. V takzvanej *recruit* fáze si hráč prakticky buduje svoju „armádu“ – nakupuje alebo predáva svojich *minionov*, mení ich usporiadanie na hracej doske, používa aktívnu *hero power* svojho hrdinu a podobne. V *combat* fáze bojuje s náhodne vybraným protihráčom. Tento zápas prebieha úplne automaticky, bez akejkoľvek interakcie hráča.

Tieto pojmy budú bližšie vysvetlené v ďalších podkapitolách.

2.2 Základné vlastnosti karty

Každá jedna karta v Hearthstone Battlegrounds má: [20]

- názov karty,
- *card art* - umelecké zobrazenie karty,
- popis karty – popisuje speciálne vlastnosti karty.

2.2.1 Rarita

Niektoré karty môžu svoju *raritu*: [20]

- *Free*
- *Common*
- *Rare*
- *Epic*
- *Legendary*

Je to pozostatok z Hearthstonu – keďže Hearthstone je *collectible card game*, v hre sa získavajú karty otvorením takzvaného *card pack*. Čím väčšia rarita, tým vzácnejšie je získanie tejto karty. Karty s vyššou raritou majú taktiež zvyčajne komplexnejšie efekty. V súčasnosti v Hearthstone Battlegrounds je však rarita veľmi málo využívaná.

2.2.2 Minion

„*Minions are persistent creatures on the battlefield that will fight for their hero.*“ [21]

Minion je základná jednotka boja tejto hry. V Hearthstone Battlegrounds sa *minioni* získavajú drvivou väčšinou času tým, že sa kúpia v takzvanej *tavern* počas *recruit* fázy. *Minion* môže byť zahraný, tým sa dostane na *board*(hraciu dosku) – následne v *combat* fáze bojujú proti sebe *minioni* nepriateľských hráčov.

Minion môže mať rôzne atribúty a efekty, avšak jeho základné dve vlastnosti sú *attack*(poškodenie) a *health*(život).

- *Attack* – táto hodnota udáva veľkosť poškodenia, ktorú *minion* spôsobí pri obrane respektíve útoku nepriateľskému *minionovi*.
- *Health* – táto hodnota veľkosť života *miniona*, ak klesne na 0 alebo nižšie, *minion* zomrie.

Každý *minion* má takisto svoj *tavern tier*(*tavern tier* bude bližšie vysvetlený v sekcii o *recruit* fáze).

2.2.2.1 Enchantment

Enchantment, je špeciálny efekt prislúchajuci *minonovi*. Prospešný *enchantment* je takisto známy ako *buff*, negatívny je známy ako *debuff*.

Každý *enchantment* má svoj názov, ikonu a popis. Môže byť generovaný *minionom*, ale aj *spellom*.

Najčastejší *enchantment* je vylepšenie útoku, respektíve života *miniona*. *Enchantment*, ktorý napríklad vylepšuje útok *miniona* o 2 a život o 3 je v hre označený skratkou *+2/+3*.

Medzi príklady patria: [22]

Ancestral Infusion Taunt.

Blessing of Might +3 Attack.

Blessing of Kings +4/+4.

Divine Spirit This minion has double Health.

Humility Attack has been changed to 1.

2.2.2.2 Minion type

Ďalšia dôležitá vlastnosť *miniona* je jeho *minion type*, takisto označovaný ako *tribe* alebo *race*. *Minioni* rovnakého typu majú medzi sebou rôzne synergie. V súčasnosti v Hearthstone Battlegrounds existujú takéto typy *minionov*: [21]

Beast: *Beasts* sú zameraní na vylepšovanie atribútov a mechaniky súvisiace so smrťou.

Demon: *Demons* je *tribe*, ktorý obsahuje silných *minionov*, ktorí ale majú nejaké negatívne efekty.

Dragon: Draci sú zameraní na časté vylepšovanie atribútov. Sú veľmi silní v neskorších fázach hry a naopak sú slabší v skoršej fáze.

Elemental: *Elementals* vedia získať veľmi veľký *attack* a *health* a v neskorších fázach hry su veľmi silní.

Mech: *Mech* je všestranný *tribe*, s unikátnou *magnetic* mechanikou.

Murloc: *Murlocs* je *tribe*, ktorého *minioni* su veľmi slabí osamote, ale v skupine majú veľmi veľké synergistické efekty.

Naga: *Nagas* sú *tribe* s unikátnou *spellcraft* mechanikou a so synergiou so *spellmi* (*spells* budú vysvetlené neskôr).

Pirate: Piráti sú agresívny *tribe*, ktorý manipuluje *gold*(mena hry, ktorá sa používa na nakupovanie *minionov* a podobne) a je zameraný na rýchle nakupovanie alebo predávanie *minionov*.

Quilboar: *Quilboars* sú *tribe* s unikátnou mechanikou *blood gem*.

Undead: *Undead* je *tribe* zameraný na mechaniky súvisiace so smrťou.

2.2.2.3 Efekty

Ako bolo spomenuté, *minion* môže mať rôzne efekty. Často používané efekty majú svoju skratku – takzvané *keyword* – pre jednoduchosť. Napríklad keyword *poisonous* je ekvivalentné s *destroy any minion damaged by this.*

Príklady *keyword* efektov sú napríklad: [21]

Taunt Tento efekt je najľahšie vysvetlený príkladom – ak útočí *minion* a ako cieľ útoku má na výber dvoch *minionov*, z ktorých jeden má *taunt* a druhý nemá, tak *vždy* najprv zaútočí na *miniona*, ktorý má *taunt* efekt.

Poisonous Ako bolo spomenuté vyššie, minion s *poisonous* zničí akéhokoľvek *miniona*, ktorému spôsobí poškodenie.

Venomous Efekt podobný *poisonous*, s tým rozdielom, že *minion* s *venomous* zničí *prvého miniona*, ktorému spôsobí poškodenie.

Windfury/megawindfury *Minion* môže obyčajne vykonať útok len raz.

Windfury, respektíve *megawindfury* spôsobí to, že *minion* môže útočiť 2-krát, respektíve 4-krát.

Divine shield *Minion* s *divine shield*, ignoruje prvé poškodenie, ktoré prijme. Následne je *divine shield* odstránený.

Reborn Po tom, ako *minion* s *reborn* zomrie, je oživený s jedným životom.

Magnetic Túto mechaniku majú len *minioni*, ktorí majú *tribe mech.* Pri zahraní *miniona* s *reborn* je na výber z dvoch možností:

1. Minion je zahraný ako obyčajný *minion.*
2. Možeš vybrať ďalšieho *mecha* na hracej doske a k nemu „prilepiť“ daného *miniona* s *reborn* – všetky atribúty a efekty sa pridajú k danému *minionovi* na *boarde.*

Medzi viacslovné efekty patria napríklad: [23]

Deathrattle: X Po smrti miniona vykonaj X akciu. Príklady:

- **Deathrattle:** Summon 1/1 Imp. (V Hearthstone skratka 1/1 znamená 1 *attack* a 1 *health*)
- **Deathrattle:** Give this minion's maximum Health to another friendly minion.
- **Deathrattle:** Give a friendly undead +1/+2.

Battlecry: X Po zahraní miniona vykonaj X akciu. Príklady:

- **Battlecry:** Give your other Pirates +3 Attack.
- **Battlecry:** Give a friendly Elemental stats equal to your Tier.

Frenzy: X Ak minion prvýkrát prežije poškodenie, vykonaj raz efekt X.

Príklad:

- **Frenzy:** Gain Divine Shield.

Avenge(N): X Vždy keď zomrie N priateľských *minionov*, aktivuj akciu X. Príklady:

- **Avenge (2)**: Give another friendly Beast +6/+6.
- **Avenge (4):** This minion sells for 1 more Gold.

Overkill: X Akcia X sa aktivuje, ak nepriateľský *minion* skončí po útoku *miniona* s týmto efektom s menej ako 0 životmi. Príklad:

- **Overkill:** Deal 3 damage to the left-most enemy minion.

Spellcraft: X Na konci kola je hráčovi pridaný dočasný *spell*, ktorý trvá do konca tvojho ďalšieho kola a má X akciu. Príklady:

- **Spellcraft:** Give a minion +2 Attack until next turn.
- **Spellcraft:** Give a minion stats equal to your Tier until next turn.

Start of Combat: X Na začiatku boja vykonaj X akciu. Príklady:

- Start of Combat: Give another friendly Dragon +2/+2 and Divine Shield.

Existujú pravdaže aj menej časté ďalšie efekty, napríklad:

- If you lost your last combat, X,
- After a card is added to your hand, X,
- After a spell is played on this, X,
- After this takes damage, X,
- At the end of your turn, X,
- a mnoho iných.

2.2.2.4 Aury

Pod aurou sa rozumie nejaký kontinuálny efekt, väčšinou patriaci k určitému *minionovi*, ktorý ovplyvňuje ostatné entity v hre (tiež väčšinou ostatných *minionov*). Medzí príklady *minionov* s aurou patria napríklad: [23]

- Humming Bird – **Your other Beasts have +2 Attack.**
- Legion Overseer – **Minions in the Tavern have +2/+1.**
- Brann Bronzebeard – **Your Battlecries trigger twice.**

2.2.2.5 Battlecries

Batlecry je špeciálny efekt, ktorý sa aktivuje tým, že sa daný *minion* s *battlecry* zahrá.

Medzi príklady *battlecries* patria napríklad: [23]

- **Battlecry:** Give a friendly Murloc +1/+1.
- **Battlecry:** Give 3 friendly minions of different types +1/+1.

- **Battlecry:** Get a 2/1 Smolderwing with "Battlecry: Give a Dragon +5 Attack."
- **Battlecry:** Get a random Elemental.
- **Battlecry:** Set a minion's Attack and Health to 15.

2.2.2.6 Zlatá verzia

Každý *minion* má svoju *golden* verziu – je to prakticky verzia *miniona* s väčším poškodením a životom a vylepšenými efektami, prípadne vylepšeným *battlecry*.

Príklady normálnej a zlatej verzie *miniona* sú zobrazené na obrázkoch 2.1 a 2.2.

(a) Obyčajná verzia (b) Zlatá verzia

Obr. 2.1: Príklad *miniona* **Imprisoner** [24]

(a) Obyčajná verzia (b) Zlatá verzia

Obr. 2.2: Príklad *miniona* **Banana Slamma** [25]

2.2.3 Spell

Spell je karta, ktorú je možné zahrať a tým sa aktivuje nejaký jednorazový efekt. Zahranie *spellu* väčšinou skonzumuje danú kartu, niektoré *spells*, napríklad takzvané *secrets* majú oneskorený efekt. [26]

V Hearthstone Battlegrounds *spelly* je možné zahrať hocikedy, nemajú žiaden *cost to play*.

Väčšina *spellov* v Hearthstone Battlegrounds vylepšuje *attack* alebo *health miniona*, prípadne mu pridá napríklad nejaký efekt ako *divine shield* a podobne, niektoré však majú aj unikátnejšie efekty.

Príklady *spellov* sú zobrazené na obrázkoch 2.3.

(a) *Spell Give A Dog A Bone*

(b) *Spell Friends and Family Discount*

(c) *Spell Angler's Lure*

(d) *Gold Coin*

Obr. 2.3: Príklady *spellov* [26]

2.2.3.1 Secret

Ako som spomenul, špeciálny typ *spellu* je takzvaný *secret*. *Secret* je zahraný normálne, jeho efekt sa však sputí, až nastane skrytý *trigger event* – skrytý v zmysle, že nepriateľský hráč nepozná daný *trigger*.

Príklady *secrets* sú zobrazené na obrázkoch 2.4.

(a) *Secret Avenge* (b) *Secret Competitive Spirit*

Obr. 2.4: Príklady takzvaných *secrets* [27]

2.3 Hrdina

Hrdina reprezentuje hráča. Podobne ako *minion*, aj hrdina má *health* – život. Avšak narozdiel od *miniona*, hráč môže mať navyše aj *armor*. Hráč príjma poškodenie teda takto:

- Ak hráč nemá *armor*, tak príjmanie poškodenia funguje rovnako ako u *miniona* – od súčasného života sa odpočíta veľkosť poškodenia
- Ak hráč má *armor*, tak poškodenie najprv príjma *armor*, až potom *health*. Príklad:

  Hráč má 30 *health*, 5 *armor* a má prijať 10 poškodenia – po prijatí poškodenia skončí s 25 *health* a 0 *armor*.

Armor sa môže použíť ako balancovacia mechanika – ak nejaký hrdina dlhodobo zaostáva za ostatnými hrdinami svojou výhernou úspešnosťou(*winrate*), tak mu zvýšia *armor*, takže prežije dlhšie.

Každý hrdina má takisto svoj *tavern tier*, ktorý môže počas hry vylepšovať (každý hráč začína na *tavern tiere* 1 a maximum je 6). *Tavern tier* hlavne ovplyvňuje akých *minionov* má hráč na výber prí kúpe – hráč si totižto môže kúpiť *miniona*, ktorý má maximálny *tavern tier* rovnaký ako daný hráč.

2.3.1 Hero power

Každý jeden hrdina má svoju unikátnu *hero power*. Táto *hero power* môže byť buď aktívna alebo pasívna:

Aktívna Daný hráč ju musí manuálne aktivovať, väčšinou je použiteľná raz za kolo.

Použitie aktívnej *hero power* často stojí niekoľko jednotiek hernej meny (takzvané *gold coins*).

Pasívna Hráč ju nemusí manuálne aktivovať, je aktivovaná napríklad sama na začiatku kola, ale môže to byť napríklad aj kontinuálny efekt, ktorý platí počas celej hru.

Príklady hrdinov s pasívnou *hero power* sú napríklad: [19]

Al'Akir *Passive* – Start of Combat: Give your left-most minion Windfury, Divine Shield, and Taunt. Pre konkrétny príklad viď obrázok 2.5.

Aranna Starseeker *Passive* – After 14 friendly minions attack, the first minion you buy each turn is free.

Cap'n Hoggarr *Passive* – After you buy a Pirate, gain 1 Gold.

Kael'thas Sunstrider *Passive* – Every third minion you play gains +2/+2.

Deathwing *Passive* – Start of Combat: Give ALL minions +2 Attack permanently.

Forest Warden Omu *Passive* – After you upgrade the Tavern, gain 2 Gold.

Master Nguyen At the start of every turn, choose from 2 new Hero Powers.

(a) Portrét hrdinu *Al'akir* (b) *Hero power* hrdinu *Al'akir*

Obr. 2.5: Príklad hrdinu *Al'akir* s pasívnou *hero power* [28]

Príklady hrdinov s aktívnou *hero power* sú napríklad: [19]

Arch-Villain Rafaam *1 gold* – Next combat, get a plain copy of the first minion you kill.

C'thun *2 coins* – At end of turn, give a friendly minion +1/+1. Repeat (time, times). (Upgrades each turn!)

Captain Hooktusk *0 coins* – Remove a friendly minion. Choose one of two from a Tier lower to keep.

George the Fallen *2 coins* – Give a minion Divine Shield. Pre konkrétny príklad viď obrázok 2.6.

Guff Runetotem *2 coins* – Give a friendly minion of each Tier +2/+2.

Malygos *0 coins* – Replace a minion with a random one of the same Tier. (Twice per turn.)

Mutanus the Devourer *0 coins* – Remove a friendly minion. Spit its stats onto another. Gain 1 Gold.

(a) Portrét hrdinu *George the Fallen* (b) *Hero power* hrdinu *George the Fallen*

Obr. 2.6: Príklad hrdinu *George the Fallen* s aktívnou *hero power* [29]

Ako môžeme vidieť, každá *hero power* je pomerne unikátna.

2.3.2 Hráč a *minioni*

Hráč manipuluje s *minionmi* – nakupuje, predáva ich a mnohé iné(táto fáza hry bude vysvetlená v nasledujúcejh sekcii). Hráč môže mať *minionov* na troch hlavných miestach(zónach):

Hand – Ak hráč kúpi *miniona*, tak je *minion* umiestnený práve do *ruky*. Podobne ak hráč získa *miniona*, prípadne *spell* iným spôsobom, je umiestnený takisto do ruky.

Hráč môže zahrať z ruky *miniona* – tým je umiestnený do zóny *board*(takisto sa tomu hovorí *put into play*), prípadne môže zahrať takisto z ruky *spell*.

Hráč môže mať v ruke maximálne 10 kariet.

Board *Board* reprezentuje miesto, kde pri bojovej fáze hry bojujú proti sebe *minioni*.

Hráč môže mať na *boarde* maximálne 7 *minionov*.

Graveyard Ak počas boja *minion* zomrie, je umiestnený práve do *graveyardu*.

2.4 *Recruit* fáza hry

Recruit fáza je najdôležitejšia časť hry. Práve tu hráč hľadá statégiu ako vyhrať hru a buduje si svoj *board* – kupuje *minionov*, hrá ich, mení ich poradie a podobne.

2.4.1 Možné akcie

Základné akcie, ktoré hráč môže vykonávať počas *recruit* fázy je nákup a predaj *minionov*. Nákup a predaj prebieha takto: [23]

- Hráč má na začiatku kola vždy na výber určitý počet *minionov* na kúpu.
  - Na začiatku každej hry sa vyberú dva
  - Počet *minionov* na výber závisí od toho, aký *tavern tier* má hráč. Ak má hráč *tavern tier* 1, má na výber 3 *minionov*. Na *tavern tiere* 2, 4 a 6 je tento limit zvýšený o 1.
  - Hráč ma vždy ná výber *minionov*, ktorí majú rovnaký alebo menší *tavern tier* ako hráč.
  - Počet *minionov* v hre nie je neobmedzený.

    Existuje takzvaný *card pool*:

    * Všetci súčasní *minioni* na predaj sú odstránení z tohto *card poolu*.
    * Pri predaji *miniona* je *minion* pridaný naspäť do *card poolu*.
    * Ak je hráč odstránený z hry, všetci jeho *minioni* sú naspäť pridaní do *card poolu*.

    Presný počet *minionov* v *card poole* sa mi bohužiaľ nepodarilo nájsť, ale podľa starých informácií to bude približne odpovedať tabuľke 2.1.

Tavern tier	Počet kópií *miniona*
1	16
2	15
3	13
4	11
5	9
6	7

Tabuľka 2.1: Počet kópií *miniona* v každom *tavern tiere*

  - *Minioni* na výber sa na začiatku kola stále resetujú.

- Ak hráč získa 3 kópie rovnakého *miniona* (rátajú sa kópe z *boardu* a ruky hráča), tak sa tieto kópie *spoja* do zlatej verzie *daného miniona*, ktorý mu je následne pridaný do ruky.
- Hráč môže hocikedy predať *miniona* zo svôjho *boardu*.
- Na ukončenie *recruit fázy* má hráč 60 až 120 sekúnd (záleží od súčasného kola).

Ďalšie akcie ktoré môže hrať vykonať počas *recruit* fázy sú: [23]

Refresh Ak hráč nie je spokojný so súčasným výberom *minonov* na kúpu, môže sa rozhodnúť ich znovu náhodne vygenerovať.

Freeze Ak si hráč chce kúpiť nejakého *miniona*, no nemá dosť hernej meny (herná mena je vysvetlená hneď v ďalšej subsekcii), tak sa môže rozhodnúť použiť takzvaný *freeze*, čo mu garantuje, že na začiatku ďalšieho kola bude mať na výber práve *minionov*, na ktorých použil *freeze*.

Vylepšenie *tavern tier* Hráč sa môže rozhodnúť vylepšiť svoj *tavern tier* o 1.

Použitie *hero power* Ak má hráč aktívnu *hero power*, tak ju môže použiť práve počas *recruit* fázy.

Zahranie *miniona*, prípadne *spellu* Hráč môže zahrať *minionov*, respektívne *spelly* práve počas *recruit fázy*.

Zmenenie usporiadania *minionov* na *boarde* Hráč sa môže rozhodnúť zmeniť usporiadanie *minionov na boarde*.

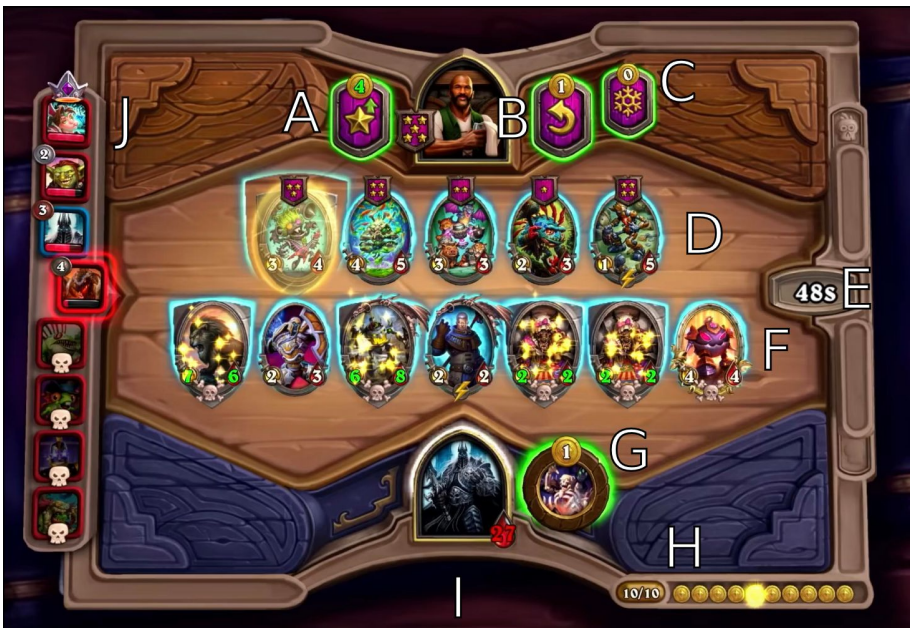
Získanie informácií o ostatných hráčov Hráč sa môže rozhodnúť zistiť určité informácie o ostatných hráčov.

Medzi tieto informácie platí napríklad:

- meno hráčov,
- koľko majú života,
- *hero power* hráčov,
- *tavern tier* hráčov,
- najčastejší *minion tribe* hráčov,

- s kým bojovali posledné dva zápasy a koľko poškodenia prijali, prípadne koľko poškodenia rozdali,
- ako dlho boli bez prehry alebo napríklad
- s kým budú bojovať ďalší zápas.

Na lepšiu predstavu o tom, ako vyzerá *recruit* fáza tu mám pripravenú jednoduchú infografiku zobrazenú na obrázku 2.7.



Obr. 2.7: UI *recruit* fázy hry
[30]

Vysvetlenie:

A Vylepšenie *tavern tieru*.

B *Refresh* ponúkaných *minionov*.

C *Freeze minionov*.

D *Minioni* ponúknutí na nákup.

E Počet sekúnd do do konca kola.

F Súčasný *board* hráča.

G *Hero power* hráča.

H Počet *coins* hráča.

I Ruka hráča.

J Informácie o ostatných hráčoch.

2.4.2 Ekonomika hry

Ekonomika hry funguje takto: [23]

- Každý hráč hry začína s práve 3 *coins*.
- Každé kolo je mu pridaný 1 *coin*, až dokým nedosiahne 10 *coins*.
- Nezáleží koľko *coins* má hráč na konci kola, na začiatku ďalšieho kola sa mu *coins* vždy resetuje na novú hodnotu (teda je zvýšené o 1 alebo na 10, ak už je hodnote 10).
- Počas kola hráč môže mať maximálne 100 *coins* – takéto veľké čísla sa dajú získať kombináciou predávania *minionov* a nejakých špeciálnych efektov *minionov* na *boarde*.

  Netreba však zabudnúť, že vždy platí vždy predchzajúce pravidlo, že na začiatku ďalšieho kola sa hráčovi *coins* vždy resetujú na novú hodnotu.
- Hráč môže minúť *coins* zvyčajne týmito spôsobmi:

  Vylepšiť *tavern tier* Hráč začína na *tavern tiere* 1 a počas hry sa môže rozhodnúť vylepšiť ho o jednu úroveň. Maximálny *tavern tier* je 6. Vylepšenie *tavern tieru* stojí 5 *coins*, každé kolo sa táto cena znižuje o 1, a po vylepšení sa cena znova resetuje na 5.

  Nakupovanie *minionov* Hráč si môže kúpiť *miniona*. *Minion* zvyčajne stojí 3 *coins*.

  Refresh Ak sa hráčovi nepáčia akí *minioni* sú momentálne na výber na kúpu, môže sa rozhodnúť získať namiesto nich náhodne nových *minionov*. Táto akcia stojí *1 coin*.

  Hero power Použitie aktívnej *hero power* stojí niekedy *coin*, zvyčajne najviac 1 až 2.
- Hráč môže získať *coins* predaním *miniona*.

  Minion sa zvyčajne predáva za 1 *coin*.

2.5 Bojová fáza hry

V bojovej fáze hry bojujú proti sebe dvaja náhodne vybraní hráči. Táto fáza prebieha bez akejkoľvek interakcie s hráčom, všetko prebieha úplne automaticky.

Počas *combat* fázy platia tieto pravidlá: [23]

- Ak nejaký hráč má pasívnu *hero power*, ktorá sa aktivuje na začiatku boja, tak sa aktivuje.
- Prvý útočí hráč s väčším počtom *minonov*. Ak obaja hráči majú rovnaký počet *minionov*, tak sa náhodne vyberie prvý hráč, ktorý útočí.
- *Minioni* útočia v smere zľava doprava. Po zaútočení posledného miniona sa poradie útočenia resetuje.

- Cieľ útoku je vždy vybraný náhodne, okrem toho, keď nepriateľskí *minioni* majú *taunt* – na *minionv* s *tauntom* sa útočí prednostne.

  Ak teda máme n nepriateľských *minionov*, z toho m má *taunt*, ako cieľ útoku sa vyberie práve jeden *minion* z m.

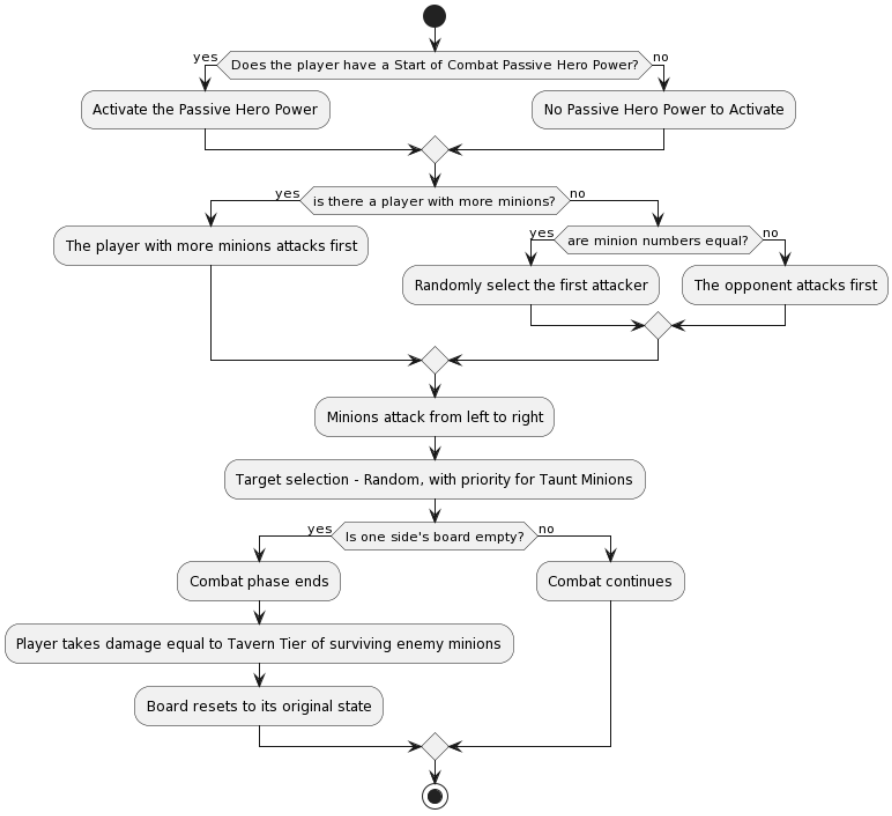
- *Combat* fáza hry sa končí vtedy, ak je jedna strana *boardu* prázdna, teda vtedy, ak jeden hráč zničil všetkých nepriateľských *minionov*.

  Hráč, ktorý prehral, následne príjme poškodenie rovné súčtu *tavern tieru* nepriateľských *minionov*, ktorí prežili.

  Na začiatku hry je toto poškodenie pravdaže malé, keďže *minioni* majú malý *tavern tier*, v neskorších fázach hry však hráč môže prijať bežne 20 a viac poškodenia za jednu prehru.

- Na konci *combat* fázy sa *board* resetuje naspäť do pôvodného stavu.

Priebeh boja je takisto zobrazený diagamom activity na obrázku 2.8.



Obr. 2.8: Diagram aktivity priebehu boja

Ak je nažive nepárny počet hráčov, tak jeden hráč z nich bojuje proti poslednému hráčovi, ktorý zomrel. Tento hráč je reprezentovaný hrdinom *Kel'Thuzad*, ktorý nemá žiadnu *hero power*, inak však boj prebieha podľa obyčajných pravidiel.

2.6 Ďalšie mechaniky

Strom akcií

Udalosti v Hearthstone sú vykonávané v postupne v takom poradí, v akom nastali(až na určité výnimky). Pri vykonávaní akcií je použitý algoritmus DFS. [31]

Zóny hry

Každá jedna karta v hre sa nachádza v určitej zóne. Pre Hearthstone Battlegrounds sú dôležité tieto zóny: [32]

Play Entita je v tejto zóne, ak sa nachádza na *boarde*.

Hand Entita je v tejto zóne, ak sa nachádza v hráčovej ruke.

Secret V tejto zóne sú zahrané *secrets*, ktoré čakajú na svoj *trigger event*.

Graveyard Do tejto zóny sa entita dostane po tom, ako zomrie.

Invalid Východisková zóna.

Kapitola 3

Návrh a implementácia

V tejto kapitole bude popísaný návrh a implementácia praktickej časti tejto diplomovej práce. Postupne sa prejde od jednoduchých komponent, cez základne entity a efekty až po fázy hry. Na konci kapitoly bude vysvetlené napojenie implementovanej knižnice na Python pomocou použitia jazyku Cython.

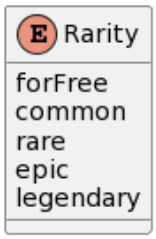
3.1 Základné komponenty

3.1.1 ICard

Táto trieda reprezentuje rozhranie pre herné karty.

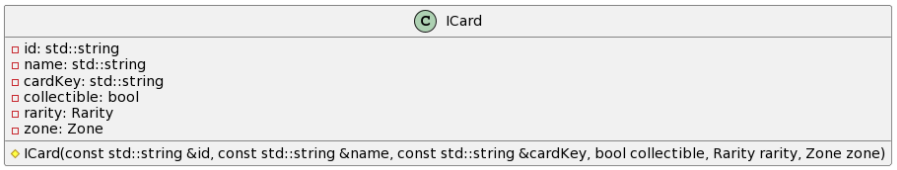
3.1.1.1 Rarity

Rarita karty je reprezentovaná enumom. To je znázornené na diagramu 3.1.



Obr. 3.1: Diagram enumu `Rarity`.

Táto trieda reprezentuje rozhranie pre herné karty.



Obr. 3.2: Diagram triedy `ICard`. Prístupové metódy sú vynechané.

Obsahuje niekoľko privátnych premenných:

`std::string id`: Unikátny identifikátor karty.

`std::string name`: Názov karty.

`std::string cardKey`: Takzvaný kľúč karty. Bude slúžiť napríklad na generovanie kariet.

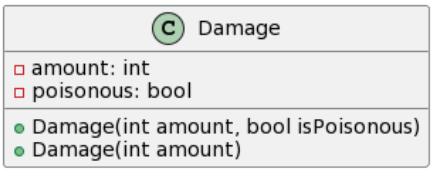
`bool collectible`: Indikátor, či sa karta dá získať normálne nákupom alebo je len napríklad vyvolaná počas boja(takzvaný *token*).

`Rarity rarity`: Enumerácia reprezentujúca vzácnosť karty

`Zone zone`: Enumerácia reprezentujúca aktuálnu zónu karty v hre.

3.1.2 Damage

Na zapuzdrenie poškodenia slúži trieda *Damage*. Jej diagram triedy je zobrazený na obrázku 3.3.



Obr. 3.3: Diagram triedy `Damage`. Prístupové metódy sú vynechané.

V súčasnej podobe je táto trieda skutočne jednoduchá. Obsahuje len 2 privátne členské premenné:

`int amount` Reprezentuje veľkosť poškodenia.

`bool poisonous` Reprezentuje či je dané poškodenie *poisonous*, teda či *minion* zomrie, ak príjme toto poškodenie.

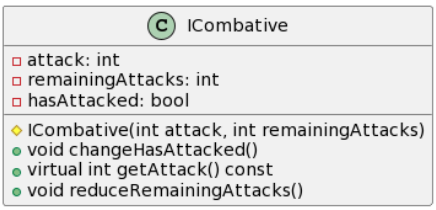
V budúcnosti môžu byť pridané aj ďalšie členské premenné, napríklad premenné súvisiace s tým či je dané poškodenie generované *spellom*, prípadne aký *spell school* ma daný *spell* a podobne.

3.1.3 Rozhrania

V tejto sekcii budú opísané rozhrania, ktoré následne implementujú jednotlivé entity *minion* a hrdina.

3.1.3.1 ICombative

Táto trieda reprezentuje rozhranie pre entity, ktoré môžu útočiť – teda majú určité poškodenie, počet zostávajúcich útokov a podobne. Jej diagram triedy je zobrazený na obrázku 3.4.



Obr. 3.4: Diagram triedy `ICombative`. Nevirtuálne prístupové metódy sú vynechané.

Trieda obsahuje tri privátne premenné:

`int attack` Táto premenná reprezentuje veľkosť poškodenia.

`int remainingAttacks` Táto premenná reprezentuje počet zostávajúcich útokov.

Minion, ktorý nemá *windfury* má túto premennú nastavenú na 1, *minion* s *windfury*, respektívne *megawindfury* má túto premennú nastavenú na 2, respektívne 4.

`bool hasAttacked` Indikátor, či entita už útočila.

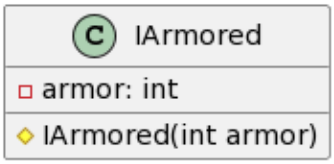
Okrem prístupových metód a konštruktoru sú dôležité tieto metódy:

`void reduceRemainingAttacks()` Táto metóda zníži počet zostávajúcich útokov o jedna.

`void changeHasAttacked()` Metóda mení stav `hasAttacked` na opačný, teda mení indikátor toho či entita útočila, respektíve, či môže znovu útočiť.

3.1.3.2 IArmored

Táto jednoduchá trieda predstavuje rozhranie pre entity, ktoré môžu mať *armor* – čo je v našom prípade prakticky len hrdina. Jej diagram triedy je zobrazený na obrázku 3.5.

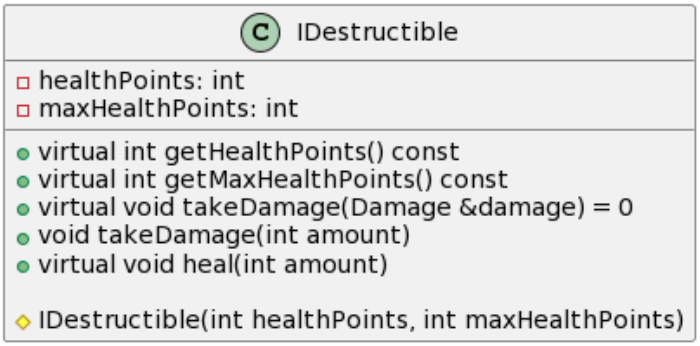


Obr. 3.5: Diagram triedy `IArmored`. Prístupové metódy sú vynechané.

Obsahuje len jednu premennú `int armor`, ktorý reprezentuje veľkosť brnenia entity.

3.1.3.3 IDestructible

Táto trieda reprezentuje rozhranie pre entity, ktoré môžu prijať poškodenie. Jej diagram triedy je zobrazený na obrázku 3.6.



Obr. 3.6: Diagram triedy `IDestructible`. Nevirtuálne prístupové metódy sú vynechané.

Trieda obsahuje dve privátne premenné:

`int healthPoints` : Predstavuje aktuálnu hodnotu života entity.

`int maxHealthPoints` : Určuje maximálnu možnú hodnotu života entity.

Okrem prístupových metód a konštruktoru sú dôležité tieto metódy:

`virtual void takeDamage(Damage &damage) = 0` – Virtuálna metóda, ktorá umožňuje entite prijať poškodenie definovanú objektom typu `Damage`.

Entity, ktoré implementujú toto rozhranie implementujú túto metódu.

`void takeDamage(int amount)` – Metóda, ktorá umožňuje entite prijať určité číselné poškodenie, definované parametrom `amount`.

Implementácia tejto metódy vyzerá takto:

```
    void IDestructible::takeDamage(int amount) {
    setHealthPoints(getHealthPoints() - amount);
    }
```

`virtual void heal(int amount)` – Virtuálna metóda, ktorá umožňuje entite vyliečiť prijaté poškodenie. Implementácia tejto metódy vyzerá takto:

```
    void IDestructible::heal(int amount) {
    if (getHealthPoints() + amount > getMaxHealthPoints())
        setHealthPoints(getMaxHealthPoints());
    else
        setHealthPoints(getHealthPoints() + amount
        );
}
```

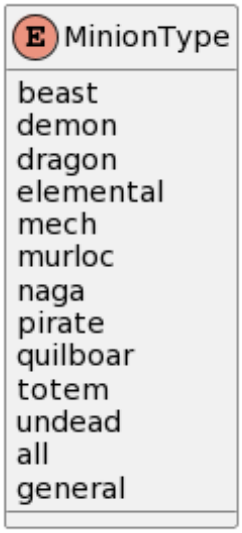
3.2 Minion

Verzia *miniona* je reprezentovaná enumom `Version`, znázorneným na obrázku 3.7.

Obr. 3.7: Diagram enumu `Version`.

3.2.1 Minion type

Typ *miniona* je reprezentovaný enumom `MinionType`, ktorý je znázornený na obrázku 3.8



Obr. 3.8: Diagram enumu `MinionType`.

Trieda `Minion` reprezentuje entitu *minion*. Táto trieda dedí zo 4 tried:

`ICard` *Minion* má základné vlastnosti karty.

`IDestructible` *Minion* môže byť zničený.

`ICombative` *Minion* môže byť útočiť.

`MinionSubject` Táto trieda a jej metódy bude vysvetlená v neskoršej sekcii 3.9.3, jednoducho povedané je to *subject* v návrhovom vzore *observer*.

Trieda obsahuje nasledujúce privátne premenné:

- `std::string id`: Členská premenná zdedená z triedy `IClass`.

  Na generovanie UUID identifikátoru je použitá trieda `IDGenerator`.

- `MinionType type`: Typ – *tribe* – miniona.

- `int tavernTier`: K akému *tavern tieru* patrí *minion*

- `std::string ownerId`: Identifikátor vlastníka *miniona*.

- `int sellCost`: Cena za predaj *miniona*.

- `Version version`: Verzia miniona.

- `std::vector<PlayerObserver*> observers`: Zoznam *observerov miniona*.

- `bool divineShield`: Indikátor toho, či *minion* má *divine shield*, teda či bude ignorovať prvé poškodenie, ktoré prijme.

- `bool taunt`: Indikátor toho, či *minion* má *taunt*.

- `bool poisonous`: Indikátor toho, či *minion* má *poisonous*.

- `std::vector<std::unique_ptr<IEffect>> effect`: Zoznam efektov *miniona*.

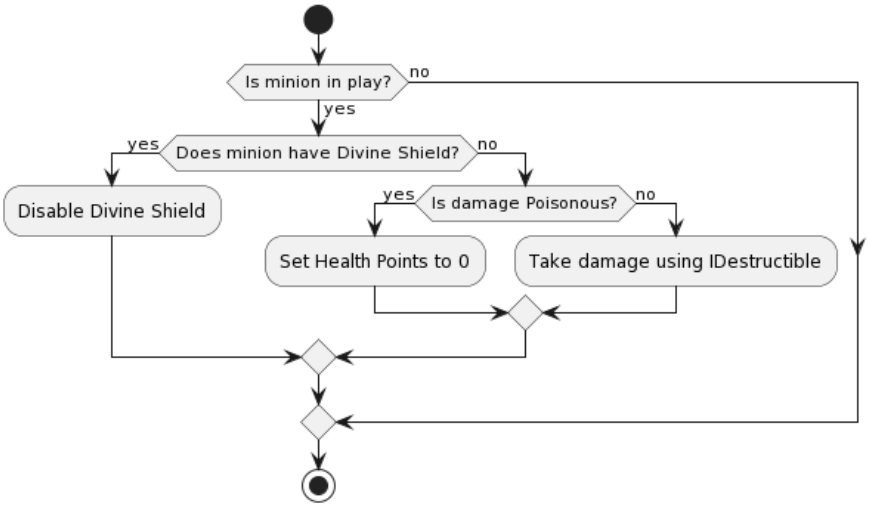
- `std::vector<Aura> auras`: Zoznam aúr *miniona*.

- `std::vector<std::unique_ptr<IBattlecry>> battlecries`: Zoznam *battlecries miniona*.

- `std::vector<std::unique_ptr<Enchantment>> enchantments`: Zoznam *enchantments miniona*.

Metódy súvisiace s efektmi, aurami a podobne budú vysvetlené v nasledujúcich sekciách ako 3.5 a 3.8. Okrem týchto metód je zaujímavá metóda `void takeDamage(Damage &damage)`.

Jej diagram aktivity je zobrazený na obrázku 3.9.



Obr. 3.9: Diagram aktivít metódy `void takeDamage(Damage &damage)` u triedy `Minion`

Táto metóda funguje takto:

- Ak sa *minion* nenachádza v zóne `play`, metóda sa ukončí.
- Ak má *minion divine shield*, tak *minion* ignoruje poškodenie, *divine shield* mu je odstránený a metóda je ukončená.
- Ak je poškodenie, ktorá má *minion* prijať *poisonous*, *health points minion* sú nastavené na 0 a metóda sa ukončí.
- Inak *minion* prijme poškodenie cez metódu `takeDamage(Damage &damage)` triedy `IDestructible`.

3.2.2 MinionRepository a MinionBuilder

Na získanie konkrétnej inštancie triedy `Minion` slúži trieda `MinionBuilder`.

Trieda obsahuje nasledujúcu metódu:

- `Minion getMinion(const std::string& cardKey)` – po zadaní `cardKey` vráti daného *miniona* s prísluchajúcim `cardKey`.

Trieda `MinionRepository` obsahuje repozitár `cardKeys` *minionov*.

Obsahuje 6 vektorov – `tier1CardKeys`, `tier2CardKeys`, ... , `tier6CardKeys`. Každý tento vektor obsahuje `cardKeys` *minionov* z daného *tavern tier*.

Ďalej obsahuje metódu `std::vector<std::string> &getTierCardKeys(int tier)`, ktorá vráti vektor *card keys* prislúchajúci danému *tavern tieru*.

Jedná sa o veľmi naivné riešenie problému, v budúcnosti bude určite potreba vytvoriť jednoduchú databázu, ktorá bude obsahovať základné informácie o daných *minionoch*.

3.3 Task

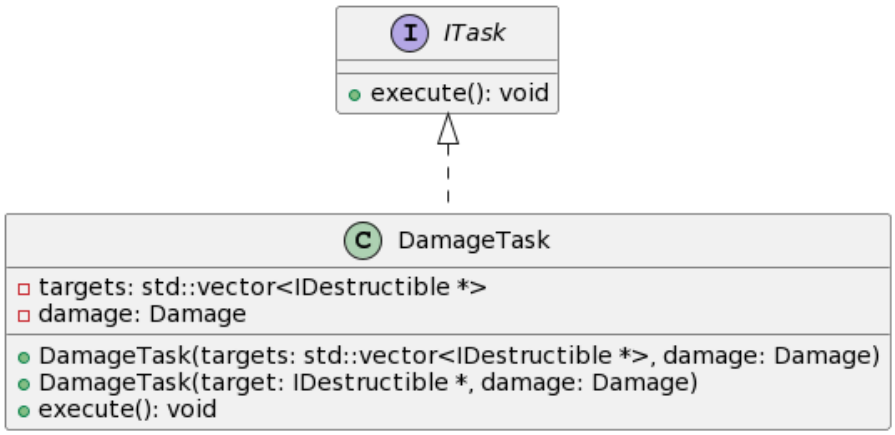
3.3.1 ITask

Pod pojmom *task*, rozumiem triedy, ktoré implementujú triedu `ITask`. Tieto triedy reprezentujú jednoduché zapuzdrenie nejakej akcie, napríklad prijaté poškodenia alebo vyvolanie *miniona*.

Trieda `ITask` je abstraktná trieda, ktorá obsahuje len jednu *pure virtual function* `void execute()`.

3.3.2 DamageTask

Ako príklad *tasku* uvediem triedu `DamageTask`. Jej diagram triedy je zobrazený na obrázku 3.10.



Obr. 3.10: Diagram triedy `DamageTask`. Prístupové metódy su vynechané.

Trieda obsahuje 2 privátne premenné:

`std::vector<IDestructible *> targets` Ciele poškodenia, ciele implementujú rozhranie `IDestructible`.

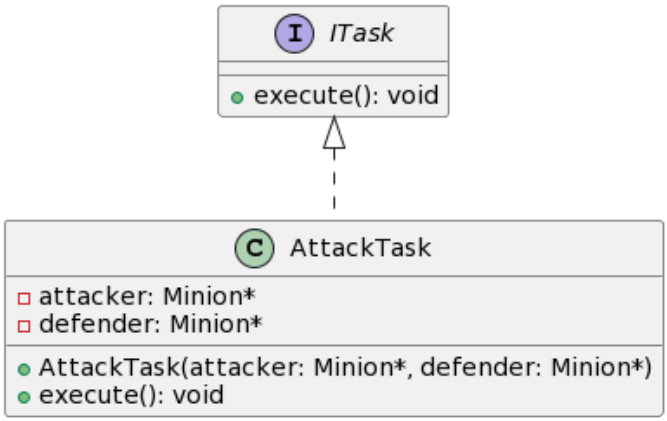
`Damage &damage` Poškodenie, ktoré majú prijať ciele.

Metóda `void execute()` následne len aplikuje poškodenie na dané ciele. Jej implementácia vyzerá takto:

```
void DamageTask::execute() {
    for (auto \&target: targets) {
        target->takeDamage(damage);
    }
}
```

3.3.3 AttackTask

Ako príklad ďalšieho *tasku* uvediem triedu `AttackTask`. Jej diagram triedy je zobrazený na obrázku 3.11.



Obr. 3.11: Diagram triedy `AttackTask`. Prístupové metódy sú vynechané.

Trieda obsahuje 2 privátne premenné:

`Minion* attacker` *Minion*, ktorý vykonáva útok.

`Minion* defender` *Minion*, ktorý je cieľom útoku.

Metóda `void execute()` následne vykonáva útok medzi útočníkom a cieľom útoku.

- Najprv si vytvoríme dve inštancie triedy `DamageTask`. Jedna inštancia bude inicializovaná veľkosťou poškodenia útočníka a indikátorom toho, či je *poisonous*, druhá bude inicializovaná veľkosťou poškodenia obrancu a indikátorom toho, či je obranca *poisonous*.
- Následne zavoláme metódu útočníka `changeHasAttacked()` a `reduceRemainingAttacks()`.
- Následne zavoláme metódu `execute()` oboch isntancií `DamageTasku`.

Implementácia logickej časti metódy `execute` vyzerá takto:

```
    auto attackerDamage = Damage(defender->getAttack(),
defender->isPoisonous());
    auto defenderDamage = Damage(attacker->getAttack(),
attacker->isPoisonous());
    auto attackerDamageTask = DamageTask(attacker,
attackerDamage);
    auto defenderDamageTask = DamageTask(defender,
                                         defenderDamage);

    attacker->changeHasAttacked();
    attacker->reduceRemainingAttacks();
```

```
        attackerDamageTask.execute();
        defenderDamageTask.execute();
```

3.4 Player

Trieda `Player` reprezentuje hrdinu v hre. Dedí zo štyroch tried:

`IDestructible` Hráč môže byť zničený.

`IArmored` Hráč môže mať brnenie.

`PlayerObserver` a `PlayerSubject` Tieto triedy a jich metódy budú vysvetlené neskôr, ale v zásade predstavuje *observera*, resp. *subject*v rámci návrhového vzoru *observer*.

Trieda obsahuje nasledujúce privátne premenné:

- `std::string id`: Unikátny identifikátor hráča, je generovaný rovnako u *miniona*.
- `int tavernTier`: Úroveň *tavern tieru*, ktorú hráč dosiahol.
- `std::vector<Minion> board`: Zoznam *minionov* na hracej ploche.
- `std::vector<Minion> graveyard`: Zoznam *minionov* v zóne *graveyard*.
- `std::vector<Minion> hand`: Zoznam *minionov* v ruke hráča.
- `std::vector<GameObserver *> observers`: Zoznam *observerov* hráča.
- `int currentGold`: Aktuálny počet *gold coins* hráča.
- `int startOfTurnGold`: Počet *gold coins* na začiatku hracieho kola.
- `const int maxMinionsOnBoard`: Maximálny počet *minionov* na hracej ploche (7).
- `const int maxMinionsInHand`: Maximálny počet *minionov* v ruke (10).
- `int levelUpCost`: Cena za zvýšenie úrovne *tavern tieru*.
- `int refreshCost`: Cena za akciu *refresh* počas *recruit fázy*.
- `int buyCost`: Cena za nákup *miniona*.
- `std::string name`: Meno hráča.

Dôležité metódy sú:

`void addMinionToBoard(Minion minion)` Ak chceme pridať *miniona* na *board*, je potreba dbať na viacero vecí:

- Treba vždy skontrolovať či hráčov *board* už nie je plný, teda či počet *minionov* na *boarde* je menší ako `maxMinionsOnBoard`.

- Treba nastaviť `ownerId` *miniona*.
- Treba zmeniť zónu *miniona* na `play`.
- Nastavenie *observerov* a registrovanie *miniona* v rámci posielania upozornení – bližšie vysvetlenie bude v sekcii 3.9.3.
- Aktualizácia áur – bližšie vysvetlenie bude v sekcii 3.8.

`void removeMinionFromBoardByIndex(int minionToRemoveIndex)` Pri odstránení *miniona* treba takisto dbať na:

- Treba nastaviť `ownerId` *miniona*.
- Treba zmeniť zónu *miniona* na `graveyard`.
- Odstránenie *observerov* a zrušenie zaregistrovania *miniona* v rámci posielania upozornení – bližšie vysvetlenie bude v sekcii 3.9.3.
- Aktualizácia áur – bližšie vysvetlenie bude v sekcii 3.8.

`int calculateDamageToDeal()` Metóda vráti súčet `tavernTier` všetkých *minionov* na *boarde*.

`void takeDamage(Damage &damage)` Táto metóda funguje takto:

- Ak je `armor` rovný 0, je zavolaná`IDestructible::takeDamage` s hodnotou poškodenia a metóda je ukončená.
- Ak je `damage amount` rovnaký ako `armor`, `armor` je nastavený na 0 a metóda sa ukončí.
- Ak je `amount` menší než `armor`, `armor` je znížený o `amount` a metóda je ukončená.
- Ak je `amount` väčší než `armor`:
  - Pôvodná hodnota `armoru` je uložená do premennej `oldArmor`;
  - `Armor` je nastavený na 0;
  - Je zavolaná metóda `IDestructible::takeDamage` s hodnotou (`damage amount - old Armor`).

Diagram aktivity tejto triedy je zobrazený na obrázku 3.12.

Metódy manipulujúce hráčove *gold coins* Ide o pomerne jednoduché metódy:

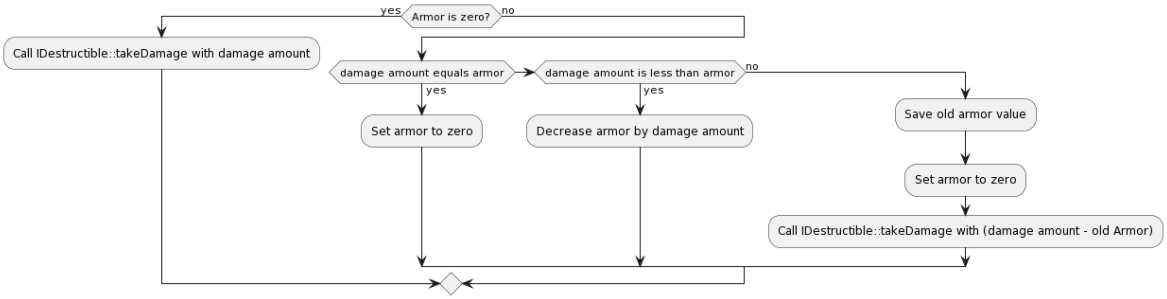
`void addGold(int amount)` Ak má hráč menej ako 100 *gold coins*, tak mu pridá `amount` *gold*.

`void spendGold(int amount)` Zníži hráčove *gold coins* o `amount`.

`void increaseStartOfTurnGold()` Ak je hráčov `startOfTurnGold` menší ako 10, tak ho zvýši o 1.

Metódy interagujúce s hráčovým *tavern tier* Ide o metódy:

- `void levelUp()` Ak je hráčov `tavern tier` menší ako 6, tak ho zvýši o 1.
- `int getMaxMinionsForTier()` Vráti maximálny počet *minionov* na kúpu v závislosti od súčasného hráčovho `tavern tieru`.



Obr. 3.12: Diagram aktivít metódy `takeDamage(Damage &damage)` v triede `Player`

3.4.1 PlayerRepository a PlayerBuilder

Veľmi podobnmý princíp ako u `MinionRepository` a MinionBuilder, viď sekciu 3.2.2.

Na získanie konkrétnej inštancie triedy `Player` slúži trieda `PlayerBuilder`.

Trieda obsahuje nasledujúcu metódu:

`Player* getPlayer(const std::string &cardKey)` Metóda po zadaní `cardKey` vráti daného ukazateľ na daného hrdinu s prísluchajúcim `cardKey`.

Trieda `MinionRepository` obsahuje repozitár `cardKeys` *minionov*.

Obsahuje 1 vektor – `playerCardKeys`. Tento vektor obsahuje `cardKeys` možných hrdinov.

Ďalej obsahuje metódu `std::vector<std::string> &getPlayerCardKeys()`, ktorá vráti vektor `playerCardKeys`.

Podobne ako u `MinionRepository`, v budúcnosti bude určite potreba vytvoriť jednoduchú databázu, ktorá bude obsahovať základné informácie o daných hrdinoch.

3.5 Effect

Väčšina efektov sa dá vyjadriť pomocou rôznych kombinácií nasledujúcich prvkov:

Trigger – Vyjadruje to, po akej udalosti sa efekt vykoná. Príklady:

- *After this take damage, do X*,
- *After this loses divine shield, do X*,
- *Whenever minion is summoned, do X*,
- *After minion gains attacks, do X* a mnoho iných.

Zdroj efektu – Vyjadruje to, od **koho** *triggerov* sa efekt vykoná. Príklady:

- *After* ***friendly pirate*** *take damage, do X*,
- *After* ***enemy minion*** *loses divine shield, do X*,
- *Whenever* ***friendly player*** *summons minion, do X*,
- *After* ***any other minion*** *gains attacks, do X* a mnoho iných.

Akcia, ktorú má efekt vykonať Príklady:

- *After friendly pirate take damage,* ***it gains +1/+1***,
- *After enemy minion loses divine shield,* ***deal 2 damage to all enemy minions***,
- *Whenever friendly player summons minion,* ***restore 2 health to all friendly minions***,
- *After any other minion gains attacks,* ***summon 1/1 Imp*** a mnoho iných.

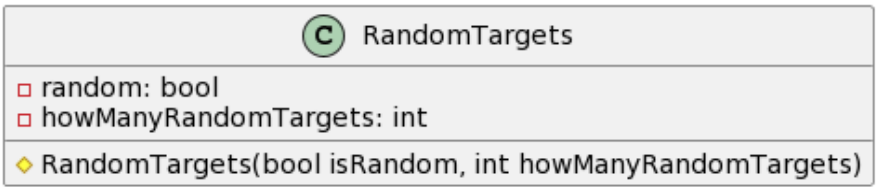
Ciele efektu Na aké ciele efekt pôsobí. Príklady:

- *After friendly pirate take damage,* ***it*** *gains +1/+1*,
- *After enemy minion loses divine shield, deal 2 damage to* ***all enemy minions***,
- *Whenever friendly player summons minion, restore 2 health to* ***all friendly minions***,
- *After any other minion gains attacks,* ***you*** *summon 1/1 Imp* a mnoho iných.

V tejto sekcii bude vysvetlený návrh a implementácia týchto prvkov.

3.5.1 RandomTargets

Táto trieda predstavuje jednoduché zapuzdrenie dát potrebných k tomu, aby sme vedeli, či budeme vyberať ciele náhodne alebo nie. Jej diagram triedy je zobrazený na obrázku 3.13.



Obr. 3.13: Diagram triedy `RandomTargets`. Prístupové metódy sú vynechané.

Trieda obsahuje dve privátne premenné:

`bool random` Táto premenná indikuje, či výber cieľov je náhodný. Ak je `true`, výber je náhodný; inak sú ciele zvolené explicitne.

`int howManyRandomTargets` Táto premenná reprezentuje počet náhodných cieľov pri náhodnom výbere.

3.5.2 Zdrojové a cieľové kritéria

V tejto sekcii bude vysvetlená reprezentácia zdrojov a cieľov efektu.

3.5.2.1 Filtre

Trieda `IFilter` predstavuje rozhranie na definovanie filtrov, ktoré možno aplikovať na entity typu `Minion` a `Player`. Poskytuje virtuálne metódy na overenie platnosti filtru. Jej diagram triedy je zobrazený na obrázku 3.18.

Pod pojmom filter myslím funkciu, ktorá berie ako parameter inštanciu triedy `Minion` a vráti `true`, respektíve `false`. Časté využitie je napríklad, ak chcem zistiť či má *minion* konkrétny *minion type*, efektoch tyu *After friendly* ***pirate*** *attacks, do X.*

Obr. 3.14: Diagram triedy `IEffect`. Prístupové metódy sú vynechané.

V triede `IFilter` sú dôležité tieto metódy:

`bool isValid(const Minion& minion) const` Táto metóda kontroluje, či daná entita typu `Minion` spĺňa kritériá daného filtra.

`bool isValid(const Player& player) const` Táto metóda kontroluje, či daná entita typu `Player` spĺňa kritériá daného filtra.

`std::unique_ptr<IFilter> clone() const` Táto metóda vráti hlbokú kópiu filtra.

Triedu `IFilter` implementujú 3 triedy:

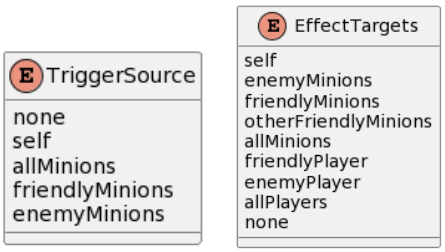
`MinionFilter` Obsahuje len filter na entity typu `Minion`.

`PlayerFilter` Obsahuje len filter na entity typu `Player`.

`MixedFilter` Obsahuje filter na entity typu `Minion` aj `Minion`.

3.5.2.2 Enumy

Zdroje a ciele efektu sú čiastočne reprezentované pomocou enumov texttttTriggerSource a `EffectTargets`. Spolu s filtrami potom poskytujú spôsob kompletnú reprezentáciu zdrojov a cieľov efektu. Tieto enumy sú zobrazené na obrázku 3.17.

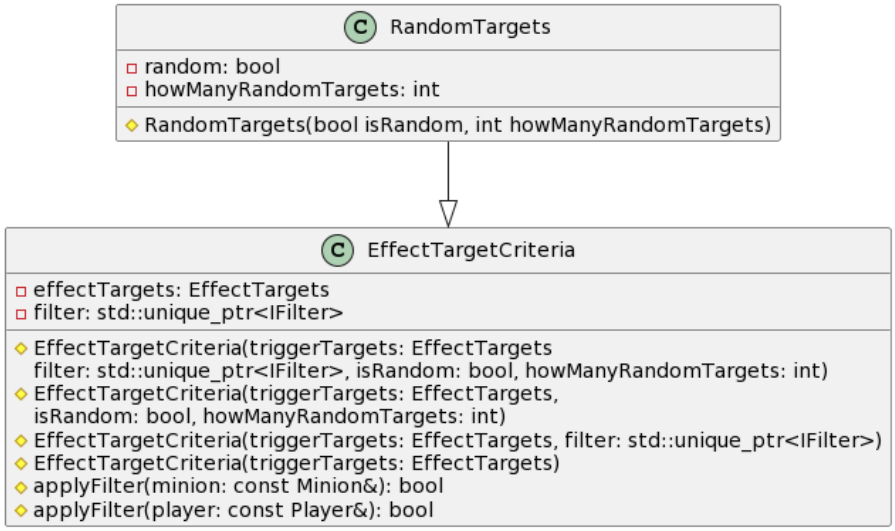


(a) Enum `TriggerSource` (b) Enum `EffectTargets`

Obr. 3.15: Diagram enumov `TriggerSource` a `EffectTargets`

3.5.2.3 EffectTargetCriteria

Táto trieda reprezentuje cieľ efektu. Jej diagram triedy je zobrazená na obrázku 3.16.



Obr. 3.16: Diagram triedy `EffectTargetCriteria`. Prístupové metódy sú vynechané.

Privátne členy triedy:

`EffectsTargets effectTargets` Reprezentuje cieľ efektu.

`std::unique_ptr<IFilter> filter`: Nepovinný filter.

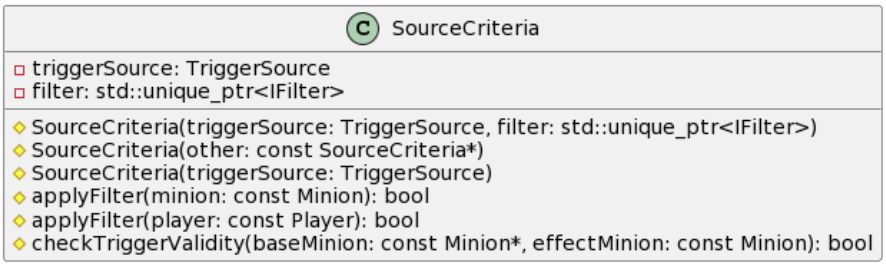
Okrem viacerých konštruktorov a prístupových metód sú dôležité tieto metódy:

`bool applyFilter(const Minion &minion)` Metóda, ktorá aplikuje filter na zadaného `miniona` a vráti `true`, ak je `minion` platný cieľ, inak `false`. Ak filter nie je inicializovaný, vráti `true`.

`bool applyFilter(const Player &player)` Metóda, ktorá aplikuje filter na zadaného hráča a vráti `true`, ak je hráč platný cieľ, inak `false`. Ak filter nie je inicializovaný, vráti `true`.

3.5.2.4 SourceCriteria

Triede `SourceCriteria` je veľmi podobná trieda `EffectTargetsCriteria`. Rozdiel je v tom, že táto trieda reprezentuje zdroj efektu. Jej diagram triedy je zobrazený na obrázku 3.17.



Obr. 3.17: Diagram triedy `SourceCriteria`. Prístupové metódy sú vynechané.

Privátne členy triedy:

`TriggerSource triggerSource` (Typ `TriggerSource`) Reprezentuje zdroj efektu.

`std::unique_ptr<IFilter> filter` Nepovinný filter.

Metódy:

`bool applyFilter(const Minion &minion) const` Rovnaký princíp ako u triedy `EffectTargetCriteria`.

`bool applyFilter(const Player &player) const` Rovnaký princíp ako u triedy `EffectTargetCriteria`.

`bool checkTriggerValidity(const Minion *baseMinion, const Minion *effectMinion) const)`

Skontroluje validitu *triggeru* – ak je napríklad `triggerSource` rovné `TriggerSource::self`, tak oba *minioni* musia mať rovnaké `id`, inak metóda vráti `true`.

Celá metóda vyzerá takto:

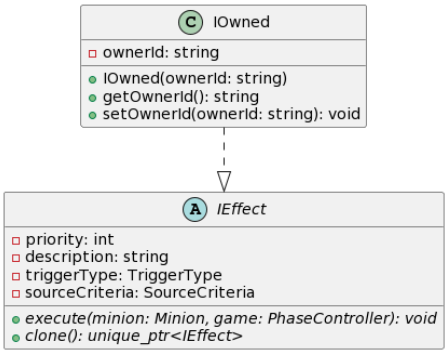
```
bool SourceCriteria::checkTriggerValidity(const Minion *
baseMinion, const Minion *effectMinion) const {
if (baseMinion == nullptr)
    return true;
if (triggerSource == TriggerSource::none) {
    return true;
} else if (triggerSource == TriggerSource::allMinions) {
    return true;
```

```
    } else if (triggerSource == TriggerSource::
    friendlyMinions) {
        return baseMinion->getOwnerId() == effectMinion->
    getOwnerId();
    } else if (triggerSource == TriggerSource::enemyMinions)
     {
        return !(baseMinion->getOwnerId() == effectMinion->
    getOwnerId());
    } else if (triggerSource == TriggerSource::self) {
        return baseMinion->getId() == effectMinion->getId();
    }
    return false;
}
```

3.5.3 IEffect

IEffect dedí z triedy `IOwned`. `IOwned` je veľmi jednoduchá trieda s jednou členskou premennou `ownerId` a prístupovými metódami. `IOwned` bude využitá na zistenie vlastníka efektu pri mechanike aury.

Diagram triedy `IEffect` je zobrazený na obrázku 3.18.



Obr. 3.18: Diagram triedy `IEffect`. Prístupové metódy sú vynechané.

Každý konkrétny efekt dedí z triedy `IEffect`.

Privátne členy triedy `IEffect`:

`int priority` Priorita efektu, východisková hodnota je 0.

`std::string description` Popis efektu.

`TriggerType triggerType` Konkrétna udalosť, ktorá vyvolá trigger. Je reprezentovaná enumom:

`SourceCriteria sourceCriteria` Zdrojové kritéria efektu.

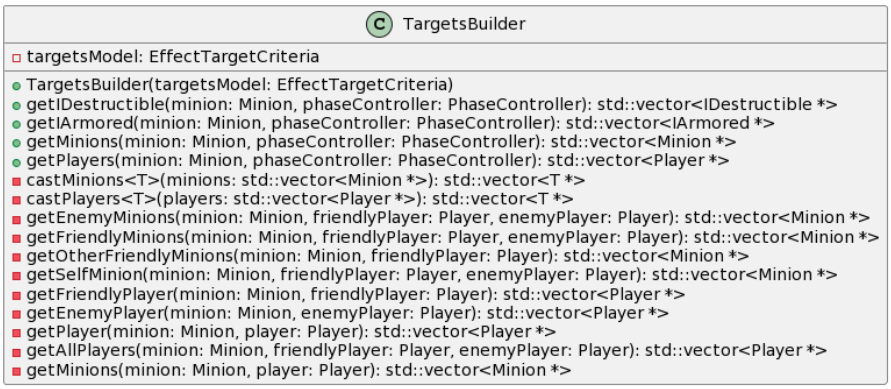
Okrem konštruktorov a prístupových metód sú dôležité tieto metódy:

`virtual void execute(Minion minion, PhaseController game) = 0`
Virtuálna metóda pre vykonanie efektu daného *miniona* v určitej fáze hry.

`virtual std::unique_ptr<IEffect>clone() = 0` Virtuálna metóda pre vytvorenie hlbokej kópie efektu.

3.5.4 TargetBuilder

Trieda `TargetBuilder` slúži na získanie cieľov efektu. Obsahuje veľké množstvo metód, tieto metódy sú však jednoduché. Jej diagram triedy je zobrazený na obrázku 3.19.



Obr. 3.19: Diagram triedy `TargetBuilder`. Prístupové metódy sú vynechané.

Privátne členy triedy:

- `EffectTargetCriteria& targetsModel`: Referencia na kritériá cieľov efektu.

Dôležité verejné metódy okrem konštruktora sú metódy:

`std::vector<IDestructible *> getIDestructible(Minion *minion, PhaseController *phaseController))`
Vráti vektor objektov implementujúcich `IDestructible`, ktoré spĺňajú dané kritériá.

`std::vector<IArmored *> getIArmored(Minion *minion, PhaseController *phaseController)`
Vráti vektor objektov implementujúcich `IArmored`, ktoré spĺňajú dané kritériá.

`std::vector<Minion *> getMinions(Minion *minion, PhaseController *phaseController)`
Vráti vektor `minionov`, ktoré spĺňajú dané kritériá.

`std::vector<Player *> getPlayers(Minion *minion, PhaseController *phaseController)`
Vráti vektor hráčov, ktorí spĺňajú dané kritériá.

Trieda `PhasController` bude vysvetlená neskôr, zatiaľ však stačí poznať iba jej tieto metódy:

- `Player* getFriendlyPlayer(Minion *minion)` a
- `Player* getEnemyPlayer(Minion *minion)`.

Tieto metódy vrátia priateľského, respektíve nepriateľského hráča vzhľadom k zadanému parametru `minion`.

Ak napríklad zavolám

```
auto vector = TargetsBuilder(effectTargets).getIDestructible(
    minion, game);
```

a premenná `effectTargets` obsahuje `EffectTargets::EnemyMinions`, funguje to takto:

1. `getIDestructible()` volá `getEnemyMinions()`:
   - Pretože `effectTargets` je nastavené na `EffectTargets::enemyMinions`, metóda `getIDestructible` volá privátnu metódu `getEnemyMinions`.
2. `getEnemyMinions()` získa *minionov* nepriateľského hráča:
   - `getEnemyMinions` získava *minionov* nepriateľského hráča pomocou `getMinions` (nepriateľský hráč sa získa z metódy parametru `phaseController`).
3. `getMinions()` filtrovanie minionov:
   - `getMinions` prechádza všetkých *minionov* hráča a filtruje ich na základe kritérií z `effectTargets`.
4. `castMinions<IDestructible>()`:
   - Na záver sa volá *template* metóda `castMinions`, ktorá používa `static_cast` na vektor inštancií triedy `Minion` a vráti vektor inštancií triedy `IDestructible`.

POdobne fungujú všetky ostatné metódy.

3.5.5 Konkrétny príklad

Na príklade `DamageEffect` si ukážeme, ako vyzerá konkrétny príklad efektu. Jej diagram triedy je zobrazená na obrázku 3.21.

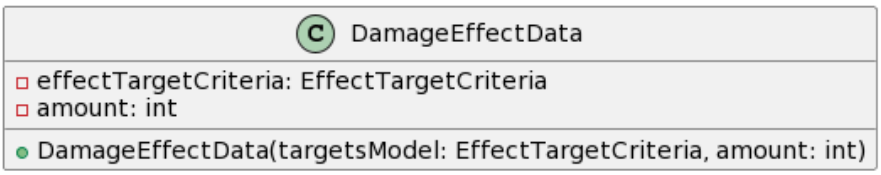
Okrem členských premenných zdedených z `IEffect` obsahuje premennú `data`, čo je inštancia triedy `DamageEffectData`. Jej diagram triedy je zobrazená na obrázku 3.20.

`DamageEffectData` obsahuje 2 privátne premenné:

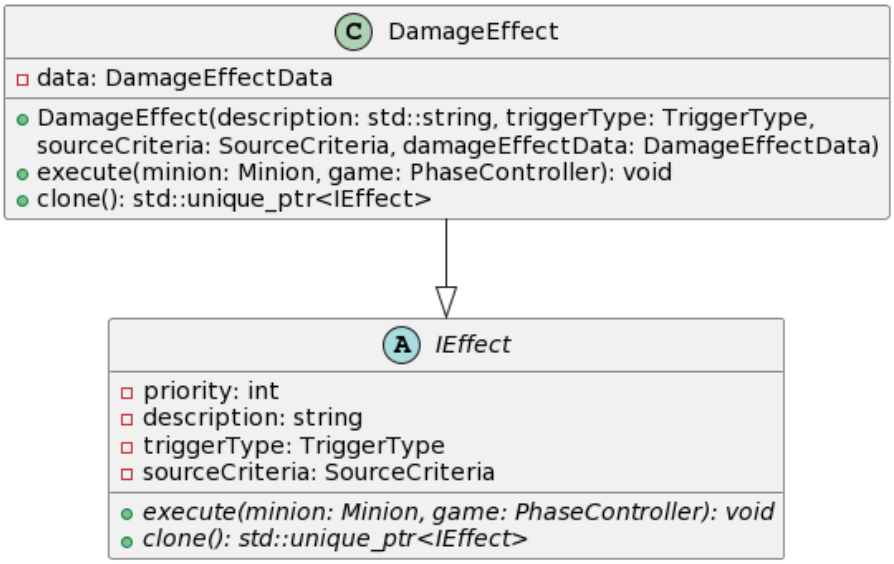
EffectTargetCriteria criteria Cieľové kritéria efektu.

int amount Veľkosť poškodenia, ktoré efekt spôsobuje.

Spolu so zdenenými premennými obsahuje všetky informácie potrebné k vykonaniu efektu.



Obr. 3.20: Diagram triedy `DamageEffectData`. Prístupové metódy su vynechané.



Obr. 3.21: Diagram triedy `DamageEffect`

Metóda `execute(Minion *minion, PhaseController *game)`, ktorú `DamageEffect` implementuje, prebieha takto:

1. Najprv sa skontroluje, či sa daný `minion` nachádza v zóne `play`. Ak nie, metóda je ukončená.
2. Získa sa vektor potrebný na vykonanie `DamageTask`.
3. Je vytvorená inštancia triedy `DamageTask` zo získaného vektora a členskej premennej `amount`.
4. Je zavolaná metóda `execute()` danej inštancie triedy `DamageTask`.

Konkrétna implementácia tejto metódy vyzerá takto:

```
void DamageEffect::execute(Minion *minion, PhaseController *game
    ) {
```

```
    if (minion->getZone() == Zone::graveyard)
        return;
    std::vector<IDestructible *> myVector = TargetsBuilder(data.
    getTargetsModel()).getIDestructible(minion, game);
    auto damage = Damage(data.getAmount());
    auto task = DamageTask(myVector, damage);
    task.execute();
}
```

3.5.6 Zhrnutie

Na podobnom princípe ako `DamageEffect` fungujú aj ostatné efekty.

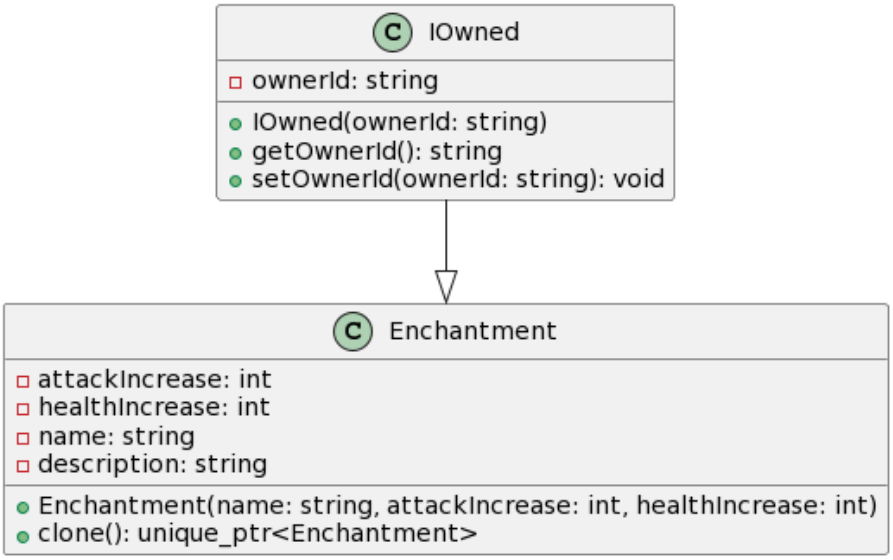
1. Z členských premenných získajú potrebné dáta.
2. Získajú ciele efektu.
3. Následne vykonajú určitý *task*.

Pridanie efektu *minionovi* je pomerne jednoduché:

1. Vytvorím konkrétnu inštanciu efektu.
2. Vytvorím z nej `std::unique_ptr` pomocou metódy `std::make_unique`.
3. Následne zavolám metódu *miniona*
   `void addEffect(std::unique_ptr<IEffect> effect)` a ako parameter jej predám konkrétny ukazateľ.

3.6 Enchantment

Na reprezentovanie *enchantmentu* slúži trieda `Enchanment`. V súčasnosti je implementovaný len zmena *attacku*, respektíve *health points miniona*, niektoré zvyšné typy *enchantmens* je však možné simulovať pomocou efektov.



Obr. 3.22: Diagram triedy `Enchantment`. Prístupové metódy su vynechané.

Obsahuje tieto privátne premenné.

`int attackIncrease` Zvyšuje hodnotu *atacku miniona* o daný počet.

`int healthIncrease` Zvyšuje hodnotu *health points miniona* o daný počet.

`std::string name` Názov daného *enchantment*.

`std::string description` Popis daného *enchantment*.

Je to jednoduchá trieda, obsahuje prakticky len prístupové metódy a metódu na vrátenie hlbokej kópie.

3.6.1 Zhrnutie

Pridanie *enchantmentu minionovi* je rovnaké ako pridanie efektu:

1. Vytvorím konkrétnu inštanciu *enchantmentu*.
2. Vytvorím z nej `std::unique_ptr` pomocou metódy `std::make_unique`.
3. Následne zavolám metódu *miniona*
   `void addEnchantment(std::unique_ptr<Enchantment> effect)` a ako parameter jej predám konkrétny ukazateľ.

V triede `Minion` prebieha interakcia s prislúchajúcimi *enchantments* takto:

- Trieda `Minion` obsahuje metódy na zistenie súčasnej zvýšenia *attacku* a *health points miniona*. Sú to metódy:

  `int getTotalHealthIncrease() const` a

`int getTotalAttackIncrease() const` . Tieto metódy získajú súčet súčasného zvýšenia *attacku*, respektíve *health points miniona*.

Implementácia metódy `int getTotalHealthIncrease() const` vyzerá napríklad takto:

```
    int Minion::getTotalHealthIncrease() const {
        int totalHealthIncrease = 0;
        for (const auto \&enchantment: enchantments) {
            totalHealthIncrease += enchantment->
    getHealthIncrease();
        }
        return totalHealthIncrease;
    }
```

- V prístupových metódach ako napríklad `int getAttack() const` alebo `int getHealthPoints() const` následne pripočítam k pôvodnej hodnote členskej premennej jej dané vypočítané zvýšenie.

  Implementácia metódy `int getHealthPoints() const` vyzerá napríklad takto:

```
    int Minion::getHealthPoints() const {
    return IDestructible::getHealthPoints() +
    getTotalHealthIncrease();
}
```

3.7 Battlecry

Battlecry, narozdiel od efektu, obsahuje len tieto prvky:

Akcia, ktorú má *battlecry* vykonať Príklady:

- ***Give** all friendly minions **+1/+1***,
- ***Give** a friendly minion **Divine Shield***,
- ***Reduce** your **next tavern tier level up cost by 1***.

Ciele efektu Na aké ciele *battlecry* pôsobí. **Veľký rozdiel je tu ten, že narozdiel od efektu ciele môžu byť vybrané aj manuálne.** Príklady:

- *Give **all friendly minions** +1/+1*,
- *Give **a friendly minion** Divine Shield*,
- *Reduce **your** next tavern tier level up cost by 1.*

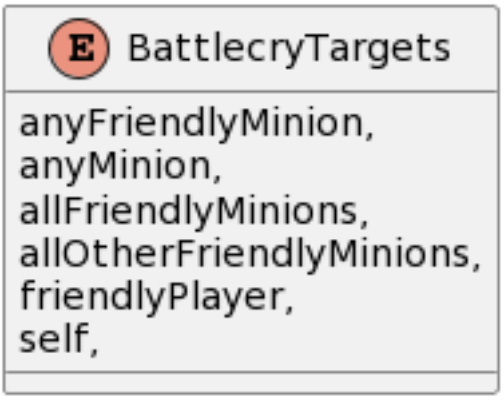
3.7.1 Cieľové kritéria

Cieľové kritéria fungujú veľmi podobne ako pri efekte, s tým rozdielom, že konkrétny enum, ktorý reprezentuje ciele je rozdielny. Kritériá cieľa pre *battlecry* reprezentuje trieda `BattlecryTargetsCriteria`. Jej diagram triedy je zobrazený na obrázku 3.23.

Obr. 3.23: Diagram triedy `BattlecryTargetCriteria`. Prístupové metódy su vynechané.

Privátne členy triedy:

`BattlecryTargets battlecryTargets` Cieľ daného *battlecry*. Je reprezentovaný nasledujúcim enumom `BattlecryTargets`, zobrazeným na obrázku 3.24.



Obr. 3.24: Diagram enumu `BattlecryTargets`.

Táto premenná reprezentuje cieľ daného *battlecry*.

`const std::unique_ptr<IFilter> filter` Nepovinný filter.

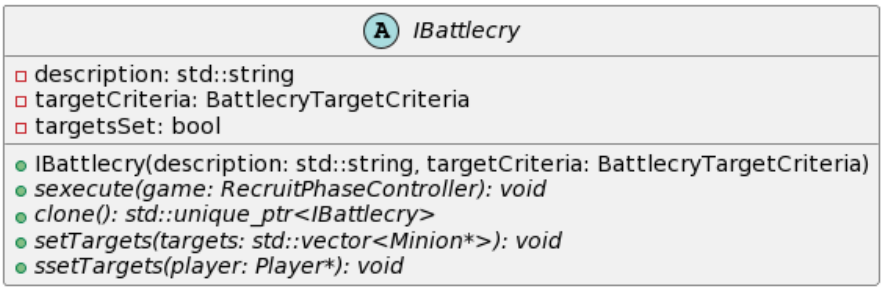
Okrem viacerých konštruktorov a prístupových metód sú dôležité tieto metódy:

`bool applyFilter(const Minion &minion)` Rovnaký princíp ako u efektu.

`bool applyFilter(const Player &player)` Rovnaký princíp ako u efektu.

3.7.2 IBattlecry

Každý konkrétny *battlecry* v hre implementuje triedu `IBattlecry`. Jej diagram triedy je zobrazený na obrázku 3.25.



Obr. 3.25: Diagram triedy `IBattlecry`. Prístupové metódy su vynechané.

Trieda obsahuje 3 privátne premenné:

`std::string description` Popis daného *battlecry*.

`BattlecryTargetCriteria targetCriteria` Kritériá cieľov pre daný *battlecry*.

`bool targetsSet` Indikátor toho, či boli nastavené ciele pre efekt. Predvolená hodnota je `false`.

Dôležité sú tieto metódy:

`void execute(RecruitPhaseController *game)` Abstraktná metóda, ktorá spustí daný *battlecry*.

`std::unique_ptr<IBattlecry> clone()` Abstraktná metóda, ktorá vytvorí hlbokú kópiu objektu `IBattlecry`.

`void setTargets(std::vector<Minion *> targets)` Abstraktná metóda, ktorá nastaví ciele pre *battlecry* – cieľ je v tomto prípade vektor *minionov*.

`void setTargets(Player *player)` Abstraktná metóda, ktorá nastaví ciele pre *battlecry* – cieľ je v tomto prípade hráč.

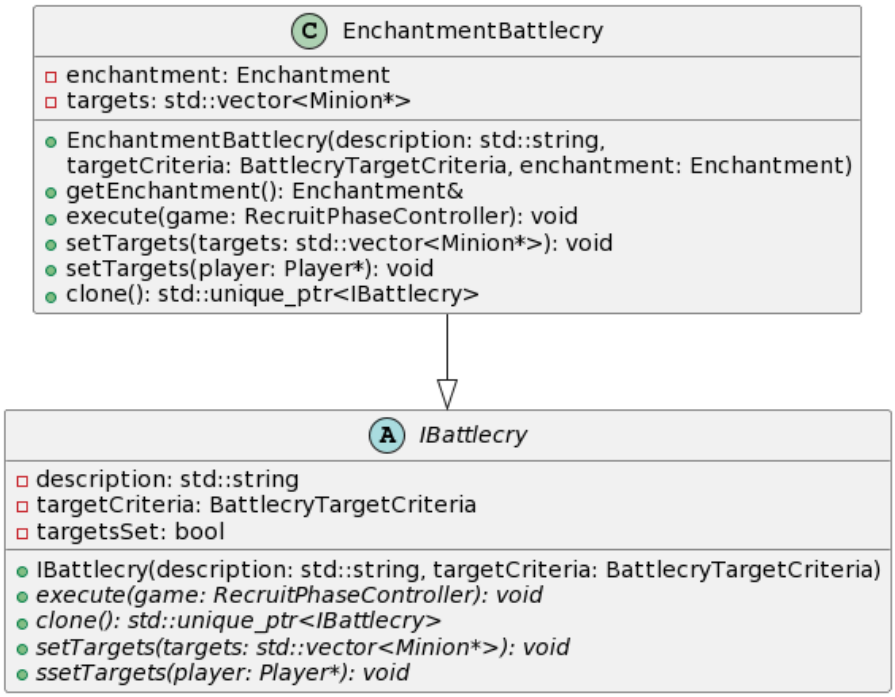
3.7.3 Konkrétny príklad

Na príklade `EnchantmentBattlecry` si ukážeme konkrétnu implementáciu *Battlecry*. Jej diagram triedy je zobrazený na obrázku 3.26.

Okrem členských premenných zdedených z `IBattlecry` obsahuje nasledujúce premenné:

`Enchantment enchantment` Konrétny *enchantment*, ktorý daný efekt aplikuje.

`std::vector<Minion *> targets` Ciele daného *battlecry*.



Obr. 3.26: Diagram triedy `EnchantmentBattlecry`. Prístupové metódy su vynechané.

Metódy na nastavenie cieľov vyzerajú takto:

`void setTargets(std::vector<Minion *> targets)` Indikátor `targetsSet` sa nataví na `true`. Následne nastavíme členskú premennú `targets`.

`void setTargets(Player *player` keďže *enchantment* môže pôsobiť len na *minionov*, je hodená výnimka.

Metóda `execute(RecruitPhaseController *game)`, ktorú `EnchantmentBattlecry` implementuje, prebieha nasledovne:

1. Najprv sa skontroluje či boli nastavené ciele pre *battlecry*. Ak nie, metóda je ukončená vyvolaním výnimky.
2. Je vytvorená inštancia triedy `EnchantmentTask` zo získaného vektora `targets` a členskej premennej `enchantment`.
3. Je zavolana metóda `execute()` danej inštancie triedy `EnchantmentTask`.

Konkrétna implementácia tejto metódy vyzerá takto:

```
void EnchantmentBattlecry::execute(RecruitPhaseController *game)
    {
    if (!targetsSet)
        throw std::runtime_error("Targets not set");
```

```
    auto enchantmentTask = EnchantmentTask(targets, enchantment)
    ;
    enchantmentTask.execute();
}
```

3.8 Aura

Trieda `Aura` reprezentuje entitu hernú mechaniku aura.

Obsahuje buď(logický XOR) efekt – ukazateľ na inštanciu triedy implementujúcu `IEffect`– alebo *enchantment* – ukazateľ na inštanciu triedy `Enchantment`.

Privátne členy triedy:

`std::unique_ptr<Enchantment> enchantment` Ukazateľ na inštanciu triedy `Enchantment`.

`std::unique_ptr<IEffect> effect` Ukazateľ na inštanciu triedy implementujúcu `IEffect`

`std::string name` Názov aury.

`std::string description` Popis aury.

`std::string ownerID` Identifikátor vlastníka aury.

3.9 Systém akcií

V tejto sekcii vysvetlím reprezentáciu stromu akcií a systém posielanie správ.

3.9.1 TreeNode

Strom akcií je reprezentovaný stromom, kde každý uzol reprezentuje rôznu akciu, ktorá sa ma vykonať:

- Je to buď nejaký `TriggerType`, napríklad *Start of Combat*.

  Koreň stromu býva zvyčajne práve `TriggerType`.
- Konkrétny efekt na vykonanie.
- Konkrétny *battlecry* na vykonanie.

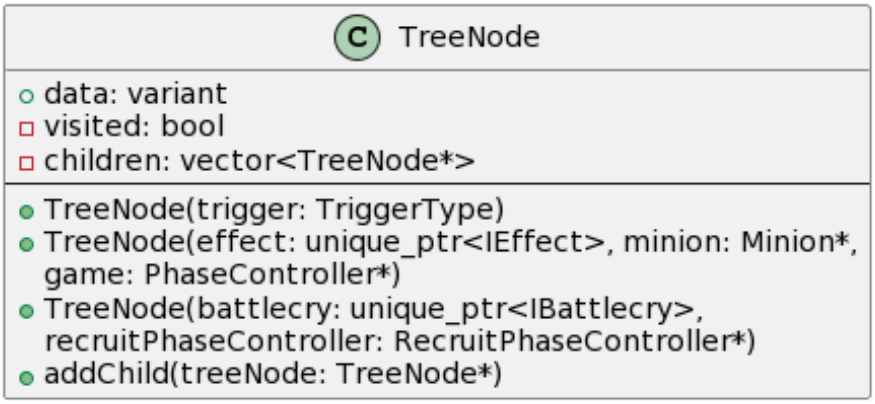
keďže efekt, respektíve *battlecry* potrebuje na zavolanie metódy `execute()` inštanciu *miniona* a inštanciu kontroléra danej fázy (fázy budú vysvetlené v sskecii 3.10), v uzli je pri efekte alebo *battlecry* uložená trojica:

- Efekt:
  - `std::unique_ptr<IEffect>`,
  - `Minion *`,
  - `PhaseController *`.

Battlecry:

- `std::unique_ptr<IBattlecry>`,
- `Minion *`,
- `RecruitPhaseController *` – keďže *battlecry* môže nastať len počas *recruit* fázy.

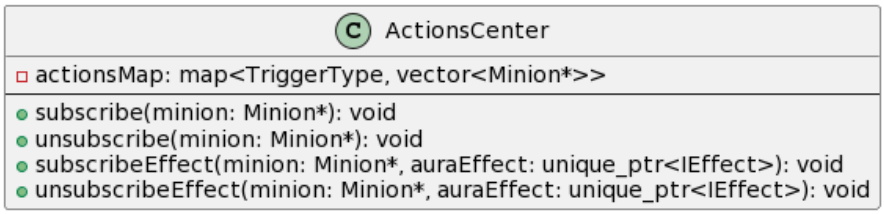
Jej diagram triedy je zobrazený na obrázku 3.27.



Obr. 3.27: Diagram triedy `NodeTree`. Prístupové metódy sú vynechané.

3.9.2 ActionsCenter

Trieda `ActionsCenter` uchováva informácie o tom, na aký daný `TriggerType` má reagovať určitý *minion*.



Obr. 3.28: Diagram triedy `ActionsCenter`. Prístupové metódy su vynechané.

Členské premenné:

`std::map<TriggerType, std::vector<Minion *>> actionsMap` Mapa, kde kľúč je `TriggerType` a hodnotou je vektor *minionov*, kde aspoň jeden efekt daného *miniona* má ako svoj `triggerType` daný kľúč *TriggerType*.

Dôležité metódy sú:

`void subscribe(Minion *minion)` Metóda na pridanie *miniona* do mapy.

- Pre každý efekt, daného *miniona*, sa získa jeho (`triggerType`).

- Následne sa pre každý `triggerType` skontroluje, či už existuje v mape `actionsMap` ako kľúč.
  - Ak neexistuje, vytvorí sa nový záznam v mape, kde kľúčom je `triggerType` a hodnotou je vektor obsahujúci jedného *miniona*.
  - Ak už existuje, zistí sa, či sa daný *minion* už nachádza vo vektore prislúcajúcemu danému `triggerType`. Ak nie, pridá sa do vektoru.

`void unsubscribe(Minion *minion)` Metóda na odstránenie *miniona* z mapy.

Pre každý záznam v mape `actionsMap` sa odstránia všetky výskyty `miniona` z prislúchajúceo vektora vektora.

`void subscribeEffect(Minion *minion, const std::unique_ptr<IEffect>& auraEffect)`
Funguje rovnako ako `void subscribe(Minion *minion)`, s tým rozdielom, že sa neprechádzajú všetky efekty *miniona*,ale získava sa `triggerType` len z `auraEffect`.

`void unsubscribeEffect(Minion *minion, const std::unique_ptr<IEffect>& auraEffect)`
Podobné ako `void unsubscribe(Minion *minion)`, s tým rozdielom, že *minion* nie je odstránený, ak má ešte nejaké iné efekty s rovnakým `triggerType` ako `auraEffect`.

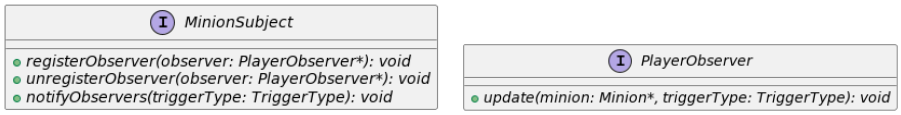
Trieda `ActionsCenter` je dôležitá pre správu a komunikáciu medzi `Minion` objektmi a efektmi v hre, zabezpečujúc registrovanie a odregistrovanie akcií a efektov pre jednotlivé `Miniony`.

3.9.3 Použitie návrhového vzoru Observer

Na registrovanie *minionov* do `ActionsCenter` a na reagovanie na nejaký `TriggerType` je využitý návrhový vzor *observer*. Diagramy tried sú zobrazené na obrázkoch 3.29 a 3.30.

V hre sú implementované 2 vzťahy *subject – observer*.

- Prvý vzťah je medzi *minionom* a hráčom:
  - *Minion* reprezentuje *subject*.
  - Hráč reprezentuje *observera*.



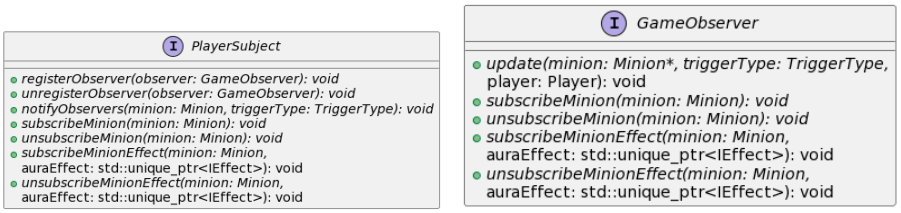
(a) *Subject* (b) *Observer*

Obr. 3.29: Prvý vzťah *subject-observer*.

- Druhý vzťah je medzi hráčom a ovládačom fázy hry:

– Hráč reprezentuje *subject*.

– Ovládač fázy hry reprezentuje *observera*.



(a) *Subject* (b) *Observer*

Obr. 3.30: Druhý vzťah *subject-observer*.

Ak chce hráč pridať určitého *miniona* do `ActionsCenter`, prebieha to takto:

1. Hráč zavolá metódu `void subscribeMinion(Minion *minion)`. Táto metóda zavolá následne pre všetkých *observerov*(ovládačov danej fázy hry) zavolá metódu `void subscribeMinion(Minion *minion)`.

2. Ovládač danej fázy hry následne volá metódu triedy `ActionCenter void subscribe(Minion *minion)`, ktorou je daný *minion* pridaný do mapy.

Veľmi podobný postup je pri tom, ak chceme zavolať metódu `unsubscribeEffect(Minion *minion, const std::unique_ptr<IEffect>& auraEffect)`.

Ak chce *minion* oznámiť, že nastal určitý `TriggerType`, prebieha to takto:

1. *Minion* zavolá metódu `notifyObservers(TriggerType triggerType)`.

2. `notifyObservers(TriggerType triggerType)` následne pre všetkých *observerov*(vlastníkov daného *miniona*) zavolá metódu `update(Minion *minion, TriggerType triggerType))`.

3. Následne hráč zavolá metódu `void notifyObservers(Minion *minion, TriggerType triggerType)`.

4. Metóda `notifyObservers(Minion *minion, TriggerType triggerType)` následne pre všetkých *observerov*(ovládačov danej fázy hry) zavolá metódu `void update(Minion*minion,TriggerType triggerType, Player *player)`.

5. `update(Minion*minion,TriggerType triggerType, Player *player)` následne spracúva dané upozornenie – vkladá nové uzly do stromu akcií.

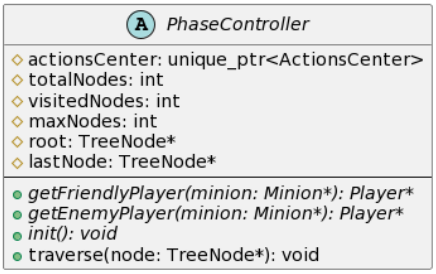
   Táto metóda bude bližšie vysvetlená v konkrétnych sekciách o fázach hry, viď 3.10.

3.10 Fázy hry

V tejto sekcii bude vysvetlená implementácia *recruit* fázy a bojovej fázy.

3.10.1 PhaseController

`PhaseController` je abstraktná trieda, ktorá poskytuje rozhranie pre riadenie hry v určitej fáze, takisto obsahuje logiku na prechádzanie stromu akcií. Jej diagram triedy je zobrazený na obrázku 3.31.



Obr. 3.31: Diagram triedy `PhaseController`. Prístupové metódy su vynechané.

Obsahuje nasledujúce privátne premenné:

`std::unique_ptr<ActionsCenter> actionsCenter`

`TreeNode *root` Koreň stromu akcií danej fázy.

`TreeNode *lastNode` Posledný spustený uzol stromu akcií danej fázy.

`int totalNodes` Udáva celkový počet uzlov v stromu akcií. Predvolená hodnota je 0.

`int visitedNodes` Udáva celkový počet navštívených uzlov v stromu akcií. Predvolená hodnota je 0.

`int maxNodes` Udáva maximálny počet uzlov. Predvolená hodnota je 250.

Obsahuje nasledujúce čisto virtuálne metódy:

- `getFriendlyPlayer(Minion *minion)`: Získa priateľského hráča vzhľadom k danému `minionovi`.
- `getEnemyPlayer(Minion *minion)`: Získa nepriateľského hráča vzhľadom k daného `minion`.
- `init()`: Inicializuje fázu hry.

Dôležitá metóda je `void traverse(TreeNode *node)`:

- Je to implementácia DFS algoritmu, slúži na prechádzanie stromu akcií danej hernej fázy.
- Ak narazím na nenavštívený uzol:
  - Ak je to inštancia `TriggerType`, tak pokračujem ďalej v DFS algoritme.

– Ak to je však trojica
`std::unique_ptr<IEffect>, Minion *, PhaseController *`
tak daný efekt vykonám – na efekt zavolám metódu `execute()` s prislúchajucou inštanciou triedy `Minion` a `PhaseController` z danej trojice. Následne pokračujem v DFS algoritme.

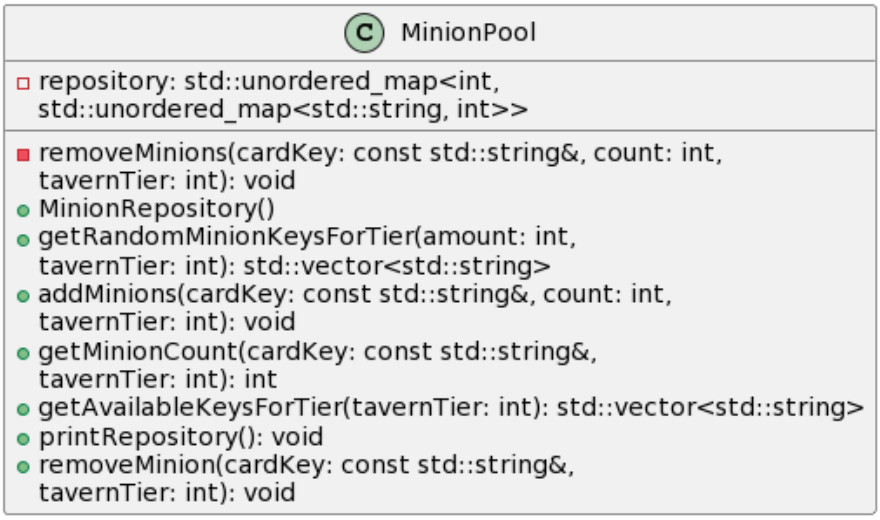
– Ak to je trojica
`std::unique_ptr<IBattlecy>,Minion*,RecruitPhaseController*`
tak pokračujem veľmi podobne ako pri efekte – získam potrebné dáta na spustenie *battlecry*, *battlecry* vykonám a pokračujem ďalej v DFS algoritme.

3.10.2 *Recruit* fáza hry

V tejto sekcii bude bližšie vysvetlený návrh a implementácia *recruit* fázy hry.

3.10.2.1 MinionPool

Táto trieda slúži na jednoduchú správu repozitára *minionov* v hre – reprezentuje takzvaný *pool minionov*, z ktorého hráči môžu *minionov* nakupovať, respektíve ich predávať naspať. Jej diagram triedy je zobrazený na obrázku 3.32.



Obr. 3.32: Diagram triedy `MinionPool`. Na diagrame nie sú zobrazené konštanty.

Obsahuje dané privátne premenné:

`std::unordered_map<int, std::unordered_map<std::string, int>> repository`
Repozitár minionov, ktorý ukladá informácie o počte `minionov` pre každý *tavern tier*.

Minion nie je reprezentovaný triedou `Minion`, ale len svojím `cardKey`.

Ďalej obsahuje 6 konštánt, ktoré reprezentujú počet *miniov* v danom *tavern tiere*, viď tabuľku 2.1. Rozdiel v porovnaní s dátami v tabuľke je ten, že ja som dané hodnoty vynásobil konštantou 3, z toho dôvodu, že zatiaľ je implementovaný menší počet *minionov* ako v reálnej hre.

Medzi dôležité metódy triedy patria napríklad:

`void addMinions(const std::string &cardKey, int count, int tavernTier)`
Pridá zadaný počet kópií *miniona* reprezentovaného `cardKey` pre daný `tavernTier`.

`void removeMinions(const std::string &cardKey, int count, int tavernTier)`
Odstráni zadaný počet kópií *miniona* reprezentovaného `cardKey` pre daný `tavernTier`.

`int getMinionCount(const std::string &cardKey, int tavernTier)`
Získa počet *minionov* pre daný `tavernTier` a `cardKey`.

`std::vector<std::string> getAvailableKeysForTier(int tavernTier)`
Získa všetky dostupné *card keys* pre daný `tavernTier`.

`void removeMinion(const std::string &cardKey, int tavernTier)`
Odstráni jedného miniona pre daný `cardKey` a `tavernTier`.

`getRandomMinionKeysForTier(int amount, int tavernTier)`
Získa náhodné *card keys* pre daný `tavernTier` a zadaný počet `minionov`.

3.10.2.2 UserInputController

Táto jednoduchá trieda slúži len príjímanie uživateľovho vstupu.

Má len jednu metódu – `std::string getUserInput(const std::string &prompt)` – ktorá získa vstup od používateľa prostredníctvom štandardného vstupu a vráti vstup reprezentovaný reťazcom.

3.10.2.3 RecruitPhaseController

Táto trieda implementuje triedy:

- `GameObserver` a
- `PhaseController`.

Obsahuje nasledujúce privátne premenné:

- `std::vector<Minion> minionsToPurchase`: Vektor obsahujúci `minionov`, ktorí sú dostupné na zakúpenie.
- `bool isFrozen`: Indikuje, či sú *minioni* na zakúpenie *frozen*, teda či hráč použil akciu *freeze*.
- `int currentTurn` Reprezentuje aktuálny ťah v hre, teda koľko ťahov uplynulo od začiatku hry.
- `Player *player` Ukazovateľ na hráča priradeného k tejto rekrutovacej fáze.

- `MinionPool *minionPool` Ukazovateľ na *pool* `minionov`.

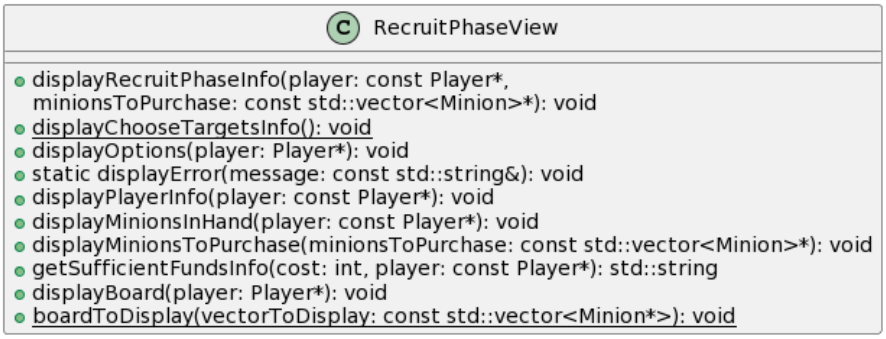
  Táto premenná je zdieľaná medzi všetkými hráčmi hry.

- `std::mutex minionPoolMutex` Mutex na riadenie prístupu k repozitáru `minionov`.

- `UserInputController userInputController` Slúži na príjímanie vstupu od užívateľa

Dôležitá premenná je `RecruitPhaseView recruitPhaseView`.

Trieda `RecruitPhaseView` slúži na zobrazovanie informácií o hráčovi a stavu hry počas *recruit* fázy. Metódy triedy sú jednoduché, zobrazujú dané informácie na štandardný výstup. Jej diagram triedy je zobrazený na obrázku 3.33.



Obr. 3.33: Diagram triedy `RecruitPhaseView`

Dôležité metódy triedy sú:

`Player *getFriendlyPlayer(Minion *minion) override` : keďže v *recruit* fáze sa nachádza len jeden hráč, táto metóda vráti vždy členskú premennú `player`.

`Player *getEnemyPlayer(Minion *minion) override` : Keďže v *recruit* fáze sa nenachádza priateľský hráč, táto metóda vždy vráti `nullptr`.

`void sortEffects(std::vector<std::tuple<Minion *, std::unique_ptr<IEffect>>>&effectsToExecute)`
Triedi efekty na základe priority efektu a poradia `minionov`.

`void subscribeMinion(Minion *minion) override`] Metóda na registráciu `miniona` do `actionsCenter`.

Podobnú funkciu ma aj metóda `subscribeMinionEffect(Minion *minion, const std::unique_ptr<IEffect> &auraEffect)`.

`void unsubscribeMinion(Minion *minion) override` Metóda na zrušenie registrácie `miniona` do `actionsCenter`.

Podobnú funkciu ma aj metóda `unsubscribeMinionEffect(Minion *minion, const std::unique_ptr<IEffect> &auraEffect)`.

`void beginTurn()` Iniciuje začiatok ťahu počas *recruit* fázy.

`void refreshMinionsToPurchase()` Nahradí dostupných `minionov` na zakúpenie za nových.

Keďže v tejto premennej pristupujem k zdieľanej premennej `minionPool`, tak v tejto metóde je využitý `std::lock_guard` na riadenie prístupu k `minionPool`.

`void changeFreeze()` Zmení indikátor `isFrozen` na opačný.

Metódy spracovanie reakcií na užívateľský vstup:

`handleOption(const std::string& option):` Spracováva užívateľský vstup.

Rozhoduje o spúšťaní jednotlivých akcií v *recruit* fáze.

`handleBuyMinionOption():` Spracováva akciu zakúpenia `miniona` z dostupných `minionov` z *minionsToPurchase*.

`handleLevelUpOption():` Spracováva akciu vylepšenia *tavern tieru* hráča.

`handleRefreshMinionsOption():` Spracováva akciu obnovenia dostupných `minionov` na zakúpenie.

`handleSellMinionOption():` Spracováva akciu predaja `miniona` z hráčovej hracej dosky.

Podobne ako v `void refreshMinionsToPurchase()`, v tejto metóde je využitý `std::lock_guard` na riadenie prístupu k `minionPool`.

`handlePlayMinionOption():` Spracováva akciu zahrania `miniona` z ruky na hráčovu dosku.

Pri tejto metóde sa spúšťajú všetky *battlecries* daného *miniona*, ktorého hráč chce zahrať. Na manuálny výber cieľov *battlecry* – ak daný *battlecry* vyžaduje manuálny výber cieľov – slúžia metódy ako:

- `setMinionsToPurchaseBattlecryTargets`,
- `setFriendlyPlayerBattlecryTargets` alebo
- `setAnyMinionBattlecryTargets`.

Tieto metódy umožnujú hráčovi manuálne vybrať cieľ daného *battlecry* pomocou štandardného vstupu.

`handleSwapMinionsOption():` Spracováva akciu výmeny pozícií dvoch `minionov` na hráčovej hernej doske.

Metódy:

- `update(Minion *minion, TriggerType triggerType, Player *player)` a
- `update(TriggerType triggerType)`

budú vysvetlené v sekcii 3.10.4.

3.10.3 Bojová fáza

3.10.3.1 CombatPhaseController

Táto trieda implementuje triedy:

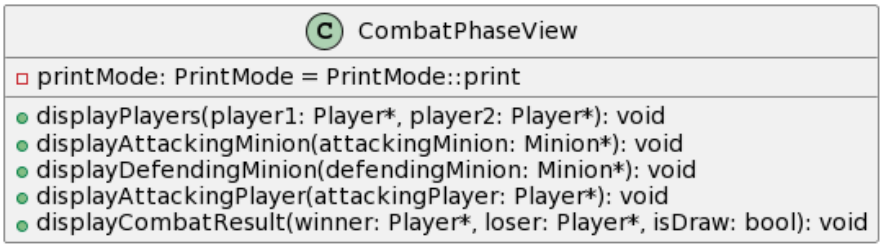
- `GameObserver` a
- `PhaseController`.

Obsahuje nasledujúce privátne premenné:

`Player *player1` Ukazovateľ na prvého hráča v súboji.

`Player *player2` Ukazovateľ na druhého hráča v súboji.

`CombatPhaseView combatPhaseView` Jednoduchá trieda na zobrazovanie informácií o súčasnom súboji. Jej členská premenná `printMode`, ktorá je inštancia enum `PrintMode` – ktorý môže mať hodnoty `PrintMode::print` alebo `PrintMode::noPrint` – rozhoduje o tom, či trieda bude vypisovať informácie na štandardný výstup. Jej diagram triedy je zobrazený na obrázku 3.34.



Obr. 3.34: Diagram triedy `CombatPhaseView`

Dôležité metódy triedy sú:

`void init()` : Inicializačná metóda, ktorá registruje všetkých `minionov` na doskách hráčov do `ActionsCenter`.

`void sortEffects(std::vector<std::tuple<Minion *, std::unique_ptr<IEffect>>> &effectsToExecute)`
Metóda triedi efekty podľa ich priority a poradia `minionov`.

`Player *getFriendlyPlayer(Minion *minion)` Metóda, ktorá vráti priateľského hráča, ktorému patrí daný `minion`.

`Player *getEnemyPlayer(Minion *minion)` Metóda, ktorá vráti hráča, ktorý je protivníkom hráča, ktorému patrí daný `minion`.

`void checkForDeadMinions()` Táto metóda odstraňuje mŕtvych *minionov* z hracej dosky hráčov.

Prebieha takto:

- Metóda skontroluje *board* oboch hráčov.

- Ak má *minion* 0 alebo menej *healthPoints*:
  1. Odstráni miniona zo zóny `play` a presunie ho do zóny `graveyard`.
  2. Vyvolá `TriggerType::deathrattle`.
  3. Odregistruje *observerov minona*.
  4. Nahradí *miniona* v cintoríne novým *minionom* pomocou metódy `getMinion(const std::string &cardKey)` triedy `MinionBuilder` – je to z toho dôvodu, že *minion* v zóne *graveyard* má mať len pôvodné vlastnosti – útok, život, efekty atď.

`void checkWinCondition()` Metóda, ktorá kontroluje výhru v súboji:

- Ak majú obaja hráči prázdny *board*, výsledok zápasu je remíza.
- Inak vyhráva hráč, ktorý nemá prázdny *board* a hráč, ktorý prehral prijme poškodenie rovné číslu, ktoré získa z metódy `calculateDamageToDeal()`.

`Player *getPlayerById(std::string id)` Metóda, ktorá vráti hráča na základe jeho identifikátora.

int findAttacker(Player *player) Vráti index *miniona*, ktorý má ako ďalší útočiť.

int findDefender(Player *player) Vráti index *miniona*, ktorý má byť obranca útoku.

`void executeCombat()` : Hlavná metóda, ktorá riadi priebeh súboja medzi hráčmi.

Prebieha takto:

- Najprv sa vyberie ktorý hráč začne súboj na základe počtu *minionov* na hracej doske.
- Následne prebiehajú útoky, až dokým hracia doska aspoň jedného hráča nie je prázdna:
  - Vyberie sa útočník a obranca pomocou metód `int findAttacker(Player *player)` a `int findDefender(Player *player)`.
  - Útočník následne vykoná toľko útokov, podľa toho aká veľká je hodnota `remainingAttacks`(alebo pravdaže dokým nezomrie).
  - Je zavolaná metoda `checkForDeadMinions()` na kontrolu, toho, či nejakí *minioni* nemajú byť odstránení.
  - Na konci útoku sa vymenia útočiaci a obraňujúci hráč.
- Na konci sa kontroluje víťaz súboja pomocou metódy `checkForDeadMinions()` a súboj je ukončený.

Metódy na registráciu *minionov* do `actionsCenter` fungujú rovnako u triedy `RecruitPhaseController`.

`Update` metódy budú vysvetlené v nasledujúcej sekcii 3.10.4.

3.10.4 Reagovanie na zmeny

Na reagovanie na *triggery* a následné aktualizovanie stromu akcií slúžia metódy:

- `void update(Minion *minion, TriggerType triggerType, Player *player)`,
- `void update(TriggerType triggerType)`.

Dôležitá je metóda `update(Minion *minion, TriggerType triggerType, Player *player)`. Táto metóda je rovnaká v oboch triedach `CombatPhaseController` aj `RecruitPhaseController`, v budúcnosti však teito implementácie môžu byť rozdielne a z toho dôvôdu som sa rozhodol k tomu, aby nebola spoločná pre obe fázy.

Máto metóda funguje takto:

1. Na začiatku sa overuje či sa nedosiahol maximálny počet uzlov v strome (`maxNodes`). Ak áno, metóda sa ukončí.
2. Inicializuje premenná `isFirst`, ktorá indikuje, či strom prázdny.
3. Ďalej z `actionsCenter` získame všetkých *minionov*, ktorí reagujú na daný `triggerType`.
4. Následne z týchto *minionov* získame všetky efekty, ktoré:
   - reagujú na daný `triggerType`.
   - Zdroj, ktorý vyvolal `triggerType`, splňuje filtre:
     - Teda metóda `bool applyFilter(...)` vracia `true`.
   - Zdroj, ktorý vyvolal `triggerType`, je validný:
     - teda metóda `bool checkTriggerValidity(const Minion *baseMinion, const Minion *effectMinion)` vracia `true`.
5. Ak žiaden taký efekt nie je, tak metóda je ukončená.
6. Keď je strom akcií prázdny, vytvorím nový uzol `TreeNode(triggerType)` a ten nastavím ako koreň.
7. Ďalej zoradím vektor efektov pomocou metódy *sortEffects* a postupne z nich vytváram nové uzly a tie pridávam ako deti súčasného koreňa.
8. Ak strom je tvorený iba jedným koreňom, tak spustím DFS algoritmus cez metódu `traverse(root)`.

   DFS algoritmus nam stačí spustiť iba na tomto mieste.
9. Na konci metódy kontrolujem, či som navštívil všetky uzly stromu.

   Ak áno, tak resetujem `totalNodes`, `visitedNodes`, `root` a `lastNode` na pôvodnú kontrolu a skontrolujem či nejakí *minioni* nezomreli cez metódu `checkForDeadMinions()`.

Konkrétna implementácia vyzerá takto:

```
void CombatPhaseController::update(Minion *minion,
TriggerType triggerType, Player *player) {
if (totalNodes == maxNodes)
    return;
bool isFirst = this->root == nullptr;
std::vector<std::tuple<Minion *, std::unique_ptr<IEffect>>>
effectsToExecute;
std::vector<Minion *> minions(actionsCenter->getActionsMap()
[triggerType].begin(),
                              actionsCenter->getActionsMap()
[triggerType].end());
if (minions.empty())
    actionsCenter->getActionsMap().erase(triggerType);

for (auto \&subscribedMinion: minions) {
    const auto \&effects = subscribedMinion->getEffects();
    for (const auto \&effect: effects) {
        /*
        if minion effect is triggered by a that trigger type
            (e.g. if the effect is triggered by after a
minion attacks, the trigger type must be afterAttack)
        and if the minion that triggered the effect is a
valid source
            (e.g. if the effect is triggered by a friendly
minion, the source must be a friendly minion)
        and if the minion that triggered the effect passes
the filter
            (e.g. if the effect is triggered by a pirate,
the source must be a pirate)
        only then we add the effect to the list of effects
to execute
         */
        if (effect->getTriggerType() == triggerType \&\&
            effect->getSourceCriteria().checkTriggerValidity
(minion, subscribedMinion) \&\&
            effect->getSourceCriteria().applyFilter(*
subscribedMinion)) {
            effectsToExecute.emplace_back(subscribedMinion,
effect->clone());
        }
    }
}
if (effectsToExecute.empty()) {
    return;
}
if (root == nullptr) {
    root = new TreeNode(triggerType);
    totalNodes++;
    lastNode = root;
```

```
    }
    sortEffects(effectsToExecute);
    for (const auto \&tuple: effectsToExecute) {
        auto effect = std::get<1>(tuple)->clone();
        auto minionPtr = std::get<0>(tuple);
        auto *effectNode = new TreeNode(effect->clone(),
    minionPtr, this);
        lastNode->addChild(effectNode);
        totalNodes++;
    }
    if (isFirst) {
        traverse(root);
    }
    if (totalNodes == visitedNodes \&\& totalNodes != 0) {
        totalNodes = 0;
        visitedNodes = 0;
        root = nullptr;
        lastNode = nullptr;
        checkForDeadMinions();}}
```

Metóda `void update(TriggerType triggerType)` len volá túto metódu a za parametre `minion` a `player` nastaví `nullptr`.

3.11 Game Manager

Trieda `GameManager` spravuje stav a priebeh hry.

Obsahuje nasledujúce premenné:

- `const int minPlayers`: Konštanta reprezentujúca minimálny počet hráčov v hre. Predvolená hodnota je 2.
- `const int maxPlayers`: Konštanta reprezentujúca maximálny počet hráčov v hre. Predvolená hodnota je 8.
- `alivePlayers`: Zoznam hráčov, ktorí sú stále nažive.
- `deadPlayers`: Zoznam hráčov, ktorí boli sú mŕtvi.
- `lastDeadPlayer`: Odkaz na posledného eliminovaného hráča.
- `minionPool`: Inštancia triedy `MinionPool`.
- `currentTurn`: Aktuálny ťah v hre.

Dôležité metódy sú:

`void addPlayer(Player *player)` Pridá hráča do hry.

`std::string addPlayer(const std::string &playerKey)` Pridá hráča identifikovaného kľúčom do hry. Návratová hodnota je `id` hráča.

`void movePlayerToDead(const std::string &playerId)` Presunie hráča s daným `playerId` do zoznamu mŕtvych hráčov.

`void startGame()` Pokiaľ `alivePlayers` neobsahuje len jedného hráča – výhercu – tak:

- Pre každého hráča z `alivePlayers` spustím *recruit* fázu.
- Následne vytvorím náhodné dvojice hráčov z `alivePlayers`, ak je počet živých hráčov párne čísla, inak vytvorím dvojicu z `alivePlayers` + `lastDeadPlayer`.

  Následne spustím pre túto dvojicu hráčov *combat fázu*.

```
void startCombatPhase(const std::string &playerId1,
const std::string &playerId2)
```

Začne bojovú fázu medzi dvoma hráčmi. Hráč je identifikovaný jeho `id`.

`void startRecruitPhase(const std::string &playerId)` Začne fázu náboru pre daného hráča. Hráč je identifikovaný jeho `id`.

```
void executeSingleCombat(const std::string &playerId1,
const std::vector<std::string> &minionIds1,
const std::string &playerId2,
const std::vector<std::string> &minionIds2,
bool isPrint = true)
```

Táto metóda vytvorí pomocou triedy `PlayerBuilder` hráčov pomocou ich *cardKeys* `playerId1` a `playerId2`. Indikátor *isPrint* rozhoduje o tom, či na štandardný výstup bude vypísaný priebeh boja.

Potom hráčovi 1 a 2 pridá pomocou triedy`MinionBuilder` *minionov* pomocou ich *cardKeys* z `minionIds1` a `minionIds2`.

Následne spustí *combat* fázu pre oboch hráčov.

3.12 Python integrácia

Kedže táto implementácia je kompletne napísaná v jazyku C++, ale cieľom tejto diplomovej práce je knižnica v jazyku Python, musíme byť možné spúšťať časti nášho kódu v jazuku Python.

Existuje viac možností ako integrovať C++ a Python, napríklad:

SWIG SWIG je nástroj pre vývoj softvéru, ktorý spája programy napísané v jazykoch C a C++ s rôznymi vysokoúrovňovými programovacími jazykmi. SWIG sa používa s rôznymi cielovými jazykmi, vrátane bežných skriptovacích jazykov ako Javascript, Perl, PHP, Python, Tcl a Ruby. [33]

SWIG sa obvykle používa na spracovanie rozhraní v jazykoch C/C++ a generovanie takzvaného *glue code* – časti kódu potrebnej pre vyššie uvedené cielové jazyky, aby mohli volať kód v jazykoch C/C++. [33]

Boost.Python Súčasť Boost knížníc, Boost Python Library je *framework* na integráciu Pythona a C++. Umožňuje rýchlo a jednoducho *expose* triedy, funkcie a objekty C++ Pythonu a naopak, bez použitia špeciálnych

nástrojov - len s C++ kompilátorom. Je navrhnutá na neintruzívne obalenie C++ rozhraní, takže by nemala byť potreba zmena c++ kód pri jeho obaľovaní. [34]

Cython Cython je optimalizujúci statický kompilátor pre programovací jazyk Python a rozšírený programovací jazyk Cython (založený na Pyrexe). Jazyk Cython je nadmnožinou jazyka Python a dodatočne podporuje volanie C funkcií a deklarovanie C typov pre premenné a triedne atribúty. To umožňuje kompilátoru generovať veľmi efektívny C kód z Cython kódu. C kód sa generuje raz a potom sa kompiluje s všetkými hlavnými C/C++ kompilátormi. [35]

Pre riešenie problematiky tejto práce by vyhovoval hociktorý spôsob, nakoniec sme však rozhodli pre Cython, kvôli jeho ľahkosti použitia.

Z nášho C++ kódu budú odhalené pre Python nasledujúce časti:

- Trieda `PlayerRepository` a jej metóda `std::vector<std::string> &getPlayerCardKeys()`.
- Trieda `MinionRepository` a jej metóda `std::vector<std::string> &getTierCardKeys(int tier)`.
- Trieda `GameManager` a jej metódy:
  - `GameManager()`,
  - `std::string addPlayer(const std::string &playerKey)`,
  - `void movePlayerToDead(const std::string &playerId)`,
  - `void startCombatPhase(const std::string &playerId1, const std::string &playerId2))`,
  - `void startRecruitPhase(const std::string &playerId)` a
  - `void executeSingleCombat(const std::string &playerId1, const std::vector<std::string> &minionIds1, const std::string &playerId2, const std::vector<std::string> &minionIds2, bool isPrint = true)`.

3.12.1 Cython

Na kompiláciu a následné spustenie Cython kódu potrebujeme 2 veci:

- .pyx súbor, obsahujúci kód nášho Python rozšírenia.
- *Setuptools* [36] súbor `setup.py`.

Teraz si ukážeme `game_manager_wrapper.pyx` súbor, ktorý odkrýva triedu `GameManager`.

Najprv si definujeme triedu `GameManager`, ktorá priamo reprezentuje našu C++ triedu.

```
cdef extern from "../../diploma-thesis-src/src/headers/
GameManager.h":
cdef cppclass GameManager:
    GameManager() except +
```

```
        string addPlayer(const string cardKey)

        void movePlayerToDead(const string playerId)

        void startRecruitPhase(const string cardKey)

        void startGame()

        void executeSingleCombat(const string playerId1,
        const vector[string] minionIds1,
        const string playerId2,
        const vector[string] minionIds2,
        bool isPrint);
    }
```

Následne si vytvoríme triedu `PyGameManager`, ktorá obsahuje ukazateľ na `GameManager`. S touto triedou budeme pracovať v Pythone.

```
cdef class PyGameManager:
cdef GameManager * thisptr  # Pointer to the C++ instance

def __cinit__(self):
    self.thisptr = new GameManager()

def __dealloc__(self):
    del self.thisptr

def addPlayer(self, str card_key):
    cdef bytes card_key_bytes = card_key.encode('utf-8')
    return self.thisptr.addPlayer(card_key_bytes)

def startRecruitPhase(self, card_key):
    if not isinstance(card_key, bytes):
        card_key = card_key.encode('utf-8')
    self.thisptr.startRecruitPhase(card_key)

def startGame(self):
    self.thisptr.startGame()

def movePlayerToDead(self, playerId):
    self.thisptr.movePlayerToDead(playerId.encode('utf-8'))

def executeSingleCombat(self, playerId1, list minionIds1,
    playerId2, list minionIds2, bool isPrint=True):
    cdef string playerId1_bytes = playerId1.encode('utf-8')
    cdef string playerId2_bytes = playerId2.encode('utf-8')

    cdef vector[string] minionIds1_vector
```

```
    for minionId in minionIds1:
        minionIds1_vector.push_back(minionId.encode('utf-8')
    )

    cdef vector[string] minionIds2_vector
    for minionId in minionIds2:
        minionIds2_vector.push_back(minionId.encode('utf-8')
    )

    self.thisptr.executeSingleCombat(playerId1_bytes,
    minionIds1_vector,
                                     playerId2_bytes,
    minionIds2_vector, isPrint)
```

V `setup.py` následne skompilujeme náš C++ kód vybraným kompilátorom a vytvoríme `Cython Extension module`.

```
os.environ["CC"] = "cl"
os.environ["CXX"] = "cl"

subprocess.run(["cmake", "-G", "Visual␣Studio␣17␣2022", "-B", "
    build", "diploma-thesis-src"])
subprocess.run(["cmake", "--build", "build", "--config", "
    Release"])

extension_modules = [
    Extension(
        "game_manager_wrapper",
        sources=["wrappers/game_manager_wrapper/
    game_manager_wrapper.pyx"],
        libraries=["battlegrounds_simulator"],
        library_dirs=["build/Release"],
        language="c++",
        extra_compile_args=["/std:c++17"]
    )]

setup(
ext_modules=cythonize(extension_modules)
)
```

Následne po vykonaní príkazu `python setup.py build_ext --inplace` môžeme v Pythone z modulu `game_manager_wrapper` importovať `PyGameManager` a používať metódy, ktoré sme odkryli.

Veľmi podobne sme postupovali pri odkrytí tried `PlayerRepository` a `MinionRepository`.

Kapitola 4

Testovanie a budúcnosť

V tejto sekci bude opísané testovanie implementovaného riešenia.

4.1 Unit testing

Existuje viecero C++ testovacích *frameworkov* na *unit testing*. Medzí príklady patrí napríklad:

Catch2 Výhoda *frameworku* Catch2 je jeho jednoduchosť a ľahkosť používania. Takisto neobsahuje žiadné vonkajšie závislosti, názvy testov nemusia byť validné identifikátory, *assertions* ako normálne C++ boolovské výrazy a sekcie poskytujú ľahké zdieľanie nastavení naprieč viacerými testami. [37]

Boost.Test Boost.Test je C++11/14/17 *unit testing* knižnica, dostupná na viacerých platformách. Táto knižnica je súčasť Boost knižnic. Poskytuje flexibilné a jednoduché rozhrania na písanie testov, organizovanie testov a kontrolu ich behu. [38]

Google Test GoogleTest je testovací *framework* od spoločnosti Google. Je založený na *frameworku* xUnit. Obsahuje *test discovery* – GoogleTest má schopnosť automaticky objaviť a vykonať testy, takže odpadá potreba manuálne vytvárať ďalšie. Obsahuje veľké množstvo asercií(spolu s možnosťou vytvárať ďalšie), má možnosť pokračovať v testoch aj keď nastane chyba a mnoho iného. [39]

Rozhodli sme sa pre *framework* Catch2, na základe jeho jednoduchosti. Testovanie malo prebehnúť vo viacerých fázach:

1. Na začiatku by boli vytvorené *unit* testy pre základné komponenty,
2. Ďalej by boli otestované entity *minion* a hrdina.
3. Následne by boli otestované základné *tasky* a efekty.
4. Pokračovalo by sa testovaním systému posielania upozornení a systému akcií.
5. Na koniec by boli otestované herné fázy.

Testovanie však nakoniec neprebehlo. Hneď na začiatku bol zistený problém so systémom vykonávania efektov. Popis problému je nasledovný:

- Problém súvisí so znehodnotením ukazateľov na *minionov* v mape `actionsMap` v triede *ActionsCentre*.
- Spustenie bojovej fázy prebehne v poriadku, takisto prebehne v poriadku registrácia *minionov* do *ActionsCentre* cez metódu `subscribe()`.
- Počas behu metódy `executeCombat()` v triede `CombatPhaseController` však niekedy nastane v členskej premennej `std::map<TriggerType, std::vector<Minion *>> actionsMap` znehodnotenie ukazateľov vo vektore `std::vector<Minion *>`.
- Následne keď nastane *trigger* asociovaný s daným vektorom a pokúsim sa pristúpiť k efektu daného *miniona*(čo je v tej dobe znehodnotený ukazateľ) tak nastane *segmentation fault*.
- Na vektor *board* hráča je na začiatku použitá metóda *std::reserve*, ktorá naalokuje potrebnú pamäť. Táto metóda by mala zaručiť, že ak mám niekde ukazateľ na nejaký prvok vo vektore, a vektor treba znovu realokovať(napríklad ak pridám do vektoru nový prvok), tak dané ukazateľe nebudú znehodnotené. Preto by dôvod tohto problému pravdepodobne nemal byť tu.
- Ďalšia možná príčina problému môže byť v zrušení registrácie *miniona* v *ActionsCentre* – metóda `unsubscribe`.

4.2 Rýchlosť

Implementácia bola spustená na laptope Lenovo X1 Carbon. Jeho parametre sú zobrazené v tabuľke 4.1.

Procesor	Intel(R) Core(TM) i7-8650U
Operačná pamäť	16.0 GB
Grafická karta	Intel UHD Graphics 620

Tabuľka 4.1: Tabuľka parametrov laptopu použitému pri testovaní

Rýchlosť implementácie bola testovaná nasledujúcim spôsobom:

1. Na začiatku boli vytvorené instancie tried `PyGameManager()`, `PyMinionRepository()` a `PyPlayerRepository()`.
2. Potom boli vygenerované 2 náhodné `player_card_keys` – `playerId1` a `playerId2` – hráčov pomocou metódy `getPlayerCardKeys()`.
3. Následne 10-krát pre každý *tavern tier* od 1 až po 6:
   a) Boli vygenerované možné *minion* `minion_card_keys` pomocou metódy `getTierCardKeys(int tier)`.

b) Z `minion_card_keys` bolo náhodne vybraných 7 hodnôt, boli uložené do premennej `minionIds1`. Podobne bolo vybraných ďalších 7 hodnôt, ktoré boli uložené do premennej `minionIds2`.

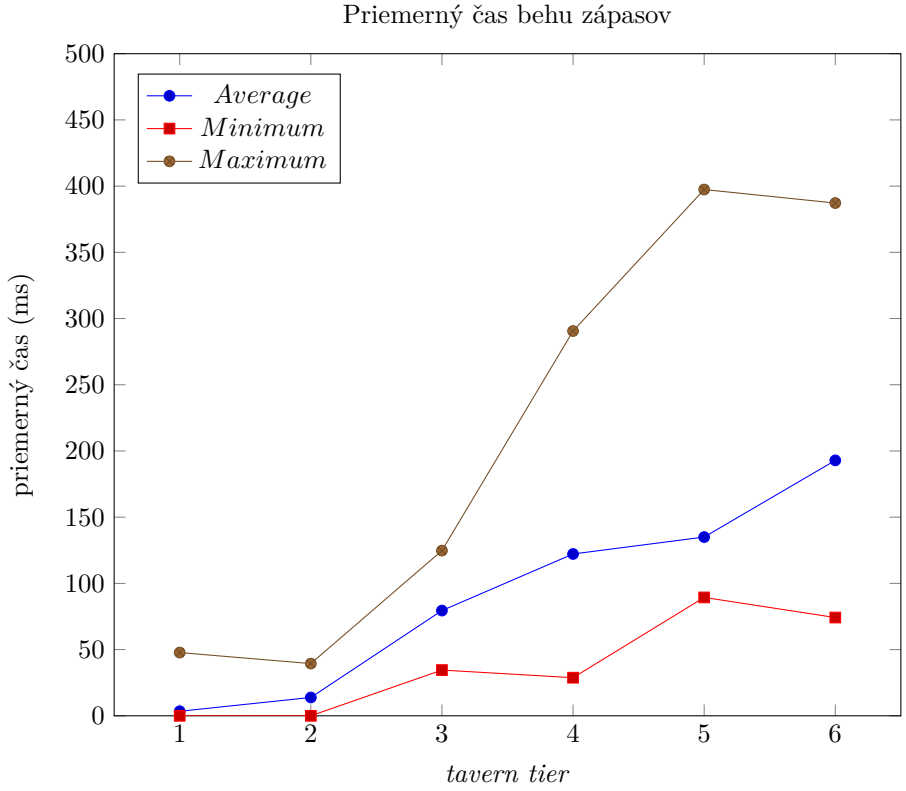
c) Bol zaznamenaný čas trvania 100 behov metódy `executeSingleCombat( playerId1, list minionIds1, playerId2, list minionIds2, bool isPrint=True)`.

4. Z nameraných časov bol zistený priemer, priemer minima a priemer maxima.

Namerané hodnoty sú uvedené v tabuľke 4.2 a graf je zobrazený na obrázku obrázku 4.1.

tavern tier	**Priemer**	**Minimum**	**Maximum**
1	3.43	0.00	47.75
2	13.85	0.00	39.44
3	79.47	34.54	124.74
4	122.21	28.78	290.57
5	214.98	89.4	397.41
6	210.88	74.21	387.21

Tabuľka 4.2: Namerané hodnoty času behov zápasov



Obr. 4.1: Namerané hodnoty času behu zápasov

4.2.1 Zlepšenia rýchlosti

Medzi spôsoby zlepšenia výkonnosti nášho riešenia v budúcnosti patria napríklad:

Použitie obyčajných ukazateľov namiesto `std::unique_ptr` V súčasnosti sú efekty, *battlecries* a *enchantments* reprezentované nie obyčajnými ukazateľmi ale iba `std::unique_ptr`. `std::unique_ptr` bol využitý kvôli eliminácii potreby manuálnej správy pamäte, a tým pádom zjednodušenia vývoja riešenia.

Je ťažké presne určiť, aké zlepšenie výkonnosti by sme týmto dosiahli. V CPP docs je uvedné, že `std::unique_ptr` má malý *overhead*: *„Manages the storage of a pointer, providing a limited garbage-collection facility, with little to no overhead over built-in pointers (depending on the deleter used).“* Každopádne, môže nastať len zlepšenie.

Zmena implementácie DFS algoritmu V súčasnosti je DFS algoritmus na prechádzanie stromu akcií implementovaný pomocou rekurzie.

V budúcnosti sa môže implementovať iteratívne riešenie tohto algoritmu. Znova ale nie je isté, či to bude mať reálny vplyv na výkonnosť nášho riešenia.

Využitie iných dátových štruktúr V súčasnosti je takmer všade v kóde, kde potrebujeme nejaký zoznam použitý `std::vector`. Používanie `std::reserve`, kde je to možné, prípadne náhrada za inú, vhodnejšiu dátovu štruktúru namiesto `std::vector`, ak je to vhodné, by malo mať pozitívny dopad na výkonnosť nášho riešenia.

4.3 Zmeny do budúcnosti

V tejto sekcii si povieme o možných zmenách do budúcnosti.

4.3.1 Testovanie

Po nájdení riešenia problému by sa malo pokračovať v *unit* testoch. Hlavné oblasti na ktoré sa treba sústrediť je

Správne fungovanie triedy `Minion` Metódy na ktoré sa treba zamerať sú:

- metódy na príjmanie a liečenie poškodenia – `takeDamage(...)` a `healDamage(...)`
- konštruktory.

Správne fungovanie triedy `Player` Metódy na ktoré sa treba zamerať sú:

- metódy interagujúce s *minionmi*, napríklad `addMinionToHand(...)` alebo `removeMinionFromBoardByIndex(...)`,
- metódy interagujúce s hráčovým `gold` a
- interakcia hráčovho *board* a aury – metódy `auraUpdate` a časť metódy `addMinionToBoard`.

Správne fungovanie základných herných mechaník Dôležité oblasti sú

- základné herné *tasky* – triedy, ktoré implementujú triedu `ITask` – a ich metódy `execute(...)` a
- efekty a *battlecries* – triedy, ktoré implementujú triedu `IEffect`, respektíve `IBattlecry` – a ich metódy `execute(...)`.

Správne fungovanie triedy `ActionsCentre` Teda to či metódy:

- na registráciu *minionov* – `subscribe(...)` a `subscribeEffect(..)` – a
- metódy na zrušenie registrácie *minionov* – `unsubscribe(...)` a `unsubscribeEffect(...)` fungujú správne.

Správne fungovanie metód `update()` v triedach `RecruitPhaseController` a `CombatPhaseController`

Teda hlavne to či:

- metóda správne hľadá registrovaných *minionov* v `ActionsCentre` a správne získava a radí ich efekty a
- to či správne napájam nové uzly(ktoré obsahujú efekty na spustenie) ako deti posledného spusteného uzlu.

Správne fungovanie bojovej fázy Ide hlavne o metódy:

- na hľadanie útočníka – `findAttacker(...)`,
- na hľadanie obrancu –`findDefender(...)` a
- hlavnú metódu *executeCombat(...)*, kde treba otestovať či
  - sa správne strieda útočiaci a obraňujúci hráč a
  - či správne funguje útočiaca sekvencia, ak má útočiaci *minion* napríklad *windfury*.

4.3.2 Herné mechaniky

Keďže cieľom tejto práce nebola presná kópia Hearthstone Battlegrounds, stále zostáva implementovať nejaké funkcionality, medzi ktoré patria hlavne *spells* a *hero power*.

4.3.2.1 Spell

Spells sú prakticky totožné s *battlecry* z pohľadu implementácie:

- *Spell* aj *battlecry* musia mať pred svojim aktivovaním nastavené svoj ciel – ten môže byť buď nastavený automaticky, napríklad vtedy ak sú cieľ všetci priateľskí *minioni*, alebo môže byť vybraný manuálne, ak cieľom je napríklad jeden priateľský *minion*.
- Konkrétne vykonanie *spellu* aj *battlecry* je veľmi podobné, jedine zdroj samotnej akcie je *spell*, nie daný *minion* s *battlecry*.

V *patchi* 28.2 pre Hearthstone Battlegrounds bola vydaná nová aktualizácia, a to to, že je možné nakupovať *spelly* podobne ako *minionov* počas *recruit fázy*. [40]

V hre by sa teda po implementácii *spellov* museli zmeniť tieto mechaniky:

- Premenné *board* a *graveyard* v triede `Player` by mali byť schopné obsahovať aj *minionov* aj *spelly*.

  Podobne by mala byť vytvorená nová premenná *secrets*, ktorá by obsahovala *secrets*, viď sekciu 2.2.3.1.

- Počas *recruit* fázy by som mal byť schopný ponúkať na predaj aj *spelly* nie len *minionov*. Premenná `minionsToPurchase` by mala byť teda schopná obsahovať aj *minionov* aj *spelly*.

Tieto problémy by sa dali vyriešiť napríklad pomocou:

- polymorfizmu – vytvorím napríklad nové rozhranie `IPlayable`, ktoré bude mať metódy na zahranie danej karty a ktoré budú implementovať triedy `Minion` aj `Spell`.

- použitím `std::variant`, podobne ako u triedy `TreeNode`, viď sekciu 3.27.

4.3.2.2 Hero power

Z pohľadu implementácie je rozdiel medzi aktívnou a pasívnou *hero power*:

- Aktívna *hero power* je prakticky *spell*(respektíve *battlecry*), ktorź prislúcha danému hráčovi a ktorý hráč môže použiť vždy(respektíve raz za kolo). Ak sa teda implementuje *spell*, implementácia aktívnej *hero power* by mala byť veľmi podobná.

- Pasívna *hero power* je zase podobná efektu(viď sekciu 3.5), má svoj:

  – *trigger*,

  – akcia, ktorú má *hero power* vykonať a

  – ciele *hero power*.

  Ak chceme implementovať aktívnu *hero power*, tak musíme zmeniť:

  – triedu `ActionsCentre` , aby si pamätala, na aký *trigger* daný hráč reaguje,

  – triedu `TreeNode` , aby bola schopná udržiavať v uzli danú pasívnu *hero power*,

  – metódu `update()` v ovládačoch fáz, aby bola schopná pridať do stromu akcií danú *hero power* a

  – metódu `traverse()` v ovládačoch fáz, aby bola schopná spúštať danú *hero power*.

4.3.2.3 Iné

Ďalšie funkcionality na implementáciu sú napríklad:

Zlaté kópie Ak chceme implementovať zlatú kópiu *miniona*, je potreba:

- vytvoriť vylepšenú verziu daného *miniona* a následne
- v `RecruitPhaseController` pridať kontrolu toho, či hráč po pridaní *miniona* do ruky nemá tri kópie daného *miniona*, a ak áno, tak tri kópie odstrániť a pridať zlatú verziu do ruky hráča.

Ďalšie *trigger types* a efekty Ak chceme pridať nový *trigger*, tak treba pridať novú hodnotu do enumu `TriggerType` a následne v odpovedajúcom mieste v kóde zavolať metódu `update(TriggerType triggerType)`.

Ak chceme pridať nový efekt, tak jediná vec, čo musí spĺňať je, aby implementoval triedu `IEffect`, inak tam je veľmi veľa možností toho, čo by mal daný efekt robiť.

Záver

Cieľom tejto práce bola implementácia simulátoru Hearthstone Battlegrounds.

Na začiatku som uviedol stručnú históriu kartových hier. Väčšia pozornosť bola venovaná zberateľským kartovým hrám – konkrétne Hearthstone. Takisto som opísal nový herný žáner *autobattler*, kde som sa zameral na Hearthstone Battlegrounds.

Práca pokračovala analýzou herných mechaník a pravidiel. Postupne som vysvetlil základné vlastnosti kariet. Dôkladne som opísal hlavnú hernú jednotku *miniona*, kde som podrobnejšie vysvetlil jeho vlastnosti a efekty a taktiež som uviedol rôzne konkrétne príklady. Ďalej som opísal hrdinu a jeho možné aktívne aj pasívne schopnosti.

V analýze boli ďalej bližšie popísané fázy hry, ekonomika hry a spôsob vykonávania akcií.

Následne prebehol návrh a implementácia riešenia v jazyku C++. Postupne som navrhol a implementoval základné herné komponenty, pokračoval som implementáciou *miniona*, kde som som sa zameral hlavne na flexibilitu a modularitu efektov. Ďalej som venoval pozornosť systému vykonávania akcií a ovládaní herných fáz. Ďalej som pomocou Cythonu odkryl časti C++ kódu tak, aby som základné metódy ovládania hry mohol volať v jazyku Python. Na konci realizačnej časti práce som popísal, ako v budúcnosti postupovať pri implementácii zatiaľ doposiaľ nerealizovaných herných mechaník.

Na záver prebehlo testovanie rýchlosti behu knižnice. *Unit testing* bohužiaľ z časových dôvod nebol vykonaný, bola však nájdená chyba v systéme akcií, ktorú bola následne popísaná a boli uvedené kroky smerujúce k riešeniu tohto problému. Po odovzdaní tejto práce sa pokúsim tieto nedostatky opraviť. Ako posledná vec v práci bolo uvedené ako postupovať v budúcnosti pri implementovaní herných funkcionalít, ktoré zatiaľ neboli realizované.

V práci boli splnené všetky ciele – od analýzy problémovej domény až návrh a implementáciu knižnice – okrem dôkladnejšieho otestovanie knižnice. Testovanie v budúcnosti by malo byť vykonané, s cieľom nájsť chyby knižnice.

Literatúra

[1] Dummett, M.: The history of cardgames. ročník 1, č. 02, 1993: str. 125–135, doi:10.1017/S1062798700000478. Dostupné z: https://www.cambridge.org/core/journals/european-review/article/abs/history-of-card-games/E9386F7675C8D9CA283502A888D46BF8

[2] Homo ludens Pragensis: příspěvek k dějinám karetní hry v pozdně středověkých a raně novověkých Čechách. *Archeologické rozhledy*, január 1997, [cit. 2024-1-1]. Dostupné z: https://kramerius.lib.cas.cz/view/uuid:e8de5143-3cfa-11e1-1872-001143e3f55c?page=uuid:e8de51af-3cfa-11e1-1872-001143e3f55c

[3] Bycer, J.: *Game Design Deep Diver: Trading and Collectible Card*. Boca Raton: CRC Press, 2023, ISBN 9781032370705.

[4] Jarvis, M.: Magic: The gathering black Lotus sells for $540,000, setting yet another record for the Holy Grail Card. [online]. Dicebreaker, Mar 2023, [vid. 18. 11. 2023]. Dostupné z: https://www.dicebreaker.com/games/magic-the-gathering-game/news/mtg-alpha-black-lotus-auction-sale-record

[5] Dahl, E.: Magic: The Gathering—a game's origins and influence at Whitman College. [online]. Whitman College, 2012, [vid. 10. 10. 2023]. Dostupné z: https://whitmanwire.com/arts/2012/11/26/magic-the-gathering-a-games-origins-and-influence-at-whitman-college/

[6] Chris Cocks Is Hasbro's Gamer in Chief. [online]. The Wall Street Journal, Jan 2022, [vid. 12. 10. 2023]. Dostupné z: https://www.wsj.com/articles/chris-cocks-is-hasbros-gamer-in-chief-11646389842

[7] Hearthstone: how a game developer turned 30m people into card geeks. [online]. The Guardian, Jan 2015, [vid. 13. 11. 2023]. Dostupné z: https://www.theguardian.com/technology/2015/jun/25/hearthstone-blizzard-strategy-trading-cards-greg-austin

[8] Game of Champions - Team 5's Ben Thompson Talks Evolving Hearthstone from Vanilla to The Grand Tournament. [online]. AusGamers News, 2024. Dostupné z: http://www.ausgamers.com/features/read/3528779

[9] World of Warcraft: 'Hearthstone began as the best card game we could make'. [online]. The Guardian, Jan 2013, [vid. 22. 12. 2023]. Dostupné z: https://www.theguardian.com/technology/2013/oct/17/world-of-warcraft-hearthstone-blizzard-ccg

[10] Hearthstone Official Game Site. [online]. Blizzard Entertainment, Jan 2024, [vid. 21. 12. 2023]. Dostupné z: https://playhearthstone.com/en-gb/expansions-adventures/

[11] Hearthstone at BlizzCon: Fireside Chat Panel Highlights. [online]. Blizzard Entertainment, Nov 2013, [vid. 21. 12. 2023]. Dostupné z: https://hearthstone.blizzard.com/en-gb/news/11524460/hearthstone-at-blizzcon-fireside-chat-panel-highlights-08-11-2013

[12] Hearthstone at BlizzCon: Fireside Chat Panel Highlights. [online]. Blizzard Entertainment, Nov 2013, [vid. 14. 10. 2023]. Dostupné z: https://hearthstone.blizzard.com/en-gb/news/11524460/hearthstone-at-blizzcon-fireside-chat-panel-highlights-08-11-2013

[13] Hearthstone Classic mode takes you back to 2014. [online]. CNET, Jan 2014, [vid. 14. 10. 2023]. Dostupné z: https://www.cnet.com/tech/gaming/hearthstone-classic-mode-takes-you-back-to-2014/

[14] Vincent, B.: Understanding what auto battler games are. [online]. Gaming Street, Mar 2020, [vid. 18. 11. 2023]. Gaming Street. Dostupné z: https://gamingstreet.com/auto-battler-games-explained/

[15] The making of Hearthstone Battlegrounds: 'We wanted a mode that didn't feel as polarizing'. [online]. Future Publishing, Jan 2024, [vid. 21. 12. 2023]. Dostupné z: https://www.pcgamer.com/the-making-of-hearthstone-battlegrounds-we-wanted-a-mode-that-didnt-feel-as-pulverising/

[16] Introducing Hearthstone Battlegrounds. [online]. Blizzard Entertainment, Jan 2024, [vid. 21. 12. 2023]. Dostupné z: https://playhearthstone.com/en-gb/blog/23156373/

[17] Hearthstone Open Beta is Here! [online]. Blizzard Entertainment, Jan 2024, [vid. 14. 10. 2023]. Dostupné z: https://playhearthstone.com/en-gb/blog/12440010/

[18] Hearthstone Wiki. [online]. Fandom, [vid. 21. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Hearthstone_Wiki

[19] Battlegrounds. [online]. Blizzard Entertainment, Jan 2024, [vid. 21. 12. 2023]. Dostupné z: https://playhearthstone.com/battlegrounds

[20] Card - Hearthstone Wiki. [online]. Fandom, Jan 2024, [vid. 21. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Card

[21] Minion - Hearthstone Wiki. [online]. Fandom, [vid. 21. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Minion

[22] Enchantment list. [online], Jan 2024, [vid. 21. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Enchantment_list

[23] Battlegrounds - Hearthstone Wiki. [online]. Fandom, Jan 2024, [vid. 29. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Battlegrounds

[24] Imprisoner - Hearthstone Card Library - Hearthstone. [online]. Blizzard Entertainment, Jan 2024, [vid. 22. 12. 2023]. Dostupné z: https://hearthstone.blizzard.com/en-us/battlegrounds/59937-imprisoner

[25] Banana Slamma - Hearthstone Card Library - Hearthstone. [online]. Blizzard Entertainment, Jan 2024, [vid. 22. 12. 2023]. Dostupné z: https://hearthstone.blizzard.com/en-us/battlegrounds/98824-banana-slamma

[26] Spell - Hearthstone Wiki. [online]. Fandom, Jan 2024, [vid. 22. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Spell

[27] Secret - Hearthstone Wiki. [online]. Fandom, [vid. 22. 12. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Secret

[28] Battlegrounds. [online]. Fandom, Jan 2024, [vid. 29. 12. 2023]. Dostupné z: https://playhearthstone.com/battlegrounds

[29] Battlegrounds. [online]. Blizzard Entertainment, Jan 2024, [vid. 29. 12. 2023]. Dostupné z: https://playhearthstone.com/battlegrounds

[30] An introduction to the Lich King in Battlegrounds. [online]. AceGameGuides, Jan 2024, [vid. 29. 12. 2023]. Dostupné z: https://acegameguides.com/an-introduction-to-the-lich-king-in-battlegrounds/

[31] Advanced rulebook - Hearthstone Wiki. [online]. Fandom, Jan 2024, [vid. 13. 10. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Advanced_rulebook

[32] Advanced rulebook. [online]. Fandom, Jan 2024, [vid. 1. 1. 2023]. Dostupné z: https://hearthstone.fandom.com/wiki/Advanced_rulebook

[33] Simplified Wrapper and Interface Generator. [online], [vid. 10. 1. 2024]. Dostupné z: https://www.swig.org/

[34] Boost-Python Tutorial. [online]. GitHub, Inc, Nov 2022, [vid. 1. 1. 2023]. Dostupné z: http://boostorg.github.io/python/doc/html/tutorial/index.html#tutorial

[35] Cython: C-Extensions for Python. [online]. GitHub, Inc, Nov 2022, [vid. 1. 1. 2023]. Dostupné z: https://cython.org/

[36] Setuptools. [online]. GitHub, Inc, Nov 2022, [vid. 1. 1. 2023]. The Python Software Foundation. Dostupné z: https://setuptools.pypa.io/en/latest/

[37] Why do we need yet another C++ test framework? [online]. GitHub, Inc, Dec 2023, [vid. 4. 1. 2023]. Dostupné z: https://github.com/catchorg/Catch2/blob/devel/docs/why-catch.md

[38] Introduction - 1.84.0. [online]. Boost.org, [vid. 4. 1. 2023]. Dostupné z: https://www.boost.org/doc/libs/1_84_0/libs/test/doc/html/boost_test/intro.html

[39] GitHub - google/googletest: GoogleTest - Google Testing and Mocking Framework. [online]. GitHub, Inc, Aug 2023, [vid. 4. 1. 2023]. Dostupné z: https://github.com/google/googletest

[40] 28.2 Patch Notes - Hearthstone. [online]. GitHub, Inc, Dec 2023, [vid. 1. 1. 2023]. Blizzard Entertainment. Dostupné z: https://hearthstone.blizzard.com/en-us/news/24008697

Kapitola 5

Seznam použitých zkratek

CCG Collectible card game

TCG Trading card game

UI User interface

RPG Role-playing game

MMORPG Massively multiplayer online role-playing game

UUID Universal Unique Identifie

DFS Depth-first search

Kapitola 6

Obsah priloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `src`
  - `impl` ..... zdrojové kódy implementácie
  - `thesis` ..... zdrojová forma práce vo formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce vo formátu PDF