FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Platforma pro soutěže AI agentů
Student:	Ondřej Vaniš
Vedoucí:	Ing. David Bernhauer
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Strategické hry se stávají populární pro programátory z hlediska tvorby umělé inteligence. Pro zachování férovosti zakazují existující hry použití externích nástrojů a programátoři tak nemůžou legálně využít svůj potenciál. Vytvořte webovou platformu pro pořádání strategických tahových her mezi AI agenty. Analyzujte možnosti komunikace AI agentů s herním serverem (minimálně REST a WebSocket). Na základě analýzy navrhněte vhodnou strukturu webové platformy a implementujte její prototyp. Zdokumentujte tvorbu modulů (her) a demonstrujte vytvořením alespoň jednoho testovacího modulu (hry, např. piškvorky, Othello). Implementujte dva jednoduché mock agenty (hráče) pro otestování funkčnosti.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 10. ledna 2019

FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

Bakalářská práce

Platforma pro soutěže AI agentů

Ondřej Vaniš

Katedra softwarového inženýrství
Vedoucí práce: Ing. David Bernhauer

16. května 2019

Poděkování

Chtěl bych poděkovat svému vedoucímu, Ing. Davidu Bernhauerovi, za podporu a vedení mé bakalářské práce. Poděkování dále patří mé rodině a nejbližším přátelům za jejich podporu v průběhu celého studia.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2019 ……

Odkaz na tuto práci

Vaniš, Ondřej. *Platforma pro soutěže AI agentů*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Tato práce se zabývá návrhem a vytvořením platformy pro soutěžení AI agentů v tahových strategických hrách. Pro řešení byla zvolena klient–server architektura. Kód AI agenta běží na straně klienta a připojuje se k hernímu serveru přes síť. Průběh hry je řízen skrze události posílané prostřednictvím protokolu WebSocket. Herní server běží v runtime prostředí Node.js. Na hře Othello byla otestována funkčnost platformy a bylo ukázáno, že Node.js je dostačující řešení pro obsluhu až 180 současně připojených dvojic agentů. Byla demonstrována lineární závislost celkové doby běhu v závislosti na počtu souběžných spojení pro hry s konstantní časovou složitostí na zpracování jednoho tahu. Bylo předvedeno, že implementované řešení je postačující pro pořádání jednoduchých tahových her.

Klíčová slova webová platforma, tahové strategie, AI agenti, klient–server, Node.js, REST, WebSocket

Abstract

This thesis focuses on the design and creation of a platform for competing AI agents in turn-based strategy games. The solution is based on the client–server architecture. The code of the AI agent runs client-side and connects to the game server via a network. The course of a game is managed by events sent via the WebSocket protocol. The game server runs in Node.js runtime environment. The functionality of the platform was tested on game Othello. It was shown that Node.js is sufficient solution for managing up to 180 simultaneously connected pairs of agents. A linear dependence of total match time was demonstrated, depending on the number of simultaneous connections for games with constant time complexity for the processing of a single turn. It was also shown that the implemented solution is satisfactory for simple turn-based games.

Keywords web platform, turn-based strategy games, AI agents, umělá inteligence, client–server, Node.js, REST, WebSocket

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Úvod

Herní průmysl zažívá v posledních letech velký rozmach. S tím je spojená rozmanitá orientace na různé cílové skupiny. Programátoři tvoří jednu z nich. V důsledku toho lze pozorovat vzrůstající popularitu her, které jako primární herní mechanismus vyžadují psaní vlastního kódu. Jeden z nejstarších podžánrů tvoří tahové strategie, ve kterých se jednotliví hráči střídají po tazích. Existující implementace těchto her jsou však většinou uzavřené systémy, kde k řešení optimální strategie nemohou programátoři plně využít svůj potenciál.

Motivací pro tuto práci je poskytnutí otevřené možnosti programátorům plně zužitkovat své know-how. Hráčům umožní naprogramovat si vlastní umělou inteligenci, která za ně bude řešit náročná herní rozhodnutí. Výsledkem tedy bude platforma, ve které proti sobě budou moct soupeřit AI agenti[1].

Cíl práce

Hlavním cílem práce je vytvoření platformy pro pořádání strategických tahových her. Součástí je analýza možností komunikace AI agentů s herním serverem. Na základě analýzy bude navržena vhodná struktura platformy. Dílčím cílem je umožnění tvorby her i ostatním programátorům. Zdokumentujeme popis tvorby herních modulů a demonstrujeme vytvořením jedné ukázkové hry. Pro otestování funkčnosti dále vytvoříme dva mock agenty, kteří budou v dané hře soupeřit proti sobě.

[1] V práci budeme dále využívat termín *AI agent* ve významu počítačového programu, který v průběhu hry zastupuje hráče v rozhodování.

Struktura práce

V kapitole 1 nejprve provedeme analýzu současných řešení. Podíváme se na možnosti, jak lze spouštět kód umělé inteligence. Zaměříme se na způsoby komunikace mezi klientem a serverem. Na závěr provedeme analýzu procesů herní platformy založené na klient–server architektuře.

V následující kapitole 2 vytvoříme návrh platformy. Rozdělíme uživatele do dvou kategorií a podíváme se na dílčí komponenty z pohledu správce. Popíšeme, jaké technologie jsme pro práci využili.

Implementací prototypu se budeme zabývat v kapitole 3. Zdokumentujeme tvorbu herního modulu. Vytvoříme REST a WebSocket rozhraní pro komunikaci a popíšeme strukturu vyměňovaných zpráv.

V závěru práce, v kapitole 4, platformu otestujeme. Ověříme její funkčnost jak na localhost, tak v rámci internetu. Vystavíme systém problematickým situacím a budeme sledovat reakci serveru a klienta. Nakonec budeme hledat limity platformy v podrobeném zátěžovém testu.

Kapitola 1

Analýza soutěžení AI

Herní umělé inteligence lze rozdělit na dva typy dle [1]. První se věnuje konkrétní hře, u které dopředu zná všechna pravidla, a zabývá se tak především tvorbě optimální výherní strategie. Druhý typ se zabývá obecným způsobem řešení tak, že lze stejného AI agenta aplikovat na hraní více her. Jedná se tedy o dynamický typ, který je schopný se učit a přizpůsobovat novým podmínkám. Pro účely této práce se budeme dále věnovat pouze prvnímu typu umělé inteligence.

V této kapitole se zaměříme na existující platformy pro soutěžení agentů. Dále se budeme zabývat možnými způsoby spouštění kódu umělé inteligence. Následně charakterizujeme tahové strategie. Poté se zaměříme na samotnou komunikaci mezi klientem a serverem. Dále probereme chybové stavy, které mohou nastat při běhu AI. Na závěr specifikujeme procesy herní platformy.

1.1 Existující soutěžní platformy

V této části provedeme analýzu existujících soutěžních platforem, ve kterých hraje hlavní roli uživatelem napsaný kód. Zaměříme se především na následující aspekty:

- způsob komunikace mezi AI agentem a hrou,
- způsob spouštění kódu,
- jazyky, které lze pro psaní AI použít.

1.1.1 Vybrané platformy a jejich aspekty

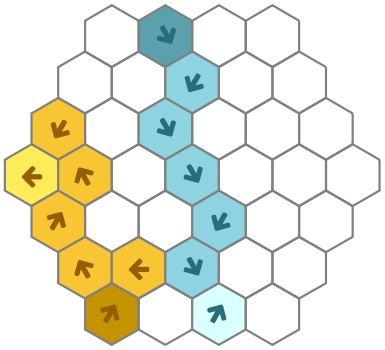
Existující platformy je možné rozdělit do dvou oblastí. První se věnuje přímo soutěžení AI agentů. Druhá se zabývá, spíše než samotnou inteligencí, výukou programování pomocí prvků gamifikace[2].

[2]Technika využívání herních prvků v neherní oblasti.

Do první oblasti lze zařadit hru Hexatron nebo platformu Riddles.io. Na pomezí první a druhé se nachází hra Screeps. V rámci druhé oblasti se zaměříme na populární CodinGame a CodeCombat.

Hexatron [2] byl vytvořen jako součást kurzu „Informatica“ na univerzitě v Gentu, kterého se dle článku [3] účastnili především studenti prvního ročníku. Jedná se o hru, ve které proti sobě soupeří dvojce AI agentů.

Hráči se pohybují po hexagonálním poli, kde zanechávají neprostupnou stopu na každém políčku, které navštíví (viz obrázek 1.1). Každý tah hry se mohou otočit o 0°, 60° nebo 120° proti nebo ve směru hodinových ručiček před tím, než se posunou na další políčko. K tomuto rozhodování dochází současně a tah se uskuteční až ve chvíli, kdy oba hráči rozhodnou o své rotaci. Cílem hry je přežít co nejdéle, aniž narazí do stěny herního pole nebo do stopy zanechané hráči.



Obrázek 1.1: Ukázka herního pole hry Hexatron [2]

Dle [3] soupeření probíhá následujícím způsobem: z databáze jsou periodicky vybráni dva uživatelé, mezi kterými je hra následně nasimulována. Celkový žebříček se aktualizuje průběžně na základě proběhlých her. Kódy jednotlivých AI agentů se ukládají a spouští na straně serveru. K zajištění bezpečnosti jsou jednotlivé instance agentů spouštěné v izolovaném prostředí.

Dle [4] můžou hráči pro kód AI agenta využít pouze jazyk Python. Pro aplikaci vlastní umělé inteligence je třeba, aby hráči implementovali rozhraní `generate_move` – funkci, která je zavolána za začátku každého tahu a která vrací rotaci hráče před posunem na další políčko.

Riddles.io [5] je platforma pro soutěžení botů. Nabízí velké množství her, ve kterých lze soupeřit proti botům ostatních hráčů. Jednotlivé hry jsou simulovány na straně serveru. Platforma vybírá automaticky vhodného oponenta pro jednotlivá utkání, ačkoliv umožňuje vyzvat i konkrétního hráče. Pro vytvoření umělé inteligence si může hráč vybrat z velkého množství podporovaných jazyků. Na straně klienta lze zpětně přehrávat záznamy odehraných zápasů v grafické podobě.

Komunikace mezi jednotlivými AI agenty a hrou probíhá skrze standardní vstup (informace o stavu hry) a standardní výstup (instrukce provedení tahu). Chybový výstup je hráči k dispozici pro ladící účely.

Screeps [6] je MMO RTS[3] hra, v rámci které hráč ovládá svou kolonii pomocí programování AI svých jednotek. Cílem je udržování a dobývání nového území na herní ploše. Hráč má k dispozici rozhraní pro zadávání kódu, který se s každou změnou ukládá na herním serveru. Jednotlivé herní akce (přesun jednotky, těžba surovin atp.) se provádí periodicky každý tick[4] hry. Všichni hráči soupeří proti sobě současně, spouštění logiky AI dochází najednou pro všechny registrované agenty. Hra nikdy nekončí, po nahrání umělé inteligence dojde k jejímu spuštění, kde běží nepřetržitě až do doby, kdy kolonie není schopná těžit energii. Není tedy nutné, aby byl programátor přítomný po dobu hraní.

Hráč může použit pouze JavaScript pro psaní AI. Vytvořený modul je spouštěný na serveru v runtime prostředí Node.js. Hráči dovoluje rozdělit logiku umělé inteligence na další moduly, které pak může importovat do hlavního modulu. Komunikace mezi agentem a hrou probíhá skrze globální objekt `Game`, který je k dispozici hlavnímu modulu AI. Voláním metod tohoto objektu lze ovládat veškeré jednotky kolonie (viz ukázka kódu 1.1).

```
var target = Game.spawns.Spawn1;
for(var i in Game.creeps) {
    Game.creeps[i].moveTo(target);
}
```

Výpis kódu 1.1: Komunikace s herním objektem ve hře Screeps [7]

CodinGame [8] je platforma zaměřená především na zdokonalování programátorských schopností prostřednictvím výzev. Nabízí možnost řešení samostatných úloh, ale také soupeření proti ostatním uživatelům. Platforma podporuje velké množství jazyků, které lze pro psaní kódu použít. Uživatelský program se vyhodnocuje na straně serveru.

Mechanismus řešení úlohy probíhá následovně: uživatel čte ve smyčce data ze standardního vstupu, vypočítá danou úlohu a na standardní výstup vypíše výsledek dané úlohy pro zadaný vstup. Jednotlivé výsledky se porovnávají na základě předpřipravených testů, kterými musí algoritmus projít.

Platforma zároveň poskytuje možnost programátorům psát vlastní výzvy. Tyto lze posléze odeslat na server a zpřístupnit je tak všem uživatelům. Vlastní moduly musí splňovat předdefinované rozhraní a lze je psát pouze v jazyce Java.

[3] Massively Multiplayer Online Real-Time Strategy Game
[4] časová jednotka, po kterou se odehrávají jednotlivé herní akce

CodeCombat [9] je populární webová role-playingová hra určená především začínajícím programátorům. Dle [10] platformu využívá v rámci vyučování jenom v USA více než 45 tis. studentů napříč 1 600 škol. CodeCombat se zaměřuje na výuku jazyků JavaScript a Python. Cílem hráče ve hře je procházet místnostmi do další úrovně. Toho docílí psaním sekvencí příkazů, které určují chování hlavní postavy v dané místnosti. Spouštění příkazů probíhá opět na straně serveru. Uživateli se zobrazuje postup vykonávání kódu v grafické podobě (hlavní postava se pohybuje po herní ploše, útočí na nepřátelské jednotky atp.).

1.1.2 Shrnutí analýzy existujících platforem

Prvním aspektem, který jsme zkoumali, byl **způsob spouštění kódu** od uživatele. Ve všech výše zmíněných případech dochází ke spouštění na straně serveru. To s sebou přináší určité výhody i nevýhody. Podrobněji se na ně zaměříme v následující sekci 1.2.

Druhým zkoumaným aspektem byla **komunikace mezi AI agentem a hrou**. Zde lze rozčlenit rozdílné přístupy z hlediska odstínění interní logiky. První přístup řeší komunikaci skrze standardní vstup a výstup, kde uživatel čte vstupní data a vrací výsledné řešení. V tomto případě rozumí agent i hra zprávám, které si vyměňují, avšak vzájemně o své vnitřní implementaci neví nic.

Druhým způsobem je implementace funkce s daným rozhraním, která se volá periodicky podle potřeby dané hry (každý tah nebo tick hry). Platforma ví o existenci této funkce a zná její rozhraní. Oproti tomu agent nezná vnitřní implementaci hry. Komunikace probíhá výhradně skrze vstupní a návratové hodnoty funkce.

Třetí možností je zpřístupnění globálních objektů přímo samotnému agentovi. Ten pak komunikuje s hrou pomocí volání metod globálních objektů. Agent zná rozhraní herních objektů, které využívá k ovládání hry.

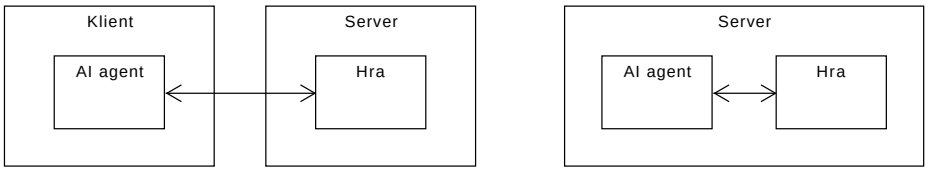
Posledním zkoumaným aspektem jsou **jazyky**, které jsou hráči k dispozici pro tvorbu vlastní umělé inteligence. Zde lze pozorovat rozdílné přístupy – buď je podporovaný pouze jeden jazyk, a platforma se tak zaměřuje spíše na využití specifických vlastností daného jazyka, nebo je podporováno větší množství jazyků, čímž se rozšiřuje dosah programátorů, kteří jsou schopní platformu používat.

1.2 Modely spouštění kódu umělé inteligence

V této sekci probereme dva přístupy ke spouštění kódu napsaného uživatelem. První možností je spouštění kódu přímo na serveru – *server-side*. Druhou možností je spouštění kódu na straně klienta – *client-side*. Komunikace pro oba modely je znázorněná na obrázku 1.2.

Server-side Vyhodnocení kódu AI agenta probíhá na straně serveru. Z bezpečnostních důvodů je nutné spouštět kód v izolovaném prostředí. V případě opačném by bylo možné zneužít systém ve prospěch hráče. Naopak výhodou je rovnocenné prostředí pro všechny AI agenty. Vyrovnané podmínky dovolují srovnání i podle dalších faktorů – např. času, který zabere vyřešení dané úlohy. Další výhodou je rychlost vyhodnocování kódu, jelikož vše běží na stejném serveru a není nutné komunikovat s dalšími zařízeními. Odpadá tak veškerá režie spojená s komunikací po síti. Sdílené prostředí navíc dovoluje použití více způsobů komunikace mezi hrou a agentem. Oproti client-side variantě mohou vzájemně komunikovat přímou cestou – skrze volání jednotlivých metod nebo funkcí.

Client-side Spouštění je prováděno přímo na straně klienta. Komunikace mezi hrou a agentem probíhá po síti. Toto řešení klade nároky na obousměrnou komunikaci mezi klientem a serverem. Výhodou je oproti server-side variantě možnost využít jakýchkoliv prostředků, které má klient k dispozici. Odpadá tak např. závislost na použití konkrétního jazyka, daného podporou platformy. Oproti tomu je nevýhodou prodleva spojená s posíláním zpráv skrze síť. Navíc je nutné předávat herní stav jednotlivým klientům.



Obrázek 1.2: Komunikace client-side vs. server-side

Při spouštění kódu na straně serveru je možné předávat informace efektivněji, jelikož celý stav může být držen v paměti, ke které mají jednotliví AI agenti přístup. Oproti tomu je při client-side variantě nutné posílat větší objem dat, neboť je po každém provedení tahu nutné klientovi poslat aktualizaci herního stavu. Z tohoto důvodu není tato varianta vhodná pro takové hry, které musí udržovat složitý a objemný herní stav. Aby nedošlo k zahlcení komunikačního kanálu, je rozumné omezit množství dat, jak nejvíce je to možné.

1.3 Charakteristika tahových strategií

Během tahových strategických her se hráč rozhoduje na základě *stavu hry* o tom, jaké ve svém tahu provede *herní akce*. Ty pak následně ovlivňují stav

hry. Současně je vždy jasné, který hráč je aktuálně na řadě.

Tahy lze rozčlenit na dva možné typy. Prvním je situace, kdy v jednu chvíli je na řadě pouze jeden hráč. Čeká se, až rozhodne o herních akcích, které provede, a až poté se dostane na řadu další hráč. Druhým typem je případ, kdy v daném tahu rozhoduje o herních akcích více hráčů najednou. Další tah může nastat až ve chvíli, kdy své akce zahrají všichni hráči. Během hry mohou *střídavé tahy* nastávat vedle *souběžných tahů*.

Příkladem může být souběžný tah na začátku hry Lodě, kdy oba hráči najednou rozmísťují své jednotky na bitevním poli. Až ve chvíli, kdy oba dokončí tuto akci, může nastat další tah. Následující tahy jsou už střídavé.

Na tahové strategie se lze také dívat z pohledu událostí. Každý zahraný tah lze reprezentovat událostí *„provést tah“*. Tato událost je adresována samotné hře a nese s sebou seznam instrukcí, které mají být provedeny. Hra tyto změny aplikuje, čímž změní aktuální stav hry. Po změně stavu vyvolá novou událost *„předání tahu“*, která je odeslána dalšímu hráči (nebo v případě souběžných tahů všem hráčům, kteří jsou na řadě). Tato událost s sebou nese informaci o stavu hry pro konkrétního hráče.

1.4 Komunikace mezi klientem a serverem

V rámci modelu spouštění umělé inteligence na straně klienta je nutné zabývat se síťovou komunikací s herním serverem. Existuje několik možných způsobů předávání informací, které lze využít. V této sekci se zaměříme na možné způsoby vyměňování zpráv mezi oběma stranami ve vztahu k potřebám herního utkání.

Z hlediska datového toku lze spojení mezi klientem a serverem obecně rozdělit na dva typy – *jednosměrné* a *obousměrné spojení*. V případě jednosměrného dochází k posílání dat pouze z jedné strany na druhou (buď od klienta na server nebo opačně). V případě obousměrného dochází k vyměňovaní zpráv oběma směry v rámci jedné relace, obě strany mohou posílat zprávy zároveň.

Z [11] vyplývá, že jednosměrné spojení je vhodné pro vyřizování jednorázových dotazů. Příkladem může být HTTP požadavek, u kterého po přijetí odpovědi nepotřebujeme se serverem dále komunikovat. Tento přístup se využívá mj. při volání dotazů do *REST API*.

Oproti tomu je obousměrné spojení vhodné v případě, že chceme umožnit oběma stranám posílat zprávy podle potřeby. Toho lze využít pro komunikaci v průběhu hry, kdy je třeba reagovat na předávání tahů mezi hráči a vracení seznamu instrukcí k provedení herních akcí. Pro obousměrnou komunikaci se nabízí využití protokolu *WebSocket*.

1.4.1 Aplikační rozhraní webového serveru

Aby mohl klient interagovat se serverem, musí být navenek zpřístupněno API. Na to se pak klient může dotazovat a skrze tento prostředek se serverem komu-

nikovat. Mezi současně používané varianty patří REST API nebo GraphQL API.

REST je *softwarový architektonický styl*, který dle [12, kap. 3.1.3] poskytuje jednotnou sémantiku API pro CRUD (Create, Read, Update, Delete) operace a manipuluje zdrojem pouze výměnou „stateless“ reprezentací. Jedná se o model orientovaný na zdroje, které jsou jednoznačně identifikovatelné pomocí URI (Uniform Resource Identifier). Ačkoliv REST samotný se neomezuje na konkrétní protokol [13, kap. 5.3.2], nejčastěji používaným je HTTP. Pro manipulaci se zdroji se pak používají zejména metody POST, GET, PUT a DELETE (analogicky pro jednotlivé CRUD operace). REST jako takový poskytuje pouze seznam pravidel a principů pro tvorbu webových služeb.

GraphQL [14] je *dotazovací jazyk*, který slouží jako robustní varianta pro selekci a filtrování dat. Poskytuje dynamický mechanismus pro vybírání specifických položek objektů. Lze tedy jednoduše vybrat např. jména z kolekce všech uživatelů, kdežto v případě REST je nutné nejprve celou kolekci získat a až následně z ní vyfiltrovat jména. GraphQL tedy poskytuje možnost efektivnějšího dotazování právě díky možnosti specifikování dat, které od serveru chceme obdržet.

Dle [15] také řeší problém s *underfetching* – nutností posílat více požadavků (např. když chceme pro každého uživatele získat také seznam odehraných her); a *overfetching* – když server vrací více dat, než potřebujeme (např. když chceme pouze posledních 5 her, ve kterých uživatel zvítězil.) Vhodné využití tak nachází v aplikacích s komplexní stavbou objektů, které jsou mezi sebou propojeny.

Pro účely herní platformy se REST API hodí např. pro vytváření nebo výpis instancí her spuštěných na serveru. GraphQL je příliš komplexní pro takto jednoduché použití a vyplatí se spíše v případě, kde chceme filtrovat v databázi z velkého množství dat.

1.4.2 Formát dat pro výměnu informací

Klient i server mohou využívat různé softwarové technologie. Aby si obě strany navzájem rozuměly, je třeba informace vyměňovat ve formátu nezávislém na platformě. Nabízí se tři možnosti, jak data kódovat – pomocí textového formátu XML (Extensible Markup Language), nebo JSON (JavaScript Object Notation), nebo skrze formát binární.

XML je značkovací jazyk, který „používá elementy a atributy pro popis dat. Tato data jsou organizována ve stromové struktuře“ [16, překlad autora].

XML popisuje strukturu dat pomocí tagů. To však způsobuje větší velikost výsledných dat, které jsou kódovány skrze XML, oproti jiným formátům dle [17].

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
   <gameId>qwertz123</gameId>
</root>
```

Výpis kódu 1.2: Ukázka formátu XML

JSON oproti XML poskytuje odlehčenou variantu kódování dat, jelikož není zatížen žádnými redundantními informacemi. Samotná syntaxe zahrnuje datové typy (zejména objekty a pole). Gramatika JSON je relativně jednoduchá a dle [18] ji lze snadno parsovat.

```
{
    "gameId" : "qwertz123"
}
```

Výpis kódu 1.3: Ukázka formátu JSON

Binární formát je dle [17] nejefektivnější způsob kódování z hlediska výsledné velikosti dat. Oproti XML a JSON však není snadno čitelný pro člověka. Všechny strany, mezi kterými jsou data posílána, potřebují znát způsob, jakým byla data zakódována.

Formát XML není příliš vhodné využít pro frekventovanou komunikaci, kvůli množství redundantních dat. Vhledem k tomu, že předpokládáme velké množství zpráv posílaných mezi klientem a serverem v průběhu hry, chceme omezit velikost jednotlivých zpráv na minimum. Především z tohoto důvodu není vhodné data kódovat v XML.

Vhodnějším způsobem by tedy bylo kódovat data v binární podobně. Zde však narážíme na problém, že klient i server musí umět data kódovat. Navíc nelze jednoduše ladit chyby z toho důvodu, že formát není pro člověka čitelný.

Nejvhodnější variantou tedy zůstává kódování dat ve formátu JSON. Jedná se o kompromis mezi XML a binárním formátem z hlediska velikosti. Avšak jeho hlavní výhodou je to, že je pro člověka dobře čitelný a na první pohled je jasné, jaké informace jsou v datech obsaženy. Navíc existuje velké množství dostupných JSON parserů pro různé jazyky (seznam některých z nich je dostupný na [19]).

1.4.3 Technologie pro obousměrné spojení klient–server

Klient se dorozumívá s webovým serverem primárně pomocí HTTP. V této sekci se zaměříme na možné varianty spojení, které jsou postavené nad tímto

protokolem a které dovolují obousměrnou komunikaci.

Short polling je technika posílání HTTP požadavků na server v pravidelných intervalech. Podle [20, kap. 2] se jedná o typ obousměrného spojení mezi klientem a serverem simulující komunikaci v „reálném“ čase. Pokud server nemá v danou chvíli dostupná data, vrací prázdnou odpověď. Tato technika je náročná na síťovou komunikaci, neboť pro každý nový požadavek je potřeba ustavení nového spojení se serverem. Dochází tak ke zpoždění příchozích zpráv. Kvůli neustálému posílání nových požadavků dochází zároveň ke zbytečnému vytěžování serveru.

Long polling se snaží zmírnit zpoždění v komunikaci a zátěž serveru oproti short polling technice. Toho dosahuje tak, že na požadavek od klienta „odpovídá pouze ve chvíli, kdy nastane nějaká událost, stav nebo po uplynutí časového limitu“ [20, kap. 2, překlad autora]. Dochází k udržování otevřeného spojení až do chvíle, než server vrátí odpověď. Po přijetí odpovědi klient ihned posílá nový HTTP požadavek, aby udržel obousměrné spojení.

HTTP streaming je dle [20, kap. 3] mechanismus umožňující serveru posílat zprávy kdykoliv potřebuje. Po ustavení spojení již nedochází k jejímu uzavření. Server pozdrží odeslání odpovědi až do chvíle, kdy nastane nějaká událost nebo změna stavu. Po odeslání odpovědi zůstává spojení otevřené a server opět vyčkává na další událost, kterou má poslat klientovi. Tato technika (oproti předchozím uvedeným) ještě více snižuje zpoždění během komunikace, protože není třeba vytvářet nové HTTP požadavky.

WebSocket je *protokol* založený na HTTP. Na rozdíl od REST je hlavním cílem WebSocket „poskytnout mechanismus aplikacím, které potřebují obousměrnou komunikaci se serverem, která nezávisí na otevírání více HTTP požadavků“[21, překlad autora]. Po ustavení spojení mohou obě strany posílat zprávy v reálném čase po dobu jeho trvání. Odpadá tím režie spojená s vytvářením nových HTTP požadavků. Tento model se hodí zejména v případě potřeby výměny většího množství zpráv v nejkratším čase mezi klientem a serverem.

Na začátku spojení je poslán tzv. otevírací hanshake, což je běžný HTTP požadavek, ve kterém je dle [21, kap. 4] specifikovaná hlavička pro upgrade protokolu na WebSocket. V rámci tohoto požadavku je možné připojit doplňující URL parametry (autentizační token, ID hry atp.), na které může server případně zareagovat (např. odmítnutí upgrade protokolu při neplatném ID hry).

Podle [22] HTTP protokol umožňuje klientovi poslat požadavek na server a zpátky získat odpověď. S vývojem webových aplikací se však začala

objevovat poptávka, aby i server mohl posílat informace klientovi za účelem obousměrné komunikace. Z tohoto důvodu vznikly metody short a long polling společně s HTTP streaming. Nicméně tyto metody obsahují určité nedostatky, zejména spojené s režií při vytváření HTTP požadavků. Plnohodnotné řešení nakonec poskytl až protokol WebSocket, který umožňuje nativní posílání zpráv v reálném čase. Pro realizaci obousměrného spojení mezi klientem a serverem je nejvhodnější variantou použití WebSocket.

1.5 Chybové stavy v průběhu hry

V průběhu hry může dojít k různým chybovým stavům. V této části se zaměříme na některé problematické prvky a popíšeme možné způsoby řešení takovýchto stavů.

Prvním problematickým místem je **výstup umělé inteligence** agenta. Může nastat případ, kdy hra obdrží nevalidní instrukce. V také situaci nelze zahrát tah a dochází k chybovému stavu. Nabízí se dvě možná řešení. Prvním je nekompromisní přerušení a ukončení hry. Druhou variantou je umožnění agentovi opravit svůj tah. Ačkoliv je pravděpodobnější, že na stejný vstup agent odpoví stejným výstupem, může v sobě obsahovat prvek náhody, a na další pokus již vrátí validní odpověď. Na druhou stranu však chceme zamezit opakovanému posílání neplatných instrukcí. To lze vyřešit limitem pro počet chybných instrukcí, které lze tolerovat, tzv. *karmou*. V případě chybné instrukce snížíme hodnotu karmy a ve chvíli, kdy dosáhneme 0, přerušíme hru jako v případě první varianty.

Druhým problematickým místem je samotný **komunikační kanál**. Vzhledem k oddělení AI agenta na straně klienta od herní instance na straně serveru je třeba se vypořádat se síťovou komunikací. V průběhu hry může dojít k přerušení spojení. Na tuto událost musí server zareagovat. V tomto případě se nabízí řešení skrze *timeout* – časový limit, během kterého musí přijít zpráva od klienta. Jakmile dojde k vypršení časového limitu, dojde k přerušení a ukončení herního utkání. O této události je třeba informovat i zbylé připojené hráče.

Speciálním případem je přerušení spojení v době, kdy se klienti připojují k nové instanci hry. Tento stav lze řešit tak, že hře předáme informaci o tom, že k odpojení klienta došlo. Bude se čekat na nového hráče, který nahradí toho odpojeného. Hra začne až ve chvíli, kdy se připojí všichni hráči.

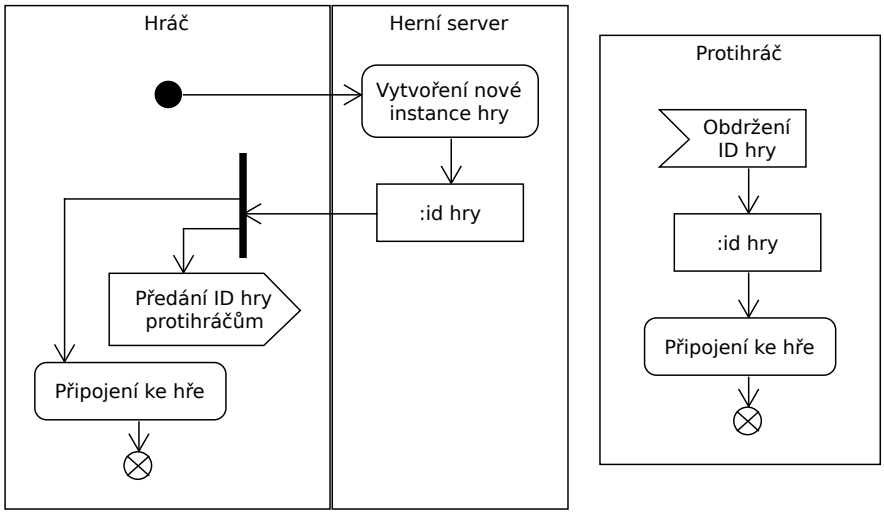
V neposlední řadě jsou problematickým bodem veškerá **příchozí data na server**. Data, která jsou na server odeslána, mohou obsahovat škodlivý vstup, která mohou narušit integritu systému. Tvoří bezpečnostní riziko, je proto potřeba, aby veškerý vstup byl kontrolovaný a ošetřený před dalším zpracováním.

1.6 Procesy herní platformy

V této sekci popíšeme procesy herní platformy. Tyto procesy jsou důležité pro následný návrh rozhraní herního modulu. Zaměříme se na popis spuštění hry a následného průběhu herního utkání a jeho ukončení.

1.6.1 Vytvoření a začátek hry

Aby mohli hráči porovnat své AI agenty, je nejprve nutné vytvořit novou instanci hry. Hráč musí zaslat tento požadavek na server, který mu po úspěšném vytvoření vrátí ID vytvořené instance. Toto ID musí dále přeposlat svým protihráčům, aby se mohli připojit ke stejné hře. Proces vytvoření hry je znázorněn na diagramu 1.3.

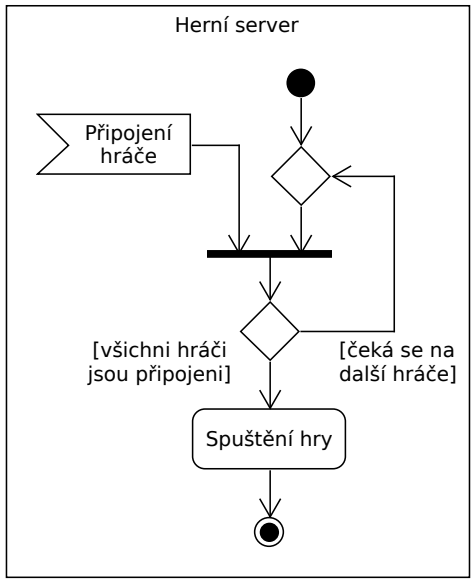


Obrázek 1.3: Vytvoření hry

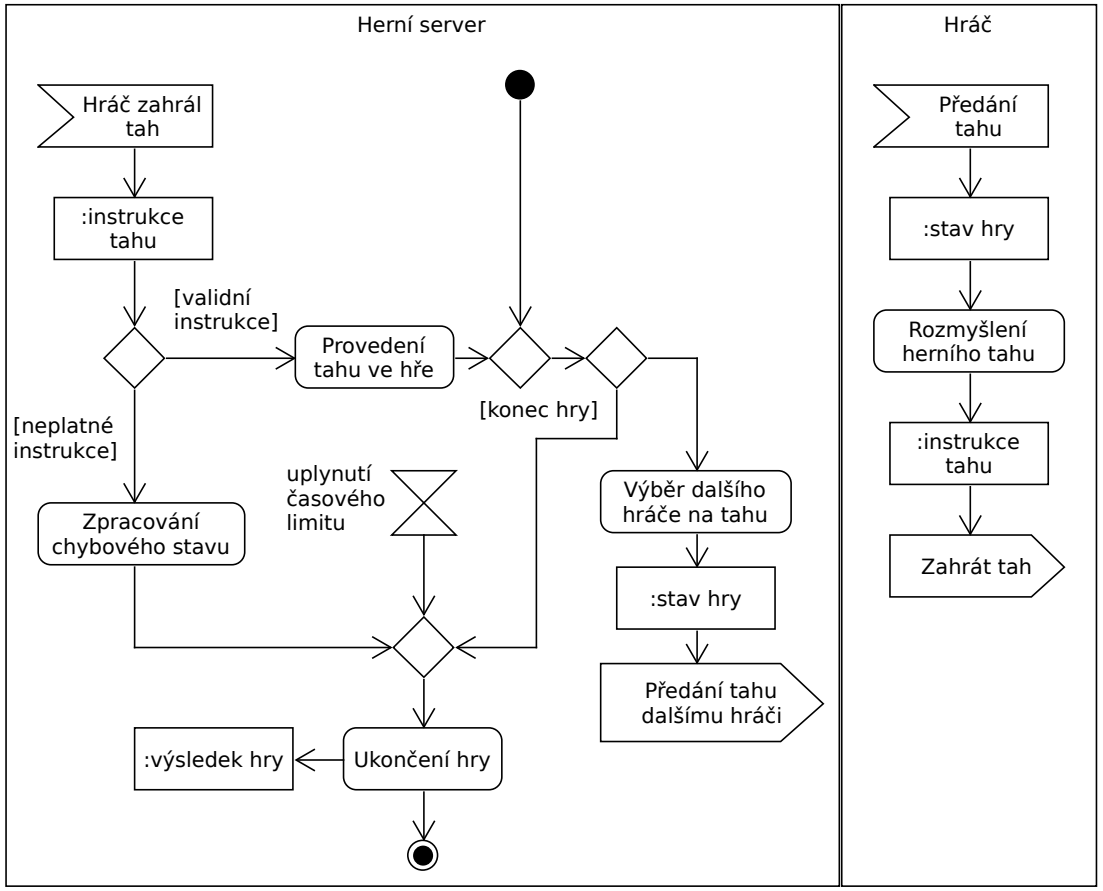
Jednotliví hráči se posléze musí připojit ke stejné instanci hry. Jakmile se připojí všichni hráči, dojde ke spuštění hry. Viz diagram 1.4

1.6.2 Průběh a ukončení hry

Na začátku každého tahu je určeno, který hráč je na řadě. Jemu je třeba následně poskytnout aktuální stav hry a předat mu tah. Hráč na základě stavu hry vytvoří seznam instrukcí, které se mají provést, a pošle je zpět na herní server. Ten data zkontroluje, a pokud jsou instrukce validní, provede daný tah. V opačném případě dojde k přerušení hry a následnému ukončení. Po provedení herních akcí začíná nový tah, pokud však v předchozím kroku nenastal konec hry. V takovém případě dojde k přerušení a řádnému ukončení. K ukončení hry může dojít také v případě uplynutí časového limitu, kdy hráč nestihne v čas zahrát svůj tah. Průběh hry je znázorněn na diagramu 1.5.



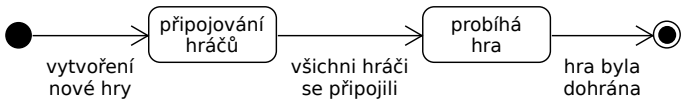
Obrázek 1.4: Začátek hry



Obrázek 1.5: Diagram průběhu hry

1.6.3 Stavy hry

V průběhu výše popsaných procesů dochází ke změně stavu hry. Po vytvoření nové instance je hra ve stavu, kdy se k ní mohou *připojovat jednotliví hráči*. Ve chvíli, kdy jsou všichni připojeni, dojde k jejímu spuštění a přechodu do stavu *probíhající hry*. Ve chvíli, kdy dojde k ukončení hry (ať už standardně nebo skrze chybový stav), přejde do koncového stavu. Stavový diagram je znázorněn na obrázku 1.6.



Obrázek 1.6: Stavový diagram hry

Kapitola 2

Návrh platformy

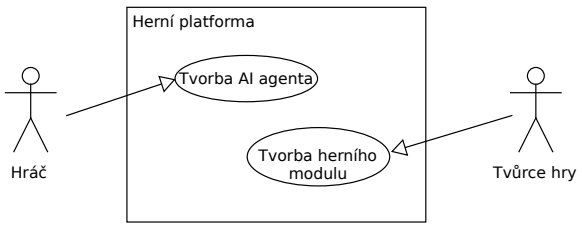
V této kapitole se zaměříme na návrh herní platformy postavené na klient–server architektuře – kód AI agenta je spouštěný na klientovi, přičemž hry běží na straně serveru. Rozdělíme uživatele platofrmy, navrhneme vhodnou strukturu komponent a zaměříme se na odpovídající výběr technologií.

2.1 Uživatelé platformy

Při návrhu systému bylo nutné vzít v potaz následující dva požadavky, které herní platforma musí umožňovat:

- odehrání hry mezi AI agenty;
- možnost vytvoření vlastního herního modulu.

V důsledku toho lze uživatele rozdělit do dvou kategorií – *hráče*, kteří vytváří své AI agenty, a *tvůrce her*, kteří programují herní moduly. Toto rozdělení znázorňuje diagram 2.1.



Obrázek 2.1: Uživatelé platformy

2.2 Komponenty z pohledu správce

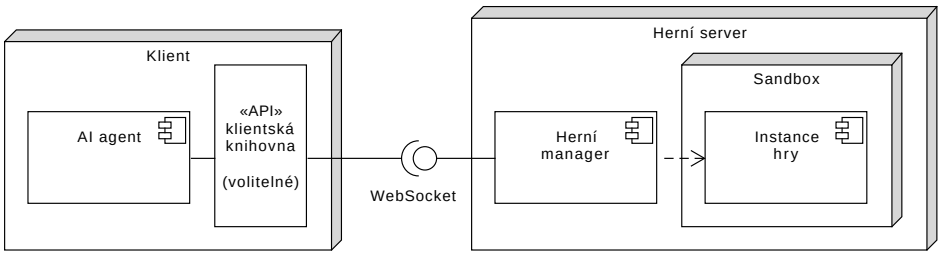
Architektura platformy se z pohledu správce dá rozdělit na následující 3 komponenty:

herní manager řeší obsluhu komunikace mezi klienty a instancí hry;

AI agent se připojuje k hernímu manageru a řeší logiku umělé inteligence;

herní modul spravuje stav a průběh hry.

Přičemž herní manager tvoří hlavní součást herní platformy. AI agenti běží na straně klienta a jsou spravováni jednotlivými hráči. Instanci hry spravuje tvůrce dané hry.



Obrázek 2.2: Přehled vazeb mezi jednotlivými částmi platformy

Herní manager přijímá požadavky od klientů. Vytváří jednotlivé instance her a obsluhuje komunikaci mezi AI, spuštěným na straně klienta, a herním modulem na straně serveru. Komunikace mezi klientem a serverem probíhá pomocí protokolu WebSocket. Jednotliví klienti mohou pro spojení využít klientské knihovny herní platformy, která bude rozebrána v následující kapitole 3. Schéma vztahů mezi jednotlivými komponentami je znázorněno na obrázku 2.2.

Mezi jednotlivými komponentami probíhá tok dat následovně. Herní modul poskytuje informaci o stavu hry na začátku tahu. Tento stav získává AI na vstupu, přičemž po zpracování musí na výstup vrátit seznam instrukcí, které se mají provést. V obou případech se jedná o jednoduché objekty, jejichž struktuře rozumí pouze daný herní modul a jednotliví agenti. Pro herní manager tato data představují pouze abstrakci, kterou přeposílá mezi klientem a instancí hry. Tento návrh vychází z analýzy herních procesů.

2.3 Volba technologií

Pro implementaci herního serveru jsme zvolili runtime prostředí Node.js [23]. Dle [24] se jedná o technologii, která je vhodná pro použití v aplikacích s vel-

kým množstvím síťových IO operací, ale zároveň není vhodná pro náročné výpočetní operace. Jedná se tedy o adekvátní nástroj, neboť od herní platformy vyžadujeme intenzivní komunikaci mezi připojenými klienty skrze server. Podmínkou je však minimalizování využití výpočetního výkonu herními moduly.

Dle provedených zátěžových testů populárních serverových technologií ve studii [25] vyšlo najevo, že Node.js dosahuje lepších výsledků při velkém množství souběžných spojení s malým množstvím posílaných dat než konkurenční řešení.

Node je dle [26] postavený na event-driven architektuře, veškeré zpracování požadavků probíhá jednovláknově v rámci tzv. event-loop. Přesto však dle [27] umožňuje efektivní zpracování současných požadavků. Je však důležité obsluhovat veškeré požadavky co nejrychleji. Z principu lze totiž v jednu chvíli vyřizovat pouze jeden požadavek. Pokud bude zpracování jednoho požadavku trvat dlouhou dobu, zdrží se automaticky všechny ostatní, které čekají ve frontě na vyřízení.

Důležitou vlastností je možnost udržovat veškeré instance herních objektů *in memory* – v operační paměti procesu. Pro ukládání stavu není tedy nutné používat externí databáze, ani přistupovat na disk, a provádět tak nadbytečné IO operace. Jednotlivé instance her z principu není nutné ukládat do databáze, vzhledem k jejich krátké životnosti. Po skončení hry uvolňujeme objekty z paměti.

Pro Node existuje množství open-source balíčků, které lze využít – od jednoduchých knihoven po celé frameworky. Balíčky lze získávat pomocí nástroje NPM [28] (Node Package Manager). Jedním z dostupných balíčků je i knihovna Socket.io [29], která poskytuje nástavbu nad protokolem WebSocket. V rámci práce ji použijeme pro obousměrnou komunikaci mezi klientem a serverem. Důvodem volby knihovny jsou především funkcionality, které poskytuje navíc oproti protokolu WebSocket. Jedná se např. o přívětivější API, automatické znovupřipojení po přerušení spojení, nebo fallback varianty (polling, long polling aj.).

Jako základ herního serveru jsme zvolili webový framework Koa.js [30]. Pomáhá zpracovávat příchozí požadavky na server skrze route. Má minimalistické jádro a veškerá funkcionalita je rozšiřována skrze další balíčky. Díky tomu je framework v základu velmi lehký, což je žádoucí vlastnost v situaci, kdy nechceme zbytečně zatěžovat výpočetní výkon serveru.

2.3.1 Izolace herních modulů

Na herním serveru chceme spouštět nezávisle vytvořené herní moduly od ostatních programátorů. Jelikož nemáme kód modulů třetích stran pod kontrolou a nemůžeme jim plně důvěřovat, je potřeba je spouštět v izolovaném prostředí, tzv. *sandbox*. Přirozeně chceme omezit přístup k určitým prostředkům, abychom tak zamezili možnému kompromitování systému. Zejména chceme

omezit přístup k build-in modulům, které by umožňovali komunikaci přes síť nebo přístup na filesystém. Zároveň nechceme, aby modul mohl přistupovat ke globálním proměnným nebo jakkoliv měnit data v rámci herního manageru.

Pro izolaci herních modulů jsme využili knihovny vm2 [31], která umožňuje spouštět herní moduly v kontrolovaném prostředí. Pomocí whitelist seznamu umožňuje specifikovat seznam build-in modulů, které může hra využít.

Jak se později ukázalo, slabým místem však zůstává způsob zpracování dat v rámci Node.js procesu. Veškeré volání funkcí probíhá v jediném vlákně. Škodlivý modul by tedy mohl spotřebovávat výpočetní prostředky v neprospěch systému. Proti tomu by bylo možné se bránit tak, že bychom jednotlivé hry spouštěly mimo hlavní vlákno. Avšak tímto bychom i tak dle [32] přišli o výhody rychlého zpracování. Jedinou účinnou obranou tedy zbývá takovýto modul zcela zakázat.

Kapitola 3

Implementace prototypu

V této kapitole ze zaměříme na implementaci herní platformy. Dále zdokumentujeme návrh rozhraní herního modulu. Znázorníme strukturu REST rozhraní. Poté se zaměříme na popis událostí, které jsou vyměňovány v průběhu hry. Na závěr popíšeme implementaci pomocné klientské knihovny pro spojení se serverem, kterou mohou hráči pro své AI agenty využít, aby si zjednodušili práci s obousměrnou komunikací přes WebSocket.

3.1 Tvorba herního modulu

Každá hra má svůj vlastní modul. Herní manager tyto moduly načítá a zprostředkovává uživatelům možnost soupeřit v nich proti sobě. Aby bylo možné herní modul použít, je potřeba splnit následující kroky:

1. vytvořit složku pro hru (např. `othello/`),
2. uvnitř této složky vytvořit soubor `index.js`, který bude exportovat třídu implementující předdefinované rozhraní (viz sekce 3.1.1), a
3. vytvořit soubor `readme.txt`, který bude obsahovat stručný popis hry a zadefinuje strukturu předávaných dat (viz sekce 3.1.2).

Minimální povinná struktura modulu je tedy následující:

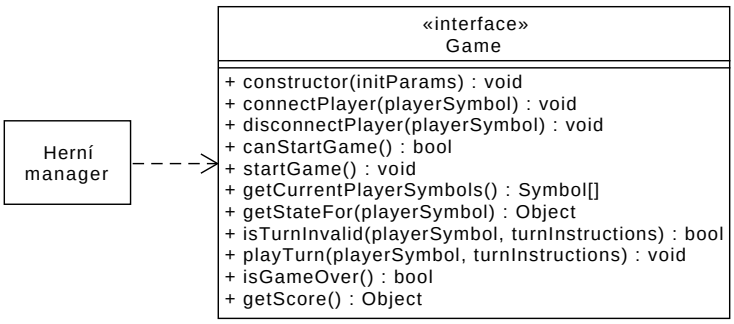
```
<nazev-herniho-modulu>
├── index.js ..... hlavní soubor hry, načítaný managerem
└── readme.txt ..... informace pro hráče s popisem dat
```

Název modulu by měl být malými písmeny a zároveň by neměl obsahovat jiné znaky než alfanumerické. Jednotlivé herní moduly jsou umístěny na herním serveru ve složce `src/games/`. Herní manager při spuštění serveru načítá moduly z této složky.

Pro každé utkání je vytvořen nový herní objekt, který existuje po celou dobu hry. Modul sám o sobě s klientem nekomunikuje, veškerou komunikaci řídí herní manager skrze volání jednotlivých metod herního objektu. Každá instance je spouštěna v izolovaném prostředí. Herní modul může používat mj. globálních proměnných pro ukládání vnitřního stavu, neboť nejsou mimo toto prostředí dostupné zvnějšku. Veškeré proměnné se udržují v paměti v průběhu hry – toho lze využít zejména pro ukládání změn herního stavu mezi jednotlivými tahy. Hernímu modulu není umožněno importovat systémové moduly (např. `fs` pro přístup k souborovému systému nelze použít). Může si však vlastní logiku rozdělit do více souborů a ty importovat uvnitř `index.js`.

3.1.1 Rozhraní herního modulu

Herní manager v sobě udržuje `mapu`[5] herních instancí. Zároveň zprostředkovává přenos informací mezi připojenými klienty a konkretními instancemi skrze volání příslušných metod. Aby mohla komunikace probíhat správně, musí herní modul implementovat rozhraní 3.1. V této sekci podrobně rozebereme jednotlivé metody seskupené podle fází – 1. vytvoření a připojení hráčů, 2. spuštění hry, 3. samotný průběh a 4. ukončení hry. (Stručný přehled metod s jejich popisem je přiložen v příloze B.)



Obrázek 3.1: Rozhraní herního modulu

3.1.1.1 Vytvoření hry a připojení hráčů

Novou instanci hry vytváří herní manager na základě žádosti od klienta poslané skrze REST API. Herní manager vygeneruje unikátní identifikátor – *ID hry* – a pod tímto klíčem uloží herní instanci do `mapy`. ID hry zároveň vrátí v odpovědi klientovi, který poslal žádost. Pomocí tohoto identifikátoru se mohou následně jednotliví agenti připojit k dané hře.

[5]Mapa je objekt udržující *key-value* páry. Konrétnímu klíči lze přiřadit její hodnotu. V tomto použití je klíčem **ID hry** a hodnotou samotná **instance hry**.

Při vytváření nové instance může klient poslat v rámci REST požadavku objekt s parametry pro inicializaci hry. Herní manager tyto data předává konstruktoru při vytváření nové instance hry. Skrze tyto parametry může hráč specifikovat určité modifikace či vstupní hodnoty atp. – např. rozdílný způsob počátečního rozložení kamenů na začátku hry Othello aj. Definice vstupních parametrů musí být předem určena herním modulem.

Jakmile je vytvořena instance hry, mohou se k ní začít připojovat jednotliví AI agenti. Agent komunikuje přímo s herním managerem (prostřednictvím protokolu WebSocket) a je reprezentován objektem `Socket`[6]. Uvnitř manageru dochází k překladu `Socket` objektů na `Symbol`[7], které jsou dále používány pro komunikaci s herní instancí. Herní modul sám o sobě nekomunikuje s jednotlivými klienty, ale využívá zástupných symbolů, které reprezentují cílové hráče.

Ve chvíli, kdy se připojí nový klient k hernímu serveru, manager vytvoří nový `Symbol`. Tento symbol předá instanci hry skrze volání metody `connectPlayer`. Hra musí uložit tento symbol, protože ho dále bude potřebovat pro komunikaci.

Pokud během fáze připojování klientů ke hře, kdy ještě nejsou všichni hráči připojení, dojde k odpojení některého z nich, manager zavolá metodu `disconnectPlayer`. Tím dá vědět, že daný klient již se serverem nekomunikuje a hra s ním dále nesmí počítat. Pokud tento odpojený hráč pokusí znovu připojit, bude mu vytvořen nový `Symbol`.

3.1.1.2 Spuštění hry

Po každém připojení klienta se zavolá metoda `canStartGame`, která udává, jestli je možné začít hru. Nutnou podmínkou pro vrácení kladné hodnoty je připojení všech hráčů. Pokud je možné začít hru, zavolá se metoda `startGame`. Ta provádí inicializaci počátečního stavu hry (např. rozložení hráčů na herní ploše atp.) a zároveň musí určit prvního hráče na řadě. Přibližný průběh spuštění hry v herním manageru je znázorněn na ukázce kódu 3.1.

```
//game je konkrétní instance herního modulu
const playerSymbol = Symbol();
game.connectPlayer(playerSymbol);
if( game.canStartGame() === true ) {
    game.startGame();
}
```

Výpis kódu 3.1: Spuštění hry v herním manageru

[6] Jedná se o objekt knihovny Socket.io, který je dostupný hernímu manageru.
[7] *Symbol* je Speciální datový typ JS sloužící jako unikátní identifikátor.

3.1.1.3 Průběh hry

Průběh hry lze rozdělit na dvě fáze. Nejprve je potřeba aktuálnímu hráči na tahu předat stav hry. Následně, po přijetí instrukcí k provedení tahu, je třeba provést validaci a vykonání herních akcí.

Protože je možné, aby v jednu chvíli bylo na tahu více hráčů najednou (tzv. *souběžný tah*), je potřeba tuto možnost zohlednit. Herní manager se před začátkem každého tahu dotazuje herní instance na seznam hráčů (pole `Symbol` objektů), kteří jsou aktuálně na řadě, skrze metodu `getCurrentPlayerSymbols`. Následně pro každého z nich potřebuje získat herní stav voláním `getStateFor`. Každému AI agentovi jsou pak následně poslány jejich příslušné herní stavy.

Jakmile se vrátí instrukce k provedení tahu zpátky na herní server, dojde ke kontrole jejich validity pomocí metody `isTurnInvalid`. Tato metoda vrací `true`, pokud jsou data neplatná, přičemž v takovém případě dochází k ukončení hry. V opačném případě jsou instrukce dále předány metodě `playTurn`. Je zaručeno, že před voláním této metody došlo k předchozí validaci.

Metoda `playTurn` musí zajistit následující úkony:

- provedení instrukcí a změnu herního stavu;
- určení následujícího hráče na tahu;
- kontrola, zda-li po zahrání tahu nenastal konec hry.

3.1.1.4 Ukončení hry

Pokud po provedení herních akcí v rámci metody `playTurn` nastal konec hry, je třeba provést vyhodnocení hry a řádně ukončit spojení s jednotlivými klienty.

Herní manager volá metodu `isGameOver`, aby zjistila, zda-li konec hry skutečně nastal. Pokud ano, dojde k zavolání metody `getScore`, která musí vrátit výsledek daného utkání. Tato data jsou před ukončením spojení poslána všem klientům.

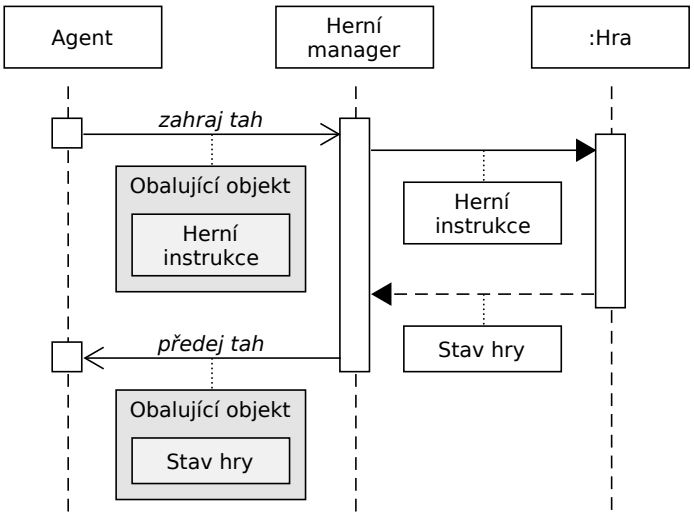
3.1.2 Reprezentace předávaných dat

Mezi klientem a serverem je potřeba po síti předávat data vztahující se k hernímu utkání. Před odesláním klientovi je nutné data serializovat do textové reprezentace a při příjmu na serveru zpátky deserializovat na konkretní objekty. Pro tuto potřebu využijeme formát JSON, který lze jednoduše parsovat pomocí nativní funkce `JSON.parse()`.

Herní manager tvoří prostředníka na cestě mezi klientem a herním objektem. V rámci komunikace chceme uvnitř jedné zprávy předávat data jak hernímu manageru, tak samotné instanci hry. Protože herní manager stojí během komunikace před samotnou instancí, lze data pro hru obalit objektem, kterému rozumí herní manager. Na tok dat lze nahlížet z pohledu dvou vrstev. **Mezi AI agentem a herní instancí** probíhá komunikace skrze *protokol*,

který je definovaný herním modulem (v povinném souboru `readme.txt`). Tato data jsou zabalena objektem „nižší“ vrstvy, v rámci které probíhá komunikace **mezi klientskou knihovnou a herním managerem**.[8] (Klientská knihovna je popsaná dále v sekci 3.4.)

Herní manager data určená hře extrahuje a předává je dále herní instanci. Uvnitř obalujícího objektu lze hernímu manageru předávat např. ladící informace apod. Na diagramu 3.2 je znázorněno předávání těchto dat napříč platformou.



Obrázek 3.2: Ukázka obalení dat během komunikace

Protože reprezentaci objektů musí rozumět jak hra, tak ale i samotný agent, je nutné zveřejnit strukturu těchto objektů. K tomuto účelu slouží soubor `readme.txt`, ve kterém by autor hry měl strukturu objektů definovat tak, aby bylo **jednoznačné, jak objekty vypadají**. Konkretně je nutné definovat následující 4 objekty, které se při komunikaci používají:

1. **vstupní parametry** pro konstruktor hry,
2. **stav hry** pro hráče na tahu,
3. **herní instrukce** pro provedení tahu a
4. objekt s **výsledkem hry**.

Vstupní parametry jsou poslány pouze jednou v rámci požadavku pro vytvoření nové hry. Jsou předány hernímu modulu přes `constructor`. Stav hry je poslán agentovi, když se dostane na řadu. Herní manager tento objekt bere

[8] Využití klientské knihovny sice není povinné, avšak v takovém případě je nutné data posílat na server v obalujícím objektu, aby jej mohl herní manager rozbalit a dále předat instanci hry.

z metody `getStateFor`. Jako odpověď posílá agent herní instrukce, které jsou následně validovány skrz `isTurnInvalid` a posléze předány do `playTurn`. Výsledek hry vrací metoda `getScore`.

Ukázka jednotlivých obalujících objektů je znázorněna tabulkou 3.1. Uvnitř JSON reprezentace jsou uvedeny zástupné symboly začínající znakem `@` – označují části, které jsou specifikovány tvůrcem hry. (Např. `@initParams` v obalujícím objektu poslaném při vytváření nové instance hry.)

	Datový objekt	Reprezentace obalujícího objektu ve formátu JSON

1	inic. parametry	`{`
		`"gameName": "othello",`
		`"initParams": @initParams,`
		`"timestamp":"2019-05-04T08:33:54.129Z",`
		`...`
		`}`

2	stav hry	`{`
		`"turnInstructions": @turnInstructions,`
		`"timestamp":"2019-05-04T08:48:23.087Z",`
		`...`
		`}`

3	herní instrukce	`{`
		`"gameState": @gameState,`
		`"timestamp": "2019-05-04T08:51:19.926Z",`
		`...`
		`}`

4	výsledek hry	`{`
		`"score": @score,`
		`"timestamp":"2019-05-04T08:55:33.372Z",`
		`...`
		`}`

Tabulka 3.1: Obalující objekty herního manageru s daty pro instanci hry

3.2 REST rozhraní herního serveru

Koncový bod pro REST požadavky se nachází na `/api`. Hlavním zdrojem dostupným skrze API je hra, rozhraní umožňuje zejména vytváření nových herních instancí. K dispozici jsou akce popsané v tabulce 3.2.

Metoda	Zdroj	Popis
`GET`	`/`	Výpis dostupných akcí API
`GET`	`/game/`	Seznam dostupných her
`GET`	`/game/{gameName}`	Zobrazí popis hry s definicí objektů pro komunikaci
`POST`	`/game/`	Vytvoří novou instanci hry

Tabulka 3.2: Akce REST rozhraní herního serveru

Při vytváření nové instance je potřeba poslat data uvnitř těla POST požadavku. Povinnou položkou je `gameName` – název hry, kterou chceme hrát. Ukázka těla požadavku je znázorněna kódem 3.2.

```
{
    "gameName": "othello",
    "initParams": {
        "startingPosition": "parallel"
    }
}
```

Výpis kódu 3.2: Ukázka požadavku pro vytvoření nové instance hry Othello

3.3 Řízení průběhu hry skrz události

Průběh hry je řízen skrze obousměrné spojení mezi klientem a serverem. Prostřednictvím něj je realizováno připojování agentů a obsluha jednotlivých tahů. Při ukončení utkání zařizuje také poslání výsledků hry všem účastníkům.

Pro realizaci obousměrného spojení na straně serveru jsme využili knihovnu **Socket.io**, která umožňuje používat jednodušší rozhraní pro volání a obsluhu událostí. Koncový bod pro toto spojení se nachází na cestě `/play`.

Ačkoliv pro připojení k serveru může klient využít vlastního řešení pracující s WebSocket protokolem, je doporučené využít knihovny **Socket.io-client** [29], která klientovi poskytuje komplementární rozhraní pro spojení s knihovou Socket.io na straně serveru. Varianty této knihovny jsou mimo JS dostupné i pro jiné jazyky (např. C++ nebo Java). V rámci řešení jsme implementovali také vlastní knihovnu (více v následující sekci 3.4), která Socket.io-client využívá.

Aby se klient mohl připojit ke konkretní hře, musí určit ID hry v parametru `gameId` v rámci handskahe. Na straně serveru dojde k vyvolání události `connect`, která zajistí připojení hráče k dané hře.

3.3.1 Události od klienta na server

Jedinou událostí, kterou během hry může klient na serveru vyvolat, je událost `makeTurn`. Ta v sobě obsahuje instrukce k provedení herních akcí. Klient touto událostí reaguje na příchozí žádost o provedení tahu ze strany serveru.

Událost	Popis	Připojená data
`makeTurn`	odeslání instrukcí k provedení tahu	instrukce tahu

Tabulka 3.3: Přehled událostí od klienta na server

V průběhu spojení může nastat několik dalších událostí, které kontroluje knihovna Socket.io. Jedná se zejména o chybové stavy – např. když dojde k odpojení klienta v průběhu utkání, je třeba na tuto událost `disconnect` adekvátně reagovat, ukončit hru a odpojit ostatní klienty od serveru.

3.3.2 Události od serveru na klienta

Hlavním úkolem herního serveru v průběhu hry je informování jednotlivých hráčů o tom, že jsou na řadě. Skrze událost `makeTurn` posílá herní stav a očekává, že klient po rozmyšlení herních akcí odpoví vyvoláním stejnojmenné události na serveru s připojeným seznamem instrukcí tahu.

Jakmile nastane situace, kdy aktuální hráč na řadě není schopný zahrát již žádný tah, dojde k ukončení hry. Herní server všem připojeným klientům odešle událost `gameOver` s připojeným objektem s výsledkem hry. Jedná se o poslední událost, která je ze strany serveru vyvolána před ukončením spojení.

Událost	Popis	Připojená data
`makeTurn`	předání tahu hráči	stav hry
`gameOver`	ukončení hry	výsledek hry

Tabulka 3.4: Přehled událostí od serveru na klienta

Klientovi jsou mimo výše uvedené události v tabulce 3.4 posílané i další, které informují o spojení mezi oběma stranami. Tyto události jsou vyvolávány samotnou knihovnou Socket.io. Zahrnují mj. následující:

`connect` se vyvolá po úspěšném připojení klienta,

`disconnect` nastane, když dojde k odpojení klienta,

`connect_error` pokud nastane chyba související se spojením,

3.4 Klientská knihovna pro komunikaci

Pro usnadnění komunikace v průběhu hry jsme implementovali vlastní klientskou knihovnu, kterou mohou hráči pro spojení se serverem využít. Jedná se o Node.js modul. Závisí na knihovně Socket.io-client, kterou na pozadí využívá pro obousměrné spojení s herním serverem.

Umožňuje programátorovi napsat jedinou funkci s vlastní logikou AI, kterou lze v rámci knihovny následně registrovat jako handler[9]. Jakmile dojde k zachycení události `makeTurn` ze strany serveru, zavolá se tento handler, který dostane herní stav skrze vstupní parametr. Návratovou hodnotou pak musí být objekt s instrukcemi pro zahrání tahu, které knihovna odešle zpět na server. Samotný hráč v případě, že použije tuto knihovnu, nemusí dále řešit komunikaci s herním serverem.

Modul vrací funkci, která po zavolání vytváří objekt s pomocnými metodami. Na vstupu vyžaduje URL herního serveru (např. `//localhost:3000`) a ID hry. Skrze pomocné metody lze následně registorvat handlery. Ukázkové použití knihovny je znázorněno kódem 3.3. K dispozici jsou následující metody:

`registerGameTurnHandler( callback )` zaregistruje callback, který se zavolá pokaždé, když ze serveru přijde událost `makeTurn`. Výstupem callbacku musí být instrukce k provedení tahu, které jsou v posléze odeslány na server;

`registerGameOverHandler( callback )` zaregistruje callback, který se zavolá, když přijde událost `gameOver`. Callback dostane skrze vstupní parametr objekt s výsledkem hry;

`connect()` připojí agenta k hernímu serveru. Lze zavolat až po té, co je zaregistrován handler obsluhující tahy.

```
const ClientAPI = require('./clientapi');
const api = ClientAPI(baseURL, gameId);
api.registerGameTurnHandler((gameState) => {
    // Samotná logika AI agenta.
    // Na základě herního stavu (gameState) vygeneruje
    // instrukce tahu (turnInstructions).
    return turnInstructions;
});
api.registerGameOverHandler((score) => {
    console.log("Výsledek hry:", score);
});
api.connect();
```

Výpis kódu 3.3: Ukázka použití klientské knihovny

[9]Callback, který je zavolaný v momentě, kdy nastane nějaká událost.

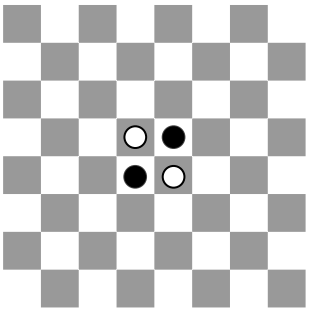
Kapitola 4

Testování funkčnosti

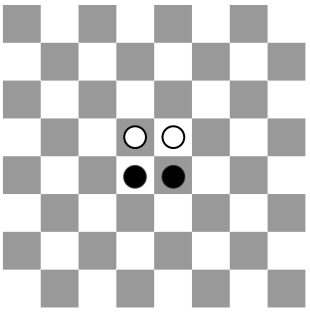
V této kapitole se zaměříme na testování funkčnosti prototypu platformy na zvolené hře Othello a dvou testovacích mock agentech. Nejprve popíšeme implementaci herního modulu. Následně se zaměříme na automatické testy REST API. Dále provedeme základní test funkčnosti. Poté podrobíme platformu testování chybových stavů a budeme zkoumat reakce klienta a serveru. Na závěr provedeme zátěžový test, ve kterém prozkoumáme chování platformy při velkém množství připojených klientů.

4.1 Imlementace hry Othello

Pro účely testování jsme implementovali variantu hry Othello. Jedná se o hru pro dva hráče, ve které se střídají po tazích v pokládání kamenů na poli 8×8. Kameny mají buď bílou, nebo černou barvu. Kámen lze položit pouze na taková pole, od kterého lze uzavřít souvislou řadu kamenů opačné barvy mezi kameny vlastní barvy. Tyto kameny jsou zajaty a nahrazeny kameny vlastní barvy. Hra končí ve chvíli, kdy hráč na řadě není schopný položit další kámen. Vítězem je ten, který na konci hry vlastní více kamenů stejné barvy.



Obrázek 4.1: Standardní rozložení



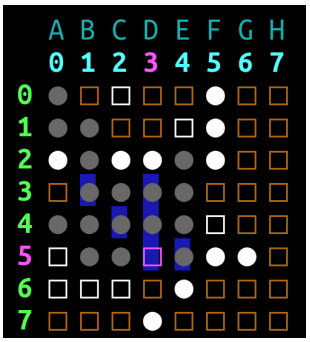
Obrázek 4.2: Vodorovné rozložení

Před začátkem hry jsou na 4 středových polích herní desky rozloženy 2 bílé a 2 černé kameny. Existují dvě možné varianty rozložení kamenů: (a) *standardní*, kdy jsou kameny položeny do kříže, nebo (b) *vodorovné*, kdy jsou 2 kameny jedné barvy položeny v první řadě a kameny druhé barvy v řadě pod ní. Rozložení lze specifikovat v rámci inicializačních parametrů předávaných při vytváření nové instance hry. Pro vodorovné rozložení by vstupní objekt (ve formátu JSON) vypadal následovně: `{ "startingPosition": "parallel" }`. Možné varianty jsou znázorněny na obrázku 4.1 a 4.2.

Herní pole je reprezentované 2D polem znaků z množiny `{_, W, B}`. Znak `_` představuje prázdné pole a znaky `W` a `B` značí pole obsazené bílým nebo černým kamenem. Pro účely ladění jsme vytvořili pomocnou funkci pro výpis herního pole na terminál. Znázorňuje vybranou pozici pro položení kamene a označuje, které kameny budou otočeny po zahrání tahu. Ukázka reprezentace herního pole v grafické podobě je znázorněna na obrázku 4.3.

```
// Herní stav
playerColor = 'W'
board =
[ [ 'B', 'B', 'W', '_', 'B', '_', '_', '_' ],
  [ '_', 'B', 'B', 'B', 'B', 'B', '_', '_' ],
  [ '_', '_', 'W', 'B', 'B', 'B', '_', '_' ],
  [ '_', '_', 'W', 'B', 'B', '_', '_', 'W' ],
  [ '_', '_', 'B', 'B', 'B', 'B', 'W', '_' ],
  [ 'W', 'W', 'W', '_', '_', 'W', '_', '_' ],
  [ '_', '_', '_', '_', '_', 'W', '_', '_' ],
  [ '_', '_', '_', '_', '_', '_', '_', '_' ] ]

// Instrukce tahu
selectedPosition = [3, 5]
```



Obrázek 4.3: Ukázka výpisu herního pole během hry

4.2 Automatické testování REST API

Pro ověření správné funkčnosti REST API jsme vytvořili automatické testy s pomocí nástroje Jest [33] a SuperTest [34]. Zaměřili jsme se na kontrolu stavových kódů, správně nastavené hlavičky `Content-Type` a částečně na obsah těla odpovědi.

Otestovali jsme, zda-li na validní vstup odpovídá server se správnými daty, ale také že na špatný vstup vrátí příslušný chybový stavový kód:

400 bad request pro nevalidní vstupní data,

404 not found pro nedefinovaný zdroj,

405 method not allowed pro neplatnou metodu na existujícím zdroji a

406 not acceptable pro nepodporovaný formát těla odpovědi.

Přehled vybraných testovaných požadavků je znázorněn v tabulce 4.1.

Akce	Stav. kód	Content-Type	Obsah odpovědi
GET /api/	200	application/json	seznam akcí REST API
GET /api/game	200	application/json	seznam dostupných her
GET /api/game/othello	200	text/plain	nápověda ke hře
POST /api/game	200	application/json	ID hry
POST /api/game[10]	400	-	-
GET /api/undefined	404	-	-
DELETE /api/game	405	-	-
GET /api/game/othello[11]	406	-	-

Tabulka 4.1: Testované požadavky na REST API

4.3 Základní test funkčnosti

Funkčnost platformy jsme otestovali pomocí dvou spuštěných instancí agentů, připojených ke hře Othello. Logiku AI agenta jsme, pro testovací účely, založili na náhodném vybírání pozice z dostupných možných. Pro realizaci jsme využili vlastní klientské knihovny pro připojení k hernímu serveru (viz kapitola 3.4 - Klientská knihovna pro komunikaci). Spuštění testu probíhalo následovně:

1. Na localhost jsme spustili herní server s nahraným modulem hry Othello.
2. Pomocí požadavku na REST API (`POST /api/game`) jsme vytvořili novou instanci hry. V odpovědi jsme zpátky získali její *ID hry*.
3. Spustili jsme 2 instance AI agenta s použitím *ID hry* získaného z předchozího bodu.

Tímto jsme ověřili funkčnost připojení agentů, obousměrnou komunikaci mezi klientem a serverem a správný průběh hry. Hra byla zdárně odehrána a test proběhl úspěšně.

Nedostatek testu však spočíval v tom, že samotná komunikace probíhala na stejném zařízení. Pro ověření komunikace přes síť jsme využili nástroje ngork [35], který umožňuje skrze bezpečný tunel vystavit lokální server na veřejně přístupný bod. To umožňuje ověřit případné problémy při komunikaci v rámci internetu. V tomto případě průběh testu probíhal obdobně akorát s tím rozdílem, že se agenti připojovali na veřejně přístupnou adresu. Při testu se projevilo zpoždění spojené s komunikací přes internet na celkové době průběhu hry. Celkové zpoždění, v rámci několika odehraných her, způsobené komunikací po síti při použití ngork serveru v regionu EU zabralo cca 4,5 s, v regionu US pak přibližně 19 s. I tento test proběhl úspěšně a hra byla odehrána bez problémů.

[10] odesláno s nevalidním vstupem
[11] odesláno s hlavičkou Accept: application/json

4.4 Testování chybových stavů

Pro ověření správného chování platformy bylo třeba otestovat i speciální případy, které vedou k chybovým stavům. Zaměříme se na testování samotného průběhu hry, která je komunikována skrze WebSocket. Tyto testy byly prováděny manuálně zejména proto, že je složité automatickými testy simulovat chování sítě a ovlivňovat obousměrné spojení. Následuje seznam provedených testů s popisem průběhu.

Neexistující ID hry
Při připojení agenta k hernímu serveru nebylo v prvním případě specifikováno ID hry v parametru `gameId` a v druhém případě bylo poskytnuto takové ID, ke kterému neexistovala žádná instance hry. Tato situace může nastat poměrně často. Hráči musí ID hry specifikovat při připojení k serveru a protože toto ID musí sdílet navzájem mezi sebou, dává to prostor pro nechtěné chyby (použití starého ID hry, špatně zkopírovaný řetězec atp.). Server v obou zmíněných případech klienta odpojil s chybou, že pro zadané ID neexistuje žádná hra, ke které se lze připojit.

Odpojení klienta od serveru
Druhým testovaným případem bylo odpojení klienta od herního serveru. Tento případ byl simulován tak, že byl ukončen běžící proces AI agenta. Testovali jsme chování serveru pro dvě varianty – když se klient odpojil **během fáze připojování hráčů** ke hře a když došlo k odpojení **v průběhu hry**. V první variantě došlo k odpojení klienta, ale ke hře s daným ID se bylo stále možné připojit. V druhém případě nastal chybový stav, který vyústil k přerušení hry a odpojení druhého klienta.

Timeout na odeslání instrukcí tahu
V dalším testu jsme se zaměřili na situaci, kdy klient nestihne z nějakého důvodu odeslat herní instrukce o provedení tahu včas. To bylo nasimulováno zpožděním na straně jednoho z připojených klientů. Po uplynutí časového limitu došlo k přerušení hry a odpojení všech hráčů.

Pád herního serveru
V tomto testu jsme zkoumali reakci připojených klientů na pád herního serveru. To jsme simulovali ukončením běžícího procesu serveru. Protože jsou veškeré instance her drženy v paměti, dojde při pádu k jejich ztrátě. Zároveň dojde k přerušení komunikace s připojenými klienty přes WebSocket kanál. Toto spojení je zajištěno pomocí knihovny Socket.io, která se snaží klienty k serveru připojit vždy, kdy dojde k přerušení spojení. Po uplynutí časového limitu 20 s dojde k vyvolání chybové události `connection_error` a dojde k pokusu o navázání nového spojení.

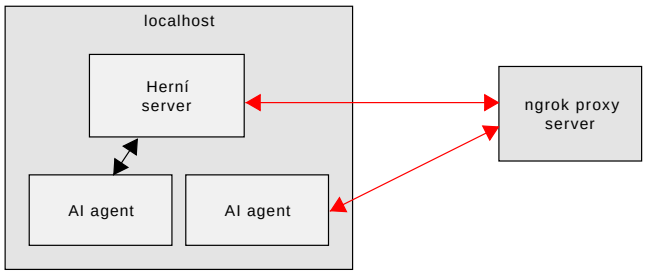
Pokud opět spustíme server, klient se k němu automaticky pokusí připojit. Je vyvolána událost `connect` – stejně jako v případě, kdy se hráč připojuje

k serveru poprvé. Při obnovení spojení je tedy opět poslán handshake požadavek se zadaným ID hry. Avšak protože po pádu serveru došlo k uvolnění paměti, není již v tento okamžik pro dané ID asociována žádná instance hry. Chybový stav je tedy následně vyřešen stejně jako v prvním testu neexistujícího ID hry, kdy dojde k odpojení klienta od serveru.

Přerušení síťového spojení

V rámci tohoto testu jsme zkoumali chování klientů v momentě, kdy dojde k přerušení spojení. Tedy situace, kdy server i klienti stále běží, avšak nastane chyba v síti. Pro simulování tohoto případu jsme využili nástroje ngrok. Přes něj byl zveřejněn herní server na internetu, ke kterému jsme poté připojili jednoho klienta. Druhý klient byl připojen v rámci localhost. Přerušení spojení jsme simulovali uzavřením přístupu počítače k internetu. Schéma spojení je znázorněno na obrázku 4.4.

Podobně jako v předchozím testu dojde k automatickému pokusu o obnovení spojení. Zde jsme však narazili na rozdíl, jestli k obnovení spojení dojde v průběhu 20s časového limitu nebo až po něm. V první variantě došlo k bezproblémovému obnovení a hra pokračovala dále. Oproti tomu v druhé variantě byla vyvolána událost `disconnect` a při opětovném navázání spojení došlo, podobně jako v předešlém testu, k vyvolání události `connect`, což následně vyústilo v chybový stav, kdy se klient snažil připojit k běžící hře, což server odmítl a klienta odpojil. Následkem toho došlo k přerušení hry a odpojení i druhého klienta.



Obrázek 4.4: Test přerušení spojení

Neplatné instrukce tahu

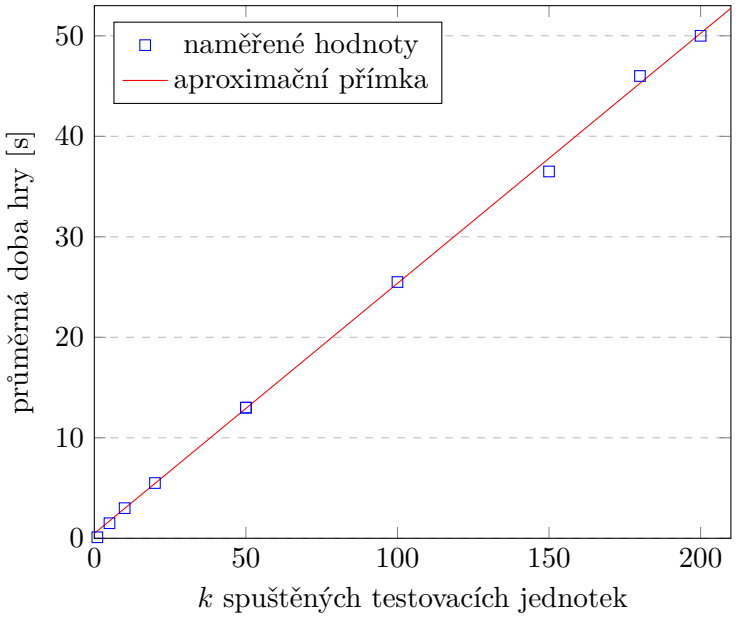
V posledním testu jsme zkoumali chování v situaci, kdy jeden z klientů odešle neplatné instrukce k provedení tahu. Byly odeslány dva vstupy. První obsahoval syntaktickou chybu. Druhý byl formátovaný správně, ale obsahoval neplatnou instrukci[12], kterou nelze zahrát. V obou případech došlo k odhalení nevalidního tahu, hra byla přerušena a klienti byli odpojeni.

[12] položení kamene na neexistující pozici $[-1, -1]$

Výše zmíněnými testy jsme ověřili chování platformy v momentech, kdy dochází k chybovým stavům. V uvedených případech došlo k vyřešení vzniklého problému skrze adekvátní reakci systému. Těmito testy jsme potvrdili funkčnost v problematických situacích.

4.5 Zátěžový test systému

Po ověření základní funkčnosti platformy jsme se dále zaměřili na otestování odezvy při velké zátěži. Protože herní server obsluhuje veškeré požadavky v rámci jediné event-loop, existuje předpoklad, že větší množství připojených klientů může zpomalit odbavování příchozích požadavků. V této sekci zkoumáme dopad na odezvu systému pod zátěží. Test probíhal na localhost.



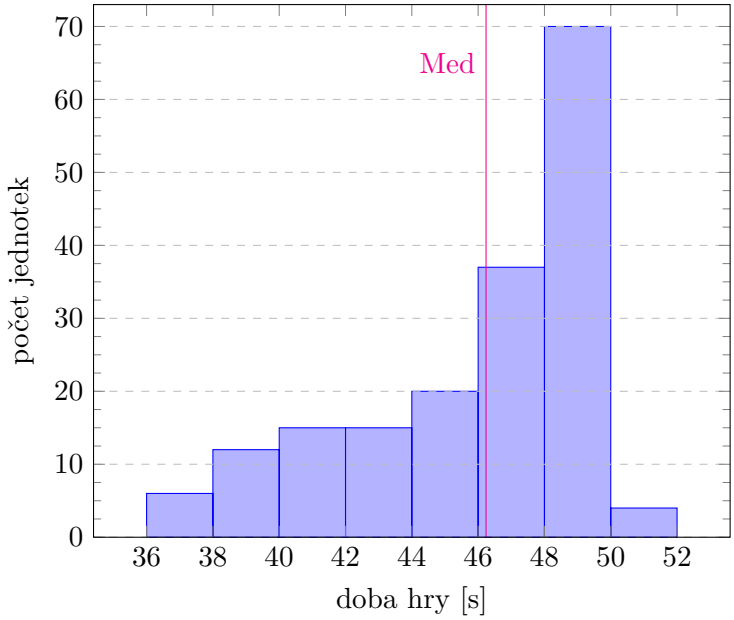
Obrázek 4.5: Průměrná doba odezvy v zátěžovém testu

Měřili jsme celkovou dobu odehrání jedné hry Othello při daném množství současně připojených klientů. Na jednu instanci hry připadá dvojice připojených klientů. Paralelně jsme spouštěli *k testovacích jednotek* najednou. Testovací jednotka se sestává z vytvoření instance přes REST API a následného připojení dvou klientů k dané hře. Postupně jsme zvyšovali počet k spuštěných jednotek. Průměrná doba hry při daném spuštění k jednotek je znázorněna na grafu 4.5.

Okolo hranice 200 testovacích jednotek se začaly objevovat chyby u < 1 % připojených klientů – nedošlo k vytvoření nové instance hry, a server tak odmítl připojit AI agenty. Tyto problémy byly pravděpodobně způsobeny vyčerpáním dostupné paměti. Z tohoto testu vyplývá, že bezpečné množství

současně spuštěných instancí se pohybuje okolo hranice **180 jednotek**, kdy všechny spuštěné hry doběhly v pořádku, avšak závisí na fyzických parametrech serveru.

Z grafu je také názorně vidět celkový dopad na odezvu propojených klientů. Odehrání tahu v testované hře Othello má při libovolném vstupu stále stejnou časovou složitost $\mathcal{O}(1)$. Avšak kvůli tomu, že veškeré požadavky jsou vyřizovány sériově v rámci jedné event-loop, se celková doba odezvy zvyšuje **u všech současně připojených klientů** rovnoměrně. Z následujícího histogramu 4.6 je toto zpoždění odezvy patrné. Naměřené hodnoty zároveň jednoznačně ukazují lineární závislost mezi počtem spuštěných jednotek a celkovou dobou běhu.



Obrázek 4.6: Histogram časové náročnosti při $k = 180$ spuštěných jednotek

Pro srovnání – jedna spuštěná instance zabrala celkově cca 0,8 sekund na dohrání. Při 180 spuštěných instancí se tedy průměrná celková doba nutná pro dokončení hry zvýšila přibližně 60krát. Jedna hra v průměru dosahuje 63 tahů, doba obsloužení jednoho tahu při velké zátěži se tedy pohybuje v intervalu < 1 s. Toto zpoždění se nachází v přiměřených mezích pro zachování plynulosti hry, a to i v případě, že do zpoždění započteme režii spojenou s komunikací po síti.

Z tohoto testu vyplývá, že platforma je použitelná i v případě většího počtu současně připojených klientů. Zátěž se rovnoměrně rozloží v důsledku zpracování všech požadavků v rámci jednoho vlákna. Toto vytváří spravedlivé prostředí pro obsluhu všech připojených klientů. Nicméně aby si platforma tuto vlastnost zachovala, je důležité, aby herní moduly zpracovávaly data co

nejefektivněji a nejúsporněji, aby nedocházelo ke zdržování vyřizování ostatních požadavků.

Závěr

V této práci jsme se věnovali tvorbě platformy pro soutěžení AI agentů v tahových strategických hrách. Řešení jsme, oproti existujícím platformám, vybudovali na klient–server architektuře. Agenti běžící na straně klienta se připojují k hernímu serveru, skrze který mezi sebou soupeří.

Zanalyzovali jsme různé způsoby komunikace mezi klientem a serverem. Zaměřili jsme se v první řadě na REST, skrze který jsme umožnili vytvářet nové instance her a získávat základní informace. V druhé řadě jsme zkoumali WebSocket jako způsob, kterým můžou klienti vyměňovat informace se serverem v reálném čase.

Na základě analýzy jsme navrhli strukturu platformy a implementovali její prototyp. Vytvořili jsme herní manager, který řídí veškerou komunikaci mezi agenty a herní instancí. Zdokumentovali jsme způsob, jakým psát herní moduly, a vytvořili testovací hru Othello.

Celou platformu jsme následně otestovali pomocí mock agentů. Testovali jsme základní funkčnost jak na localhost, tak v rámci internetu. Zaměřili jsme se také na testování chybových stavů a kontrolovali, zda-li server a klienti reagují správně i v problematických situacích.

Provedli jsme také zátěžový test, ve kterém jsme k serveru současně připojovali zvyšující se počet klientů. Hledali jsme limit, kdy je ještě možné obsluhovat příchozí požadavky. Z testu vyšlo najevo, že průměrná časová složitost na dohrání hry roste lineárně, avšak pouze do určitého bodu závislého na parametrech herního serveru. Ve chvíli, kdy dojde k vyčerpání dostupné paměti, přestává být server schopný efektivně obsluhovat požadavky.

Od určité hranice bude třeba zabývat se škálováním platformy, aby byla schopná obsluhovat větší množství současně připojených klientů. Práci lze do budoucna rozšířit tak, že zátěž bude rozložena na více serverů. Komunikaci bude nutné řídit skrze load balancer.

Bibliografie

1. THIELSCHER, Michael. General Game Playing in AI Research and Education. In: BACH, Joscha; EDELKAMP, Stefan (ed.). *KI 2011: Advances in Artificial Intelligence*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, s. 26–37. ISBN 978-3-642-24455-1.
2. MOLLE, P.; ÇATAL, O.; CONINCK, E.; VANDEWIELE, G. *Hexatron* [software]. 2010 [cit. 2019-04-29]. Dostupné z: `https://hexatron.ilabt.imec.be/`.
3. VANDEWIELE, Gilles. *A simple, but strong AI bot for Tron/Lightriders on a hexagonal grid.* [online]. Towards Data Science, 2019 [cit. 2019-04-20]. Dostupné z: `https://towardsdatascience.com/a-simple-but-strong-ai-bot-for-tron-lightriders-on-a-hexagonal-grid-2d7662ffcbf6`.
4. VANDEWIELE, G. *Creating a platform for AI competitions in Python* [online]. Towards Data Science, 2019 [cit. 2019-04-20]. Dostupné z: `https://towardsdatascience.com/creating-a-platform-for-ai-competitions-1184666f32e4`.
5. RIDDLES.IO. *The codesports platform* [software]. 2019 [cit. 2019-05-09]. Dostupné z: `https://www.riddles.io/`.
6. IP CHIVCHALOV. *Screeps: MMO RTS sandbox for programmers* [software]. 2019 [cit. 2019-05-09]. Dostupné z: `https://screeps.com/`.
7. SCREEPS. *Screeps Documentation: Global Objects* [online]. 2017 [cit. 2019-05-09]. Dostupné z: `https://docs.screeps.com/global-objects.html`.
8. CODINGAME. *CodinGame* [software]. 2019 [cit. 2019-04-30]. Dostupné z: `https://www.codingame.com/`.
9. CODECOMBAT. *CodeCombat* [software]. 2019 [cit. 2019-04-30]. Dostupné z: `https://codecombat.com/`.

10. CodeCombat Enhances Learn-to-Code Platform. *Entertainment Close - Up*. 2016.

11. BOZDAG, E.; MESBAH, A.; VAN DEURSEN, A. A Comparison of Push and Pull Techniques for AJAX. In: *2007 9th IEEE International Workshop on Web Site Evolution*. 2007, s. 15–22. ISSN 1550-4441. Dostupné z DOI: 10.1109/WSE.2007.4380239.

12. MCCABE, F.; BOOTH, D.; HAAS, H.; CHAMPION, M.; FERRIS, Ch.; ORCHARD, D.; NEWCOMER, E. *Web Services Architecture* [online]. 2004 [cit. 2019-04-20]. Dostupné z: https://www.w3.org/TR/ws-arch/. W3C Note. W3C. http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.

13. FIELDING, Roy Thomas. *Architectural Styles and the Design of Network-based Software Architectures*. 2000. Disertace. UNIVERSITY OF CALIFORNIA, IRVINE.

14. FACEBOOK INC. *GraphQL: A query language for APIs.* [software]. 2019 [cit. 2019-04-30]. Dostupné z: https://graphql.org/.

15. PRISMA. *How to GraphQL: GraphQL vs REST - A comparison* [online]. 2018 [cit. 2019-04-30]. Dostupné z: https://www.howtographql.com/basics/1-graphql-is-the-better-rest/.

16. LIN, B.; CHEN, Y.; CHEN, X.; YU, Y. Comparison between JSON and XML in Applications Based on AJAX. In: *2012 International Conference on Computer Science and Service System*. 2012, s. 1174–1177. Dostupné z DOI: 10.1109/CSSS.2012.297.

17. MAEDA, K. Performance evaluation of object serialization libraries in XML, JSON and binary formats. In: *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*. 2012, s. 177–182. Dostupné z DOI: 10.1109/DICTAP.2012.6215346.

18. GOYAL, G.; SINGH, K.; RAMKUMAR, K. R. A detailed analysis of data consistency concepts in data exchange formats (JSON amp; XML). In: *2017 International Conference on Computing, Communication and Automation (ICCCA)*. 2017, s. 72–77. Dostupné z DOI: 10.1109/CCAA.2017.8229774.

19. *Introducing JSON* [online] [cit. 2019-05-09]. Dostupné z: https://www.json.org/.

20. LORETO, S.; SAINT-ANDRE, P.; SALSANO, S.; WILKINS, G. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP* [online]. 2011 [cit. 2019-04-20]. Dostupné z: https://tools.ietf.org/html/rfc6202. RFC. IETF.

21. FETTE, I.; MELNIKOV, A. *The WebSocket Protocol* [online]. 2011 [cit. 2019-04-20]. Dostupné z: `https://tools.ietf.org/html/rfc6455`. RFC. IETF.

22. LENGSTORF, Jason; LEGGETTER, Phil. What Is Realtime? In: *Real-time Web Apps: With HTML5 WebSocket, PHP, and jQuery*. Berkeley, CA: Apress, 2013, s. 3–13. ISBN 978-1-4302-4621-3. Dostupné z DOI: `10.1007/978-1-4302-4621-3_1`.

23. NODE.JS FOUNDATION. *Node.js* [software]. 2019 [cit. 2019-04-30]. Dostupné z: `https://nodejs.org/`.

24. LEI, K.; MA, Y.; TAN, Z. Performance Comparison and Evaluation of Web Development Technologies in PHP, Python, and Node.js. In: *2014 IEEE 17th International Conference on Computational Science and Engineering*. 2014, s. 661–668. Dostupné z DOI: `10.1109/CSE.2014.142`.

25. MCCUNE, Robert Ryan. Node. js paradigms and benchmarks. *Striegel, Grad Os F*. 2011, roč. 11.

26. NODE.JS FOUNDATION. *The Node.js Event Loop, Timers, and process.nextTick()* [online]. 2019 [cit. 2019-05-11]. Dostupné z: `https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/`.

27. TILKOV, S.; VINOSKI, S. Node.js: Using JavaScript to Build High-Performance Network Programs. *IEEE Internet Computing*. 2010, roč. 14, č. 6, s. 80–83. ISSN 1089-7801. Dostupné z DOI: `10.1109/MIC.2010.145`.

28. NPM, INC. *npm: The Node Package Manager* [software] [cit. 2019-04-30]. Dostupné z: `https://www.npmjs.com/`.

29. AUTOMATTIC. *Socket.IO* [software]. 2019 [cit. 2019-05-12]. Dostupné z: `https://socket.io/`.

30. KOA CONTRIBUTORS. *Koa.js* [software]. 2019 [cit. 2019-05-12]. Dostupné z: `https://koajs.com/`.

31. PATRIK SIMEK AND CONTRIBUTORS. *vm2* [software]. 2019 [cit. 2019-05-14]. Dostupné z: `https://www.npmjs.com/package/vm2`.

32. NODE.JS FOUNDATION. *Don't Block the Event Loop (or the Worker Pool)* [online]. 2019 [cit. 2019-05-11]. Dostupné z: `https://nodejs.org/en/docs/guides/dont-block-the-event-loop/`.

33. FACEBOOK INC. *Jest: Delightful JavaScript Testing* [software]. 2019 [cit. 2019-05-10]. Dostupné z: `https://jestjs.io/`.

34. TJ HOLOWAYCHUK. *supertest* [software]. 2019 [cit. 2019-05-10]. Dostupné z: `https://www.npmjs.com/package/supertest`.

35. SHREVE, Alan. *ngrok: Public URLs for exposing your local web server.* [software] [cit. 2019-04-30]. Dostupné z: `https://ngrok.com/`.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

CRUD Create, Read, Update, Delete

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

NPM Node.js Package Manager

REST Representational state transfer

JS JavaScript

URI Uniform Resource Identifier

URL Uniform Resource Locator

XML Extensible markup language

Příloha B

Metody herního modulu

V této příloze je přehledně rozepsán popis jednotlivých metod rozhraní herního modulu, které musí tvůrce hry dodržet. Tyto metody slouží pro komunikaci s herním managerem, který následně koordinuje předávání informací mezi hrou a připojenými AI agenty. Aby mohla tato komunikace správně probíhat, je nutné aby autor hry dodržet rozhraní popsané v kapitole 2. Následuje stručný popis jednotlivých metod, které herní modul musí implementovat.

`constructor(initParams) : void`
: Kontruktor je volán při vytváření nové instance hry (po zavolání POST požadavku na REST API). Skrze parametr `initParams` lze modulu předat informace potřebné k inicializaci herního objektu.

`connectPlayer(playerSymbol) : void`
: Informuje herní modul o připojení nového hráče ke hře. Hráč je reprezentovaný `Symbol` objektem, který je předaný skrze vstupní parametr metody. Tento symbol si musí herní modul uložit pro pozdější použití.

`disconnectPlayer(playerSymbol) : void`
: Informuje herní modul, že došlo k odpojení hráče, reprezentovaného symbolem `playerSymbol`. Herní modul již s tímto hráčem nesmí dále počítat.

`canStartGame() : bool`
: Vrací `true`, pokud lze začít hru, resp. pokud jsou všichni hráči připojení. Volá se pokaždé, když dojde k připojení nového klienta k hernímu serveru.

`startGame() : void`
: Metoda je zavolána ve chvíli, kdy metoda `canStartGame` vrátila hodnotu `true`. Uvnitř této metody může herní modul provést základní nastavení herního stavu před začátkem hry. Zároveň musí určit, který hráč je na řadě.

`getCurrentPlayerSymbols() : Symbol`
: Metoda vrací `Symbol` (nebo pole Symbol) hráčů, kteří jsou aktuálně na řadě. Metoda je volána v průběhu hry až od chvíle, kdy dojde ke spuštění hry přes metodu `startGame`.

`getStateFor(playerSymbol) : Object`
: Metoda, která pro zadaného hráče `playerSymbol` vrací příslušný objekt s herním stavem. Herní stav je následně poslán hráči, který na základě něj vygeneruje instrukce k tahu.

`isTurnInvalid(playerSymbol, turnInstructions) : bool`
: Zkontroluje, zda-li instrukce `turnInstructions`, které byly poslané hráčem `playerSymbol`, jsou validní. Pokud lze tyto instrukce aplikovat, metoda vrací `false`. V opačném případě `true`.

`playTurn(playerSymbol, turnInstructions) : void`
: Aplikuje instrukce a změní herní stav. Zároveň musí aktualizovat informaci o aktuálním hráči na řadě. Po zahrání tahu musí rozhodnout, zda-li došlo k ukončení hry nebo ne. Pokud ano, je potřeba tuto informaci uložit do herního stavu a předat ji v následujícím volání metody `isGameOver`.

`isGameOver() : bool`
: Volá se ihned po předchozím volání metody `playTurn`. Vrací `true`, pokud došlo k ukončení hry.

`getScore() : Object`
: Pokud došlo k ukončení hry (metoda `isGameOver` vrátila `true`), herní manager zavolá tuto metodu. Ta musí vrátit objekt s informací o výsledku hry, který je následně předán všem připojeným hráčům. Jedná se o poslední metodu, která se v průběhu hry volá na instanci herního modulu.

Příloha C

Obsah přiloženého CD

/
- `readme.txt` ..... stručný popis obsahu CD
- `ukazka.mkv` ..... záznam základního testu funkčnosti
- `src`
  - `impl` ..... zdrojové kódy implementace
    - `client` ..... klientská část
    - `server` ..... herní server
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF