Mendelova univerzita v Brně
Provozně ekonomická fakulta

Návrh a implementace řídicího systému pro exponát Kulatý stůl ve VIDA! science centru

Diplomová práce

Vedoucí práce:
Mgr. Tomáš Foltýnek, Ph.D.

Autor:
Bc. David Savič

Brno 2017

Rád bych poděkoval vedoucímu mé práce Mgr. Tomáši Foltýnkovi, Ph.D. za cenné rady a připomínky, VIDÁtorům Bc. Oliverovi Velichovi a Mgr. Martině Nekolové, a dále rodině, přátelům a všem, kteří mě při tvorbě práce podporovali.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Návrh a implementace řídicího systému pro exponát Kulatý stůl ve VIDA! science centru** vypracoval/a samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*

Jsem si vědom/a, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.

Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

V Brně dne 22. května 2017 ____________________________

Abstract

Bc. David Savič. Design and implementation of control system for exhibit Round Table in VIDA! Science Centre. Diploma thesis. Brno: Mendel University, 2017.
In the text is described implementation of control system for launching multiplayer games and their administration on the exhibit in VIDA! Science Centre, which consists of 12 touch screens, central computer and central projector.

Keywords

Control system, Client-Server, projector, multiplayer games, Java, JavaFX.

Abstrakt

Bc. David Savič. Návrh a implementace řídicího systému pro exponát Kulatý stůl ve VIDA! science centru. Diplomová práce. Brno: Mendelova univerzita v Brně, 2017. V textu je popsaná implementace řídicího systému, zajišťující možnost spouštění a správy her pro více hráčů na exponátu ve VIDA! science centru, který je tvořen 12 dotykovými obrazovkami, centrálním počítačem a středovým projektorem.

Klíčová slova

Řídicí systém, klient-server, projektor, hry pro více hráčů, Java, JavaFX.

Obsah

Obsah 9

1 Úvod a cíl práce 12

1.1 Úvod 12

1.2 Cíl práce 12

2 Metodika 13

2.1 Prototypový přístup 13

2.2 Modularita 14

2.3 Postup práce 14

2.4 Výběr programovacího jazyka 15

3 Teoretický základ 16

3.1 Programovací jazyk Java 16

3.1.1 Novinky v Java 8 16

3.1.2 Grafické uživatelské rozhraní 17

3.1.3 Platforma JavaFX 17

3.1.4 Dotykové ovládání v JavaFX 18

3.2 Další použité technologie 18

3.2.1 Apache Maven 18

3.2.2 Git 18

3.2.3 IntelliJ IDEA 19

3.2.4 MVC 19

3.3 Síťová komunikace 19

4 Analýza současného stavu 21

4.1 Exponát Kulatý stůl 21

4.2 Současný stav exponátu 21

4.3 Požadavky 21

4.3.1 Řídicí systém a klientská aplikace 21

4.3.2 Hry exponátu Kulatý stůl 23

4.3.3 Výběr hry ... 24

5 Návrh řešení 25

5.1 Návrh klientské aplikace řídicího systému ... 25

5.2 Návrh serverové aplikace řídicího systému ... 30

5.3 Návrh hry SnakeXSnake ... 30

5.4 Tvorba grafiky ... 30

5.5 Zprávy posílané mezi klientem a serverem ... 31

5.6 Dotykové ovládání ... 33

5.7 Promítání ... 33

5.8 Logování ... 34

6 Implementace 36

6.1 Struktura celého projektu ... 36

6.1.1 Adresář resources ... 37

6.2 Řešení síťové komunikace ... 38

6.3 Rozdělení do balíků ... 40

6.4 Popis modulu VIDAControlSystem_common ... 40

6.4.1 Třída ColorHandler ... 40

6.4.2 Třída FileHandler ... 40

6.4.3 Třída Game ... 40

6.4.4 Třída GameLoader ... 41

6.4.5 Třída GameProcess ... 41

6.4.6 Třída IPAddressConfig ... 41

6.4.7 Třída SimpleModalWindow ... 41

6.4.8 Třída TransitionFactory ... 41

6.5 Popis modulu VIDAControlSystem_client ... 42

6.5.1 Třída Client ... 42

6.5.2 Třída GameClient ... 44

6.6 Popis modulu VIDAControlSystem_server ... 44

6.6.1 Třída ClientHandler ... 44

6.6.2 Třída ConsoleLogger 45
6.6.3 Třída GameServer 45
6.6.4 Třída ProjectorWindowEffects 45
6.6.5 Třída Server 45
6.7 Popis modulu VIDAControlSystem_snake_common 46
6.7.1 Třída Snake 46
6.7.2 Třída SnakePart 47
6.8 Popis modulu VIDAControlSystem_snake_client 47
6.9 Popis modulu VIDAControlSystem_snake_server 47
6.10 Popis modulu VIDAControlSystem_utils 47
6.10.1 Třída LogUtils 47

7 Nasazení řídicího systému 49
7.1 Připomínky a jejich řešení 49

8 Diskuze 52
8.1 Dosažené výsledky a možná rozšíření 52
8.2 Možná využití projektoru 54

9 Závěr 56

10 Literatura 57

A Struktura přiloženého CD 60

B Snímky obrazovky ŘS a klientské aplikace 61

C Ukázka zdrojových kódu vybraných tříd 79

1 Úvod a cíl práce

1.1 Úvod

VIDA! science centrum je zábavní vědecký park, který slouží pro podporu vědy v Brně. Pro návštěvníky, kteří sem chodí především za poznáním, rozvojem osobnosti a zábavou, se zde nachází přes 170 interaktivních exponátů. Návštěvník si zde může vyzkoušet například exponáty simulace tornáda, oceán v lahvi, simulátor zemětřesení, moderování počasí před kamerou, zmražení vlastního stínu a mnoho dalších.

Většina exponátů by však nemohla fungovat bez softwaru na míru, ať už na nižší úrovni, například v podobě automatizace mechanických částí, nebo na úrovni vyšší v podobě grafického rozhraní na obrazovkách a projekcích. Díky spolupráci s Provozně ekonomickou fakultou Mendelovi univerzity v Brně mají studenti možnost podílet se na vývoji exponátů Kulatý stůl, Planetary Control Room a Time Room, které většinou využívají několik obrazovek a projektorů. Náplní této práce bude vývoj exponátu Kulatý stůl.

1.2 Cíl práce

Cílem této práce bude navrhnout a implementovat potřebné aplikace pro chod exponátu Kulatý stůl. Celé řešení by mělo umožňovat spouštění her (popř. jiných typů aplikací) na koncových stanicích (dále jen klienti), a jejich správu a řízení na centrálním počítači (dále jen server). Součástí řešení má být:

- Grafické uživatelské rozhraní pro dotykové obrazovky (dále jen klientská aplikace), pomocí kterého má uživatel (návštěvník VIDA! science centra) možnost vybrat hru, spustit ji a zahrát si spuštěnou hru proti ostatním hráčům.
- Řídicí systém (serverová aplikace, dále jen ŘS), který zajistí správu a řízení her přes centrální počítač. ŘS bude řídit klientské aplikace, starat se o spouštění her při splnění potřebného množství hráčů, umožňovat běh více her zároveň (na každé dotykové obrazovce klienta však pouze jedna), a bude využívat projektor.
- Vytvoření jedné hry, která bude v nabídce her hotového systému a bude využívat projektoru.

2 Metodika

Na vývoji exponátu Kulatý stůl se se mnou podílel ještě jeden student, a to pan Ing. Jakub Drobný (dále jen kolega Drobný) v rámci své diplomové práce. Cílem jeho práce bylo navrhnout a implementovat výukovou hru Distribuce limonád založenou na využití mentálních modelů v oblasti řízení dodavatelského řetězce. Tato hra je určena pro 4 hráče a je přidána do výsledného ŘS. Kolega Drobný odevzdával svoji práci dříve. A protože byl vývoj jeho hry závislý na ŘS, bylo potřeba již na začátku specifikovat dosud známé požadavky na ŘS, a vytvořit princip fungování a seznam požadavků, které musí splňovat hra, aby byla kompatibilní s ŘS.

Ideálním případem bylo mít hotovou předběžnou verzi ŘS, s omezenou funkcionalitou, pro testování společně s hrou kolegy Drobného. Z tohoto důvodu by zvolen prototypový přístup vývoje. Tedy kvůli potřebě průběžného testování předběžných verzí systému, s nekompletními požadavky, které se navíc přidávali a měnili v průběhu vývoje.

Protože celkový projekt (ŘS a hra) je složen z mnoha oddělených částí (aplikací), bylo potřeba vhodně rozdělit jeho strukturu. Proto jsem zvolil koncept rozdělení aplikací do jednotlivých modulů.

2.1 Prototypový přístup

Prototypový přístup vývoje, je model založený na iterativním vytváření fungujících modelů softwaru s omezenou funkcionalitou. Tento přístup se používá v případě, že nejsou dány detailní informace k vstupním a výstupním požadavkům. Předpokládá se, že ne všechny požadavky jsou známy od začátku vývoje. Tento model umožňuje uživatelům vyzkoušet si pracovní model systému, kterému se říká prototyp. Během kterékoliv fáze se může uživatel rozhodnout, že chce udělat změny v systému. (Thakur, 2017)

Prototyp je vytvářen následujícími způsoby:

- vytvořením hlavního uživatelského rozhraní bez nějakého hlubšího programování, aby si mohl uživatel vyzkoušet, jak bude systém vypadat,
- vytvořením zjednodušené verze systému, který má omezenou funkcionalitu,
- vytvářením systémových komponent pro ilustraci funkcionality, která bude zahrnuta ve vyvíjeném systému. (Thakur, 2017)

Prototyp je vyvíjen na základně aktuálně známých požadavků. Vývoj samozřejmě zahrnuje návrh, implementaci a testování, ale každá z těchto fází je řešena pouze okrajově a nezachází se do detailů. Při vytvoření každého prototypu, má koncový uživatel a zákazník možnost prototyp vyzkoušet. Vývojáři je následně poskytnuta zpětná vazba: co je správně, co je potřeba upravit, co chybí, co není potřeba atd. Na základě zpětné vazby je prototyp upraven tak, aby odpovídal některým navrženým požadav-

kům a následně opět vyzkoušen koncovým zákazníkem. Tento proces se opakuje tak dlouho, dokud není systém kompletní a zákazník s ním je spokojený. (Thakur, 2017)

2.2 Modularita

Modularita je obecným konceptem, použitelným při vývoji softwaru. Tento koncept umožňuje vývoj jednotlivých modulů, s většinou stanoveným rozhraním, které slouží pro komunikaci mezi moduly. Jde o podobné rozdělení jako mezi objekty v objektově orientovaném programování, jen ve formě modulů, na vyšší úrovni abstrakce. Rozdělení systému do modulů pomáhá snižovat nadměrnou provázanost (která vede k nízké soudržnosti) a vede k snáze udržovatelnému kódu. (Blewitt, 2009)

Při návrhu Javy nebylo počítáno s moduly (umožňuje pouze rozdělení do balíků). Kterákoliv knihovna Javy je ve výsledku obdobou modulu, od knihovny *Log4J* přes *Tomcat* až po *Hibernate*. Knihovny mají tedy stejný efekt jako moduly. Sice nemusí mít jednotné rozhraní pro komunikaci s nimi, ale většinou mají veřejné aplikační rozhraní (dané klíčovým slůvkem *public*) a mohou mezi sebou mít závislosti. (Blewitt, 2009)

Modularita se hodí v případě, že chceme aplikaci (nebo celý projekt) rozdělit do logických částí, které mohou být vyvíjeny a testovány odděleně. Aby se knihovna dala použít jako modul, je potřeba do ní zabalit informaci, kterou je možno použít pro závislosti mezi knihovnami, a to nástroji pro sestavování aplikací, jako je například *Apache Maven* (informace o knihovně je pak obsažená v konfiguraci v souboru *pom.xml*). (Blewitt, 2009)

2.3 Postup práce

Na začátku bylo potřeba specifikovat veškeré známé požadavky, které se od ŘS očekávají. Bylo potřeba specifikovat, jakým způsobem by měl ŘS fungovat a co se očekává od každé hry, která bude součástí ŘS. Tedy požadavky, které hra musí splňovat, aby byla kompatibilní s ŘS. Následně jsem nastudoval danou problematiku s důrazem na síťovou komunikaci a práci s dotykovou obrazovkou v jazyce Java. Dále jsem se seznámil s exponátem Kulatý stůl a shromáždil nově získané požadavky. Na základě všech známých požadavků jsem vytvořil první prototyp ŘS. První prototyp byl otestovaný na exponátu, včetně otestování kompatibility s hrou kolegy Drobného. Po prvním testování jsem shromáždil nově získané požadavky, připomínky a nalezené problémy. Následovalo opakování fází návrhu, implementace, testování a aktualizace požadavků. Stejným cyklem probíhal vývoj hry. Cyklus vývoje se dále opakoval s testováním v ostrém provozu. Vše se opakovalo tak dlouho, dokud nebyly splněny veškeré požadavky na ŘS, hru a opraveny veškeré známé problémy. Nakonec byly shromážděny veškeré připomínky a jejich řešení, a celkové řešení bylo zhodnoceno.

2.4 Výběr programovacího jazyka

Již při vytváření zadání této diplomové práce jsem pro vývoj ŘS zvolil programovací jazyk Java. Vybral jsem si jej především kvůli mým dlouholetým zkušenostem, a protože bohatě poskytuje vše podstatné, co je pro vývoj tohoto ŘS potřeba. Vhodných programovacích jazyků je však více.

Pro vývoj tohoto ŘS jsem zvolil požadavky pro vhodný programovací jazyk. Zvolený programovací jazyk by měl mít:

- rozšířenost a popularitu, kvůli možnosti budoucího rozšiřování ŘS jiným vývojářem,
- možnost spouštět aplikace příkazem s parametry na operačním systému Windows,
- pokročilé uživatelské rozhraní,
- možnost dotykového ovládání a
- možnost síťové komunikace.

3 Teoretický základ

V této kapitole jsou popsány použité technologie a přístupy k vývoji software.

3.1 Programovací jazyk Java

Java byla roku 1991 vytvořena firmou Sun Microsystems. Jazyk byl původně pojmenovaný jako *Oak* (dub), ale byl přejmenovaný na Java v roce 1995. Překvapivé, v té době bylo, že impulzem k vytvoření Javy nebyly webové aplikace. Místo toho bylo primárním cílem vytvořit platformově nezávislý jazyk, který je možno použít pro vytvoření softwaru pro širokou škálu zařízení koncových uživatelů. Později se však druhým a nakonec důležitějším faktorem stalo programování na World Wide Webu. Nakonec tedy právě Internet vedl k velkému úspěchu a rozšíření Javy. (Schildt, 2014)

Pro zjednodušení webového vývoje byl pro Javu, vedle standardní aplikace, vynalezen nový typ programu, nazývaný *applet*. Applet je přenášený přes Internet, zaveden do paměti a automaticky spuštěný standardním prohlížečem, který je kompatibilní s Javou. (Schildt, 2014)

Java má blízko k programovacím jazykům *C* a *C++*. Syntaxe Javy vychází z jazyka C a objektový model je převzat z C++. U Javy je klíčem k přenositelnosti, že forma výstupu z překladače není spustitelný kód, ale *bajtkód* (*bytecode*). Bajtkód je vysoce optimalizovaný seznam instrukcí, které spouští (interpretuje) virtuální stroje *Java Virtual Machine* (*JVM*). JVM tedy řeší nejen přenositelnost programu, ale i jeho bezpečnost, protože JVM kód kontroluje a může tak zabránit vytvoření vedlejších efektů na systému vně aplikace. (Schildt, 2014)

Java byla navržena jako interpretovaný jazyk, jejichž problémem je jejich pomalost ve srovnání s kompilovanými jazyky. Překlad bajtkódu do strojového jazyka (spustitelného souboru) počítače je řešen za běhu a optimalizaci zde řeší *JIT kompilátor* (*Just In Time – JVM*). JIT compilátor je součástí JVM. (Schildt, 2014)

3.1.1 Novinky v Java 8

Nejnovější verze Java 8 přináší řadu novinek. Největší novinkou v Javě 8 jsou určitě lambda výrazy. Zde je výčet těch nejzajímavějších novinek:

- Lambda výrazy – jde v podstatě o zkrácený zápis funkce, která pro dané vstupní hodnoty určuje výstupní hodnotu. Lze tak tedy předat funkci v parametru metody.
- Odkazy na metody a konstruktory – jde o odkazy, které jsou konvertovány na funkce, přes zápis pomocí dvou dvojteček `::`.
- Výchozí metody rozhraní – díky nim, může mít nově rozhraní neabstraktní metody. Jde o metody, které obsahují kromě hlavičky i implementaci. Označují se klíčovým slovem *default*.

- *Stream API* – jde o třídy v novém balíku *java.util.stream*, které se používají s kolekcemi. Základní metodou je zde metoda *stream*, která zřetězí prvky dané kolekce a nad kterými lze provádět funkcionální operace. (What's New in JDK 8, 2016)

Další verze, Java 9 má vyjít v červenci 2017, avšak datum vydání se neustále oddaluje, protože původním datem vydání bylo září 2016, pak březen 2017 a nakonec se vydání posunulo o 4 měsíce. (Sharwood, 2016)

3.1.2 Grafické uživatelské rozhraní

Java podporovala vývoj grafického uživatelského rozhraní již od verze 1.0, a to za použití *AWT* (*Abstract Window Toolkit*), knihovny která byla představena v roce 1996. Na knihovně AWT bylo v té době nejpokrokovější, že umožnovala programátorovi napsat aplikaci, která bude vypadat vzhledově stejně, jak na operačním systému Windows od Microsoftu, tak na systémech UNIX. (Ebbers, 2014)

Později byla knihovna AWT nahrazena knihovnou *Swing*, která byla uživatelsky přívětivější, ale stále jí chyběla možnost vytvářet moderně vypadající widgety, produktivní vývoj, efektivního renderování grafického rozhraní, data bindingu, snadno použitelné 2D a 3D knihovny, a podpory stylů. (Sharan, 2015)

V roce 2008 byla poprvé vydána platforma JavaFX, jako nástroj pro vytváření Bohatých internetových aplikací (*Rich Internet Application*) psaných v deklarativním jazyce *JavaFX Script*. Více pozornosti si však získala s verzí JavaFX 2.0, vydanou v roce 2011, která umožnila vývojářům psát JavaFX programy v programovacím jazyce Java. Od verze JavaFX 8 je JavaFX zahrnuta v Javě v prostředí Java Runtime Environment (JRE). Výhodou JavaFX je podpora dotykových a vícedotykových gest, která ve Swingu chybí. Pro implementaci grafického uživatelského rozhraní jsem si zvolil právě platformu JavaFX. (Sharan, 2015)

3.1.3 Platforma JavaFX

JavaFX je platforma založená na Javě, která slouží pro vývoj bohatých klientských aplikací. Je porovnatelná s ostatními platformami jako *Adobe Flex* a *Microsoft Silverlight*, a je považována za nástupce knihovny Swing v tvorbě grafického uživatelského rozhraní. Protože je napsána v jazyce Java, tak lze využívat všech výhod Javy, jako jsou vlákna, lambda výrazy či generika, a lze ji použít v libovolném editoru (třeba NetBeans). (Sharan, 2015)

JavaFX nabízí dvě možnosti jak psát uživatelské rozhraní – buď za použití kódu v Javě nebo za použití *FXML* pro deklarativní zápis. FXML je založeno na značkovacím jazyce *XML*. Oracle navíc poskytuje nástroj pro návrh rozhraní *Scene Builder*, který generuje kód v FXML. JavaFX je postavena na architektuře MVC (viz kapitola *MVC*). (Sharan, 2015)

3.1.4 Dotykové ovládání v JavaFX

Jak jsem již zmínil, JavaFX podporuje dotyková a vícedotyková gesta na zařízeních s dotykovým ovládáním. Při dotyku obrazovky je vygenerována událost dotyku, a navíc je vygenerována příslušná událost myši. Pokud tedy uživatel například klepne prstem na tlačítko, je kromě události dotyku vygenerována stejná událost jako kdyby na něj kliknul myší. Každá akce je dána bodem, ve kterém došlo ke kontaktu, tzv. *bod dotyku* (*touch point*). Dotykové obrazovky je možné se dotknout více prsty – taková událost obsahuje více bodů dotyku. Každý stav bodu dotyku, například zmáčknuto (*pressed*), uvolněno (*released*), a tak dále, generuje událost dotyku. Bod dotyku určuje cíl této události. Tedy, ten grafický prvek, na který bylo klepnuto, je v události veden jako cíl. (Sharan, 2015)

Uživatelé mají možnost interagovat s JavaFX aplikací použitím gest. Gesto je složeno z více bodů dotyku a akcí dotyku. Gestem může být například otáčení, skrolování, posun (*swipe*) a přiblížení (*zoom*). Gesto otáčení je provedeno vzájemným otočením dvou prstů kolem sebe. Gesto skrolování je provedeno posouváním prstu po obrazovce. Gesto posunu je provedeno posunem prstu (nebo více prstů) po obrazovce v jednom směru. A gesto přibližování je určeno pro přiblížení grafického prvku, přiblížením nebo oddálením dvou prstů. (Sharan, 2015)

3.2 Další použité technologie

3.2.1 Apache Maven

Maven je velice populární nástroj pro správu a sestavování aplikací postavených nad platformou Java. Maven používá deklarativní přístup, kterým popisuje strukturu a obsah projektu. Díky Mavenu odpadá závislost na vývojovém prostředí, protože veškeré informace potřebné ke kompilaci a sestavení programu jsou přímo obsaženy ve speciálním souboru *pom.xml* (*POM* - project object model). Je součástí všech velkých vývojových prostředí (*Netbeans*, *Eclipse*, *IDEA*), které usnadňují práci s Mavenem, a posyktují pro jeho ovládání i uživatelské rozhraní. (Horděјčuk, 2017)

Popis projektu je obsažený v souboru *pom.xml* a nachází se v kořenovém adresáři daného projektu. Tento soubor obsahuje detailní popis projektu, informaci o jeho verzi, konfiguraci, závislosti, zdroje pro aplikaci a její testování, strukturu a mnoho dalšího. (Smart, 2005)

3.2.2 Git

První verze *Gitu* byla vydána v roce 2005 a autorem je Linus Torvalds, který je jedním z hlavních vývojářů Linux kernelu. Git je systém pro správu verzí. Zaznamenává změny souborů v průběhu času a uživatel má tak možnost kdykoliv obnovit kteroukoliv verzi. Dává nám tedy přístup k celé historii projektu a poskytuje přehled o tom, kdo vytvořil jakou část kódu a jaké jsou změny v každé verzi. Díky větvení stromu historie

lze vytvářet novou funkcionalitu, aniž by to mělo dopad na zbytek projektu. Hodí se pro správu větších projektů a spolupráci na vývoji v týmu. Je většinou integrovaný do vývojových prostředí a dále existuje řada nástrojů, které pro jeho ovládání poskytují grafické rozhraní, například *SourceTree*. (Valkovič, 2014)

3.2.3 IntelliJ IDEA

IntelliJ IDEA je produkt od firmy *JetBrains* se sídlem v Praze. Jde o inteligentní vývojové prostředí (*IDE*) pro vývoj především v Javě a zaměřuje se hlavně na produktivitu vývojářů. Je dodávána ve dvou verzích, *Community Edition*, která je zdarma, a *Ultimate Edition*, která je za peníze. (Kavalec, 2011)

Klíčové vlastnosti IDEA jsou: inteligentní doplňování kódu, chytrá navigace a vyhledávání, řada různých refaktoringů, které pomáhají zvýšit kvalitu kódu, analýzu kódu, pokročilé ladění kódu, podporu webových a enterprise aplikací, Unit testování, analýza pokrytí kódu testy a mnoho dalšího. (About IntelliJ IDEA, 2017)

3.2.4 MVC

Model-View-Controller (*MVC*) je velice oblíbený architektonický vzor (někdy také označovaný jako návrhový vzor), který rozděluje datový model aplikace, uživatelské rozhraní a řídící logiku na tři oddělené komponenty (*Model*, *View*, *Controller*). Komponenta *Model* obsahuje doménové třídy, které reprezentují skutečné objekty, se kterými se pracuje, převedené do kódu. Komponenta *View* a *Controller* obsahuje prezentační objekty, které se zabývají se vstupem, výstupem, uživatelskými událostmi a elementy grafického rozhraní. Komponenta Controller přijímá vstupy od uživatele a rozhoduje, jak s nimi naložit. Komponenta View zobrazuje výstup na obrazovku. Tyto dvě komponenty jsou tedy úzce provázaný a navzájem se ovlivňují. Komponenta Controller mění stav komponenty Model. (Sharan, 2015)

3.3 Síťová komunikace

Architektura TCP/IP je na rozdíl od referenčního modelu OSI (*Open System Interconnection*), který má sedm vrstev, rozčleněna do čtyř vrstev:

- aplikační vrstvu (application layer),
- transportní vrstvu (transport layer),
- síťovou vrstvu (internet layer) a
- vrstvu síťového rozhraní (network interface). (Brookshear, 2013)

Rodina protokolů TCP/IP je sada protokolů pro komunikaci v počítačové síti a je hlavním protokolem celosvětové sítě Internet. Ve skutečnosti nejsou TCP (*Transmission Control Protocol*) a IP (*Internet Protocol*), které jsou v názvu, jediné protokoly z této sady. Síťová vrstva má na starosti přesnost datových paketů přes

případné mezilehlé uzly až k cílovému uzlu. Velkou roli zde hraje směrování (routing), který zajišťuje směr přenosu až ke svému cíli. V TCP/IP má směrování na starosti protokol IP. Ten funguje jako nespojovaný, což v praxi znamená, že přenáší jednotlivé pakety (tzv. *datagramy*) nezávisle na sobě. Protokol TCP definuje jednu z možností transportní vrstvy. Je jednou z možností, protože sada protokolů TCP/IP poskytuje více než jeden způsob implementace transportní vrstvy. Jeden z nich je definovaný protokolem UDP (*User Datagram Protocol*). Transportní vrstva TCP/IP obsahuje pouze tyto dva protokoly, mezi kterými jsou značné rozdíly. Podle typu služby se jednotka aplikační vrstvy může rozhodnout, jestli posílat data přes protokol TCP nebo UDP. (Brookshear, 2013)

Jedním z rozdílů mezi protokoly TCP a UDP je, že TCP je spojovaný a UDP je nespojovaný. U TCP to v praxi znamená, že před posláním zprávy, kterou si vyžádala aplikační vrstva, pošle transportní vrstva zprávu cíli s informací o tom, že bude poslána zpráva. Následně počká na potvrzení této zprávy, před tím než ji pošle. TCP tedy nejdříve naváže spojení, zatímco transportní vrstva postavená na UDP takovéto spojení nenaváže, pouze pošle zprávu na určenou adresu a nic dalšího ji nezajímá. Může se tedy stát, že cílový uzel ani není aktivní. (Brookshear, 2013)

Dalším rozdílem mezi TCP a UDP je spolehlivost doručení a formát v jakém jsou předávána data. Protokol TCP vytváří aplikacím ke vzájemné komunikaci spolehlivé obousměrné spojení na způsob roury. Jde o spojitý přenosový proud, který přenáší jednotlivé byty. Do této, v zásadě přenosové roury, aplikace z jedné strany postupně vkládá jednotlivé byty a druhá aplikace si je na druhé straně byte po bytu odebírá. Představuje tak proudový přenos – ten je tvořen řadou datových segmentů. Protokol UDP se drží základní představy blokového přenosu, protože si předává data mezi aplikacemi již v nařezané formě, na bloky zabalené do datagramů. Jde o přenos jednotlivých paketů, na rozdíl od přenosu datového proudu, jak je tomu u protokolu TCP. Spolehlivost je u TCP dále dána tím, že opakovaně posílá ztracené nebo poškozené pakety a zajišťuje jejich správné pořadí, na rozdíl od UDP. (Brookshear, 2013)

Důležitým úkolem transportní vrstvy je rozlišovat mezi různými odesílateli a příjemci v rámci každého uzlu. Datové pakety (IP datagramy), jsou určeny vždy pouze přijímacímu uzlu. V rámci uzlu je však potřeba rozlišovat celou řadu možných příjemců (a odesílatelů), protože na jednom uzlu může běžet více aplikací, které komunikují po síti, aby bylo zajištěno, že každá aplikace dostane svá data. Také může na jednom uzlu navíc pracovat více uživatelů. Transportní vrstva řeší problém tím, že k absolutní adrese (síťové adrese reprezentující uzel – zde IP adresa), přidává ještě „relativní“ adresu, tzv. číslo portu. Port je jednoznačně daný svým číslem (číslem portu), a je přiřazený k příslušné entitě aplikační vrstvy (aplikačnímu procesu, systémové úloze apod.). (Peterka, 1999)

4 Analýza současného stavu

Tato kapitola analyzuje stav, ve kterém byl exponát před vývojem tohoto ŘS. Přibližuje podobu exponátu a vstupní požadavky, ze kterých vycházel počáteční návrh.

4.1 Exponát Kulatý stůl

Exponát Kulatý stůl, jak již název napovídá, je velký kulatý stůl, po jehož obvodu je rozmístěno 12 dotykových obrazovek, a shora je na jeho střed promítán kulatý obraz. Přes dotykové obrazovky mají návštěvníci možnost výběru různých interaktivních her pro více hráčů, které mohou využívat promítaného obrazu jako sdílené plochy. Tato plocha se dá využít například pro rozšíření herní plochy nebo zobrazování informací z průběhu her.

Protože her bude na výběr několik, může jich být spuštěno více naráz a ke každé hře se bude moci připojit několik hráčů, je potřeba chod systému nějakým způsobem řídit a spravovat. K tomu má sloužit právě řídicí systém.

4.2 Současný stav exponátu

Exponát se momentálně používá – v minulosti pro něj již byl vytvořen systém na míru, který však umožňuje spustit pouze jednu hru. Jde o hru s názvem Rybolov, jejímž cílem každého hráče je vylovit maximální množství ryb, aniž by ryby spolu s ostatními hráči ryby vyhubil.

Tento systém je však nedostačující a není nadále použitelný. Důvodem je nemožnost systém rozšířit a nemožnost přidat do něj další hry. Hra Rybolov se navíc nebude moci přidat do nově vytvářeného ŘS, protože je úzce spjata s původním ŘS a nedá se oddělit.

S původním řešením systému byl používán program *WatchDog*. Jde o program, který „sleduje“ ostatní aplikace a zajišťuje, že jsou neustále spuštěny. Pokud například dojde k pádu některé aplikace, tento program ji může znovu spustit. Tento program VIDÁtoři používají pro automatické spouštění aplikací při spuštění počítačů exponátu a zajištění jejich neustálého běhu. Tento program je nadále používán i s mým řešením ŘS.

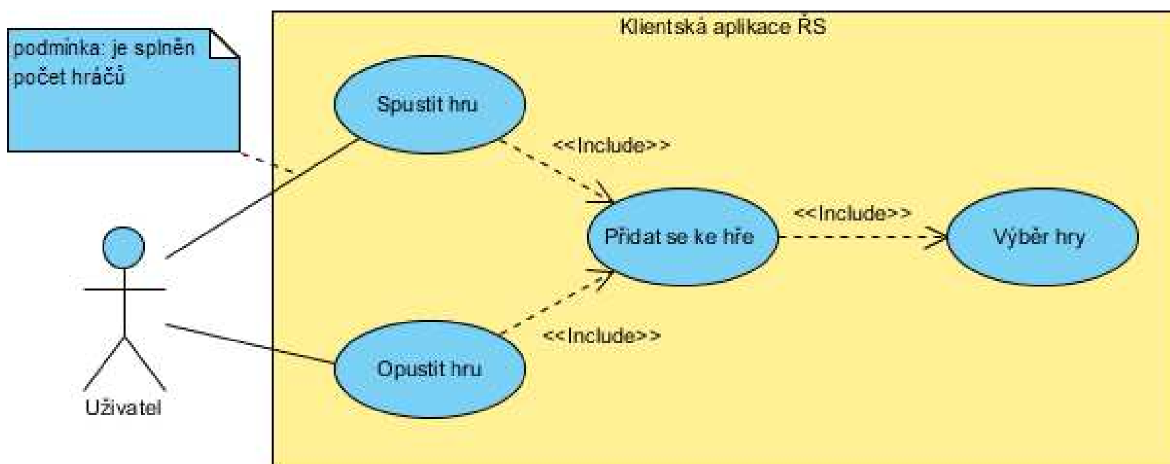
4.3 Požadavky

4.3.1 Řídicí systém a klientská aplikace

Před jakoukoliv implementací jsme, já a vedoucí mé práce, ještě před návštěvou VIDA! science centra, dali dohromady seznam požadavků, které by měl ŘS systém a klientské aplikace splňovat. Šlo o první představu o tom, jak by se měl systém chovat. Byly stanoveny následující body:

- Na každé klientské stanici budou na výběr hry, které je možné hrát (grafické menu).
- Běží-li hra využívající projektor, není možné spustit další hru využívající projektor.
- Každá hra má minimální a maximální počet hráčů.
- Kdokoliv může zvolit hru, v tu chvíli je na všech ostatních klientech možnost se se k této hře připojit. Není možné vytvořit novou instanci této hry, dokud se hra nezačne hrát.
- Je-li počet hráčů, kteří se připojili ke hře, dostatečný (tedy je splněn minimální počet hráčů pro danou hru), má kdokoliv možnost hru spustit.
- Je-li naplněn maximální počet hráčů, už není možnost se ke hře přidávat. Čeká se na zahájení.

Na základě stanovených požadavků jsem pro klientskou aplikaci vytvořil diagram případů užití.



Obr. 1: Diagram případů užití klientské aplikace

Dále jsme stanovili chování při spouštění hry:

- Řídicí systém spustí serverovou aplikaci, které předá jako parametr číslo portu, na kterém bude běžet.
- Řídicí systém spustí všechny klientské aplikace a předá jim číslo portu serverové aplikace.

A také jsme stanovili chování při ukončení hry:

- Řídicí systém periodicky kontroluje, že proces hry na klientských aplikacích pořád běží. Pokud ne, je potřeba zobrazit úvodní obrazovku a ukončit klient-

ské aplikace i na ostatních koncových stanicích, které byly připojeny ke stejné instanci hry.

Při první návštěvě VIDA! science centra jsme předvedli navržené požadavky pro ŘS a klientskou aplikaci, a můj první prototyp ŘS průvodcům centra (tzv. VIDÁtoři – lektoři, animátoři, kteří mají na starost pomoc při obsluze exponátů a předvádění vědeckých pokusů). S návrhem řešení souhlasili, až na jedinou funkcionalitu systému, a to instancemi hry. Nepožadovali tedy, aby bylo možné spouštět více instancí jedné hry. Důvodem bylo, aby nebyl uživatel systému zmatený, pokud na sousedních obrazovkách bude běžet jiná instance stejné hry, kterou právě hraje na obrazovce před sebou. Při téměř dokončeném systému bylo však domluvené zadání změněno: instance hry byly nakonec požadovány a následně jsem je přidal do systému.

4.3.2 Hry exponátu Kulatý stůl

ŘS musí být navržen tak, aby do něj bylo možné přidávat další hry a hru kolegy Drobného. Bylo tedy potřeba stanovit požadavky, které musí každá hra splňovat:

- Hra musí být rozdělena na 2 aplikace – klientskou a serverovou.
- Každá aplikace musí být spustitelná příkazem přes konzoli na operačním systému Windows.
- Aplikace musí používat pouze relativní cesty k potřebným souborům a musí být spustitelná nehledě na tom, kde je v adresářové struktuře uložena.
- Klientská aplikace musí být spustitelná se 2 parametry – číslem portu a IP adresou serveru, a to právě v tomto pořadí.
- Serverová aplikace musí být spustitelná s 1 parametrem – číslem portu.
- Číslo portu určuje na kterém portu klientská a serverová aplikace hry bude komunikovat. Je zvolena ŘS, ne danou hrou.
- Klientská aplikace musí být spuštěna v celoobrazovkovém režimu.
- Pokud hra využívá projektor, tak musí serverová aplikace zobrazovat promítaný obraz (viz kapitola *Promítání*)
- Spuštění aplikací je třeba otestovat přímo na exponátu a mít nainstalovaný potřebný software na klientech/serveru pro běh aplikací (knihovny, běhové prostředí apod.).

Díky spouštění her příkazem s parametry přes konzoli, je jedno, jak je každá hra implementována. Například hra kolegy Drobného je implementována v programovacím jazyce C#, zatímco ŘS je implementovaný v Javě.

4.3.3 Výběr hry

Součástí řešení této práce je implementace jedné hry, která bude do výsledného ŘS přidána a bude využívat projektor. Z VIDA! science centra byl k dispozici dokument se seznamem her, ze kterých je možné vybírat. Nabídka her (a simulace) byla následující:

- Simulace *Dálnice*. Cílem je naimplementovat simulaci, která je dostupná na odkaze *http://www.mtreiber.de/MicroApplet_html5* (spolu s odkazy na volně dostupný zdrojový kód a vysvětlením modelů) s danými modifikacemi.
- Hra *Distribuce limonád*. Jsi vedoucím velkoobchodu. Tvým úkolem v této hře je objednávat si od distributora tolik sudů limonády, aby pokryly poptávku od maloobchodníka.
- Hra *Parlament*. Cílem hry je vyzkoušet si vyjednávání podpory pro kontroverzní zákony. Hra je pro 3-12 hráčů, hlasováno je tedy o 3-12 „zákonech". Každému hráči je přidělen zákon, jehož je autorem, a je mu řečeno, se kterými ze zákonů prozatím spíš souhlasí (třetina ze všech), spíš nesouhlasí (třetina) a na které zatím nemá názor (třetina). Pro schválení zákona je třeba nadpoloviční většina. Cílem hráče je, aby byl schválen zákon, jehož je autorem.
- Hra *Vězňovo dilema*. Hra, u které se po spuštění hry zobrazí pokyny „Od této chvíle s nikým až do konce hry nekomunikuj!" a „Ty a tvůj spoluhráč jste spáchali trestný čin. Policie vás nyní vyslýchá v oddělených celách a dává ti nabídku. Když svého společníka udáš, pustí tě a tvůj společník si odsedí 20 měsíců. Když budeš zapírat, půjdeš na 1 měsíc do vězení za menší přestupky spáchané dříve. Víš, že pokud se s tvým společníkem udáte navzájem, půjdete do vězení oba na 5 měsíců. Co uděláš?", následně se zobrazí tlačítka „budu zapírat" a „udám svého společníka". Hra se dále nese v tomto duchu. Dává na výběr možnost volby a probíhá celkem v 5 kolech. Při vyhodnocení je zobrazena přehledná tabulka, jak se kdo v kterém kole zachoval, a ponaučení.

Kolega Drobný měl již dané téma s hrou Distribuce limonád. Další navržené hry se mi zdály velice strohé, nezáživné a málo interaktivní pro budoucí uživatele klientské aplikace ŘS (návštěvníky VIDA! science centra). Po návštěvě VIDA! science centra a konzultaci s VIDÁtory jsme všichni dospěli ke stejnému názoru. Místo výběru navržených her mi byla nabídnuta možnost implementovat hru s vlastním nápadem.

Rozhodl jsem se pro hru, která zabaví a naplno využije možnosti hrát proti ostatním hráčům. Pro hru jsem zvolil název *SnakeXSnake*. Tato hra je implementací klasické hry *Had*, kde hráč ovládá hada, který sbírá potrav, aby rostl, a snaží se dorůst co největší velikosti, aniž by narazil do zdi herní plochy nebo do svého vlastního těla. Co bude hra nabízet navíc je zpestření ve formě více hadů na herní ploše, kde každý hráč ovládá svého hada.

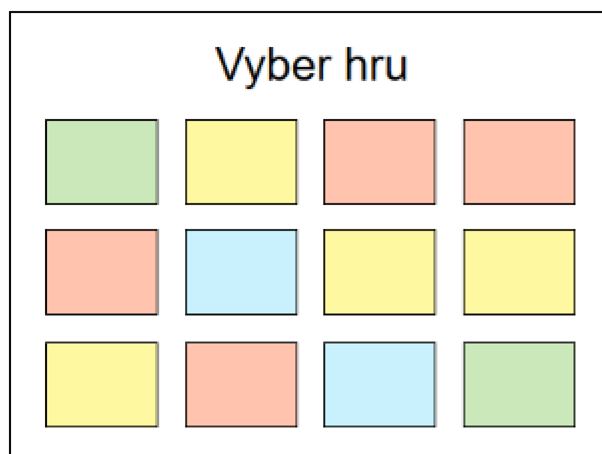
5 Návrh řešení

Tato kapitola popisuje, jakým způsobem jsem ŘS a hra SnakeXSnake navrhl, a jakým způsobem byla řešena funkcionalita jednotlivých technologií. Kapitola je úzce spjatá s teoretickým přehledem a přenáší tak tyto znalosti do praktického využití. U klientské aplikace ŘS jsem blíže popsal návrh grafického uživatelského rozhraní, protože u ní byl stěžejní. Je totiž hlavním aplikací exponátu, kterou ovládá návštěvník VIDA! science centra.

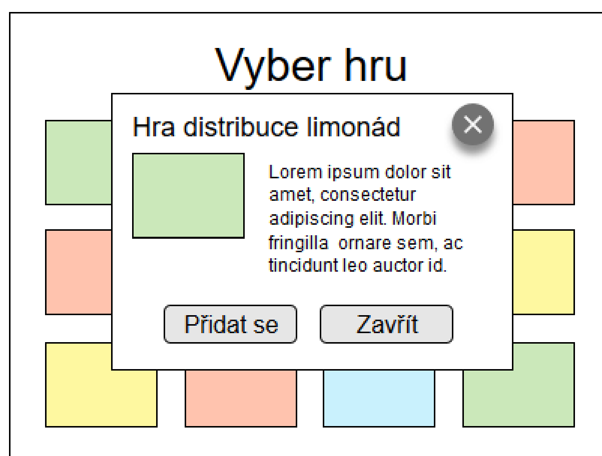
5.1 Návrh klientské aplikace řídicího systému

Od klientské aplikace se očekává, že bude mít možnost výběru hry, zobrazení popisu vybrané hry, přidání se do fronty na hru, opuštění fronty na hru a případně spuštění hry (při splněném počtu hráčů). Navrhl jsem 2 varianty grafického uživatelského rozhraní, které by tyto volby umožňovaly.

Hlavní obrazovka prvního návrhu klientské aplikace zobrazuje náhledy všech dostupných her. Výhodou prvního návrhu je, že uživatel vidí okamžitě přehled všech dostupných her a může si zobrazit detail hry, která mu „padla do oka“ podle jejího náhledu. Detail hry se zobrazí jako modální okno klepnutím na náhled hry.



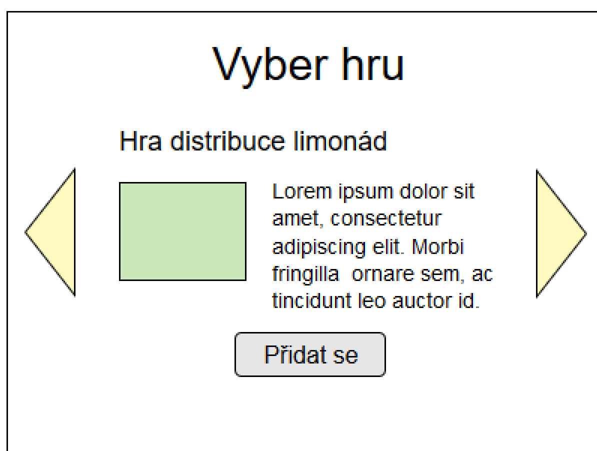
Obr. 2: První návrh hlavní obrazovky klientské aplikace pro výběr her



Obr. 3: První návrh klientské aplikace pro detail vybrané hry

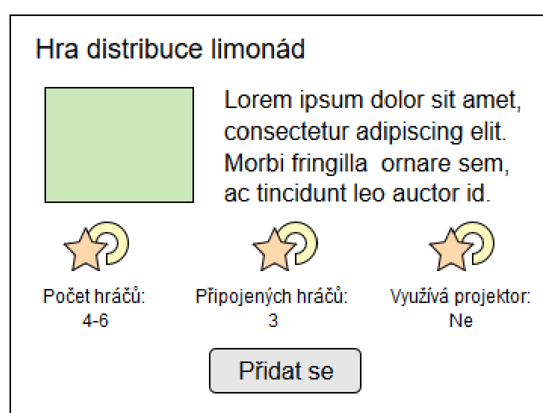
Bylo by dobré, aby byla aplikace interaktivnější a uživatel měl přehled o průběžných informacích. Jako například, u které hry zbývá už jen pár hráčů pro její spuštění, která hra je právě spuštěna nebo třeba která hra využívá projektor. Tyto informace by bylo možné zobrazovat na popiscích u každého náhledu hry.

U druhého návrhu je detail hry zobrazen již na hlavní obrazovce pro hru, kterou si uživatel zrovna prohlíží, a mezi hrami se dá přepínat pomocí šipek. Není tak potřeba zobrazovat žádné dialogové okno a tlačítko pro zavření tohoto okna (případně křížek v pravém horním rohu) jako u prvního návrhu.



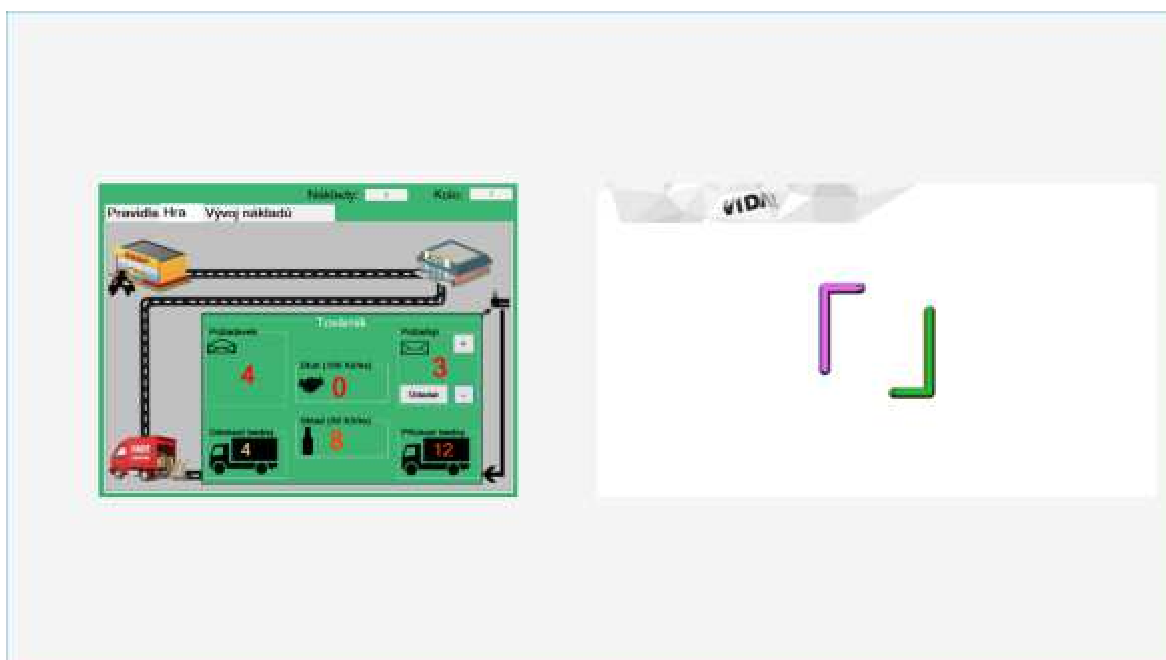
Obr. 4: Druhý návrh vzhledu klientské aplikace pro výběr hry s detailem, který zobrazuje i detail hry

Detail hry je u obou návrhů grafického rozhraní velice podobný. Dále jsem si musel udělat představu o tom, jak tento detail hry bude vypadat, a co bude obsahovat. Detail hry musí zobrazovat název hry, její náhled, popis a tlačítka s volbami. Také je potřeba zobrazit počet potřebných hráčů, počet připojených hráčů, a zdali hra využívá projektor.

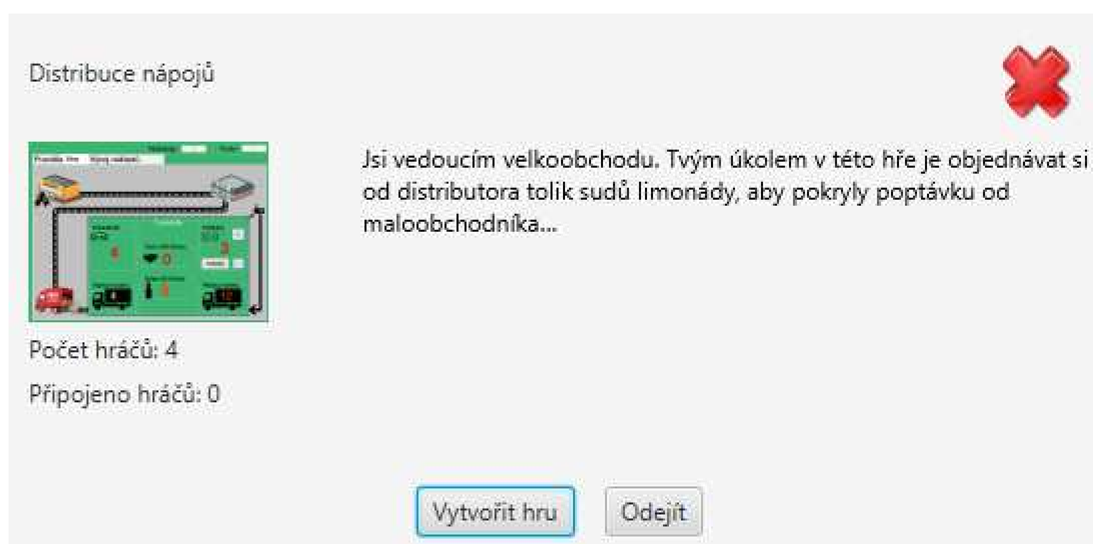


Obr. 5: Návrh části grafického uživatelského rozhraní klientské aplikace s detailem hry

Pro první prototyp ŘS a klientské aplikace jsem zvolil první návrh grafického rozhraní. Prototyp měl již implementovanou základní funkcionalitu: umožňoval zobrazovat detail vybrané hry, přidávání ke hrám, spouštění her apod. U prvního prototypu jsem nekladl důraz na vzhled klientské aplikace, ale pouze na vyzkoušení funkcionality.



Obr. 6: Hlavní obrazovka prvního prototypu klientské aplikace



Obr. 7: Modální okno detailu hry prvního prototypu klientské aplikace

Při první návštěvě VIDA! science centra se návrh nesetkal s velkou oblibou. Velkou nevýhodou bylo nevyužité místo na hlavní obrazovce při výběru her, protože ŘS ze začátku bude obsahovat pouze dvě hry, moji a hru kolegy Drobného, a další hry budou přidávány v budoucnu. Právě z tohoto důvodu jsem se přiklonil k druhému návrhu grafického rozhraní (snímek hotové klientské aplikace viz *Obr. 15*).

Většina uživatelů exponátů (návštěvníků VIDA! science centra) budou lidé kteří se seznámí s klientskou obrazovkou ŘS poprvé, lidé kteří se jdou bavit, a velkou část uživatelů budou tvořit děti. Z tohoto důvodu jsem se při návrhu klientské aplikace zaměřil na uživatelskou přívětivost a jednoduché ovládání. Aplikace vždy obsahuje pouze jediné tlačítko, a případně šipky pro přechod mezi hrami.

Klientská aplikace se může nacházet ve 4 stavech:

- Stav *můžeSePřidat* (viz *Obr. 15*). V tomto stavu se aplikace nachází, když není vybraná ani spuštěná žádná hra. Do tohoto stavu se aplikace vrátí, pokud je ukončena spuštěná hra. Uživatel má možnost přepínat se mezi hrami klepnutím na jednu z šipek nebo přejetím po displeji (gesto *swipe*). Dále má možnost přidat se k frontě na hru klepnutím na tlačítko *Přidat se*.

  Také jsem potřeboval navrhnout chování aplikace, pokud se chce uživatel přidat ke hře využívající projektor, který zrovna není dostupný (je využívaný jinou hrou). V tomto případě nemá uživatel možnost se ke hře přidat (viz *Obr. 19*)
- Stav *můžeOdejít* (viz *Obr. 16*). V tomto stavu se aplikace nachází, když byla vybrána některá z her, ale není splněný potřebný počet hráčů pro její spuštění. Uživatel může pouze počkat na další hráče nebo odejít z fronty na hru klepnutím na tlačítko *Opustit hru*.
- Stav *můžeSpustitHru* (viz *Obr. 17*). V tomto stavu aplikace nachází, když byla vybrána některá z her a je splněný potřebný počet hráčů pro její spuštění. Uživatel může pouze klepnout na tlačítko *Spustit hru* (pokud na něj dříve neklepne jiný z přidaných hráčů).
- Stav *hraBěží* (viz *Obr. 18*). V tomto stavu se aplikace nachází, když ŘS spouští hru a po dobu hraní hry. Uživatel nemá možnost provádět žádnou akci s klientskou aplikací ŘS, dokud není hra ukončena. Uživatel obrazovku aplikace vidí pouze při spouštění hry, protože jakmile je hra spuštěna, hráč vidí pouze okno spuštěné hry.

Klientskou aplikaci jsem navrhl tak, aby uživatele v každém jejím stavu prováděla a uživatel vždy věděl, co má v daném okamžiku dělat. To jsem zajistil velkým poblikávajícím nadpisem nad detailem hry. Tento nadpis uživatele informuje o tom, co se v aplikaci děje (obsahuje text „Spouští se hra“ nebo „Tato hra potřebuje projektor, který je nyní využívaný jinou hrou“) nebo jej pobízí k akci (obsahuje text „Vyber hru“, „Počkej, až se přidá dostatek hráčů pro spuštění hry“ nebo „Můžeš spustit hru“).

Uživatele navíc navádí i další poblikávající prvky grafického rozhraní. Ve stavu *můžeSePřidat* poblikávají tlačítka (šipky) pro přechod mezi hrami a tlačítko pro přidání se ke hře. Ve stavu *můžeOdejít*, kde uživatel čeká na další hráče, poblikává číslo představující počet přidaných hráčů. Ve stavu *můžeSpustitHru* poblikává tlačítko pro spuštění hry.

5.2 Návrh serverové aplikace řídicího systému

Z uživatelského hlediska je serverová aplikace ŘS velice jednoduchá. Obsahuje pouze konzoli (textové pole), ve které jsou zobrazovány změny stavu ŘS a výjimky (viz *Obr. 20*). Složitější je dění na pozadí, protože serverová aplikace ŘS je jádrem celého ŘS – uchovává stav o hrách, klientech, stavu ŘS a zajišťuje odpovídání na požadavky klientských aplikací.

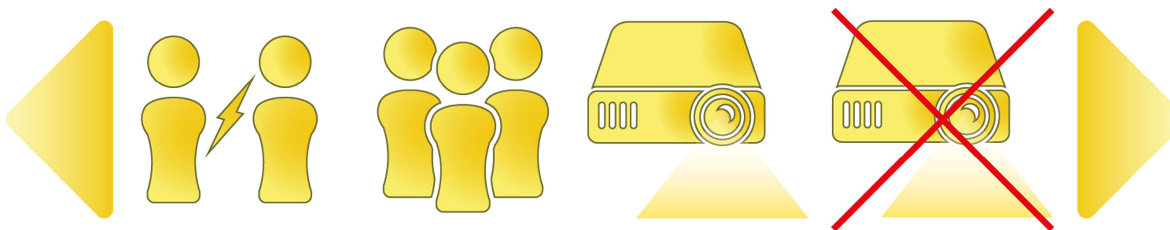
5.3 Návrh hry SnakeXSnake

Hra SnakeXSnake vychází z jednoduchého principu. Jde o implementaci klasické hry had, s tím rozdílem že je určena pro více hráčů. Hráč tedy musí svého hada vést tak, aby se vyhýbal ostatním hadům a zůstal ve hře jako poslední naživu. Ke strategizování mu může pomoci sbírání potravy, protože díky tomu had roste a má větší kontrolu nad herním polem. Pokud hráči chtějí, mohou i spolupracovat.

Při spuštění aplikace je na výběr barva hada, kterého bude hráč ovládat (viz *Obr. 25*). Na to má hráč 10 sekund a po skončení odpočtu je zobrazena herní plocha s hady všech hráčů s jejich zvolenými barvami. Pro zorientování se je před uvedením hadů do pohybu opět odpočítáváno (viz Obr 26). Jakmile jsou hadi uvedení do pohybu, hráči je ovládají gestem posunutí. Pokud hráč narazí do zdi nebo do jiného hada, tak prohrál. Vyhrává ten hráč, jehož had zůstane jako poslední naživu.

5.4 Tvorba grafiky

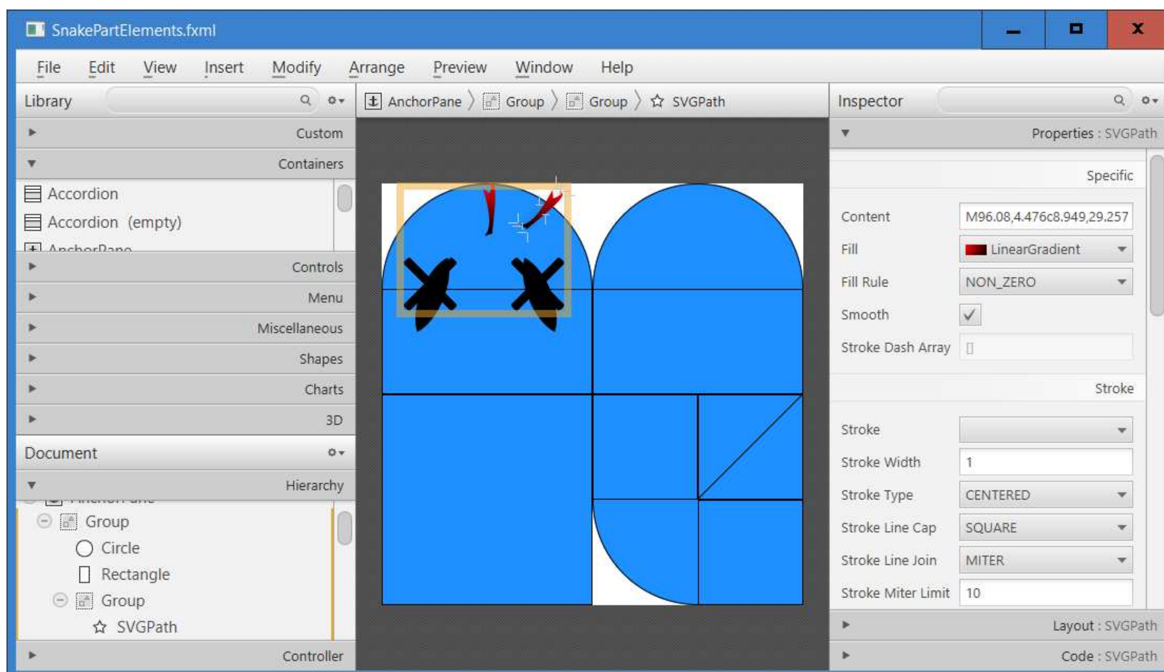
Grafické rozhraní aplikací a efekty animací jsem implementoval pomocí JavaFX. Co se však týče grafiky, potřeboval jsem ještě kromě na webu volně dostupných ikonek a obrázků, vytvořit ještě pár vlastních ikonek a části hadů. Ikonky jsem vytvořil v programu *Adobe Illustrator CS5*.



Obr. 8: Ikonky vytvořené pro klientskou aplikaci ŘS

Pro hru SnakeXSnake bylo potřeba navrhnout a vytvořit části hadů, u kterých lze programově změnit barvu. Tvar částí hadů jsem navrhl opět pomocí programu Adobe Illustrator CS5 jako vektorovou grafiku ve formátů SVG. Výplň těchto tvarů (barevné přechody) jsem však potřeboval řešit přímo v kódu, abych mohl nastavovat různé barvy hadů. Naštěstí je jazyk FXML (pro definici grafického rozhraní, který je

součástí JavaFX) postavený na XML stejně jako jazyk SVG. Tyto dva jazyky lze tak mezi sebou převádět pomocí knihoven třetích stran. Pro převod jsem použil knihovnu *svg2fxml*, která se spouští přes příkazový řádek.



Obr. 9: Grafická podoba části hada ve formátu FXML otevřeném v programu SceneBuilder

Podoba veškerých částí hada je definovaná v jednom souboru. Bylo však potřeba jednotlivé části oddělit a seskupit grafické elementy do skupin. K tomu slouží v JavaFX třída *Group* z balíku *javafx.scene*. Výplň jednotlivých části jsem vytvořil pomocí barevných přechodů *LinearGradient* a *RadialGradient*, které jsou součástí balíku *javafx.scene.paint*. Výslednou podobu hadů lze vidět na obrázcích hotové aplikace (například *Obr. 27*).

5.5 Zprávy posílané mezi klientem a serverem

Klientské a serverové aplikace spolu komunikují pomocí zpráv. Bylo potřeba definovat, které zprávy si mají vzájemně posílat a jaká bude jejich podoba. Zprávy jsou tvořeny klíčovým slovem definující požadavek a případnými parametry.

Požadavky, které posílá klient serveru u ŘS, jsem vybral následovně (*X* značí pořadové číslo hry):

- *JOIN X* – požadavek pro přidání se ke hře *X*. Přidáním je zde myšleno přidání se do fronty hráčů čekajících na spuštění hry.
- *LEAVE X* – požadavek pro opuštění fronty na hru *X*.
- *START X* – požadavek pro spuštění hry.

- *GET_INFO X* – požadavek pro získání počtu připojených hráčů ke hře *X*.
- *FINISH X* – požadavek pro ukončení běžící hry (posílá se v případě, kdy hra skončila, hráč ze hry odešel nebo v klientské aplikaci nastala chyba).
- *GET_PROJECTOR_INFO* – požadavek pro získání informace, zdali je projektor používán.

Požadavky (popř. odpověď či oznámení stavu), které posílá server klientovi, jsem vybral následovně (*X* značí pořadové číslo hry, *Y* počet připojených hráčů a *P* číslo portu)

- *ACCEPT X* – odpověď o přijmutí žádosti připojení klienta ke hře *X*.
- *CONNECTED X Y* – oznámení o počtu hráčů připojených ke hře *X*.
- *LAUNCH X P* – požadavek, na spuštění hry *X* (klientského procesu na klientovi) na portu *P*. Posílá se po spuštění serverového procesu na serveru.
- *END X* – požadavek, aby klientská aplikace ukončila svoji běžící hru.
- *PROJECTOR_IN_USE* – oznámení, že je projektor využívaný některou z her.
- *PROJECTOR_NOT_IN_USE* – oznámení, že projektor není využívaný žádnou hrou.
- *KILL_YOURSELF* – požadavek, aby klientská aplikace byla ukončena. Používá se v případě, kdy je ukončena serverová aplikace. Pokud je, server pošle všem klientům tuto zprávu, aby byly také ukončení. Důvodem je možnost rychlého restartu celého ŘS, a protože klientská aplikace je bez serverové zbytečná.

Oznámení, které posílá klient serveru u hry SnakeXSnake jsou následující:

- *MY_COLOR_IS C* – oznámení o vybrané barvě *C* hada při spuštění hry. *C* zde představuje hexadecimální kód barvy.
- *MOVE_ME D* – oznámení o požadovaném směru (*D* je zde jedna z hodnot *UP*, *RIGHT*, *DOWN*, *LEFT*).

Požadavky a oznámení, které posílá server klientovi u hry SnakeXSnake jsou následující:

- *COUNTDOWN_COLOR T* – požadavek na aktualizaci odpočtu. Server oznamuje klientovi, aby zobrazil číslo *T* jako počet zbylých sekund do ukončení výběru barvy hada.
- *COUNTDOWN_START T* – požadavek na aktualizaci odpočtu. Server oznamuje klientovi, aby zobrazil číslo *T* jako počet zbylých sekund do startu hry (uvedení hadů do pohybu).
- *SEND_ME_COLOR* – požadavek na poslání zvolené barvy hada. Je poslán při dokončení odpočtu pro výběr barvy.

- *INIT_BOARD INIT_STRING* – oznámení o konfiguraci herní desky. *INIT_STRING* je řetězec složený z informace o náhodně zvoleném pozadí, počtu hadů a jejich hráči zvolené barvy. Poslání této informace zajistí, že klient a server mají stejnou počáteční konfiguraci herní desky.
- *MOVE DIRECTIONS_STRING* – požadavek na pohyb hadů podle směrů obsažený v řetězci *DIRECTIONS_STRING*.
- *ADD_FOOD P C* – požadavek na přidání potravy s danou barvou *C* na danou pozici *P*.

5.6 Dotykové ovládání

Dotykové ovládání v JavaFX generuje stejné události, jako ovládání myší (viz kapitola *Dotykové ovládání v JavaFX*). Není tedy potřeba implementovat speciální chování při klepnutí, klepnutí a držení atd. Protože uživatelé exponátu Kulatý stůl budou ovládat ŘS a hry pomocí dotykových obrazovek, je vhodné jim práci usnadnit dotykovými gesty.

Grafické rozhraní ŘS jsem navrhl tak, aby veškeré události pomocí dotykových gest byly proveditelné i pomocí klepnutí. Dotykové gesto jsem nakonec zvolil pouze jedno, a to gesto posunu při výběru her. Pokud uživatel přejíždí prstem ze strany na stranu, tak se přepíná mezi dostupnými hrami. Má však možnost mezi hrami přepínat i pomocí virtuálních tlačítek.

Co se týče hry SnakeXSnake, přemýšlel jsem, jakým způsobem by hráč mohl ovládat hada. Nabízelo se hned několik možností řešení:

- gestem posunu ve směru, kterým má had směřovat,
- klepnutím na jednu z virtuálních šipek,
- ovládáním virtuální páčky (joysticku),
- klepnutím na část obrazovky (pravá část by znamenala pohyb doprava, horní část pohyb nahoru apod.),
- ovládáním hada, který by následoval poslední známou polohu klepnutí nebo tažení prstem

Nakonec jsem se rozhodl pro nejjednodušší variantu a to ovládání pouhým posunem prstu ve směru, kterým má had směřovat. Ve výsledku jsem tedy použil u ŘS a hry pouze jediné gesto. Toto gesto generuje událost *SwipeEvent* ze standardního balíku *javafx.scene.input*.

5.7 Promítání

Promítání je u exponátu Kulatý stůl zajištěno projektorem umístěným nad kulatým stolem, na který je promítáno. Na promítání se nepoužívá žádný speciální program

ani nebylo potřeba žádného speciálního nastavení. Projektor totiž pouze promítá duplikovaný obraz centrálního počítače. Není promítán obraz celé obrazovky, ale pouze její část, a to čtverec středu obrazovky o straně 700 obrazových bodů. Stačilo tedy, aby každá serverová aplikace zobrazovala to, co je potřeba promítat. U serverové aplikace ŘS jsem to vyřešil zobrazitelným oknem, které zůstává na popředí. A pokud je potřeba s aplikací pracovat (podívat se do logovací konzole), stačí toto okno zavřít, a po dokončení práce s aplikací znovu otevřít pro zobrazení promítacího okna. U serverové aplikace hry SnakeXSnake je promítána hrací deska hry na které se hadi pohybují (jde pouze o duplikaci toho, co vidí hráči na klientských počítačích).

U ŘS navíc bylo potřeba nastavit chování tak, aby na popředí obrazovky bylo vždy okno, které je potřeba promítat. Tedy promítací okno serverové aplikace ŘS v případě, že neběží žádná hra využívající projektor, a okno serverové aplikace dané hry v opačném případě (hry, která využívá projektoru).

5.8 Logování

U ŘS a hry SnakeXSnake jsem potřeboval zaznamenávat informace o průběhu a výskytu chyb v aplikacích. To bylo potřeba především u serverové aplikace ŘS, protože je centrem celého exponátu a stará se o správu her běžících na 12 koncových zařízeních (počítače s dotykovou obrazovkou). Pro logování všech potřebných informací jsem zvolil hned dva způsoby. Jedním z nich je vytvoření textové konzole jako součást serverové aplikace ŘS, do které se vypisují tyto informace a události (viz *Obr. 20*) a druhým z nich je použití logovacího nástroje *Log4j*.

Do logovací konzole ŘS jsou posílaný veškeré informace o změně stavu – pokud se k serveru připojí nový klient, pokud se přidá hráč ke hře, pokud je hra spuštěna nebo ukončena apod. Do logovací konzole jsou dále zaznamenávány veškeré zachycené výjimky ŘS. Protože ŘS může být na exponátu spuštěný i celý den, musel jsem vyřešit i mazání starých záznamů z konzole a jejich archivaci. Problém jsem vyřešil vytvořením časovače, který každou hodinu uloží záznam z konzole do souboru a promaže konzoli pro možnost dalšího záznamu (viz *Třída ConsoleLogger*).

Logování do konzole serverové aplikace ŘS poskytuje pouze základní informace o stavech her, připojování klientů a výpis chyb. Logování však bylo potřeba řešit i u ostatních aplikací (klientské aplikace ŘS a aplikací hry SnakeXSnake) a není vhodné pro zjištění příčiny problému (ladění). Při testování aplikací při vývoji nebo při testování přímo na exponátu jsem občas potřeboval zjistit příčinu chování nebo příčinu různých problémů, abych je mohl řešit. Potřeboval jsem znát, jaké metody se zavolaly, s jakými parametry, hodnoty vybraných proměnných apod. A to vše za běhu systému, pokud nějaký problém zrovna nastal – běžné ladění aplikace pomocí vývojového prostředí tedy nestačilo. Problém jsem vyřešil použitím logovacího nástroje Log4j. Závislost na této knihovně a její stažení mi zajistil Maven. Konfiguraci této závislosti jsem tedy pouze přidal do souboru *pom.xml*:

```
<!-- logging -->
<dependency>
    <groupId>log4j</groupId>
    <artifactId>log4j</artifactId>
    <version>1.2.17</version>
</dependency>
```

Pro použití Log4j pro logování bylo dále potřeba vytvořit soubor *log4j.properties* ve složce *resources* s konfigurací tohoto logovacího nástroje. Jedním z nastavení bylo například, kam se má log ukládat. To jsem definoval relativní cestou, pomocí následujícího řádku:

```
log4j.appender.file.File=logs\\application.log
```

Pro zalogování události, že se zavolala metoda a s jakými parametry, stačí pouze přidat jeden řádek na začátek dané metody. Například následující řádek vypíše, že byla zavolána metoda *multiply* s parametry *a* a *b*:

```
LogUtils.logMessage(this, "multiply", a, b);
```

6 Implementace

Tato kapitola blíže popisuje strukturu celého systému (ŘS i hry SnakeXSnake). Popsal jsem zde vše, od rozdělení projektu na moduly, přes rozdělení modulů na balíky a rozdělení balíků danými identifikátory, až po implementaci jednotlivých tříd. Je popsána pouze implementace tříd příslušících do balíků s identifikátorem *model*, protože jde o doménové třídy, které reprezentují logiku systému, zatímco balíky s identifikátorem *view* pouze deklarují grafické rozhraní systému a balíky s identifikátorem *controller* pouze reagují na události a zajištují změny v grafickém rozhraní a doménových třídách.

6.1 Struktura celého projektu

Jak ŘS, tak hra se skládá ze dvou aplikací – klientské a serverové. Z tohoto důvodu jsem celý projekt rozdělil do jednotlivých modulů. Protože klientské a serverové aplikace mají část funkcionality stejnou, vytvořil jsem i modul, který je určen pro použití jak klientskou aplikací, tak serverovou aplikací. Příkladem stejné funkcionality je třeba herní plocha hry SnakeXSnake, kterou zobrazuje jak klientská aplikace, tak serverová. Tento modul jsem si pracovně pojmenoval identifikátorem *common*.

Dále jsem vytvořil modul, který slouží pro použití oběma projekty, a díky jeho obecné implementaci jej lze v budoucnu využít i jinými projekty, popřípadě do něj připsat další funkcionalitu. Tento modul jsem si pracovně pojmenoval identifikátorem *utils*.

Modulů je tedy celkem 7:

- *VIDAControlSystem_client* – klientská aplikace ŘS,
- *VIDAControlSystem_common* – knihovna se sdílenou funkcionalitou pro aplikace ŘS,
- *VIDAControlSystem_server* – serverová aplikace ŘS,
- *VIDAControlSystem_snake_client* – klientská aplikace hry SnakeXSnake,
- *VIDAControlSystem_snake_common* – knihovna se sdílenou funkcionalitou pro aplikace hry SnakeXSnake,
- *VIDAControlSystem_snake_server* – serverová aplikace hry SnakeXSnake,
- *VIDAControlSystem_utils* – knihovna se sdílenou funkcionalitou pro všechny aplikace.

U těchto modulu jsem potřeboval definovat komunikaci a závislosti mezi nimi. Komunikace modulů probíhá přes stanovené rozhraní. V případě komunikace mezi klientskou a serverovou aplikací jsou rozhraním zprávy, které si navzájem posílají mezi sebou. Tyto zprávy jsem si již předem definoval (viz Zprávy posílané mezi klientem a serverem). Komunikaci zde zajišťuje protokol TCP.

Závislosti mezi moduly jsem definoval pomocí nástroje Apache Maven (viz kapitola *Apache Maven*) v konfiguračním souboru *pom.xml*, který obsahuje každý modul. Výsledný konfigurační soubor vypadá například následovně:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                             http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>cz.mendelu.savic.vida_cs_client</groupId>
    <artifactId>VIDAControlSystem_client</artifactId>
    <version>1.0-SNAPSHOT</version>
    <dependencies>
        <dependency>
            <groupId>cz.mendelu.savic.vida_cs</groupId>
            <artifactId>VIDAControlSystem_common</artifactId>
            <version>1.0-SNAPSHOT</version>
        </dependency>
    </dependencies>

    <properties>
        <maven.compiler.source>1.8</maven.compiler.source>
        <maven.compiler.target>1.8</maven.compiler.target>
    </properties>

</project>
```

Z tohoto souboru se dá mimo jiné zjistit, že jde o klientskou aplikaci ŘS, její verzi, že má definovanou závislost na modulu common projektu ŘS, a že kompilátor bude pracovat s Javou ve verzi 1.8.

6.1.1 Adresář resources

Adresář s hotovým řešením ŘS neobsahuje pouze spustitelné aplikace. Výsledné řešení ŘS totiž navíc obsahuje různé konfigurace uložené v souborech, použité obrázky, aplikace her, které jsou součástí ŘS a adresář kam si ukládá logy. Na to vše jsem vytvořil speciální adresář s názvem *resources*, kterou využívá jak klientská tak serverová aplikace ŘS. Tento adresář obsahuje 3 konfigurační soubory. Soubor *ipConfig.txt*, ve kterém je uložená IP adresa serverové aplikace. Soubor *resolution.txt*, ve které je definované rozlišení obrazovky centrálního počítače. A soubor *games.csv*, ve kterém je uložena konfigurace všech her, které jsou součástí ŘS.

Rozlišení obrazovky lze v JavaFX v kódu zjistit pomocí zápisu *Screen.getPrimary().getVisualBounds().* Metoda *getVisualBounds()* však vrací pouze část obrazovky, ve které může být aplikace zobrazena. V systému Windows je to celá obrazovka kromě hlavního panelu (*taskbar*). Pro vycentrování promítaného obrazu

jsem však potřeboval zjistit rozlišení celé obrazovky a nenašel jsem úplně vhodné řešení. Rozhodl jsem se proto informaci uložit do souboru, a to ještě z jednoho důvodu, a tím je že projektor promítá obraz na střed stolu s určitou odchylkou. Změnou hodnot v tomto souboru se tedy dá i kalibrovat centrování promítaného obrazu.

Asi nejdůležitější částí adresáře *resources* je konfigurace her a adresář se spustitelnými aplikacemi těchto her. Některé z obsažených informací o hrách jsou zobrazeny v klientské aplikaci při výběru her, některé definují jak má ŘS s hrami zacházet. V souboru *games.csv* jsou pro každou hru nastaveny následující informace:

- název hry,
- popis hry,
- informace, zdali hra používá projektor,
- minimální a maximální počet hráčů,
- příkaz pro spuštění klientské aplikace hry a
- příkaz pro spuštění serverové aplikace hry.

Z toho příkazy pro spuštění definují relativní cestu ke spustitelným aplikacím uloženým také v tomto adresáři.

6.2 Řešení síťové komunikace

Pro komunikaci po síti mezi klientem a serverem jsem jak u ŘS, tak u hry musel zvolit jeden z protokolů transportní vrstvy. Pro komunikaci mezi aplikacemi jsem si stanovil následující požadavky:

- spojení a přenos dat musí být spolehlivý – když si uživatel vybere hru, když si chce hru spustit, když chce pohnout hadem, tak očekává, že se tak opravdu stane,
- komunikace musí být obousměrná – je potřeba, aby klient posílal zprávy serveru a naopak,
- není potřeba posílat velké množství dat – zprávy určené pro komunikaci mezi klientem a serverem jsou jednoduché a je jich relativně málo (jsou posílány v poměrně dlouhých intervalech).

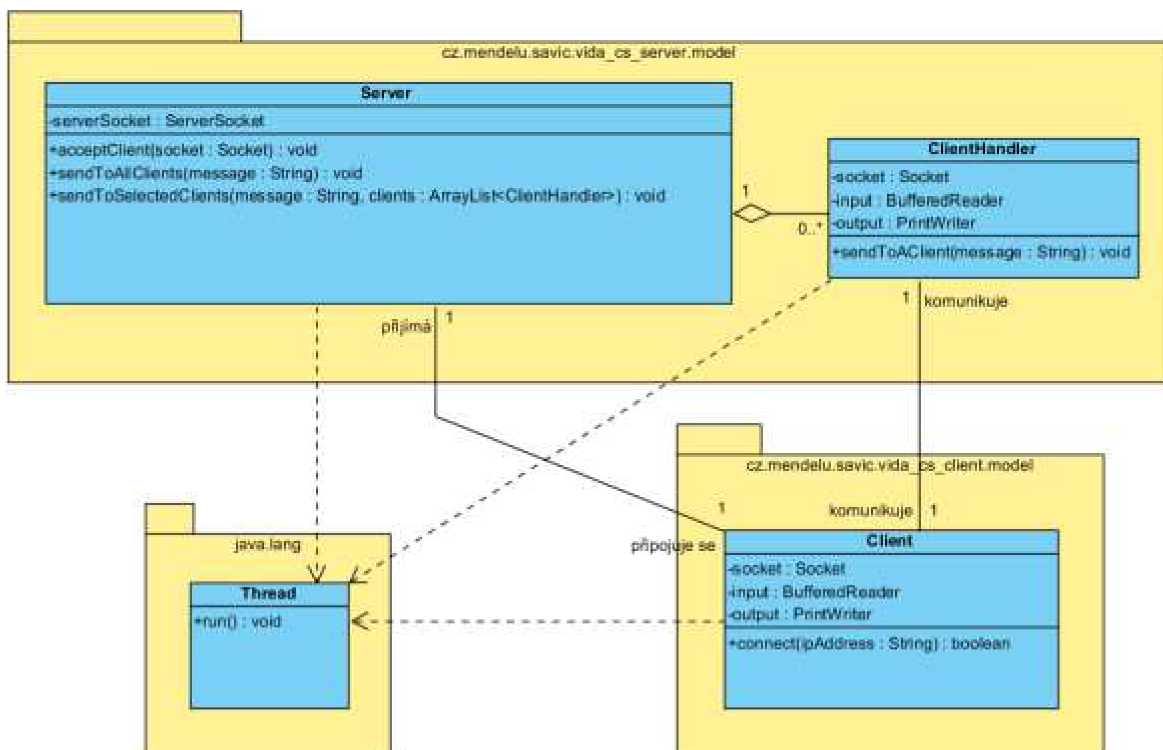
Na základě těchto požadavků jsem pro komunikaci zvolil protokol TCP, který nabízí přesně to, co požaduji.

V Javě lze pro síťovou komunikaci použít standardní knihovnu *java.net*. Pro použití protokolu TCP lze z této knihovny použít třídy *Socket* a *ServerSocket*. Síťový socket je koncový bod pro komunikaci mezi dvěma zařízeními.

Třída *Socket* implementuje klientský socket. Nabízí metody *getInputStream()* a *getOutputStream()*, které jsou použity pro získání vstupního a výstupního proudu

tohoto socketu. Pomocí těchto metod jsou inicializovány proměnné *input* a *output*, které slouží pro po příjem a posílání zpráv.

Třída *ServerSocket* implementuje serverový socket, jehož úlohou je čekání na požadavky (*request*). Toto čekání zajišťuje metoda *accept()*, která po přijmutí požadavku vrátí klientský socket, který požadavek poslal.



Obr. 10: Diagram tříd použitých pro síťovou komunikaci

Pro implementaci síťové komunikace jsem zvolil 3 třídy. Třídu *Client*, která se k serveru připojuje pomocí metody *connect()*. Třídu *Server*, která čeká na požadavek připojení od klientů. A třídu *ClientHandler*, jejíž instance je vytvořena pro každého klienta hned po jeho připojení k serveru. To serveru umožňuje odpovídat na dotazy více klientům. Všechny tyto třídy dědí třídu *Thread* ze standardního balíku *java.lang*, aby mohli běžet jako samostatná vlákna, a díky tomu asynchronně posílat a přijímat zprávy (v případě třídy *Server* přijímat klienty) nezávisle na chodu programu.

Třídy *Client* a *ClientHandler* spolu komunikují posíláním zprávy po klientském socketu. Pokud tedy klient potřebuje poslat zprávu serveru, provede to právě třída *Client* posláním zprávy třídě *ClientHandler*, která zavolá příslušné metody třídy *Server*, popř. pošle klientovi nazpět zprávou odpověď (pomocí metody *sendToAClient()*). A naopak, pokud třída *Server* oznamuje klientům změny, pošle zprávu buď všem klientům (pomocí metody *sendToAllClients()*) nebo pouze vybraným klientům (pomocí metody *sendToSelectedClients()*) a třída *Client* se podle toho zachová.

Tyto 3 třídy, implementující síťovou komunikaci obsahuje jak ŘS, tak hra SnakeXSnake, protože komunikace probíhá obdobným způsobem. Liší se pouze implementací, protože zprávy posílané v ŘS a zprávy posílané ve hře se liší.

6.3 Rozdělení do balíků

Pro přehlednější a snazší orientaci je vhodné rozčlenit každý modul do jmenných prostorů – v Javě jsou k tomu k dispozici balíky. Dle jmenné konvence jsou balíky pojmenovány podobně jako internetové domény v opačném pořadí a hierarchicky členěné identifikátory (odpovídající adresářům) jsou odděleny tečkami.

Všechny balíky začínají prefixem *cz.mendelu.savic* a dále jsou balíky pojmenovány podle modulu, ve kterém se nachází. Celkem je tedy pro rozdělení použito 7 identifikátorů: *utils*, *vida_cs*, *vida_cs_client*, *vida_cs_server*, *vida_cs_snake*, *vida_cs_snake_client* a *vida_cs_snake_server*.

Pokud modul implementuje i grafické rozhraní je dále rozčleněn identifikátory *model*, *controller* a *view*. Důvodem rozdělení je použitá architektura MVC (viz kapitola *MVC*). Moduly, které představují spustitelnou aplikaci, obsahují třídu *MainLoader*, která spouští aplikaci. Tato třída je umístěna v balíku bez tohoto identifikátoru.

6.4 Popis modulu VIDAControlSystem_common

Modul obsahuje 8 tříd, které jsou napsány obecně pro použití jak na serverové části, tak na klientské části ŘS.

6.4.1 Třída ColorHandler

Tato třída slouží pro náhodné generování barev (barva je reprezentována v JavaFX třídou *Color* ze standardní knihovny *javafx.scene.paint*).

6.4.2 Třída FileHandler

Třída *FileHandler* obsahuje metody pro jednoduché čtení/zápis ze/do souboru. Ke čtení/zápisu používá tříd *BufferedReader* a *BufferedWriter* ze standardní knihovny *java.io*.

6.4.3 Třída Game

Třída *Game* si uchovává veškeré údaje her v ŘS. U každé hry se eviduje její název, popis, cesta k obrázku ze hry, zdali používá projektor, možný počet hráčů, příkaz pro spuštění její klientské aplikace, příkaz pro spuštění její serverové aplikace a číslo reprezentující její pořadí v systému. Tato třída se nepoužívá přímo, slouží jako předek pro třídy *GameClient* a *GameServer*, které ji dále rozšiřují pro použití v klientské/serverové aplikaci.

6.4.4 Třída GameLoader

Třída *GameLoader* slouží pro načtení všech her použitých v ŘS, které jsou uloženy v souboru *games.csv*.

6.4.5 Třída GameProcess

Instance třídy *GameProcess* představuje proces běžící hry. Obaluje třídu *Process* ze standardní knihovny *java.lang*. Třída *Process* umožňuje v operačním systému spustit příkaz jako oddělený proces. Lze ji použít, jak pro spuštění klientské aplikace hry, tak pro spuštění serverové aplikace hry. Aby měl ŘS přehled o tom, zda proces stále běží, je součástí třídy *GameProcess* vlákno, které běh procesu periodicky kontroluje (metodou *isAlive()*, která je součástí třídy *Process*). Důležitou součástí této třídy je metoda, která se má spustit v případě, že proces skončí. Při skončení procesu se v případě klientské aplikace volá metoda, která pouze pošle serveru zprávu *FINISH.* V případě serverové aplikace se volá metoda *endGame()*.

6.4.6 Třída IPAddressConfig

Třída *IPAddressConfig* slouží pouze pro načtení IP adresy centrálního počítače, na kterém běží serverová aplikace, ze souboru, popř. pro zápis do něj (přepsání IP adresy).

6.4.7 Třída SimpleModalWindow

Třída *SimpleModalWindow* představuje jednoduché modální okno se zprávou. To je použito jako oznámení při čekání na připojení klientské aplikace k serverové aplikaci. Dá se dále použít podle potřeby pro případných budoucí úpravy ŘS.

6.4.8 Třída TransitionFactory

Třída *TransitionFactory* slouží pro vytvoření animací (přechodů), které jsou součástí třídy JavaFX. V ŘS jsem použil třídy *FadeTransition* (pro efekt schování/zobrazení prvku grafického rozhraní), *RotateTransition* (pro efekt otáčení) a *ParallelTransition* (animace složená z více efektů) ze standardního balíku *javafx.animation*. V ŘS je tato třída použita pro efekt poblikávání textu a dalších prvků grafického rozhraní. Efekt otáčení je pak použit pro otáčení středu obrazu, který promítá projektor na stůl. Jako parametr při volání metod této třídy se předává prvek grafického rozhraní (instanci třídy *Node*), kterému chceme animaci nastavit. Metody pak vrací výsledný efekt (instanci třídy *FadeTransition*/*RotateTransition*).

Implementace metod pro vytvoření efektu „poblikávání" grafického prvku vypadá následovně:

```
public static FadeTransition createFadingNode(Node node) {
    return createFadingNode(node, 0.7, 1, 0.25);
}
```

```
public static FadeTransition createSlightlyFadingNode(Node node) {
    return createFadingNode(node, 0.7, 1, 0.6);
}

public static FadeTransition createFadingNode(Node node, double dura-
tionInSeconds, double fromValue,
        double toValue) {
    FadeTransition transition = new FadeTransiti-
on(Duration.seconds(durationInSeconds), node);

    transition.setFromValue(fromValue);
    transition.setToValue(toValue);
    transition.setAutoReverse(true);
    transition.setCycleCount(Transition.INDEFINITE);

    return transition;
}
```

A implementace metod pro otáčení grafického prvku (použita projektorem pro otáčení celého promítaného obrazu) vypadá následovně:

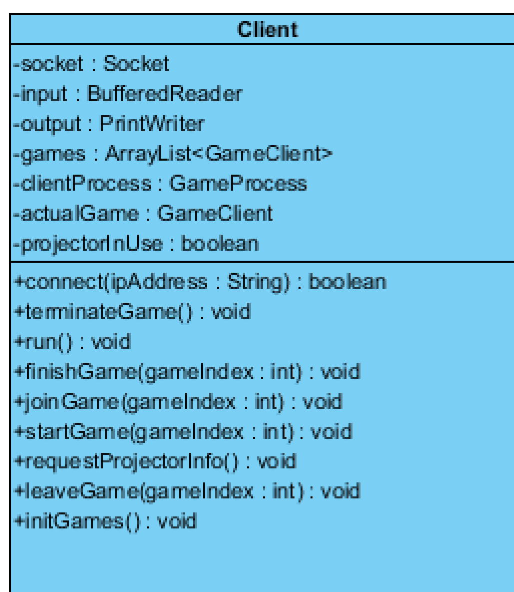
```
public static RotateTransition createSpinningNode(Node node) {
    RotateTransition transition = new RotateTransition();
    transition.setNode(node);
    transition.setByAngle(360);
    transition.setCycleCount(Transition.INDEFINITE);
    transition.setInterpolator(Interpolator.LINEAR);
    transition.setRate(0.03);

    return transition;
}
```

6.5 Popis modulu VIDAControlSystem_client

Modul obsahuje implementaci klientské aplikace. Protože tato aplikace má grafické uživatelské rozhraní, rozdělil jsem ji do balíků s identifikátory podle architektury MVC.

6.5.1 Třída Client

Třída *Client* slouží pro připojení k serverové aplikaci, následnou komunikaci s ní (posílání a přijímání zpráv), a spouštění a ukončování procesu hry.



Obr. 11: Struktura třídy Client vyjádřená diagramem tříd

Proměnné *socket*, *input*, *output* a metoda *connect()* jsem popsal již dříve (viz kapitola *Řešení síťové komunikace*). Zbytek třídy slouží pro správu stavu hry, se kterou uživatel právě interaguje přes grafické rozhraní, a případné spouštění a ukončování hry využitím třídy *GameProcess* (viz kapitola *Třída GameProcess*).

Třída si dále uchovává seznam her (proměnná *games)*, instanci procesu hry klienta (proměnná *clientProcess*), hru se kterou uživatel klientské aplikace právě interaguje (proměnná *actualGame*), a zdali je projektor momentálně využívaný některou hrou (proměnná *projectorInUse*).

Aby mohla třída *Client* přijímat zprávy ze strany serveru, tak je spuštěna jako vlákno (dědí třídu *Thread* ze standardního balíku *java.lang* a implementuje metodu *run()*). Implementace metody *run()* je důležitou součástí této třídy.

Metody *finishGame()*, *joinGame()*, *startGame()* a *requestProjectorInfo(), leaveGame()* jsou jednoduché metody, určené pro poslání zprávy serveru. Například metoda *startGame()* pouze posílá zprávu *START*:

```
public void startGame(int gameIndex) {
     output.println("START " + gameIndex);
}
```

Tyto metody jsou volány z metody *run()*, třídou *GameInfoController* (když uživatel klikne na tlačítko) nebo při spuštění klientské aplikace.

Metoda *terminateGame()* se stará o ukončení procesu klientské aplikace běžící hry. Metoda *initGames()* slouží pro inicializaci seznamu her a aktualizaci jejich stavu dotázáním serveru posláním požadavku *GET_INFO*.

6.5.2 Třída GameClient

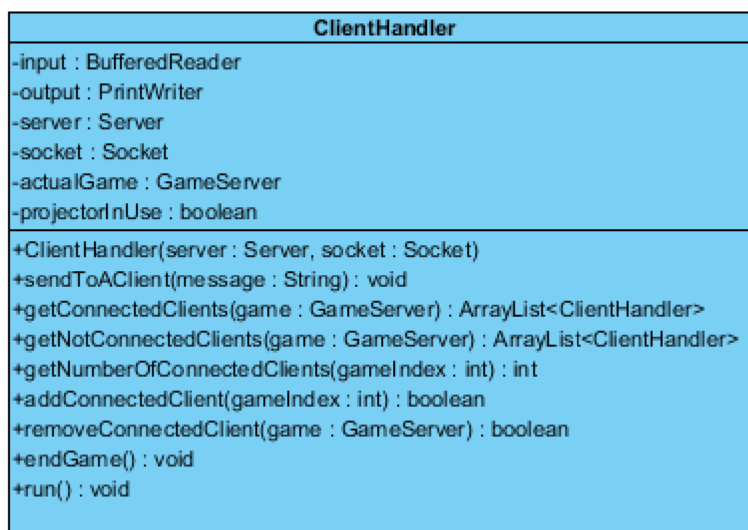
Třída *GameClient* pouze rozšiřuje třídu Game o číselnou proměnnou *connectedPlayers*. Tato proměnná slouží pro uchování aktuálního počtu hráčů u každé hry. Třída *Client* si aktualizuje tuto hodnotu pokaždé, když obdrží od serveru zprávu *CONNECTED*. Tuto zpráva server průběžně posílá všem klientům při změnách v počtu připojených hráčů ke hrám. Druhým případem je poslání této zprávy jako odpověď na zprávu *GET_INFO*. Tu posílá klient při svém spuštění.

6.6 Popis modulu VIDAControlSystem_server

Tento modul obsahuje implementaci serverové aplikace. Podobně jako u klientské aplikace jsem strukturu rozdělil do balíků s identifikátory podle architektury MVC.

6.6.1 Třída ClientHandler

Třída *ClientHandler* slouží pro komunikaci s klientskou aplikací. Pro každou klientskou aplikaci, která se připojí k serveru, je vytvořena právě jedna instance této třídy pro obsluhu daného klienta.



Obr. 12: Struktura třídy ClientHandler vyjádřená diagramem tříd

Proměnné *socket*, *input* a *output* jsem popsal již dříve (viz kapitola *Řešení síťové komunikace*). Proměnná *actualGame* obsahuje odkaz na aktuálně vybranou hru. Třída dále obsahuje metody *sendToAClient()*, která pošle klientovi danou zprávu, *getConnectedClients()*, která získá všechny připojené klienty k dané hře, getNonConnectedClients(), která naopak získá všechny nepřipojené klienty k dané hře, getNumberOfConnectedClients(), která získá počet připojených klientů k dané hře, *addConnectedClient()* a *removeConnectedClient()*, která přidá/odebere klienta této třídy k/z dané hře/hry, *endGame()*, která ukončí momentálně spuštěnou hru

a implementaci metody *run()*, která neustále čeká na zprávy od serveru a reaguje na ně.

6.6.2 Třída ConsoleLogger

Tato třída slouží pro periodické logování serverové konzole. Doba, po které se loguje, jsem momentálně nastavil na 1 hodinu. Třída *ConsoleLogger* po uběhnutí této doby zapíše veškerý text ze serverové konzole do textového souboru a text z konzole smaže (je tedy připravena pro nový záznam událostí). Jméno souboru obsahuje datum a čas zalogování (název souboru může být např. *consoleLogFor_2016-12-18_14-37.txt*). Po tomto zalogování jsou smazány staré soubory, kromě pár nejnovějších (momentálně nastaveno na 5).

Konzole obsahuje záznam událostí v serverové aplikaci a záznam výjimek. V případě, že by došlo k problému v ŘS, tak se z logů dá vypozorovat, o jakou chybu šlo a proč nastala.

Periodické spouštění logování zde zajišťuje třída *Timer* ze standardního balíku *java.util*, která funguje jako časovač. Pro spuštění časovače je použita metoda *scheduleAtFixedRate*(), která jako jeden z parametrů přijímá instanci třídy *TimerTask* (zde samotný *ConsoleLogger*, který je potomkem této třídy). *TimerTask* představuje úkol, který se má periodicky vykonávat. Třída *TimerTask* (a tedy i *ConsoleLogger*) běží jako vlákno (dědí třídu *Runnable*) a neovlivňuje tedy běh serverové aplikace. Úkol pro periodické vykonávání je definován v implementaci třídy *run()*. V případě třídy *ConsoleLogger* je úkolem pouze zavolání metody *logConsole()*, která zaloguje text z konzole.

6.6.3 Třída GameServer

Třída *GameServer* je potomkem třídy *Game* (viz kapitola *Třída Game*). Tuto třídu navíc rozšiřuje o proměnné *running* (typu *boolean*), *instance* (typu *int*) a *serverProcess* (třída *GameProcess*).

Do proměnné *running* se ukládá, zdali je hra, kterou *GameServer* představuje, momentálně spuštěna nebo ne.

Protože v ŘS je uloženo několik her a každá hra může být v jednom okamžiku spuštěna několikrát, tak je potřeba rozlišovat jednotlivé instance každé hry. K tomu slouží proměnná *instance*, která obsahuje pořadové číslo instance dané hry.

Proměnná *serverProcess* je proces serverové aplikace dané hry a touto třídou je spouštěn přes metodu *startServerProcess()*.

6.6.4 Třída ProjectorWindowEffects

6.6.5 Třída Server

Tato třída je jádrem celého modulu. Stará se totiž o správu všech připojených klientů (se kterými komunikuje třída *ClientHandler,* viz kapitola *Třída ClientHandler*), a správu všech her a jejich instancí. Třída je potomkem třídy Thread ze standardního

balíku *java.lang*. Chová se tedy jako vlákno. Vlákno, které na pozadí neustále čeká, jestli se nepřipojí nějaký klient. To zajišťuje metoda *accept()* třídy *ServerSocket* ze standardního balíku *java.net*. Návratovým typem této metody je třída Socket, představující socket, pomocí kterého může probíhat komunikace s daným klientem. Pokud se tedy připojí klient, je pro něj vytvořena instance třídy *ClientHandler*, který s ním komunikuje. Třída server si uchovává seznam těchto instancí (proměnná *clientHandlers*).

Třída server si dále uchovává seznam her. Každou hru představuje třída *GameServer* (viz kapitola *Třída GameServer*). Pokud je některá z her spuštěna, a přijde třeba požadavek pro přidání se ke hře, je v tomto případě vytvořena nová instance hry, ke které je uživatel přidán.

6.7 Popis modulu VIDAControlSystem_snake_common

Modul slouží jako sdílená knihovna pro klientský a serverový modul aplikace SnakeXSnake, tedy vše, co je v obou aplikacích stejné. Jsou to třeba výčtové typy:

- *DirectionEnum* – obsahuje hodnoty 4 směrů (*UP*, *RIGHT*, *DOWN*, *LEFT*), kterými se může had pohybovat.
- *MoveValidity* – obsahuje hodnoty, určující validitu tahu hráče. Hodnoty jsou následující: *VALID* (tah byl validní), *HIT_SNAKE* (had narazil do jiného hada), *HIT_WALL* (had narazil do zdi), *NOT_VALID* (nevalidní – hráč chtěl s hadem táhnout stejným směrem, kterým se již pohybuje) a *IS_ALREADY_DEAD* (had již jednou narazil a není tedy ve stavu, aby s ním bylo možné dále pohybovat).
- *SnakePartEnum* – obsahuje celkem 19 hodnot, 18 z nich pro všechny možné části hada (například hodnotu *CORNER_DOWN_RIGHT*, představující roh hada ve směru odspodu doprava), a jednu speciální, která představuje hadí potravu.

Dalším obsahem modulu jsou grafické komponenty. Jedna z komponent obsahuje grafické prvky tvořící herní plochu. Další zajímavou komponentou je grafická reprezentace částí hada (definovaná v souboru *SnakePartElements.fxml*). Způsob jakým jsem části hada vytvořil je již popsaný dříve (viz kapitola *Tvorba grafiky*). Modul dále obsahuje třídy popsané v následujících podkapitolách (jsou vybrány jen ty důležité).

6.7.1 Třída Snake

Třída reprezentující hada, kterého hráč ovládá. Obsahuje následující proměnné: seznam částí hada, šířku části hada, která slouží pro škálování velikosti dle hrací desky, barvu hada, jeho aktuální směr a zdali je naživu. Dále obsahuje metodu na pohyb hada a nastavení směru, kterým se má had ubírat.

6.7.2 Třída SnakePart

Třída reprezentující část hada. Uchovává v sobě jednu z hodnot z výčtu *SnakePartEnum*, pozici části na herní ploše, grafickou komponentu části hada, danou třídou *Group* z balíku *javafx.scene* a barvu této části.

6.8 Popis modulu VIDAControlSystem_snake_client

Modul je implementací serverové aplikace hry SnakeXSnake. Strukturu jsem rozdělil do balíků s identifikátory podle architektury MVC. Využívá většiny tříd z modulu *VIDAControlSystem_snake_common*. Proto nebudu rozebírat další třídy, protože jde pouze o třídy z balíku s identifikátorem *controller*, které pouze obsluhují grafické rozhraní.

Třídou, která zde stojí za zmínku je pouze třída *GameTimer*. Tato třída implementuje časovač, který řídí chod hry. Časovač během každé časové periody zavolá metodu pro posun hadů.

Další zajímavou třídou je třída *Client*. Ta funguje obdobným způsobem jako stejnojmenná třída ze serverové aplikace ŘS. Princip fungování těchto tříd je popsaný již dříve (viz kapitola *Řešení síťové komunikace*).

6.9 Popis modulu VIDAControlSystem_snake_server

Modul je implementací serverové aplikace hry SnakeXSnake. Strukturu jsem rozdělil do balíků s identifikátory podle architektury MVC. Využívá většiny tříd z modulu *VIDAControlSystem_snake_common*. Proto nebudu popisovat další třídy, protože jde pouze o třídy z balíku s identifikátorem *controller*, které pouze obsluhují grafické rozhraní.

Třída *ClientHandler* a *Server* funguje obdobným způsobem jako stejnojmenné třídy ze serverové aplikace ŘS. Princip fungování těchto tříd je popsaný již dříve (viz kapitola *Řešení síťové komunikace*).

6.10 Popis modulu VIDAControlSystem_utils

Modul slouží jako sdílená knihovna pro všechny ostatní moduly. Vytvořil jsem jej tedy proto, aby obsahoval třídy, které se dají obecně využít v kterékoliv aplikaci. Modul nakonec obsahuje pouze jedinou třídu – *LogUtils*, která zajišťuje logování (blíže popsáné v kapitole *Logování*).

6.10.1 Třída LogUtils

Jde o třídu, ve které jsem implementoval metody pro vlastní způsob logování. Tato třída využívá pro logování nástroje Log4j. Obsahuje metody určené pro zalogování

začátku metody (*logMethod()*), konce metody (*logMethodEnd()*), výjimky (*logException())* a libovolné informace (*logMessage()*).

7 Nasazení řídicího systému

Abych zajistil, že se bude vývoj ŘS a hry ubírat správným směrem a výsledné řešení nebude obsahovat neočekávané chyby, potřeboval jsem ŘS řádně otestovat na exponátu. Několikrát jsem tedy navštívil VIDA! science centrum a vyzkoušel si běh aplikací v praxi. Hotový systém jsem také řádně otestoval v ostrém provozu.

7.1 Připomínky a jejich řešení

Z testování a jednoho týdne běhu v provozu vznikla sada připomínek – od návrhů a požadavků, až po nalezené chyby. Tyto připomínky jsem průběžně řešil. Zde jsou popsány veškeré připomínky a jejich řešení (pokud jde o banální řešení, tak není uvedeno).

Nalezené chyby v klientské aplikaci ŘS:

- *Hry se dají měnit přejetím po displeji (gestem swipe), i když se hráč přidal do čekání na hru:* Tato chyba byla objevena hned při prvním testování výběru her. Pokud je aplikace ve stavu, kdy hráč vybírá hru (viz *Stav můžeSePřidat*), tak má možnost listovat mezi hrami. Pokud však aplikace v tomto stavu není, hráč by neměl tuto možnost mít. Chybou bylo, že gesto swipe tuto možnost umožňovalo v kterémkoliv stavu, protože bylo stále aktivní. Problém jsem vyřešil přepsáním metody pro tuto událost, tak aby se hra přepínala pouze, pokud je aplikace ve stavu výběru hry.
- *Nadpis se po některých událostech neustále zmenšuje:* Nadpis se nastavuje na jednu z velikostí – výchozí nebo malá (v případě dlouhého textu nadpisu). Velikost byla nastavována proporcionálně podle výchozí velikosti nadpisu v metodě pro inicializaci komponenty zobrazující popis hry. Chybou zde bylo, že tato inicializační metoda byla při používání aplikace volána několikrát a nadpis se tedy neustále zmenšoval. Vyřešil jsem voláním inicializační metody pouze jednou.
- *Spuštění hry využívající projektor přeruší promítání právě běžící hry (pokud je nějaká hra spuštěná a používá projektor):* Důvodem byla chyba v návrhu, kdy se nepočítalo s případem, že se spustí více her, využívající projektor. Vyřešil jsem schováním tlačítka pro spuštění u her využívající projektor, pokud již projektor používá jiná hra. Nemožnost spustit hru je uživateli oznámena nadpisem (viz *Obr. 19*).

Návrhy a požadavky ke klientské aplikaci ŘS:

- *Možnost spouštět více instancí jedné hry:* Jde o požadavek, který byl změněn VIDÁtory v průběhu vývoje. V původním návrhu bylo požadavkem mít možnost spustit každou hru maximálně jednou. ŘS jsem tedy přepsal tak, aby spravoval a umožňoval spuštění více instancí jedné hry.

- *Ukázat oznámení, pokud se aplikace připojuje k serverové aplikaci a mít možnost připojování přerušit:* Při spouštění se klientská aplikace nejdříve připojuje k serveru a čeká na odpověď. Pro toto připojování jsem nastavil určitou časovou prodlevu. Jde o pár sekund, avšak uživatel (správce exponátu), který aplikaci spustil, není informován o tom, co se děje. Důvodem je, že se aplikace spouští (připojuje) na pozadí. Problém jsem vyřešil zobrazením dialogového okna s informací o tom, že se aplikace připojuje. Navíc má uživatel možnost připojování zrušit klepnutím na křížek tohoto dialogového okna.
- *Ve stavu, kdy hráč může spustit hru (viz Stav můžeSpustitHru), mít také možnost hru opustit místo spuštění:* Požadavek jsem vyřešil přidáním tlačítka pro opuštění hry.
- *Ukončit klientské aplikace při ukončení serverové aplikace:* Požadavek vznikl kvůli nutnosti ukončovat klientské aplikace na všech počítačích (třeba když je restartovat ŘS) a protože spuštěná klientská aplikace je bez serverové aplikace zbytečná. Toto jsem vyřešil přidáním zprávy (viz zpráva *KILL_YOURSELF*), kterou pokud klient obdrží, tak se ukončí. Tuto zprávu posílá server všem klientům při jeho ukončení (ať už jde o zavření nebo spadnutí aplikace).

Nalezené chyby v serverové aplikaci ŘS:

- *Všechny instance stejné hry běží na stejném portu:* Problém jsem vyřešil generováním portu na základě ordinálního čísla instance.
- *Zvětšit promítaný obraz:* Promítaná kulatá plocha by měla mít průměr 700 obrazových bodů. Při vyzkoušení jsme zjistili, že je promítaná plocha o kousek větší. Zvětšil jsem tedy velikost promítaného obrazu.
- *Promítaný obraz není vycentrovaný:* Jde o drobnou odchylku způsobenou nepřesně vycentrovaným projektorem. Se změnou rozlišení a odchylkami promítaného obrazu jsem počítal – stačilo tedy pouze přepsat záznam o rozlišení v souboru *resolution.txt* (viz kapitola *Adresář resources*), podle kterého se centruje okno s promítaným obrazem.
- *Promítací okno je občas uzavřeno jinou událostí (nezůstává na popředí):* Problémem bylo špatně implementované chování promítaného okna. U okna lze v JavaFX nastavit, jestli má být vždy v popředí a lze jej skrýt/zobrazit. Ve výchozím stavu bylo promítané okno nastaveno tak, aby vždy zůstávalo v popředí. Pokud měla být spuštěna aplikace, toto nastavení se zrušilo a okno bylo skryto, a při skončení aplikace bylo nastavení opět vráceno a okno zobrazeno. Chování jsem však musel upravit tak, aby promítané okno nebylo na popředí pouze v případě, že je spuštěna hra využívající projektor. Dalším problémem bylo, že pokud se aplikace spouštěla znatelně delší dobu, okno bylo schováno a během této doby nebylo v popředí žádné okno určené k promítání (pouze konzole serverové aplikace, kterou uživatel nepotřebuje vidět). Zrušil jsem tedy neustálé zavírání/otevírání okna, které stejně nebylo potřeba.

- *Pokud na tlačítko Spustit kleplo více uživatelů, hra se spustila několikrát:* Problém jsem vyřešil ošetřením, že lze danou instanci dané aplikace spustit pouze jednou.

Návrhy a požadavky k serverové aplikaci ŘS:

- *Při spuštění aplikace mít otevřené promítací okno:* Promítací okno lze na serverové aplikaci spustit klepnutím na tlačítko *Promítat* (viz *Obr. 20*), a pokud je potřeba pracovat se serverovou aplikací (prohlížet si konzoli, zavřít ji atd.), lze promítací okno zavřít. Ve výchozím nastavení bylo okno zavřené. Požadavkem od VIDÁtorů však bylo mít okno spuštěné již při spuštění aplikace. Důvodem byla automatizace spouštění systému celého exponátu, protože VIDÁtoři občas spouští systém na dálku. Ve výsledku pouze nechají spustit veškeré počítače exponátu Kulatý stůl, a potřebné aplikace jsou spuštěny při spuštění operačního systému. Při spuštění serverové aplikace je tedy potřeba ihned promítat. Řešením bylo pouhé zobrazení okna již při vytvoření hlavního okna serverové aplikace.

Nalezené chyby v klientské a serverové aplikaci SnakeXSnake:

- *Hra se občas špatně vykreslila – některé části hada se například zobrazily na jiných částech herní plochy:* Problémem bylo nesprávné používání metody *runLater()* ze standardního balíku *javafx.application*, která určuje, co má vykreslovací vlákno vykreslit ve chvíli, kdy nebude zaneprázdněné. Protože toto vlákno vykresluje asynchronně s během systému, není dobré jej zatížit nadměrným využíváním nebo dlouhými výpočty. Problém jsem vyřešil voláním této metody pouze tam, kde je to nezbytně nutné.
- *Had se v klientovi občas pohybuje jinak, než na serveru a občas se hra zasekne:* Důvodem bylo nadměrné logování, které způsobovalo zpomalování aplikace. Bohužel jsem nepřišel na příčinu problému, a vyřešil jsem jej pouze vypnutím logování při běhu aplikace v provozu.

8 Diskuze

Návrh tématu pro tuto diplomovou práci vzešel z požadavku na bohatší ŘS nad exponátem Kulatý stůl. U tohoto exponátu totiž nebyl plně využitý potenciál. Původní ŘS byl velice jednoduchý a umožňoval spouštět pouze jednu hru. Navíc jej nebylo možné rozšířit, protože firma, která vytvořila toto řešení, neposkytla zdrojové kódy VIDA! science centru. Vznikl tedy požadavek na ŘS, který je především více konfigurovatelný, umožňuje spouštět více různých her, které lze v budoucnu postupně přidávat, a má otevřený kód pro jeho další rozšiřovaní. Výsledkem je ŘS, který tyto požadavky splňuje a splnil očekávání VIDÁtorů. ŘS byl po dobu jednoho týdne řádně testovaný v ostrém provozu a byly zpracovány veškeré připomínky. To výrazně pomohlo při optimalizaci řešení a zapracování finálních připomínek. ŘS je nyní nasazený v ostrém provozu a návštěvníci mají možnost si jej vyzkoušet.

Tato kapitola dále popisuje, čeho se dosáhlo, co by mohlo být dále vylepšeno a přidáno do ŘS, a jaké jsou další možnosti pro plné využití potenciálu exponátu Kulatý stůl.

8.1 Dosažené výsledky a možná rozšíření

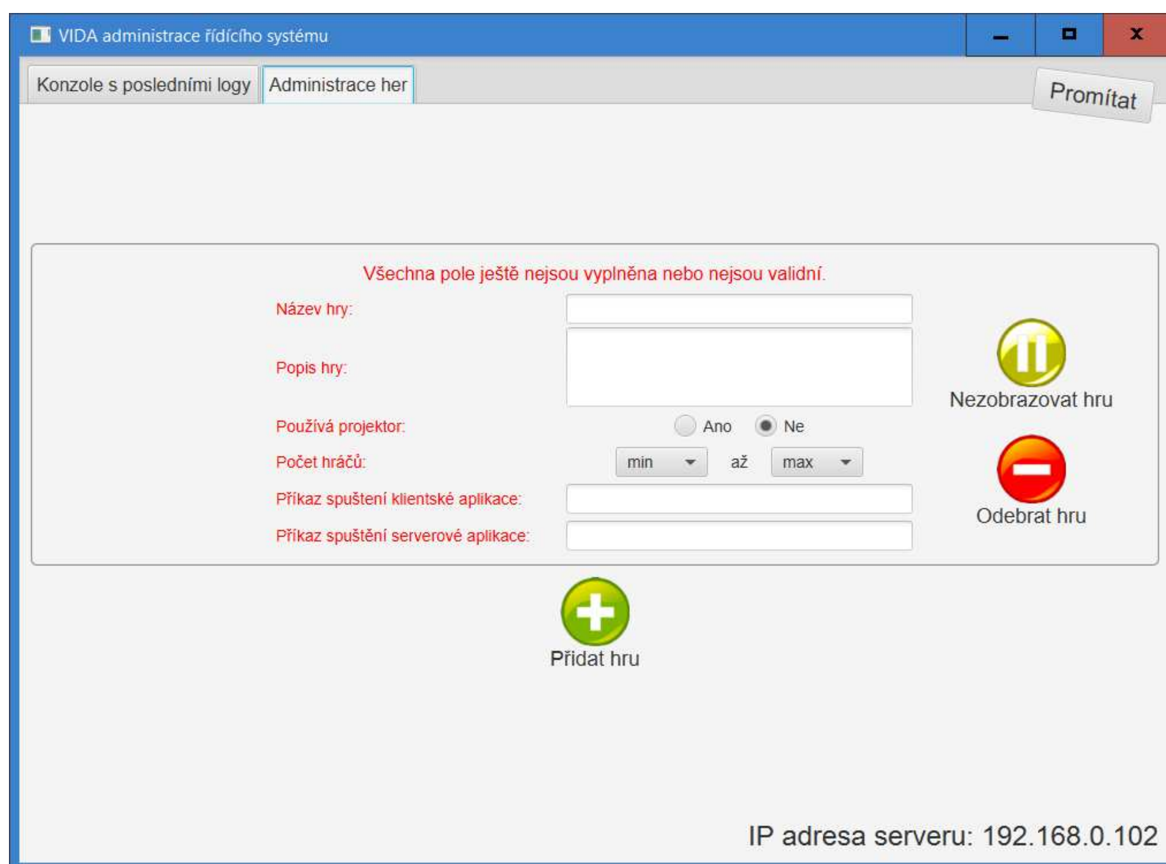
Zvolená metodika prototypového přístupu vývoje se ukázala jako vhodně zvolená. ŘS a následně i hru SnakeXSnake jsem tedy vyvíjel postupně od menších verzí (prototypů) až k finálnímu řešení. To mi umožnilo prototypy průběžně testovat přímo ve VIDA! science centru, i když ještě neměly veškerou funkcionalitu. Každý prototyp tedy přinesl něco nového, co bylo možné vyzkoušet, předvést VIDÁtorům a dostat zpětnou odezvu. Díky tomu jsem mohl při vývoji každého prototypu, zároveň s přidáním nové funkcionality, zapracovat požadavky a připomínky z předchozího prototypu.

Členění do modulů velice pomohlo v rozdělení celého projektu na logické celky, se kterými se lépe pracovalo a bylo možné je vyvíjet samostatně. K označení modulů, definici jejich závislostí a závislosti na knihovně třetí strany Log4j pomohl nástroj Maven.

Zvolený jazyk Java hodnotím jako správný výběr. Jedinou hlavní nevyužitou výhodou je zde přenositelnost. Protože bylo potřeba implementovat řešení na míru, na počítače s operačním systémem Windows, tak zde není přenositelnost potřeba a působí spíše jako nevýhoda. Programy jsou z tohoto důvodu, v Javě obecně, trochu pomalejší, zavádění aplikací do paměti při jejich spuštění trvá déle a aplikace jsou náročnější na paměť ve srovnání s nativními aplikacemi, které jsou staticky zkompilované pro danou platformu. Tato nevýhoda byla vidět hlavně při spouštění aplikací, které se spouštějí poměrně pomalu. Zrovna u tohoto ŘS, který je určen pro spouštění aplikací, je to poměrně velkou nevýhodou. U ŘS to však v praxi nepůsobí pro uživatele značné potíže. Vývojem v Javě jsem tedy dosáhl všeho potřebného, ale jako vhodnější řešení bych doporučil použití programovacího jazyka C++.

Při návrhu a během vývoje ŘS vznikla spousta nápadů na jeho možná vylepšení jak ze strany VIDÁtorů a vedoucího mojí práce, tak ze strany mojí. Požadavky vycházející ze zadání a mnoho dalších požadavků se podařilo splnit, na některé další však nezbyl prostor. Jedním z nich je například možnost volby jazyka, ve kterém systém komunikuje s uživatelem, a to aspoň mezi češtinou a angličtinou. Jazyk by bylo možné vybrat z obrazovky klientské aplikace ŘS a v ideálním případě by se volba přenesla do spuštěných her. Výběr jazyka by mohl být vyřešený třeba umístěním ikonek s vlajkami krajin do jednoho z rohů hlavní obrazovky. Spuštěné hry by pak byly spouštěny s parametrem navíc, který by specifikoval zvolený jazyk, ve kterém se má hra spustit.

Dalším návrhem, je jednodušší a přehlednější správa her, které ŘS obsahuje. Současným řešením je správa přes soubor ve formátu csv. Pro přidání nové hry do systému stačí do tohoto souboru přidat řádek s popisem a konfigurací dané hry. Toto řešení je jednoduché a postačující, ale není uživatelsky přívětivé pro správce systému. Lepším řešením by bylo vytvoření grafického rozhraní, které by bylo součástí okna serverové aplikace ŘS. Toto rozhraní by mohlo obsahovat formuláře pro editaci již existujících her a možnost vytvoření nové hry (přidání prázdného formuláře). Tento nápad vzešel z mojí strany, a vytvořil jsem pro něj návrh grafického rozhraní bez funkcionality.



Obr. 13: Možná podoba administrace her, která nebyla dokončena.

Myšlenkou zde bylo, že při každé editaci formuláře hry je příslušně změněna její konfigurace. Formulář by bylo navíc potřeba validovat, aby nedocházelo k chybným konfiguracím, které by mohly způsobovat chyby v systému. Protože klientské aplikace ŘS vycházejí ze stejné konfigurace, bylo by potřeba konfiguraci her při každé změně přenést na klientské počítače. Také by bylo potřeba všechny aplikace ŘS (klientské i serverovou) po změnách restartovat, aby nedocházelo k problémům kvůli změnám v konfiguraci her za chodu aplikací. Tento nápad se nakonec nedočkal implementace funkcionality a grafické rozvržení jsem z výsledného ŘS odebral.

8.2 Možná využití projektoru

Jak jsem již zmínil na začátku této kapitoly, potenciál exponátu nebyl původně plně využitý. Exponát Kulatý stůl vznikl spíše jako takový pokus s nedotaženou myšlenkou. Uspořádání exponátu, jako kulatého stolu s dvanácti dotykovými obrazovkami pro spouštění aplikací a projektorem, promítajícím na střed tohoto stolu, je zajímavou myšlenkou a řešení konceptu je velice dobré. Méně promyšlený je však Kulatý stůl po obsahové stránce aplikací a jeho provázání her s projektorem. Na začátku jsem měl

od VIDÁtorů k dispozici návrh možných her, ze kterého jsem mohl vybírat hru pro implementaci (viz kapitola *Výběr hry*). Tyto hry jsou však poměrně jednoduché, většinou založeny na nějakém poučení a podle mého názoru by nebyly pro uživatele moc záživné ani interaktivní. VIDA! science centrum nabízí řadu exponátů, které jsou především naučné a zábavné. Jejich princip by se dal tedy označit třeba heslem „věda hrou". Toho by se měl držet i exponát Kulatý stůl. Hru SnakeXSnake jsem navrhl pro oživení exponátu, protože je především interaktivní, zábavná a navíc kolektivní (popř. i kooperativní).

Co je zde však nevyužito, je promítací plocha, která by mohla být využita mnoha různými způsoby. Promítací plocha je rozšířením k dění na dotykových obrazovkách, a je společná pro všechny hráče. Může sloužit pro zobrazení informací o průběhu spuštěné hry, může sloužit jako rozšíření herní plochy apod. Hra Distribuce limonád není určena pro promítání a u hry SnakeXSnake je na promítací plochu pouze duplikována herní plocha, kterou hráči vidí také na dotykových displejích. Jako nejzajímavější řešení hry by však mohlo být právě rozšíření herní plochy.

Uvedu příklad hry, která by promítanou plochu mohla plně využít. Představme si třeba kteroukoliv sportovní hru, například curling. Hráči se snaží po ledové ploše dopravit kameny do vyznačeného prostoru. Tento vyznačený prostor by mohl být právě součástí promítané plochy, a hráči by ovládali kameny přes své dotykové obrazovky. Bylo by díky tomu využito i dotykového ovládání. Pro dopravení kamenu do vyznačeného prostoru by hráči mohli kámen ovládat třeba gestem posunutí a následně by sledovali trajektorii kamene, který by se postupně posunoval z plochy na dotykovém displeji až do vyznačeného prostoru na promítaném obrazu. Zajímavým oživením by tedy mohla být hra, která ovládáním dotykového displeje ovlivňuje dění na ploše promítaného obrazu. Takový typ hry by byl však náročnější na implementaci, protože by musel mít dobře vytvořenou fyziku, úzké provázání dotykové plochy s plochou promítací a ideálně pěkné grafické zpracování. Pro implementaci by tedy bylo vhodné použít některý herní engine (*Unity 3D*, *Unreal Engine* apod.). To bylo však nad rámec této diplomové práce. Nápad tedy může sloužit jako inspirace pro možná rozšíření nebo pro vytvoření tématu pro jinou diplomovou práci. Jedinou nevýhodou exponátu je, že promítací plocha je poměrně malá oproti ploše celého stolu (viz Obr. 31).

9 Závěr

Cílem práce bylo vytvořit řídicí systém pro exponát Kulatý stůl, který slouží pro spouštění her pro více hráčů, a jedné hry. Důraz byl kladen především na rozšiřitelnost jak samotného řídicího systému, tak her které obsahuje, přehlednost a uživatelskou přívětivost.

Po úvodním seznámení je čtenář seznámen s metodikou. Tato část přibližuje způsob, jakým jsem přistupoval k vývoji a jaké byly vstupní podmínky a požadavky pro řešení.

Následuje teoretický základ, kde jsou přiblíženy pojmy a metody dané problematiky, a technologie, které jsem při vývoji používal.

Dále je analyzována současná podoba exponátu a návrh řešení. Řešení je úzce spjato s metodami a technologiemi popsanými v teoretickém základu, vychází z nich a převádí je do praktického využití.

Čtenář je dále blíže seznámen se strukturou projektu a implementací daného řešení.

Důležitou součástí práce je zapracování připomínek získaných z průběžného testování prototypů a řádného testování v ostrém provozu. Dosažené výsledky jsou rozebrány a jsou navržena další možná rozšíření exponátu.

Výsledné řešení je nasazeno na exponátu Kulatý stůl a může si jej vyzkoušet každý návštěvník VIDA! science centra. Pro využití řídicího systém jsem implementoval hru SnakeXSnake. Byly splněny veškeré požadavky a výsledné řešení tak nahrazuje řešení původní. Řídicí systém byl navržený tak, aby do něj bylo možné jednoduše přidávat další hry, díky specifikaci jasných požadavků a rozhraní těchto her. Samotný řídicí systém může být dále rozšířen především o možnost přepínání jazykových verzí aplikací a uživatelsky přívětivější administraci her. Největším přínosem by bylo především naplnění řídicího systému dalšími hrami, protože momentálně obsahuje pouze hru Distribuci limonád a hru SnakeXSnake.

Cíl práce byl splněný a splnil očekávání. Výsledný řídicí systém je uživatelsky přívětivý, rozšiřitelný a spolu se hrou je komunikace po síti je rychlá a spolehlivá.

10Literatura

Použité zdroje a literatura:

About IntelliJ IDEA. *ComponentSource: Software Superstore for Developers & IT Pros* [online]. 2017 [cit. 2017-05-21]. Dostupné z: https://www.componentsource.com/product/intellij-idea/about

ANDERSON, G. a P. ANDERSON. *Essential JavaFX™*. California: Sun Microsystems, 2009, 361 s. ISBN 978-0-13-704279-1.

BLEWITT, Alex. Modular Java: What Is It? *InfoQ: Software Development News, Videos & Books* [online]. 2009 [cit. 2017-05-19]. Dostupné z: https://www.infoq.com/articles/modular-java-what-is-it

BLOCH, Joshua. *Java efektivně: 57 zásad softwarového experta*. 1. vyd. Praha: Grada, 2002, 230 s. Moderní programování. ISBN 80-247-0416-1.

BOOCH, Grady. *Object-oriented analysis and design with applications*. 2nd ed. Redwood City, California: Benjamin/Cummings Pub. Co., 1994, 720 s. ISBN 08-053-5340-2.

BRACKEEN, David, Bret BARKER a Laurence VANHELSUWÉ. *Vývoj her v jazyku Java*. Praha: Grada, 2004, 710 s. Moderní programování. ISBN 80-247-0874-4.

BROOKSHEAR, J. Glenn, David T. SMITH a Dennis BRYLOW. *Informatika*. Brno: Computer Press, 2013, 608 s. ISBN 978-80-251-3805-2.

EBBERS, H. *Mastering JavaFX™ 8 Controls*. New York: McGraw-Hill Education, 2014, 338 s. ISBN 978-0-07-183378-3.

GAMMA, Erich. *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, Mass.: Addison-Wesley Professional, 1995, 395 s. ISBN 0-201-63361-2.

HEROUT, P. *Učebnice jazyka Java.* 5. vyd. České Budějovice: Kopp, 2010. 386 s. ISBN 978-80-7232-398-2.

HORDĚJČUK, Vojtěch. Maven. *Vojta Hordějčuk aka voho: Software Engineer and Bedroom Music Producer* [online]. 2017 [cit. 2017-05-19]. Dostupné z: http://voho.eu/wiki/maven/

Java™ Platform, Standard Edition 8 API Specification. *Oracle: Documentation* [online]. 1993-2016 [cit. 2016-12-21]. Dostupné z:

http://docs.oracle.com/javase/8/docs/api/

JavaFX API: Overview. *Oracle: Documentation* [online]. [cit. 2017-05-19]. Dostupné z: https://docs.oracle.com/javase/8/javafx/api/

KAVALEC, Jaroslav. IntelliJ Idea pro vývoj v Javě. *AspectWorks* [online]. 2011 [cit. 2017-05-21]. Dostupné z: http://www.aspectworks.com/2011/12/intellij-idea-pro-vyvoj-v-jave/

MOHAN, P. *Beginning JavaFX™*. New York: Apress, 2010, 337 s. ISBN 978-1-4302-7199-4.

MORRIS, S. *JavaFX in Action*. Greenwich: Manning Publications, 2010, 385 s. ISBN 978-1-933988-99-3.

PETERKA, Jiří. TCP a UDP. *EArchiv.cz* [online]. 1999 [cit. 2017-05-19]. Dostupné z: http://www.earchiv.cz/anovinky/ai1864.php3

SHARAN, K. *Learn JavaFX™ 8: Building User Experience and Interfaces with Java 8*. New York: Apress, 2015, 1210 s. ISBN 978-1-4842-1143-4.

SHARWOOD, Simon. JDK 9 release delayed another four months. *The Register: Sci/Tech News for the World* [online]. 2016 [cit. 2017-05-19]. Dostupné z: http://www.theregister.co.uk/2016/09/14/jdk_9_release_delay/

SCHILDT, Herbert. Java Fundamentals. *Java: A Beginner's Guide* [online]. Sixth edition. New York: McGraw-Hill Education, 2014, 30 s. [cit. 2017-05-19]. ISBN 978-0071809252. Dostupné z: http://www.oracle.com/events/global/en/java-outreach/resources/java-a-beginners-guide-1720064.pdf

SCHINDL, Tom. SVG To FXML Converter as Commandline util. *Tomsondev Blog* [online]. 2012 [cit. 2017-05-19]. Dostupné z: https://tomsondev.bestsolution.at/2012/08/22/svg-to-fxml-converter-as-commandline-util/

SMART, John F. An introduction to Maven 2: How applied best practices can optimize the Java build process. *JavaWorld* [online]. 2005 [cit. 2017-05-19]. Dostupné z: http://www.javaworld.com/article/2072203/build-ci-sdlc/an-introduction-to-maven-2.html

TAMAN, M. *JavaFX™ Essentials*. Birmingham: Packt Publishing Ltd.,2015, 224 s. ISBN 978-1-78439-802-6.

THAKUR, Dinesh. Prototyping Model in Software Engineering. *Computer Notes* [online]. 2017 [cit. 2017-05-19]. Dostupné z: http://ecomputernotes.com/software-engineering/explain-prototyping-model

TOPLEY, K. *JavaFX™ Developer's Guide*. New Jersey: Pearson Education, 2011, 1150 s. ISBN 978-0-321-60165-0.

VALKOVIČ, Patrik. 1. díl - Git - Historie a principy. *Itnetwork.cz* [online]. 2014 [cit. 2017-05-21]. Dostupné z: https://www.itnetwork.cz/software/git/git-tutorial-historie-a-principy

What's New in JDK 8. *Oracle: Integrated Cloud Applications and Platform Services* [online]. 2016 [cit. 2017-05-21]. Dostupné z: http://www.oracle.com/technetwork/java/javase/8-whats-new-2157071.html

Přílohy

A Struktura přiloženého CD

- adresář VIDAControlSystem – který obsahuje zdrojové kódy všech modulů a následující spustitelné aplikace:
    - v adresáři VIDAControlSystem_server – obsahuje spustitelný soubor serverové aplikace ŘS VIDAControlSystem_server.jar
    - v adresáři VIDAControlSystem_client – obsahuje spustitelný soubor klientské aplikace ŘS VIDAControlSystem_client.jar
- adresář diplomova_prace – obsahuje elektronickou verzi diplomové práce (soubor diplomova_prace.pdf)

B Snímky obrazovky ŘS a klientské aplikace

Obr. 14: Obrazovka klientské aplikace ŘS při nevybrané hře

Počkej až se přidá dostatek hráčů pro spuštění hry

Distribuce nápojů

Požadovaný počet hráčů:

4

Připojeno hráčů:

1

Využívá projektor:

Ne

Jsi vedoucím velkoobchodu. Tvým úkolem v této hře je objednávat si od distributora tolik sudů limonády, aby pokryly poptávku od maloobchodníka...

Opustit hru

VIDA!

Vytvořil David Savič

Obr. 15: Obrazovka klientské aplikace ŘS při přidání se ke hře

Obr. 16: Obrazovka klientské aplikace ŘS při splněném počtu hráčů a možnosti spuštění hry

Spouští se hra

Distribuce nápojů

Požadovaný počet hráčů:

4

Připojeno hráčů:

0

Využívá projektor:

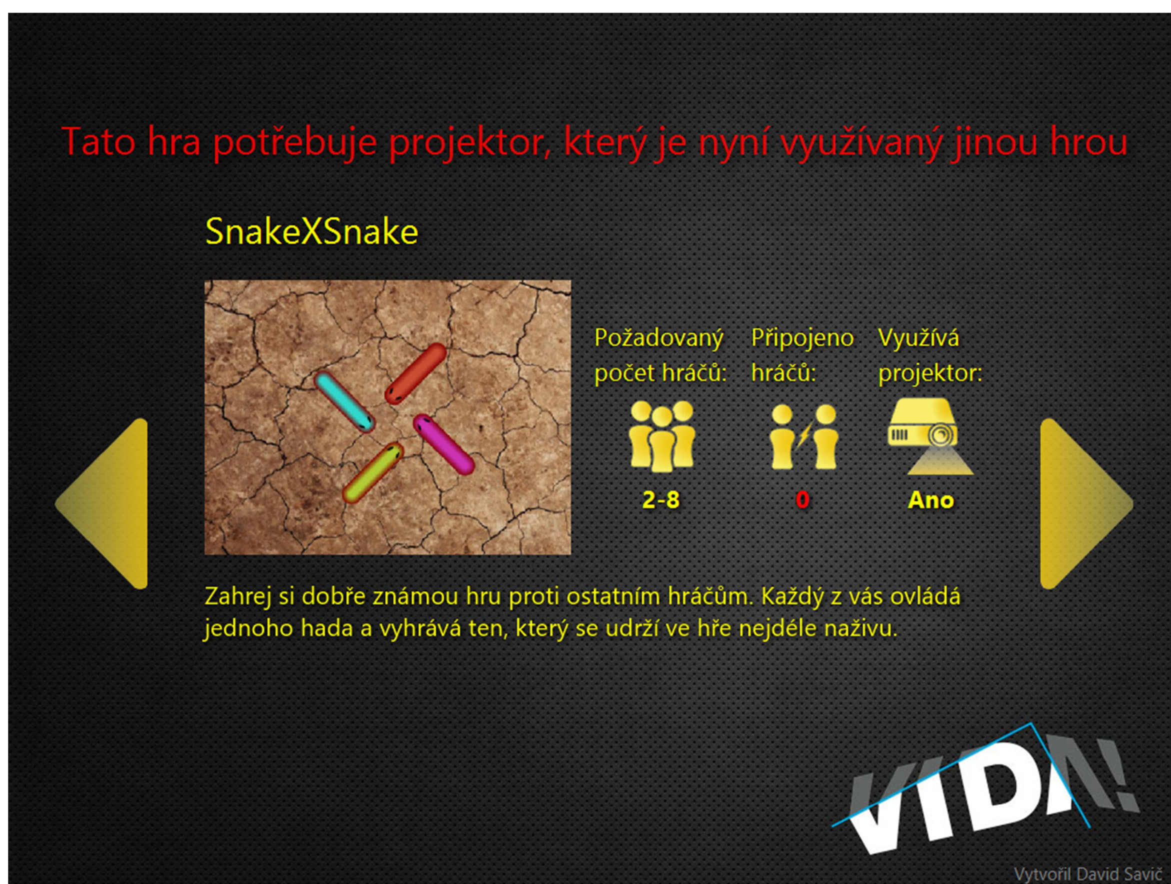
Ne

Jsi vedoucím velkoobchodu. Tvým úkolem v této hře je objednávat si od distributora tolik sudů limonády, aby pokryly poptávku od maloobchodníka…

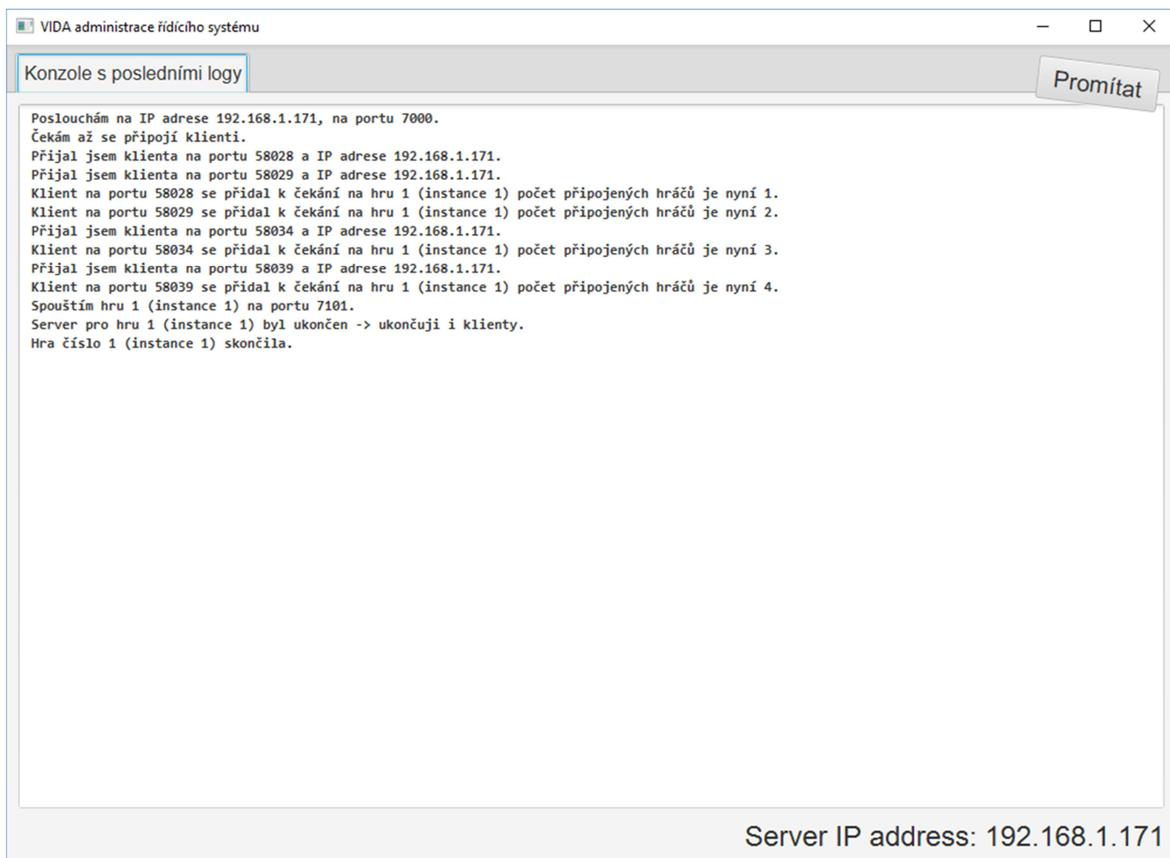
VIDA!

Vytvořil David Savič

Obr. 17: Obrazovka klientské aplikace ŘS při spouštění a hraní hry



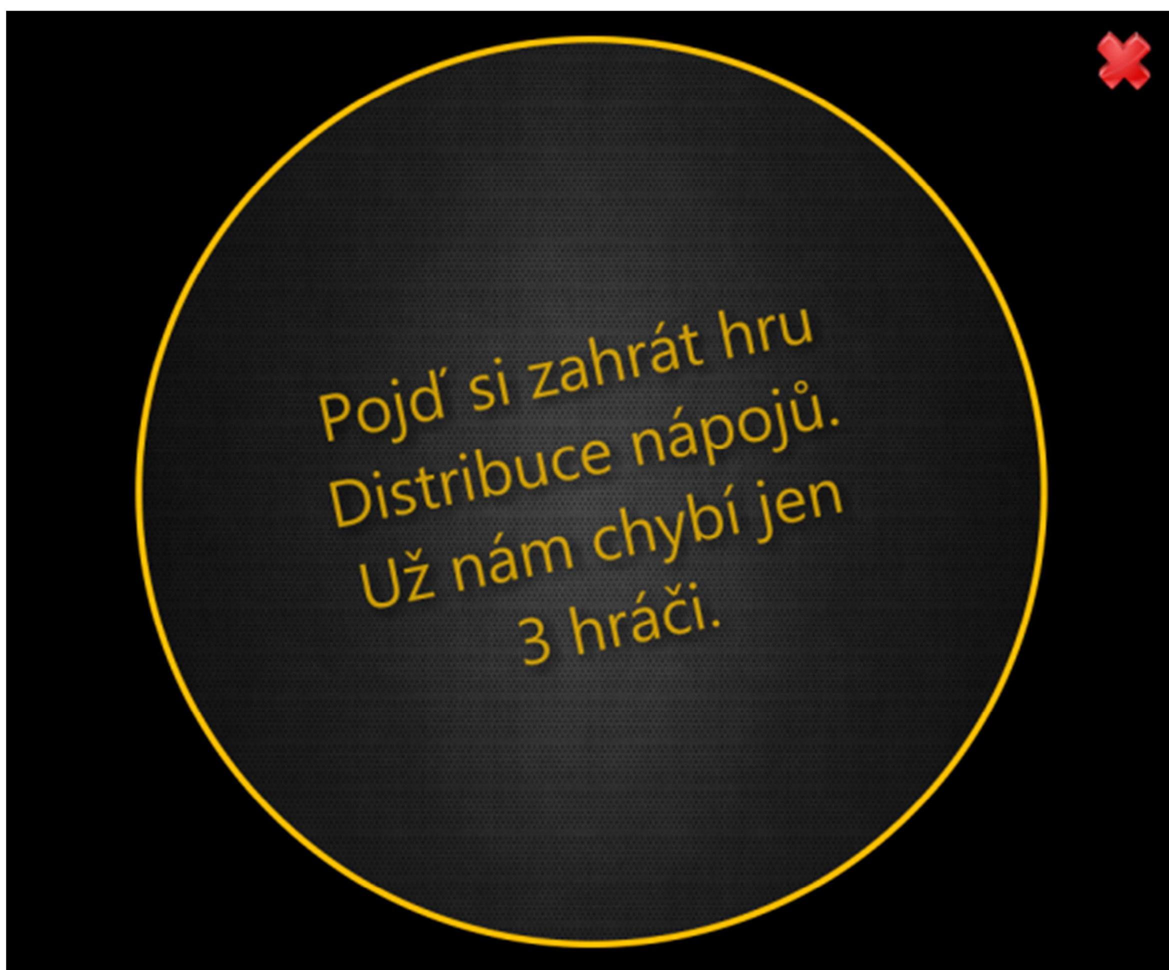
Obr. 18: Obrazovka klientské aplikace ŘS při přechodu na hru využívající projektor, který není zrovna dostupný



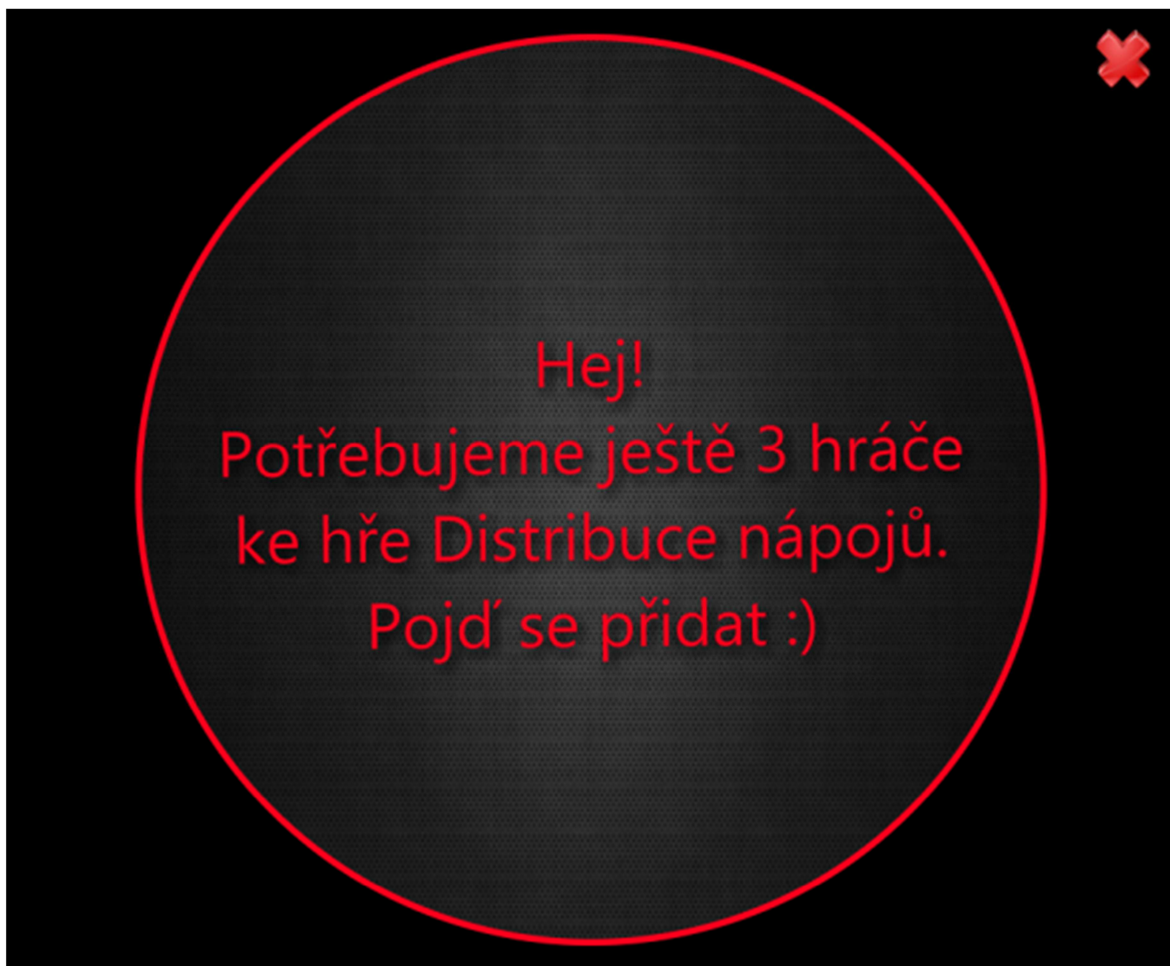
Obr. 19: Obrazovka serverové aplikace ŘS při zobrazené konzoli

Obr. 20: Obrazovka promítaného obrazu s logem VIDA! science centra

Obr. 21: Obrazovka promítaného obrazu s informací o výběru hry

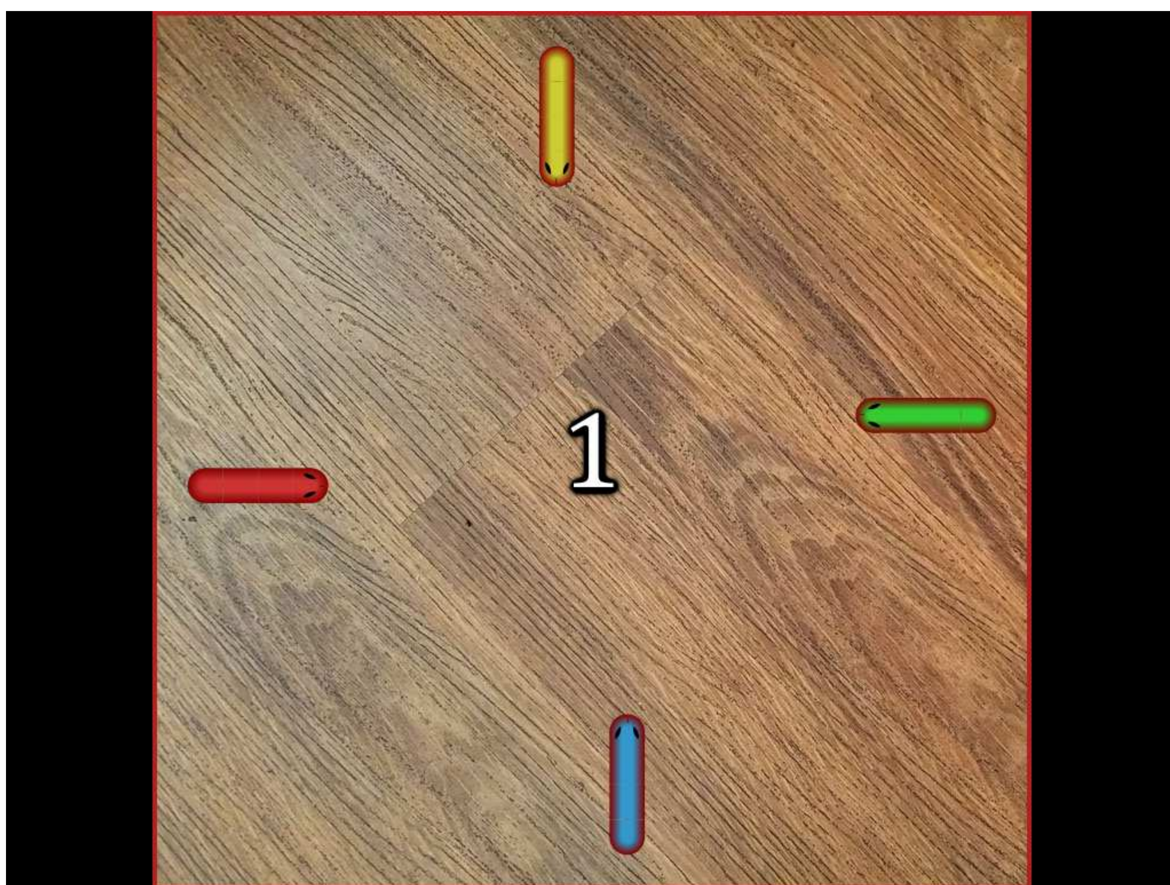


Obr. 22: Obrazovka promítaného obrazu s informací o chybějících hráčích



Obr. 23: Obrazovka promítaného obrazu s další variantou informace o chybějících hráčích

Obr. 24: Obrazovka klientské aplikace hry SnakeXSnake s volbou výběru barvy hada před začátkem hry



Obr. 25: Obrazovka klientské aplikace hry SnakeXSnake při odpočtu do začátku hry

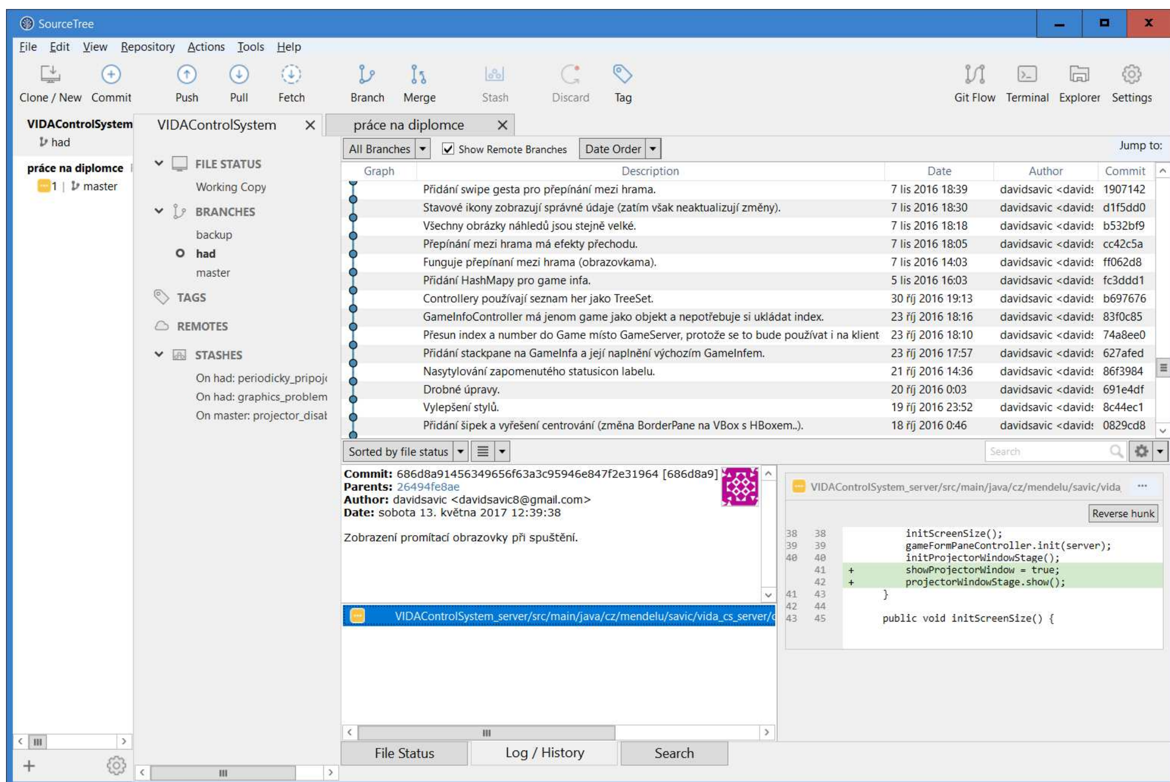
Obr. 26: Obrazovka klientské aplikace hry SnakeXSnake při skončení hry

Obr. 27: Obrazovka klientské aplikace hry SnakeXSnake při skončení hry

Obr. 28: Fotografie celého exponátu Kulatý stůl – stolu s dotykovými obrazovkami a projektoru

Obr. 29: Fotografie exponátu Kulatý stůl při výběru her

Obr. 30: Fotografie exponátu Kulatý stůl při výběru barvy hada po spuštění hry SnakeXSnake



Obr. 31: Snímek obrazovky nástroje SourceTree, ve kterém vidět lze použití Gitu pro verzování nad ŘS a hrou SnakeXSnake

C Ukázka zdrojových kódu vybraných tříd

Metoda run() třídy Client

```
@Override
public void run() {
    String response;

    while (true) {
        try {
            response = input.readLine();
            String[] responseParts = response.split(" ");

            if (response.startsWith("CONNECTED")) {
                int gameIndex = Integer.valueOf(responseParts[1]);
                int connectedPlayers = Integer.valueOf(responseParts[2]);
                games.get(gameIndex)
                        .setConnectedPlayers(connectedPlayers);
                Platform.runLater(()
                        -> this.mainLoader.updateGameInfoController());
            } else if (response.startsWith("ACCEPT")) {
                int gameIndex = Integer.valueOf(responseParts[1]);
                actualGame = games.get(gameIndex);
                mainLoader.hideArrows();
            } else if (response.startsWith("END")) {
                terminateGame();
            } else if (response.startsWith("LAUNCH")) {
                Platform.runLater(() -> mainLoader
                        .updateGUIAfterStartGame());

                int gameIndex = Integer.valueOf(responseParts[1]);
                int gamePort = Integer.valueOf(responseParts[2]);

                String arguments = gamePort + " "
                        + new IPAddressConfig().getIPAddress();
                clientProcess = new GameProcess(
                        games.get(gameIndex).getRunCommandClient(),
                        arguments, () -> finishGame(gameIndex));

                clientProcess.start();

            } else if (response.startsWith("PROJECTOR_IN_USE")) {
                projectorInUse = true;
                mainLoader.updateGUI();
            } else if (response.startsWith("PROJECTOR_NOT_IN_USE")) {
                projectorInUse = false;
                mainLoader.updateGUI();
            } else if (response.startsWith("KILL_YOURSELF")) {
                // close app
                Platform.exit();
                System.exit(0);
            }
        } catch (SocketException e) {
            // server disconnected
            try {
                socket.close();
```

```
            } catch (IOException e1) {
                LogUtils.logException(this, e1);
            }
            return;
        } catch (IOException e) {
            LogUtils.logException(this, e);
        } catch (Exception e) {
            LogUtils.logException(this, e);
        }
    }
}
```