FAKULTA
INFORMAČNÍCH
TECHNOLOGIÍ
ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	2D hra Liero v JavaFX
Student:	Jan Potočiar
Vedoucí:	Ing. Miroslav Balík, Ph.D.
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2018/19

Pokyny pro vypracování

Navhněte a implementujte vlastní verzi 2D hry Liero v prostředí JavaFX, proveďte analýzu konkurence a srovnání.

Požadované vlastnosti hry:
- Bude podporovat hru více hráčů - jak hru dvou hráčů na jednom počítači, tak více hráčů na lokální síti stylem klient/server.
- Protivníci budou řízeni umělou inteligencí nebo člověkem.
- Součástí hry budou alespoň 3 předvytvořené mapy, generátor náhodných map a aplikace s grafickým rozhraním pro vytváření vlastních map.
- Implementovaná jako standalone aplikace, hratelná pod operačními systémy rodin MS Windows a Unix.

Ve svém řešení dbejte na dodržování zásad čistého kódu. Součástí práce budou i jednotkové testy vytvořené ve frameworku JUnit.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

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

V Praze dne 13. února 2018

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra softwarového inženýrství

Bakalářská práce

2D hra Liero v JavaFX

Jan Potočiar

Vedoucí práce: Ing. Miroslav Balík, Ph.D.

10. května 2018

Poděkování

Chtěl bych poděkovat všem, kdo mi pomohli při psaní práce. V prvé řadě svému vedoucímu Ing. Miroslavu Balíkovi, Ph.D. za vedení a cenné potřehy, které mi pomohly při psaní této práce. Dále bych chtěl poděkovat své rodině a přítelkyni za jejich podporu, kterou mi po celou dobu projevovali. Na závěr bych rád také poděkoval autorům knih a článků, ze kterých jsem čerpal, neboť díky nim jsem se v oboru opět posunul o něco dále.

Prohlášení

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

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 10. května 2018 .....

Odkaz na tuto práci

Potočiar, Jan. *2D hra Liero v JavaFX*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2018.

Abstrakt

Cílem práce je navrhnout a implemetovat vlastní verzi kdysi populární hry Liero, která je hratelná pod moderními operačními systémy. Aplikace je napsaná v jazyce Java a její platformě pro tvorbu desktopových aplikací JavaFX. V řešení je kladen důraz na dodržování zásad čistého kódu.

Klíčová slova Liero, 2D hra, JavaFX

Abstract

The goal of this thesis is to design and implement a new Liero like game, which is playable under modern operating systems. The application is written in Java and its platform JavaFX for creating and delivering desktop applications. The source code is written as clean as possible.

Keywords Liero, 2D game, JavaFX

Obsah

Seznam obrázků

Seznam tabulek

Úvod

Pokrok v oblasti technologií jde neustále kupředu, což ovšem znamená, že starší aplikace nemusí být spustitelné na dnešních počítačích, hry nevyjímaje. Nové neznamená vždy lepší, což si uvědomují různá herní studia po celém světě a investují prostředky do udržování starších her, dokonce i vytváření tzv. remaků, tedy her, které mají stejný základ jako některá starší hra, stejné základní mechanismy, liší se pouze v grafické úpravě a podpoře moderních technologií. Jednou ze starších slavných her je i Liero.

Původní Liero spatřilo světlo světa roku 1998 ve Finsku. K originální hře brzy začalo vznikat mnoho dalších klonů, zachovávajících základní charakteristiky hry, ale zároveň i rozšiřujících původní verzi, například ve hraní po síti nebo multiplatformovosti. Popularita Liera v jakékoli formě rychle stoupala, především na území Evropy. Nyní však, o 20 let později, jen málokterá verze hry je spustitelná pod moderními operačními systémy.

Mým cílem je vytvořit úplně novou verzi hry, která je hratelná pod moderními operačními systémy, podporuje hru po síti, tvorbu vlastních map, řízení protivníků umělou inteligencí a mnoho dalšího. Svým způsobem tedy kombinuje to nejlepší, co se za celou dobu v Lieru a jeho klonech objevilo.

V práci se nejprve věnuji analýze herního průmyslu a současného stavu hry, tedy originální verze a jejími vybranými klony. Dále následuje teorie týkající se vývoje softwaru, počínaje specifikací požadavků. Jelikož se při kontrukci aplikace řídím zásadami čistého kódu, popíši jejich obsah a význam v další kapitole. Teoretickou část zakončí kapitola věnující se testování. Následuje praktická část, ve které aplikuji principy z teoretické části, počínaje výčtem funkčních a nefunkčních požadavků na hru a návrhem samotné aplikace. Dále se věnuji zajímavým implementačním detailům hry a testování. Závěrečnou kapitolu tvoří uživatelská dokumentace.

Kapitola 1

Herní průmysl

1.1 Ekonomická síla

Proč se vůbec zajímat o vývoj her? Tržby herního průmyslu rostou v celosvětovém, ale i národním měřítku.

1.1.1 Situace u nás

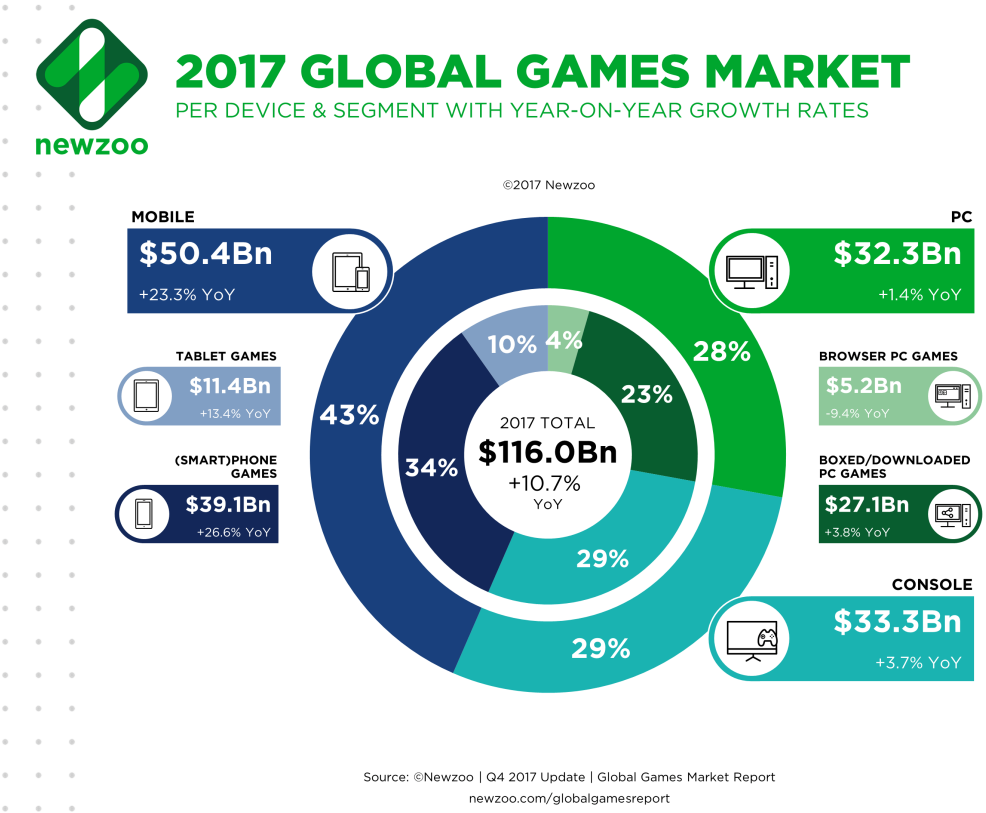
V roce 2013 měl místní herní průmysl obrat 2,5 miliardy Kč [1]. Před pár měsíci[1] byla vydána jedna z nejočekávanějších her tohoto roku Kindom Come: Deliverance od českého herního studia Warhorse Studios, které se za týden po vydání prodalo přes milion kusů [2]. Vývoj her je v naší republice obzvláště významný, jelikož se jedná o největší český kulturní export [3].

1.1.2 Situace a trendy ve světě

Situace ve světě není jiná. Na obrázku 1.1 vidíme, že odhadovaný obrat světového herního průmyslu za rok 2017 činí 116 miliard dolarů. Každým rokem jsou vydávány další herní tituly, s nimi roste popularita her, i obrat celého průmyslu. Hraní her se stává mainstreamovou záležitostí, tudíž se mění i publikum, ale i platformy, které se využívají ke hraní her. V posledních letech pozorujeme, že roste podíl her určených na mobilní zařízení, a pomalu klesá popularita her na PC – konkrétně z obrázku 1.1 můžeme vyčíst, že obrat z her určených na mobilní zařízení tvoří 43% celkového obratu, zatímco z her na PC pouze 28%.

[1] 13.2.2018

Obrázek 1.1: Světový herní průmysl v číslech



1.1.3 Srovnání s filmovým průmyslem

Dlouhou dobu byl filmový průmysl výdělečnější než herní, toto se však v posledních letech začíná měnit. Herní průmysl podle odhadů již dosáhl na obrat filmového, dokonce jej již předehnal. [4] Dříve nebylo výjimkou, že mnohé hry vznikají na motivy úspěšných filmů. Karty se však postupně obrací, a filmový průmysl vnímá rostoucí popularitu her a začíná tvořit filmy podle úspěšných her, což můžeme vidět na titulech jako Tomb Raider či Warcraft.

1.2 Herní žánry neumírají

Starší hry netvoří zapomenutou historii, ale základ dnešního herního průmyslu.

1.2.1 Herní série

Podobně jako u knih a filmů, i u her existují velice úspěšné série. Za zmínku stojí například série GTA , Elder Scrolls, Warcraft, Diablo, Fallout, Starcraft, Call of Duty, FIFA, Far Cry, Unreal a mnohé další. Všechny tyto série jsou stále aktuální, většina z nich tu s námi je již více než 20 let. Tituly v sérii obvykle sdílejí stejné herní prostředí a základní herní mechanismy, liší se pak především po technické a obsahové stránce, např. novější díly navazují dějově na díly starší.

1.2.2 Samostatné tituly

V herní historii nalezneme i pár titulů, které obstály samy o sobě a dodnes se aktivně hrají. Vznikají hry, které mají tak dobrý herní základ, že jim stačí pravidelné dodávky obsahu, v herní branži se takovýmto rozšířením říká datadisky. Vznikají například ke hře World of Warcraft, která si stále drží svou základnu hráčů. Další úspěšnou hrou je Diablo 2, které i dnes existuje bez dodávek obsahu, jelikož má velmi dobrou znovuhratelnost. Že je hra stále hraná dokazuje fakt, že Blizzard k ní vydal patch, díky kterému je možné ji hrát pod moderními operačnímy systémy[5], a to po téměř patnácti letech po vydání původní hry[2].

[2]Diablo 2 bylo vydáno roku 2000, patch vyšel roku 2016

Kapitola 2

Analýza Liera

2.1 Analýza žánru

Původní hra dala vzniknout žánru Liero jako takovému, proto jejímu popisu věnuji tuto kapitolu.

2.1.1 Prostředí hry

Hra je zasazena do podzemí, konkrétně do prostředí sestávajícího se z kamenů, které představují nezničitelné objekty, a hlíny, která tvoří jakousi výplň mapy, která se dá zničit. Do něj jsou umístěni dva hráči v podobě červů, kteří se na příkaz hráče pohybují a bojují proti sobě různými zbraněmi moderního typu, tedy střelnými zbraněmi, výbušninami apod. Kromě zbraní mají červi k dispozici i zachycující se lano, kterým se mohou přitáhnout ke vzdálenějším objektům. V celém prostředí funguje jednoduchá fyzika v podobě gravitace.

2.1.2 Charakteristika červa

Červ se v pojetí Liera může pohybovat vpravo a vlevo, může skákat a používat tzv. ninja rope, kterým se může přitahovat k objektům. Každá žížala má k dispozici 5 zbraní, mezi kterými může libovolně přepínat. Každá ze zbraní má svůj zásobník, který se po vyprázdnění musí naplnit a červ po dobu naplňování nemůže zbraň používat. Zbraní se dá mířit, nicméně jelikož je hra zasazena do 2D prostředí, dá se zaměřovat pouze ve dvou dimenzích. Na závěr se hodí podotknout, že slovo liero označuje v překladu z finštiny žížalu[3].

[3]Ze zoologického hlediska sice červ a žížala označují různé živočichy, nicméně jelikož se pro postavy v tomto herním žánru na našem území ujal pojem červ, budu tato různá označení libovolně zaměňovat.

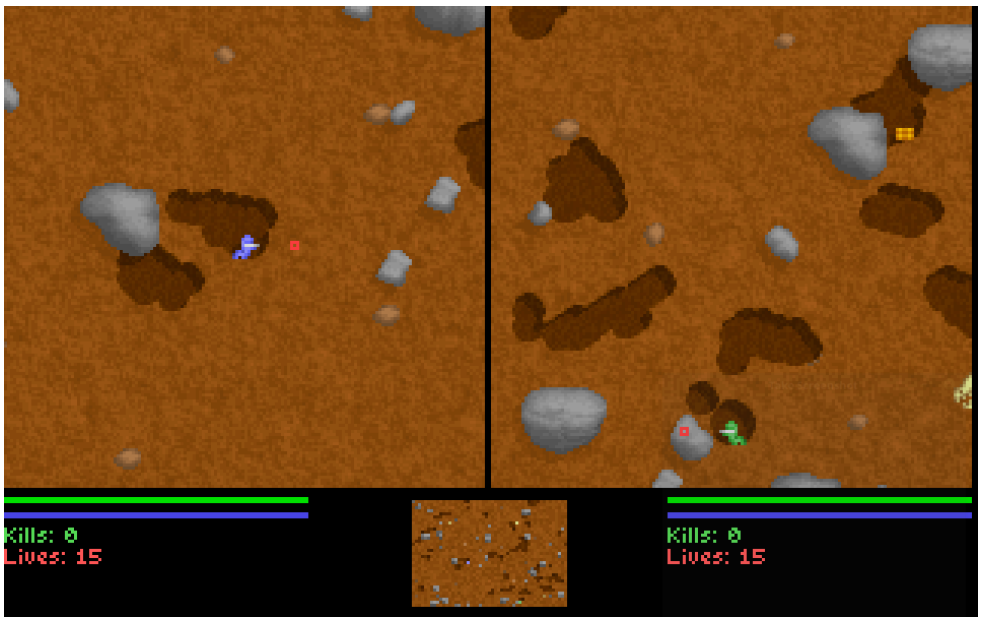
2.1.3 Ovládání hráči

Hra je tvořena pro dva hráče na jednom počítači, sdílí tedy jednu klávesnici i monitor. Každý z hráčů si v nastavení zvolí 7 kláves, kterými bude ovládat svého červa. Jejich význam je následující - dvě klávesy jsou určeny k otočení a pohybu červa do stran, další dvě slouží k míření se zbraní, další klávesa ovládá skákání červa, další slouží k výstřelu ze zbraně, a poslední klávesa slouží k zobrazení názvu červem aktuálně používané zbraně. Existují také speciální významy pro tyto kombinace kláves - pohyb vlevo + pohyb vpravo, který slouží k sežrání hlíny červem, a skok + zobrazení názvu zbraně, která vyvolá vrh lana. Lano se pak dá pustit stiskem klávesy pro skok.

2.1.4 Grafické rozhraní

I když je Liero teprve 20 let starou hrou, svým grafickým provedením připomíná spíše starší hry, hlavně kvůli své „pixelovosti“. Jelikož herní plocha, neboli mapa, je rozsáhlá a nevejde se na celou obrazovku, obrazovka je rozdělena na dvě části, přičemž každý hráč vidí výsek mapy, na kterém se aktuálně nachází. Pod zobrazeným herním prostředím se nachází statistiky jednotlivých hráčů - stav života, zásobníku, počet zabití a zbývající počet životů. Mezi těmito statistikami se nachází zmenšenina mapy, na které je zvýrazněna poloha jednotlivých červů.

Obrázek 2.1: Hra lierolibre, jedna z oficiálních verzí Liera



2.2 Analýza klonů

K původnímu Lieru začaly brzy po jeho vydání vznikat klony, které kopírovaly většinu jeho mechanismů, navíc ale původní hru o něco rozšiřovaly a vylepšovaly.

2.2.1 lierolibre

Projekt, který osvobodil Liero od všech jeho proprietárních komponent a umožnil být svobodně distribuován. [6](překlad vlastní)

Autorem této verze Liera je Martin Erik Werner. Tento klon je, co se mechanik týče, plně věrný originální verzi, neboť jeho základ tvoří originální Liero ve verzi 1.35b. Navíc je snáze modifikovatelný. Umělá inteligence umí chodit a prožírat se mapou. Neumožňuje hru po síti, počet červů na mapě je omezen na dva - buď dva hráči, nebo jeden hráč a jeden bot. Nelze tedy založit hru, kde by byli dva hráči a víc červů ovládaných umělou inteligencí. Aktuálně se jedná o jednu z oficiálních verzí Liera. K dostání i v Ubuntu repozitáři[4].

2.2.2 OpenLieroX

Jeden z nejpopulárnějších klonů Liera, jehož autorem je australský programátor Jason Boettcher. Vznikl v roce 2006 ze hry LieroX, a to zveřejněním zdrojového kódu. Podporuje hru po místní síti, dokonce i hru na Internetu. Má v sobě zabudovaný editor map a různé druhy umělé inteligence. Dokonce i nyní, v dubnu 2018, bylo možné se připojit do rozehrané hry na Internetu. Jediná jeho nevýhoda spočívá v neudržované verzi pro Linux.

2.2.3 LieroAI

Další z populárních klonů, zaměřen na některé nové herní mechaniky, hru po lokální síti a bohatý obsah, včetně nových prvků do mapy, jako je voda či jed. Umělá inteligence se neumí pohybovat, pouze stojí na místě a útočí na ostatní. Klient funkční pod operačními systémy MS Windows i Linux.

[4]https://packages.ubuntu.com/artful/lierolibre

Kapitola 3

Specifikace požadavků

Na počátku každého softwarového projektu je třeba vyjasnit si, k čemu má vlastně projekt sloužit a co by mělo být jeho výstupem. Tomuto procesu se říká specifikace požadavků. Stojí za to mu věnovat pozornost, jelikož chyby způsobené při analýze požadavků jsou zodpovědné za 40–60 procent všech chyb softwarových projektů.[7]

3.1 Smysl požadavků

Požadavky v kontextu softwarových projektů specifikujeme hned z několika důvodů. Tím prvním je, že zjišťujeme, co se od aplikace chce. Ale zároveň i nechce. Z požadavků musí jasně plynout, jakou funkcionlitu bude aplikace poskytovat, a jakou nikoliv. Máme pak jistotu, že zadavatel bude spokojený a my nebudeme pracovat na něčem zbytečném. Specifikace požadavků je pak vstupem pro návrh a implementaci aplikace, ale zároveň i testování, konkrétně akceptační testy.

Pokud vyvíjíme software pro zákazníka, pak vzniklou specifikaci požadavků odsouhlasí obě strany, čímž získáváme právní ochranu a tudíž i jistotu, že požadovaná aplikace má být taková a žádná jiná. Pokud vyvíjíme software pro vlastní potřeby, i zde se nám vyplatí kvalitně provedená analýza požadavků. Lidská paměť není dokonalá a sepsání si požadavků na papír či jiné médium nám pomůže v objasnění si, co vlastně chceme. Díky koncepci takového dokumentu pak není problém provádět jejich revizi a další požadavky jednoduše přidávat.

3.2 Žádoucí vlastnosti požadavků

Úplnost, správnost, proveditelnost, nepostradatelnost, priorita, jednoznačnost a ověřitelnost jsou dobrými vlastnostmi jednotlivých požadavků. Pro specifi-

kaci jako celek je vhodné dodržovat úplnost, jednotnost, přizpůsobitelnost a dohledatelnost.[7]

Jinými slovy, požadavek nesmí nic zatajovat, musí být aktuální, implementovatelný a otestovatelný, ideálně by mu měla být nastavena priorita. Požadavky musí tvořit celek, který si zachovává svou jednotnou formu, je přehledný a poskytuje všechny informace, které potřebujeme k návrhu aplikace, implementaci a uživatelského testování.

3.3 Typy požadavků

- **Funkční požadavky** kladou nároky na funkcionalitu aplikace, tedy co vše má umět a k čemu má být dobrá. Funkční požadavky jsou brány v potaz při tvorbě designu aplikace.
- **Nefunkční požadavky** se již ze samotného názvu zabývají takový nároky na aplikaci, které nejsou popsané ve funkčních požadavcích. Obvykle se týkají výkonu aplikace, škálovatelnosti, udržitelnosti, stabilitou, bezpečností či rozšiřitelností. Nefunkční požadavky jsou obvykle řešeny na úrovni architektury.

Kapitola 4

Čistý kód

Kód může být funkční a čitelný pro kompilátor, ale nemusí být už čitelný pro člověka, udržovatelný, nebo rozšiřitelný. V časech, kdy softwarové projekty jsou velmi rozsáhlé a komplexní a je do nich zapojeno stále více lidí, je třeba dbát na srozumitelnost kódu z lidského hlediska. Proto byl zaveden pojem čistý kód.

4.1 Zásady čistého kódu

4.1.1 DRY

Každý kus znalosti musí mít jednu, jednoznačnou, autoritativní reprezentaci napříč systémem. Alternativou je mít stejnou věc vyjádřenou na dvou, nebo i více místech. Když změníte jedno, musíte si pamatovat, že musíte změnit i ostatní, nebo váš program přestane fungovat. Není otázkou, zda si to budete pamatovat: je otázkou času, kdy zapomenete.[8](překlad vlastní)

DRY, don't repeat yourself, je jedním z nejdůležitějších principů čistého kódu. Duplicitní kód je problematický hned z několika důvodů, z nichž zřejmě nejvýznamnější je popsán výše. Porušení principů DRY se někdy nazývá WET, což obecně zastává výrazy „Write Every Time“, „Write Everything Twice“, „We Enjoy Typing“, či „Waste Everyone's Time“.

4.1.2 GRASP

General Responsibility Assignment Software Patterns, jak již název napovídá, je soubor vzorů, které se zabývají zodpovědnostní a jejím rozdělováním v softwarových projektech.

4.1.2.1 Slabá provázanost (low coupling)

Slabá provázanost je vlastností struktur, zejména modulů a objektů, co nejméně záviset na ostatních strukturách. Prvky, které závisí co nejméně na ostatních

prvcích, vnímáme v menším kontextu, a jsou tudíž lepší na pochopení. Zároveň roste znovupoužitelnost takových prvků a je snažší je modifikovat, neboť existuje málo prvků, které jsou na nich závislé.

„*Princip slabé provázanosti se dá aplikovat na mnoho dimenzí softwarového vývoje; popravdě řečeno je jedním ze základních cílů při budování softwaru.*“[9]

4.1.2.2 Vysoká soudržnost (high cohesion)

Soudržnost je míra různorodosti zodpovědností, které prvek má. Prvky, které mají nízký počet silně souvisejících zodpovědností, mají soudržnost vysokou, narozdíl od prvků, které mají zodpovědností mnoho, a tyto zodpovědnosti se vzájemně velmi liší. Vysoká soudržnost je žádoucí, jelikož prvky s touto vlastností jsou snadnější na pochopení, lépe se udržují, méně často se musí měnit a mají vysokou znovupoužitelnost.[10]

4.1.3 SOLID

SOLID je soubor pěti principů návrhu tříd sestavené Robertem C. Martinem. Dodržování těchto pravidel by mělo vést k větší srozumitelnost, flexibilitě a stabilitě aplikací. Samotý název SOLID je akronynem vzniklý z počátečních písmen oněch pěti principů.

4.1.3.1 Single responsibility principle

Třída by měla mít jeden, jeden jediný, důvod ke změně.[11](překlad vlastní) Princip jedné odpovědnosti představuje pravidlo o vysoké soudržnosti dohnané do extrému. Dá se aplikovat na třídy, ale i funkce a moduly.

4.1.3.2 Open/closed principle

Třída by měla být otevřená vůči rozšířením, ale uzavřená vůči modifikacím.

„*Klíčem k aplikaci tohoto principu je především abstrakce a polymorfismus. Třídy by tedy měly být vytvářeny tak, že z nich abstrahujeme společnou funkčnost a konkrétní implementační detaily, které se mohou měnit, definujeme až v odvozených třídách. Při potřebě vytvořit nový typ třídy s jinou funkčností pak můžeme pouze vytvořit nového potomka abstraktní třídy s vlastní implementací a nemusíme zasahovat do implementace dalších konkrétních tříd.*“[12]

4.1.3.3 Liskov substitution principle

Princip zabývající se třídami a jejich podtřídami. Říká, že instance třídy by měla být v kódu zaměnitelná s libovolnou instancí svých podtříd, bez změny funkčnosti.

4.1.3.4 Interface segregation principle

Je jedním ze způsobů zvýšení soudržnosti rozhraní. V podstatě říká, že namísto jednoho rozsáhlého rozhraní s nízkou soudržností, které využívá různě mnoho klientů, je lepší vytvořit několik menších rozhraní. Taková rozhraní jsou lépe udržovatelná a obecně se s nimi díky vysoké soudržnosti lépe manipuluje.

4.1.3.5 Dependency inversion principle

Prvky s vyšší abstrakcí by nikdy neměly záviset na prvcích s nižší abstrakcí. Obecně by jakékoli prvky měly záviset na co nejabstraktnějším rozhraní, které bývá neměnné. Pokud je totiž třída závislá na konkrétní implementaci, která se často mění, přenáší se tato nestabilita i na závislou třídu. Dodržování toho principu zvyšuje stabilitu tříd.

4.2 Refaktorizace

Refaktorizací rozumíme takové činnosti, v rámci kterých dojde k takovým změnám v kódu, které mění jeho strukturu, neovlivňují ale jeho vnější funkcionalitu. Nedochází k žádným opravám chyb, ani žádným jiným změnám v chování programu. Náš uživatel si žádného rozdílu nevšimne.

4.2.1 Proč refaktorovat

K čemu to tedy je? Refaktorizace je tu pouze pro nás, programátory. Kód, který projde refaktorizací, by měl být ve výsledku čistší, čitelnější a mělo by se nám s ním, jako vývojářům, lépe pracovat. Refaktorizace tedy stojí nějaký čas, ovšem čas vyložený na refaktorizaci se nám jistě vrátí, až se k danému kusu kódu opět dostaneme my, nebo někdo jiný, v podobě rychlejšího pochopení, jak kód funguje.

4.2.2 Co a jak refaktorovat

Během psaní našeho kódu bychom si měli dát pozor na různé programátorské neduhy:

4.2.2.1 Duplicitní kód

Duplicitní kód je jedním z nejběžnejších programátorských prohřešků typický zejména pro začátečníky. Pokud máme v programu dva, nebo dokonce více identických bloků kódu ve stejném modulu, je to vážná chyba. Pokud potřebujeme takový kód pozměnit, musíme tuto změnu provést úplně na všech místech užití. To je jednak pracné, jednak můžeme zapomenout tuto změnu provést na skutečně všech místech. Ideálním řešením je pro tento blok kódu

vymyslet jméno a používat jako funkci, kterou budeme volat na původních místech užití.

```
void doMorningRoutine() {
   feedScratchy();
   feedGarfield();
   feedTom();
   makeBreakfast();
}

void doEveningRoutine() {
   feedScratchy();
   feedGarfield();
   feedTom();
   makeDinner()
}

/* Co k sobe patri dame do jedne funkce */

void feedCats() {
   feedScratchy();
   feedGarfield();
   feedTom();
}

void doMorningRoutine() {
   feedCats();
   makeBreakfast();
}

void doEveningRoutine() {
   feedCats();
   makeDinner()
}
```

4.2.2.2 Příliš dlouhá funkce

Pokud je funce delší než jedna obrazovka, stojí za zvážení, zda ji nerozdělit na několik menších, vhodně pojmenovaných. Krátké funkce jsou přehlednější, soudržnější a lépe testovatelné než delší. Nemusíme se bát ani velmi krátkých funkcí, Samer Buna dokonce doporučuje, že by žádná funkce neměla být delší než 10 řádků.[13]

4.2.2.3 Související data nejsou ve společné třídě

Jestliže používáme některá data vždy společně, stojí za zvážení, zda pro ně nevytvořit jednu třídu. Typickým příkladem jsou například položky *Jméno* a

Příjmení, pro které můžeme vytvořit třídu *Osoba*. Jiným příkladem mohou být souřadnice x, y a z, které mohou tvořit třídu *Souřadnice*.

4.2.2.4 Příliš dlouhý seznam parametrů

Ideální počet parametrů funkce je nula. Další je jeden, krátce následovaný dvěma. Třem argumentům by se mělo vyhýbat, kdykoli je to možné. Více než tři parametry vyžadují velmi speciální opodstatnění – a pak by stejně neměly být použity[14](překlad vlastní).

Toto může znít extrémně, nicméně pokud se zároveň dodržují dvě výše zmíněná pravidla, jedná se o jejich logický důsledek. Robert C. Martin dále argumentuje, že nízký počet parametrů zvyšuje srozumitelnost funkce a zároveň značně ulehčuje testování.

4.2.2.5 Chabá pojmenování

Proměnné, funkce, třídy, moduly - to vše by mělo být korektně pojmenované. Proměnná podle toho, co obsahuje za hodnotu. Funkce podle toho, co dělá.

```
MyClass myClass = new MyClass(); // K cemu trida slouzi? Vubec nevime
MyClass2 tmp = doSomething(); // Co obsahuje promenna, co dela funkce?
myClass.addObject(tmp); // Co dela metoda?

/* Pojmenovavejme vse korektne */

PersonDatabase personDatabase = new PersonDatabase();
Person supervisor = getSupervisor();
personDatabase.addPerson(supervisor);
```

Přijít na přesné pojmenování dané jednotky chce jistou kreativitu, nicméně pokud nad jménem přemýšlíme dlouho a nemůžeme přijít na vhodné jméno, stojí za zamyšlení, zda se daná jednotka stará opravdu jen a pouze o jednu věc.

4.2.2.6 Magické hodnoty

Někdy je dobré pojmenovat i je - tedy vytvořit pojmenovanou konstantu, které přiřadíme danou hodnotu a v kódu ji dál používáme pod svým jménem. To vede jednak k menší duplicitě kódu, jednak daný kód začne dávat větší smysl.

Dobré pravidlo ověřené praxí říká, že jediná čísla, která by se měla vyskytovat v těle programu jsou 0 a 1. Jakákoli jiná čísla by měla být zastoupena něčím více popisným.[15] (překlad vlastní)

```
for (int i = 1; i <= 12; i++) { /* 12? mesicu v roce? podlazi v
    budove? delka seznamu? */
   sum += collection.at(i).value();
}
```

4.2.2.7 Nedodržování konvencí

Každý jazyk má nějaké své konvence, které by se měly dodržovat, stejně tak tým vývojářů by si měl nějaké konvce určit a dodržovat, případně i vývojář sám pro sebe. Konvence vedou k jednotnosti kódu, která vede k efektivnější práci v týmu, rovněž i k efektivnější práci programátora samotného. Jakmile jsou konvence určené, nemusíme činit tolik rozhodnutí jak co psát - mnoho věcí už máme určené a zažité, a tím pádem šetříme svou mentální kapacitu, kterou můžeme využít pro řešení důležitějších problémů.

```
int managerID = getManager();
int supervisorId = getSupervisor();
int testerid = getTester();
...
String managerName =
   personDatabase.getNameById(/* managerid? managerId? managerID? */);

/* camelCase patri k zakladnim konvencim, je spolecna mnoha jazykum */

int managerId = getManager();
int supervisorId = getSupervisor();
int testerId = getTester();
...
String managerName =
   personDatabase.getNameById(managerId); // nyni bez pochyb vime
```

4.2.3 Kdy refaktorovat

Je nutné si uvědomit, že refaktorizace je proces, se kterým se každý programátor setkává velmi často, aniž by o tom třeba tušil. Za refaktorizaci lze považovat rozdělení jedné velké třídy do několika menších, ale stejně tak i prostá změna pojmenování proměnné. Jelikož se většinou jedná o malé změny, není problém je provádět průběžně během psaní kódu, tedy kdykoliv vidíme prostor pro zlepšení kvality kódu. Během průběžné kontroly kódu většinou opravíme menší záležitosti. Na ty větší je dobré se podívat například po dopsání celé třídy, kdy zvážíme, zda struktura dané třídy skutečně odpovídá tomu, jak byla zadána a jak ji chápeme.

Kapitola 5

Teorie testování

Programové testování je velmi důležitou součástí jakéhokoli netriviálního programu, větší projekt se bez něj neobejde. Testování má dvě nesporné výhody. Tou hlavní je, že zjistíme, zda výsledný program skutečně odpovídá specifikaci požadavků, tedy funguje jak má a jak očekáváme. Dále se při refaktorizaci nemusíme tolik bát, že něco pokazíme, neboť pokud se něco pokazí, kvalitní testy nás na to upozorní.

5.1 Typy testů

5.1.1 Dělení dle znalosti implementace

- **Black box** testy píšeme bez ohledu na vnitřní implementaci, testujeme pouze dané rozhraní.

- **White box** testy píšeme s přihlédnutím na implementaci testovaného kódu, typicky řešíme mezní hodnoty funkcí nebo korektní vyhazování výjimek.

5.1.2 Dělení dle velikosti testovaného celku

- **Unit (jednotkové) testy** kontrolují správnost nejmenších logických celků, tedy funkčnost jednotlivých objektů a jejich metod. Objekt by měl být v tomto případě od ostatních objektů izolován. Toho je dosaženo používáním tzv. stubů, objektů, které mají předdefinované chování pro účely testování, a nahrazují tak objekty reálně používané v produkčním kódu.

- **Integrační testy** předpokládají existenci unit testů a testují, jak samostatně otestované objekty dokážou společně fungovat.

5.2 F.I.R.S.T. – pravidla pro psaní testů

F.I.R.S.T. je dalším akronymem z tvorby Roberta C. Martina[14], který je tvořen prvními písmeny následujících pěti principů.

- **Fast** (rychlé) – testy nesmí být pomalé, neboť je potřebujeme spouštět často.
- **Independent** (nezávislé) – testy na sobě musí být nezávislé, aby šly spouštět samostatně a lépe se hledaly chyby.
- **Repeatable** (opakovatelné) – testy musí být napsané tak, aby se daly spouštět opakovaně, kdykoliv a kdekoliv, a vracely stejné výsledky.
- **Self-validating** (samy se vyhodnocují) – testy by měly skončit buď úspěchem, nebo neúspěchem, aby byly výsledky jednoduše interpretovatelné.
- **Timely** (správně načasované) – testy by měly být napsané co nejdříve.

5.3 Test driven development

Test driven development, neboli vývoj řízený testy, je jedním z prvků agilního přístupu k vývoji softwaru. Vývojář při takovémto stylu vývoje píše, především jednotkové, testy paralelně s implementací. Při psaní testů tedy již musíme znát, nebo dotváříme rozhraní dané třídy či funkce. Psaní testů před samotnou implementací nás vede ke psaní lépe otestovatelných rozhraní, a tudíž i čistších.

5.3.1 Tři pravidla TDD

Svou představu TDD definoval Robert C. Martin v následujících pravidlech [14] (překlad vlastní)

1. Nemůžete psát produkční kód, dokud jste nenapsali selhávající jednotkový test.
2. Nemůžete psát více jednotkových testů, než kolik jich stačí, aby některý selhal, nebo se nedal zkompilovat.
3. Nemůžete psát více produkčního kódu než je zapotřebí, aby prošly aktuálně selhávající testy.

Tato tři pravidla programátora udržují v jakémsi cyklu, ve kterém píše paralelně produkční a testovací kód, přičemž testovací vždy předchází produkční.

5.3.2 Analogie s podvojným účetnictvím

TDD přirovnává Robert C. Martin[16] k podvojnému účetnictví. Argumentuje, že účetnictví se velmi podobá vývoji softwaru, jelikož oba obory zacházejí s komplexními dokumenty, které jsou plné „tajemných symbolů“, a musí být naprosto korektní, neboť i malá chyba může vést k velkým problémům. Dále kdykoliv je potřeba do takového dokumentu něco přidat, je nutné poté zkontrolovat, že kvůli tomuto zásahu nenastala chyba nikde jinde.

Hledání analogií s jinými obory není u softwarového inženýrství nic neobvyklého. Jedná se o velmi mladý obor, který ještě není tak dobře prozkoumaný a popsaný, ale přitom velmi rychle narůstá na důležitosti. Proto bývají jednotlivé části vývoje softwaru připodobňované něčemu, co jako lidstvo dlouho známe, máme to více prozkoumané, a ostatně pro lidskou mysl je to i lépe uchopitelné, než mnohdy velmi abstraktní pojmy.

Kapitola 6

Analýza a sběr požadavků

6.1 Funkční požadavky

F1 Grafické prostředí hry

- F1.1 Aplikace musí mít pevné rozlišení 800x600 pixelů.
- F1.2 Aplikaci musí být možno spustit jak v okně, tak přes celou obrazovku, se zachováním poměru rozměrů.
- F1.3 Celé prostředí musí mít „nostalgický nádech“, a to konkrétně v podobě jisté „pixelovosti“, tedy nízkých detailů a ostrých hran.

F2 Hra více hráčů

- F2.1 Hra musí podporovat hru dvou hráčů sdílející jednu klávesnici.
- F2.2 Hra musí podporovat hru více hráčů na lokální síti. Jeden z hráčů zakládá hru a ostatní hráči se připojují. Jedna hra po síti pojme až 8 různých hráčů.

F3 Herní mapy

- F3.1 K dispozici bude generátor náhodných map.
- F3.2 K samotné hře bude dodávána samostatná aplikace na tvorbu vlastních map.
- F3.3 Dostupné minimálně 3 pevné předvytvořené mapy.

F4 Umělá inteligence

- F4.1 Protivníci budou řízení umělou inteligencí – minimálně budou stát na místě a útočit na ostatní červy, ideálně se budou umět pohybovat.

F5 Nastavení kláves

F5.1 Při prvním spuštění bude nastaveno výchozí nastavení kláves.

F5.2 Nastavení kláves musí být nastavitelné uživatelem přímo ve hře.

F5.3 Pokud uživatel změní nastavení kláves, musí se toto nové nastavení uložit a automaticky se nastavit při každém dalším spuštění hry.

6.2 Nefunkční požadavky

NF1 Hra bude v anglickém jazyce, jako je tomu u her v tomto žánru zvykem.

NF2 Hra musí být hratelná pod operačními systémy MS Windows (verze 7, 8, 10).

NF3 Hra musí být hratelná pod operačními systémy rodiny Unix.

6.3 Výběr technologií

Ve výběru technologií jsem vzal v potaz především podporu multiplatformovosti, výkonnost, rozšířenost a výhledy dané technologie do budoucnosti, tedy na to, zda tu s námi bude i za další jednotky, možná i desítky let. Všechny tyto požadavky splňuje Java se svým rozhraním JavaFX.

6.4 Analýza konkurence

U vybraných verzí Liera jsem porovnal, které klíčové funkcionality obsahují. Svou verzi Liera jsem nazval Liero FX.

Tabulka 6.1: Analýza konkurece

	NF2	NF3	F3.2	F2.2	F4
LieroLibre	✓	✓	X	X	✓
OpenLieroX	✓	X	✓	✓	✓
Liero AI	✓	✓	X	✓	✓
Liero FX	✓	✓	✓	✓	✓

Musím podotknout, že OpenLieroX oficiálně podporuje i linuxové distribuce. Nicméně když jsem se aplikaci pokoušel spustit na svém linuxovém stroji, pokus skončil neúspěchem, stejně jako manuální kompilace, která vyžadovala příliš mnoho závislostí, a to mnohdy velmi starých, které se mi nepovedlo ani po hodině najít a doinstalovat. Jelikož se tedy jedná o verzi neudržovanou a v mém případě ani po značném úsilí nespustitelnou, udělil jsem hře v dané kategorii hodnocení „nesplněno“.

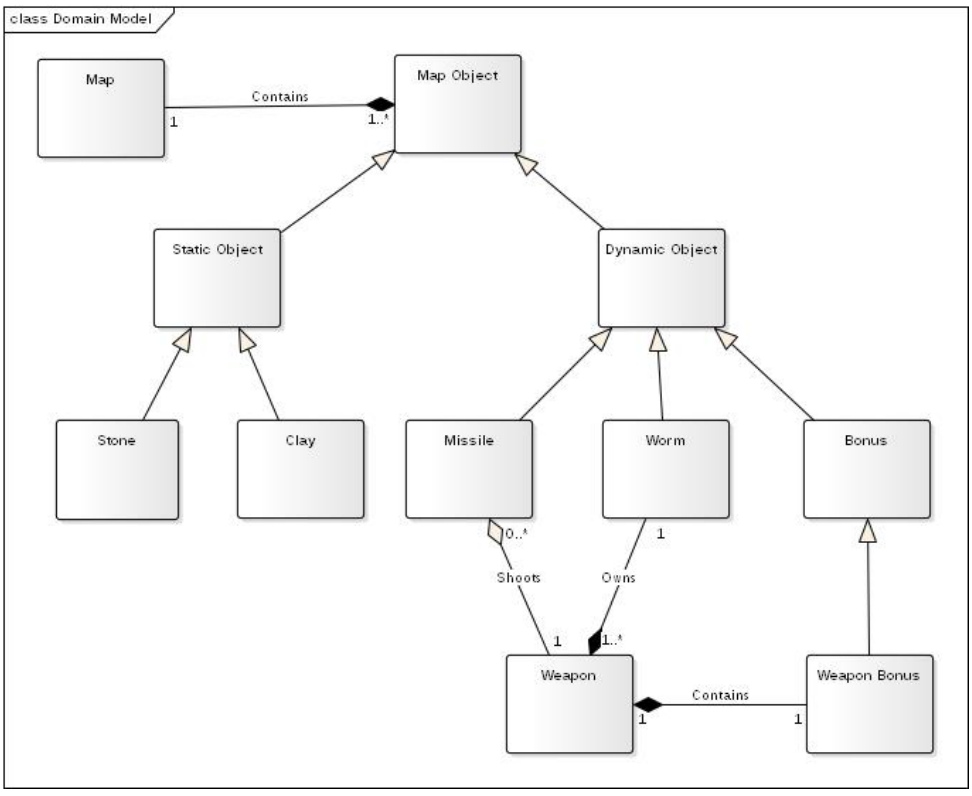
Kapitola 7

Architektura a design

7.1 Liero FX

7.1.1 Doménový model

Obrázek 7.1: Doménový model



Obrázek 7.1 představuje základní doménový model. Každá mapa obsahuje mnoho objektů, které můžou být buď statické, nebo dynamické. Ty se od sebe

liší tím, že statické se nepohybují, zatímco dynamické ano. Statické objekty mohou být dvojího druhu – buď hlína, která může být zničena, nebo kámen, který nikoliv. Dynamické objekty se pak dělí na střely, červy a bonusy. Červ může vlastnit několik zbraní, zbraně mohou střílet mnoho střel. Objekty typu *Weapon Bonus* obsahují právě jednu zbraň.

7.1.2 Architektura hry

Architektura aplikace je založena na architektuře model-view-controller. Model v našem případě představují například třídy z doménového diagramu výše, view vrstvu obstarává JavaFX rozhraní, konkrétně ve formě tří objektů typu *Canvas*, které mají stejnou velikost a vzájemně se překrývají. Nejspodnější vrstva tvoří pozadí aktivní hry, dále následuje vrstva, na kterou se vykreslují statické objekty, kterou překrývá vrstva, na kterou se vykreslují dynamické objekty. Platí, že nejspodnější vrstva se nikdy nepřekresluje, nejvrchnější se překresluje každý snímek a prostřední se překresluje, když je třeba. Controller vrstvu obstarávají podtřídy třídy *Context*, které zpracovávají vstup uživatele.

7.1.3 Design hry

Ve svém designu jsem častokrát definoval rozhraní v abstraktních třídách, které obsahují i základní chování společné všem myslitelným potomkům. Potomci pak implementovali všechny abstraktní metody. Tím bylo zajištěno dodržování *DRY* principu a hojného užití polymorfismu. Ze známých návrhových vzorů je použil například návrhový vzor *singleton* pro různé podtřídy typu *Context*.

7.1.4 Hra po síti

Hra po síti je realizovaná stylem client – server.

Klient je v tomto případě „tlustý“, obstarává veškerou logiku aplikace. Každý snímek je zpracován vstup z klávesnice, příslušně přeložen na příkazy červům (typu vystřel, vyskoč atd.) a poslán serveru. Jakmile přijde odpověď serveru, klient ví, jak se který člověkem řízený červ má pohnout.

Server je v tomto případě „tenký“ a jeho jediná funkce spočívá v tom, že za každý snímek sebere příkazy od všech klientů, vytvoří z nich kolekci a tuto kolekci pošle zpět všem klientům. Jelikož jsou ve hře některé komponenty řízeny náhodně, je k této kolekci příkazů přiloženo náhodně generované číslo. Každý klient daný snímek pak operuje s tímto číslem. Tím je dosaženo deterministického chování klientů, neboť každý klient daný snímek operuje se stejným vstupem.

Výsledná komunikace je na data nenáročná, neboť se přenáší pouze to, co je nezbytně nutné. Ve svém řešení jsem vzal v úvahu i variantu s tlustým serverem a tenkým klientem, nicméně při 60 snímcích za sekundu a několika

klientech by se muselo přenášet velmi často velké množství dat, a pokud by síť neměla dostatečnou propustnost, byl by to problém.

7.2 Editor map

7.2.1 Návrh aplikace

Aplikace je navržena jako FXML aplikace, tedy aplikace v platformě JavaFX, která má vzhled a názvy událostí (events) popsané v souboru *FXMLDocument.fxml*.

Editor map je ve výchozím nastavení oddělen od samotné hry, což může být z uživatelského hlediska nepříjemné, jelikož pokud si chce uživatel zahrát vytvořenou mapu, je třeba ji zkopírovat do adresáře hry, nicméně oddělenost těchto dvou aplikací má i své výhody, například může tímto způsobem jasně oddělovat hotové a rozpracované mapy. Pokud se uživatel rozhodne, že chce mít složku *maps* sdílenou oběmi aplikacemi, je to možné, a to prostým zkopírováním Editoru (jar a ostatních pomocných souborů) do kořenového adresáře hry.

7.2.2 Formát mapy

Jako formát mapy jsem zvolil binární soubor s následujícím obsahem. Nejprve obsahuje barvu všech pixelů mapy, tedy celkem 800x600 položek, vyjádřených pomocí tří bajtů, zastupující barvy červenou, zelenou a modrou. Poté následuje seznam speciálních objektů vyjádřených jako: ID typu objektu (byte), souřednice (x a y - obojí integer) a případné další parametry objektu, které se vážou k danému typu.

Binární formu souboru jsem zvolil proto, že jsem chtěl mít všechny informace o mapě pohromadě v jednom souboru, aby se dal dobře sdílet. Zároveň jsem chtěl mít obraz mapy naprosto přesný, bez jakékoli komprese, takže jsem potřeboval zachytit všech 800x600 pixelů. Podíl barvy se dá v binární podobě zachytit jedním bytem, zatímco v textové jsou na ni potřeba dva bajty (zapsáno hexadecimálně). Binární formát je tedy zvolen i jako úspornější varianta.

Kapitola 8

Implementace

8.1 Generátory map

V aplikaci jsou k dispozici dva různé generátory map.

8.1.1 Generátor pevných map

Tento generátor dostane na vstupu mapu, která byla vytvořena v přiložené aplikaci na tvorbu vlastních map.

8.1.2 Generátor náhodných map

Někdy nechceme mít pevnou mapu, ale chceme se nechat počítačem překvapit, co pro nás vymyslí. Tento účel plní popisovaný generátor.

Samotná generace náhodné mapy sestává ze dvou základních kroků:

1. Vyplnění celé mapy hlínou.
2. Umístění pevného počtu různě tvarovaných kamenů.

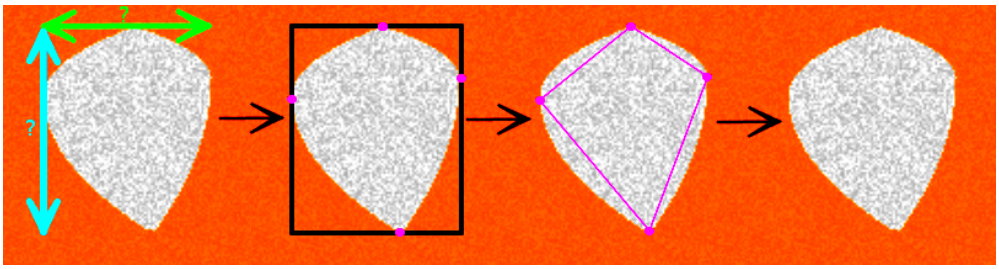
Podrobněji popíšu druhý krok.

8.1.2.1 Modelování kamenů

Poté, co je určen počet kamenů, které do mapy umístíme, se každý z nich vymodeluje následujícím způsobem (viz obrázek 8.1 pro snažší pochopení procesu).

1. Náhodné určení výšky a šířky kamene v zadaných mezích.
2. Náhodný výběr 4 bodů v zadaných mezích, každého na jedné straně „obalujícího obdélníku“. Zadanými mezemi mám na mysli to, že tento náhodný bod se musí nacházet poblíž středu strany, konkrétně nesmí mít od středu strany dál, než ke konci strany.

Obrázek 8.1: Modelování kamene



3. Vytvoření propojení mezi těmito 4 body tak, aby byl každý spojen s bode ze sousedních stran. Toto propojení nemůže být přímé (úsečkou), protože pak by nám vznikly hranaté kameny. Stěny výsledného kamenu chceme mít oblé a konkvexní, proto pomyslné úsečky propojující sousední body kamene „ohneme“ směrem ven.

4. Vyplnění vzniklého útvaru.

8.2 Umělá inteligence

Protivníci řízení umělou inteligencí mají jednoduché a přímočaré chování, skládající se ze tří kroků.

1. Spočítat vzdálenosti od všech červů na mapě a najít nejbližšího.

2. Zamířit na něj.

3. Vypálit.

Umělá inteligence se tedy nehýbe, stojí na místě, stejně jako tomu je u hry Liero AI. Na první pohled to může vypadat jako nedostatek, nicméně v reálné hře to tolik nevadí. Absence pohybu totiž z pohledu obtížnosti poražení protivníka vyvažuje fakt, že umělá inteligence míří absolutně přesně.

8.3 Fyzika a gravitace

Veškerá fyzika a pohyb objektů v aplikaci je řešena skrze třídu *VectorOfMovement*, jejíž instanci obsahuje každý dynamický objekt. Tento vektor je tvořen dvěmi souřadnicemi (x a y) a má velmi jednoduchou interpretaci. Určuje totiž, jak se má daný objekt posunout mezi dvěma snímky[5] hry.

Gravitace, stejně jako každý jiný pohyb s objektem, je tedy řešena skrze tento vektor, který se požadovaně mění.

[5]Hra má 60 snímků za sekundu

8.4 Pohyb po kružnici

Několikrát v aplikaci je třeba popsat pohyb po kružnici, například při vykreslování zaměření zbraně, či při destrukci bonusu obsahující zbraň, kde má daná zbraň vystřelit do všech směrů (což se dá popsat skrze různé, stejně vzdálené body na kružnici). Toto je kód sloužící k vystřelení ze zbraně do všech možných směrů.

```
private final Weapon WEAPON;
private static final int SHOTS_ON_DESTROY = 10;
private static final int VECTOR_SIZE = 20;
 //...
public void actionOnDestroy() {
   for (int i = 0; i < SHOTS_ON_DESTROY; i++) {
       double fractionOfTheWhole = (double) i / SHOTS_ON_DESTROY;
       double currentAngleInRadians = fractionOfTheWhole *
           (Math.PI * 2);
       VectorOfMovement vec = new VectorOfMovement();
       vec.setX(VECTOR_SIZE * Math.sin(currentAngleInRadians));
       vec.setY(VECTOR_SIZE * Math.cos(currentAngleInRadians));
       Map map = ActiveGameContext.getInstance().getMap();
       WEAPON.absoluteReload();
       WEAPON.trigger(map, vec, coordinates);
   }
}
```

Pohyb po kružnici jsem elegantně vyřešil pomocí goniometrických funcí sinus a cosinus. Výsledná metoda je závislá pouze na třídní proměnné *SHOTS_ON_DESTROY*, která se dá změnit dle potřeby, metoda se jí přizpůsobí.

8.5 Detekce kolizí

Při pohybu dynamických objektů je potřeba ohlídat kolize. To je řešeno jednoduchým, intuitivním algoritmem. Ve fázi hry, kde se posouvají dynamické objekty (aplikováním instance *VectorOfMovement* výše popsaným), se zkonstruuje dráha, kterou daný objekt za snímek urazí. Následně se zkontroluje, zda na dané dráze existuje objekt, se kterým by daný objekt mohl kolidovat (ne všechny objekty kolidují se všemi, např. instance *Missile* nekolidují mezi sebou).

Kapitola 9

Testování

9.1 Programové testování

Aplikace obsahuje jednotkové a některé integrační testy vytvořené ve frameworku JUnit. Testy jsem psal většinou až po implementaci dané funcionality. Když jsem se ale (bohužel relativně pozdě) dozvěděl, jak správně aplikovat TDD, vyzkoušel jsem ho a skutečně se mi zdálo, že jsem v rámci tohoto procesu tvořil čistší produkční kód. K efektivní aplikaci TDD bych ale potřeboval mnohem více zkušeností.

9.2 Uživatelské testování

Aplikace byla testovaná několika uživateli na různých strojích, lišící se výkonností, operačním systémem, i rozlišením monitoru. Jmenovitě mezi testovanými operačními systémy byly MS Windows 7 a 10, Linux Mint 17 a 18, Ubuntu 16 a Gentoo. Mezi testovanými rozlišeními byly 1200x900, 1366x768 a 1920x1080.

Při testování se narazilo na několik menších chyb. Fullscreen mod na noteboocích o rozměrech 15.6 palců a s rozlišením 1366x768 nebyl plně centrovaný, obraz byl posunut vlevo od středu. Na slabších počítačích se hra s velkým množstvím červů (více než 20 červů řízených umělou inteligencí) „mírně sekala“, jako problém bylo identifikováno ozvučení hry, které by mohlo být implementováno lépe. Na silnějších počítačích byla i extrémní hra se 100 červy plynulá.

Během testování přišlo ze strany uživatelů mnoho připomínek, se kterými jsem mnohdy souhlasil, týkalo se to především různých menších chyb a návrhů na změny. V kódu jsem rychle našel místo, které bylo potřeba upravit, abych danou funkcionalitu změnil. Vždy stačilo upravit jen pár řádků. Tato skutečnost mě ujistila, že hra je opravdu dobře navržená, neboť její údržba byla velmi snadná.

Kapitola 10

Uživatelská dokumentace

Pro spuštění aplikace je třeba mít v operačním systému JRE minimálně ve verzi 8, která podporuje platformu JavaFX. Ke stažení například zde: https://java.com/en/download/.

10.1 Liero FX

10.1.1 Prostředí aplikace

Aplikace má dáno rozlišení 800x600 pixelů a je možné ji spustit jak v okně, tak ve fullscreen modu. Stačí z hlavního menu zajít do *Settings*, a zde kliknout na *Change screen mode*.

Ve hře se vůbec nepoužívá myš, veškeré ovládání je řízeno klávesnicí. V menu se dá pohybovat kurzorovými šipkami, pro výběr položky se používá klávesa *ENTER*, pro návrat / pauzu aktivní hry se používá klávesa *ESCAPE*.

10.1.2 Lokální hra

Pokud si chcete zahrát sami proti umělé inteligenci, či ve dvojici na jednom počítači, zvolte lokální hru. Zvolte si mapu a můžete začít hrát.

10.1.3 Hra po síti

Liero je možné hrát na místní síti společně s ostatními hráči. Pro spojení je nutné, aby jeden z hráčů „založil hru“, tedy v menu *Network Game* zvolil *Host Game*, vybral mapu a nastavil ostatní parametry hry. Ostatní hráči vyberou v menu *Network Game* naopak *Join Game*, a pokud nikde není žádný problém, automaticky by se měla nalézt založená hra. Až se připojí všichni hráči, zakladatel hru zahájí.

10.1.4 Ovládání červa

10.1.4.1 Výchozí nastavení

1. Výchozí nastavení kláves pro levého hráče je kombinace R + F pro zaměření nahoru / dolů, D + G pro pohyb vlevo / vpravo, dále Y pro skok, X pro zobrazení názvu zbraně, a C pro výstřel.

2. Výchozí nastavení kláves pro pravého hráče je kombinace kurzorových šipek pro zaměření nahoru / dolů a pohyb vlevo / vpravo, dále „.“ pro skok, „-“ pro zobrazení názvu zbraně, a CTRL pro výstřel.

10.1.4.2 Změna výchozího nastavení

Výchozí nastavení kláves se dá změnit z menu aplikace, stačí vlézt do menu *Settings*, vybrat hráče, u kterého chcete klávesy přenastavit (*Left Player / Right Player*) a poté již funkci, pro kterou chcete změnit klávesu. Stačí na ni najet, stisknout ENTER a poté klávesu, kterou chcete nastavit.

10.1.4.3 Kombinace kláves

Ve hře je pár klávesových kombinací, kterými lze dosáhnout různých událostí. Pro vrh lana je nutno za daného červa stisknout klávesy pro skok a zobrazení názvu zbraně. Pro puštění lana pak slouží klávesa pro skok. Pro změnu zbraně je nutné stisknout klávesu pro zobrazení názvu zbraně a pohyb vlevo / vpravo.

10.1.5 Protivníci řízení umělou inteligencí

Ve hře jsou dva druhy protivníků řízených umělou inteligencí.

- Boti představují klasické červy řízené umělou inteligencí. Mají, stejně jako červi řízení člověkem, 10 životů, 5 zbraní a vzhled klasického červa.

- Roboti mají jiný vzhled, vydrží o něco více, ale mají pouze jeden život. Roboti na sebe vzájemně neútočí. Lze je v editoru map umístit na konkrétní místa, dokonce jim lze přednastavit zbraň. Po zničení po sobě zanechají zbraň, kterou vlastnili.

V nastavení hry lze nastavit počet náhodně generovaných botů i robotů. Toto nastavení se aplikuje na všechny mapy a ukládá se do souboru *bots.txt*.

10.2 Editor map

Tato aplikace umožňuje vytvářet, a upravovat již existující mapy.

10.2.1 Vytvoření nové mapy

K vytvoření nové mapy je třeba nejprve zvolit prostředí mapy v podobě obrázku. Takovýto obrázek musí mít rozlišení 800x600 pixelů. Obrázek se zpracuje následujícím způsobem: Algoritmus zjistí barvu každého pixelu a podle barvy rozhodne, zda daný pixel reprezentuje nezničitelný objekt, zničitelný objekt nebo pozadí mapy. Pokud je barva daného pixelu stupně šedi (hexadeximálně vyjádřeno jako 0xAAA, kde A nabývá hodnot od 0x0 do 0xFF), pak je daný pixel identifikován jako nerozbitný objekt, pokud má barva pixelu zápis 0x502800, pak se jedná o průhledné pozadí. Jinak se jedná o zničitelný objekt.

Pro usnadnění tvorby vlastních map je v kořenovém adresáři projektu přiložen obrázek s barvou průhledného pozadí o požadovaných rozměrech, najdete ho pod názvem *background.png*.

10.2.2 Přidání objektů do mapy

Do mapy je možné přidat speciální objekty a upravovat jejich pozici. V aktuální verzi je možné přidávat do mapy objekty typu robot. Zde je na výběr z několika typů robota – lze vybrat robota s náhodnou zbraní, či robota s pevně danou zbraní.

10.2.3 Uložení mapy

Pro uložení mapy klikněte na tlačítko *Save map*. Ve složce *Maps* (v adresáři programu) se vygeneruje nová složka s názvem vaší mapy, ve které budou dva soubory - binární soubor se jméne vaší mapy a příponou *.map* a obrázek, který slouží jako náhled vaší mapy. Abyste si mohli danou mapu zahrát, zkopírujte tuto složku do složky *Maps* v kořenovém adresáři samotné hry.

10.2.4 Načtení mapy

Dříve uloženou mapu je samozřejmě možné načíst a požadovaně upravit. Stačí v menu kliknout na *Load map* a vybrat daný **.map* soubor.

Závěr

V práci jsem se zabýval analýzou, návrhem a implementací vlastní multiplatformní verze hry Liero, jejíž součástí je hra po síti a umělá inteligence, navíc jako samostatná aplikace je dodávaný editor map.

Výsledné aplikace jsem psal tak čistě, jak jsem uměl, snažil jsem se dodržovat všechna pravidla popsaná v teoretické části práce. Šlo by to určitě ještě lépe, nicméně s výslednými aplikacemi jsem spokojen, jelikož i při své velikosti (kolem 200 tříd) se mi s aplikacemi pracuje velmi dobře, modifikovat a rozšiřovat funkcionalitu je velmi snadné, což se projevilo při uživatelském testování, kde jsem byl schopný pochopit připomínky uživatelů, velmi rychle je v kódu najít a příslušné místo upravit.

Téměř všechny cíle se mi podařilo splnit, bohužel u hry po síti jsem stihl jen návrh a základní implementaci, která ještě nebyla dosatečně uživatelsky testovaná. Musím ještě sebekriticky přiznat, že výsledná verze hry nebude pro hráče příliš zážívná, neboť ji chybí větší míra obsahu. Nutno ovšem zdůraznit, že díky dobrému návrhu je přidání obsahu jednoduchou záležitostí, která není tak mentálně náročná, jelikož všechny základní mechanismy jsou již implementovány a počítají s přidáním obsahu.

Rozšíření aplikace o obsah tedy vidím jako nutné, pokud má být hra nejen hratelná, ale i zábavná v dlouhodobém měřítku. Rozšířením mám na mysli především nové mapy, které jsou díky editoru map velmi jednoduché na vytvoření samotným uživatelem, a zbraně, které musí být vytvořeny vývojářem.

Na závěr bych dodal, že díky dobrému návrhu má aplikace dobrou znovupoužitelnost, takže je možné vzít jádro mého řešení a poměrně málo změnami vytvořit úplně jinou hru fungující na podobných principech (2D hra s jednoduchou fyzikou), jako např. Mario.

Literatura

[1] Šmíd, M.: Herní průmysl v ČR a SR vyrostl v roce 2013 o 6,4 procenta. In: *konzolista.tiskali.cz* [Online]. Konzolista.cz, 2014. [cit. 2018-04-10]. Dostupné z: https://konzolista.tiscali.cz/articles/onearticle?link=herni-prumysl-v-cr-a-sr-vyrostl-v-roce-2013-o-6-4-procenta-179658

[2] Vítek, J.: Kingdom Come: Deliverance se prodal už milion kopií. In: *svethardware.cz* [Online]. Svethardware.cz, 2018. [cit. 2018-04-10]. Dostupné z: https://www.svethardware.cz/kingdom-come-deliverance-se-prodal-uz-milion-kopii/46121

[3] Švelch, J.: Jaroslav Švelch: Počítačové hry jsou největší český kulturní export In: *euro.cz* [Online]. Euro.cz, 2014. [cit. 2018-04-10]. Dostupné z: https://www.euro.cz/byznys/jaroslav-svelch-pocitacove-hry-jsou-nejvetsi-cesky-kulturni-export-1139065

[4] Bhalla, A.: Let's Face It; The Future Of Entertainment Is In Video Games Not In Movies & Here's Why In: *mensxp.com* [Online]. TIMES INTERNET LIMITED, 2017. [cit. 2018-04-10]. Dostupné z: https://www.mensxp.com/technology/gaming/36201-let-s-face-it-the-future-of-entertainment-is-in-video-games-not-in-movies-here-s-why.html

[5] Ferreira, B.: Blizzard releases 1.14a update for Diablo II In: *techreport.com* [Online]. The Tech Report, 2016. [cit. 2018-04-10]. Dostupné z: https://techreport.com/news/29842/blizzard-releases-1-14a-update-for-diablo-ii

[6] Werner, M. E.: lierolibre In: *liero.be* [Online]. Official website of Liero, 2013. [cit. 2018-04-10]. Dostupné z: http://www.liero.be

[7] Wiegers, K. E.: *Požadavky na software*. Brno: Computer Press, a.s., 2008. 448 s. ISBN 978-80-251-1877-1.

[8] Hunt, A.; Thomas, D.: *The pragmatic programmer: from journeyman to master*. Boston: Addison Wesley Professional, 1999. 352 s. ISBN 978-0-2016-1622-4.

[9] Larman, C.: *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. Třetí vydání. Boston: Addison Wesley Professional, 2004. 736 s. ISBN 0-13-148906-2.

[10] Jonáš, M.: GRASP – 2 – High cohesion In: *zdrojak.cz* [Online]. Devel.cz Lab s.r.o., 2012. [cit. 2018-04-10]. Dostupné z: `https://www.zdrojak.cz/clanky/grasp-2-high-cohesion/`

[11] Martin, R. C.: The Principles of OOD In: *butunclebob.com* [Online]. Butunclebob.com, 2005. [cit. 2018-04-10]. Dostupné z: `http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod`

[12] Jonáš, M.: Návrhové principy: SOLID In: *zdrojak.cz* [Online]. Devel.cz Lab s.r.o., 2012. [cit. 2018-04-10]. Dostupné z: `https://www.zdrojak.cz/clanky/navrhove-principy-solid/`

[13] Buna, S.: The Mistakes I Made As a Beginner Programmer In: *medium.com* [Online]. Medium, 2018. [cit. 2018-04-10]. Dostupné z: `https://medium.com/@samerbuna/the-mistakes-i-made-as-a-beginner-programmer-ac8b3e54c312`

[14] Martin, R. C.: *Clean code: a handbook of agile software craftsmanship*. New Jersey: Prentice Hall, 2009. 431 s. ISBN 0-13-235088-2.

[15] McConnell, S.: *Code complete*. Druhé vydání. Washington: Microsoft Press, 2004. 960 s. ISBN 978-0-7356-1967-8.

[16] Martin, R. C.: Clean Coder Blog - Excuses In: *blog.cleancoder.com* [Online]. The Clean Code Blog, 2017. [cit. 2018-04-10]. Dostupné z: `http://blog.cleancoder.com/uncle-bob/2017/12/18/Excuses.html`

Příloha A

Seznam použitých zkratek

AI Artificial intelligence

GRASP General Responsibility Assignment Software Patterns

GTA Grand Theft Auto

GUI Graphical user interface

TDD Test driven development

Příloha **B**

Obsah přiloženého CD

- `readme.txt` ..... stručný popis obsahu CD
- `exe` ..... adresář se spustitelnou formou implementace
- `src`
  - `impl` ..... zdrojové kódy implementace
  - `thesis` ..... zdrojová forma práce ve formátu LaTeX
- `text` ..... text práce
  - `thesis.pdf` ..... text práce ve formátu PDF