VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

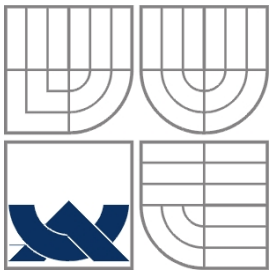
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KLIENT/SERVER APLIKACE HRY MARIÁŠ

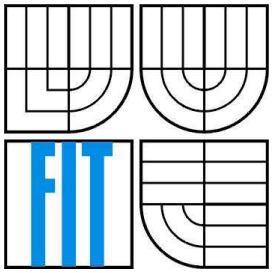
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE JAROSLAV FABIAN
AUTHOR

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

KLIENT/SERVER APLIKACE HRY MARIÁŠ

CLIENT/SERVER APPLICATION OF THE GAME WHISP

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — JAROSLAV FABIAN
AUTHOR

VEDOUCÍ PRÁCE — ING. MICHAL DOLEŽEL
SUPERVISOR

BRNO 2011

Abstrakt

Bakalářská práce popisuje návrh a implementaci karetní hry Mariáš. Hra je určená pro tři až čtyři hráče, kteří ji hrají přes počítačovou síť prostřednictvím protokolu TCP/IP. Hra je vytvořená pro systémy Microsoft Windows XP, Vista a 7 v programovacím jazyce Java.

Abstract

The bachelor's thesis describe predesign and implementation of the card game the Whisp. Game is dedicated for three or four players, who play it using computer network and protocol TCP/IP. Game is developed for Microsoft Windows XP, Vista, 7 operating systems. Programming language is Java.

Klíčová slova

mariáš, java, multiplayer, síťová hra

Keywords

the whisp, java, multiplayer, network game

Citace

Fabian Jaroslav: Klient/server aplikace hry mariáš, bakalářská práce, Brno, FIT VUT v Brně, 2011

Klient/Server aplikace hry mariáš

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Michala Doležela. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Jaroslav Fabian
12.05.2011

Poděkování

Rád bych poděkoval vedoucímu mé bakalářské práce panu Ing. Michalovi Doleželovi za projevené úsilí a čas, který mi věnoval.

Obsah

1 Úvod

Počítačové hry zaznamenali za posledné dve dekády obrovský vzostup. Technológie na tvorby hier stále napredujú a dnes existuje veľa herných titulov s takmer dokonalým technickým spracovaním. Aj napriek obrovskému množstvu kvalitných a zložitých hier dostupných na rôznych platformách, sú stále veľmi populárne aj jednoduchšie hry, ktoré z hľadiska technickej stránky práve neexcelujú. Primárnou funkciou každej hry je zábava. Pokiaľ je hra zábavná a pritiahne k sebe dostatok hráčov, nie je nutné aby to bola graficky náročná, 15 gigabajtov zaberajúca aplikácia. Táto premisa platí aj opačne a nie každá náročná hra, ktorej vývoj možno zabral roky, je automaticky zábavná, pretože jednoducho neobsahuje žiadnu pridanú hodnotu. Pri počítačových hrách je dôležitá v prvom rade originálna myšlienka, nápad, ktorý ľudí zaujme.

Práve k tým jednoduchším hrám patria aj hry kartové, ktoré majú predovšetkým na operačných systémoch typu Windows obrovskú popularitu už takmer dve desaťročia. V tejto práci sa budeme venovať kartovej hre mariáš. Mariáš patrí k najobľúbenejším hrám na Slovensku a v Českej republike a na rozdiel od rôznych nenáročných kartových hier (sedma, faraón a pod.), ktoré často hrávame, pri mariáši musí človek aj rozmýšľať.

Cieľom práce je teda vytvorenie aplikácie, ktorá umožňuje hrať kartovú hru mariáš prostredníctvom sieťového protokolu TCP/IP trom až štyrom hráčom. Zvolený variant pre túto kartovú hru je tzv. volený mariáš (resp. pauzovaný pre štyroch hráčov). Aplikácia musí obsahovať užívateľské rozhranie, ktoré má jednoduchým spôsobom poskytnúť užívateľovi prístup k jednotlivým nastaveniam. Pre účely hry bude vytvorené jednoduché grafické rozhranie s animáciami kariet.

Na implementáciu aplikácie sa použije programovací jazyk Java a vývojové prostredie NetBeans 6.8. Užívateľské rozhranie bude vytvorené pomocou knižnice Swing.

Kapitola 2 má za úlohu oboznámiť čitateľa s pravidlami kartovej hry mariáš, stručne popísať jej históriu a tiež poskytnúť základné informácie o iných kartových hrách, ktoré sa dnes vyskytujú na osobných počítačoch.

Kapitola 3 popisuje jazyk Java, niektoré jeho dôležité knižnice a ďalšie nástroje. Vyskytuje sa tu stručný popis jazyka UML a niektorých UML diagramov.

Kapitola 4 zahŕňa analýzu a návrh aplikácie, ako aj samotný účel aplikácie. Možno sa v nej dočítať o problematike a možnostiach aplikácií typu klient/server, ako aj o zvolenom sieťovom variante. V tejto kapitole sa využívajú UML diagramy spomínané v kapitole 3.

Kapitola 5 popisuje postup implementácie jednotlivých častí programu. Hlavné časti sú práca s dátami, klient/server protokol a sieť, užívateľské rozhranie a implementácia herného enginu. Sú tu popísané ďalšie implementačné súčasti a aj problémy počas programovania aplikácie.

Kapitola 6 sa venuje testovaniu a skúmaniu problémov, ktoré vznikli počas chodu aplikácie. Ďalej sú tu obsiahnuté minimálne hardvérové a softvérové požiadavky a prípadné obmedzenia.

Kapitola 7 obsahuje záverečné celkové zhodnotenie práce a možné budúce rozšírenia.

2 Mariáš

2.1 Kartové hry na PC

Kartové hry na osobných počítačoch zaznamenali vysokú popularitu vďaka spoločnosti Microsoft, ktorá ich implementuje do svojich operačných systémov už od roku 1990, kedy bola prvýkrát zahrnutá hra Solitér do Windows 3.0. Od verzie Windows 3.11 sa začala objavovať hra Srdcia a od operačného systému Windows 95 bol pridaný aj FreeCell. Od roku 1998 sa stáva súčasťou operačných systémov Windows aj hra Pavúčí Solitér. Všetky tieto hry sú veľmi obľúbené pre ich jednoduchosť a relatívne efektívne a zábavné zabíjanie času (napr. v práci).

2.2 O mariáši

Mariáš je kartová hra, ktorá sa hrá s balíčkom 32 kariet nemeckého typu (na Slovensku známe ako „sedmové“ a v Čechách ako „mariášky“). Mariáš má pôvod v Nemecku, kde vznikol na prelome 18. a 19. storočia. Jeho názov pochádza z francúzskej hry *Mariage(svadba)*, ktorá bola jeho predchodcom. Okrem toho mariáš prevzal niektoré prvky z hier ako *Taroky*, *Skat* a *Špád*. Hra patrí k bodovaným hrám, kde hrajú najvýznamnejšiu úlohu esá a desiatky. Mohli by sme ju zaradiť do skupiny mariášových hier, kde okrem iných patrí aj nemecká *Schnapsen*, španielska *Tute* alebo maďarská *Ulti*.[10]

Kartová hra mariáš je známa predovšetkým v Českej republike, ale nájde si svojich priaznivcov aj na Slovensku. Keďže sa nejedná o celosvetovo známu hru, neexistuje príliš veľa aplikácií, ktoré by sa Mariášom zaoberali. Úplne prvá videohra Mariáš vznikla na DOS-ovej platforme v roku 1993 pod názvom Flek! Dnes už máme k dispozícii niekoľko aplikácií, ktoré ponúkajú vylepšené možnosti a prívetivejšie užívateľské prostredie. Samotné čaro hry však ostáva rovnaké. Všetky tieto aplikácie sú vytvorené pre jedného hráča a súpermi sú počítačom riadení protivníci. Možnosť zahrať si online proti skutočným hráčom ponúka iba jediný server v Českej republike.

2.3 Pravidlá hry

Táto podkapitola prevzatá z [7].

2.3.1 Všeobecné zásady

Hra je určená pre 3-4 hráčov. Zvolený implementovaný variant sa nazýva *volený mariáš* (resp. *pauzovaný* pre 4 hráčov). Mariáš sa hrá s kartami obsahujúcimi 32 listov, ktoré majú hodnoty sedem, osem, deväť, desať, dolník, horník, kráľ a eso. Každá z hodnôt sa nachádza v balíčku v štyroch farbách – zeleň, červeň, guľa a žaluď.

Pokiaľ je jedna farba ustanovená ako tromfová, prebíja akúkoľvek hodnotu zvyšných troch farieb. Pri variantoch, ktoré majú stanovený tromf, je hodnotovo najsilnejšia karta eso a za ním nasleduje desiatka. Ostatné karty sa ďalej radia podľa bežného poradia. Pri netromfových variantoch je desiatka zaradená na obvyklé miesto, a teda medzi deviatku a dolníka.

Hráč, ktorý je naľavo od rozdávajúceho, sa nazýva *predák* a hráč napravo *zadák* (platí pri rozdávaní v smere hodinových ručičiek, pri opačnom smere je predák a zadák v opačnej pozícii). Miešanie kariet sa uskutočňuje iba na začiatku hry a od tej doby sa už karty nemôžu miešať. Po

dohraní predošlého kola sa začína ďalšie rozdanie (posunie sa pozícia rozdávajúceho, predáka a zadáka), pričom iba rozdávajúci má právo zobrať karty uložené na kôpkach a zložiť ich do balíčku (karty nemieša). Potom položí karty pred zadáka a požiada ho o „seknutie“ balíčka.

Povinnosťou každého hráča je priznať farbu. To znamená, že pokiaľ je vynesená karta nejakej farby, tak ostatní hráči musia použiť kartu rovnakej farby, pokiaľ takúto farbu majú medzi svojimi kartami. Zároveň platí, že musia použiť vyššiu kartu rovnakej farby, ak majú takú k dispozícii a nebola použitá karta tromfovej farby. Ak niektorý z hráčov nemá kartu rovnakej farby ako je vynesená karta, je povinný použiť kartu tromfovej farby. Ak sa ani tromfová farba v jeho repertoári nenachádza, ostáva na ľubovôli hráča, akú kartu použije.

Kartová hra mariáš sa hrá na tzv. štichy alebo tiež zdvihy. Štich obsahuje 3 karty (od každého hráča jednu) a berie ho ten hráč, ktorý vlastní kartu s najvyššou hodnotou (viď hodnotový rebríček kariet vyššie) s ohľadom na farbu. Hráč, ktorý získal posledný zdvih, vynáša na začiatku ďalšieho.

2.3.2 Priebeh hry

Rozdávajúci hráč rozdá predákovi 7 kariet, potom zadákovi a sebe po 5. Následne rozdá ešte každému hráčovi po 5 kariet. Predák vyberá z prvých siedmich kariet (ostatných päť musí nechať ležať lícnou stranou dolu) tromfovú farbu. Tromf môže aj náhodne zvoliť vybratím jednej karty zo zvyšných (zakrytých) piatich (tento spôsob vyberania tromfu sa nazýva aj voľba z „národa“). Vybratú tromfovú kartu položí predák pred seba lícnou stranou dole a svojim protihráčom ju v žiadnom prípade zatiaľ neukazuje. Vezme si zakrytých 5 kariet a rozhodne sa pre dve karty, ktoré odloží do talónu (je zakázané odhodiť desiatky a esá) alebo si tromf vezme, čím hráč volí netromfovú hru (potom môže do talóna odhodiť aj desiatky a esá).

Pokiaľ chce hrať netromfovú hru, zahlási *Betl* (malý) resp. *Durch* (veľký). Ak sa rozhodne pre tromfovú hru, musí dať príležitosť ostatným hráčom na hru netromfovú. Tak to urobí vyhlásením „farba“. Ostatní dvaja hráči majú možnosť súhlasiť s týmto vyhlásením a ak obaja súhlasia, hrá sa tromfová hra, kde je predák aktérom a ďalší dvaja hráči sú obranou. Vybratá tromfová karta je teraz ukázaná brániacim sa hráčom. V tejto chvíli má aktér pred sebou niekoľko možností. Môže hru ukončiť a vyplatiť protihráčom základnú sadzbu alebo zahlásiť obyčajnú hru, prípadne k nej pridať *sedem*. Ďalšou možnosťou je zahlásiť *sto* alebo *stosedem*. Toto sú záväzky, ktoré chce aktér splniť. Každý z týchto záväzkov sa dá flekovať a následne druhou stranou kontrovať prakticky donekonečna (v Českej republike sa používa „Re!“, pri ďalšom kontrovaní „Tutti“ a potom „Boty“), pričom hodnota kontrovaného záväzku sa vždy zdvojnásobuje. Kontrovať sa dá vždy len jeden záväzok. V prípade, že obaja obrancovia vyhlásením „Dobrý“ nechcú hru alebo sedmu flekovať, a teda predpokladajú, že aktér svoj záväzok splní, hra sa končí a aktér je vyplatený.

V prípade, že niektorý z hráčov nesúhlasí s ohlásením farby na začiatku hry, môže zobrať karty v talóne a ohlásiť Betl resp. Durch. V tomto prípade sa on sám stáva aktérom, vyberie karty, ktoré zahodí do talóna. Ak tento hráč ohlásil Betl, môže následne ešte niektorý z hráčov ohlásiť Durch, vziať karty z talóna a tým sa aj stať aktérom. V prípade, že všetci súhlasia s Betlom alebo Durchom, môže sa hra začať. Ďalšou možnosťou je ešte zvyšovanie hodnoty hry. Zahlásením „Kontra!“ sa vždy hodnota hry zvyšuje na dvojnásobok.

2.3.3 Varianty, sadzby a bodovanie

Hra mariáš obsahuje niekoľko variantov. Niektoré tieto varianty je možné kombinovať. Variant sa volí vždy pred začiatkom každej hry. Aktér volí ako prvý, pričom obrana má vždy možnosť voliť vyšší variant (viď Sadzby).

Varianty:

Všetky varianty môže hlásiť aj aktér, aj obrana. Nasledujúce varianty sú tzv. tromfové (t.j. je vždy určený tromf, počítajú sa body a desiatka je druhá najsilnejšia karta).

Hra **–** Cieľom je získať viac bodov ako protistrana, pričom obrancom sa ich body sčítavajú.

Sedem (resp. hra a sedem) – Cieľom je získať posledný štich tromfovou sedmou. Pri tomto variante sa hra a sedem vyhodnocuje samostatne. Pokiaľ sa „neflekne“ ani sedma, ani hra, tak sa toto kolo nehrá, ale hráči musia vyplatiť základné čiastky.

Sto – Cieľom je získať 100 bodov s pomocou jedinej hlášky, teda je predpoklad, že hráč, ktorý tento variant hlási, vlastní nejakú hlášku. Pokiaľ hráč záväzok splní, získa navyše „vyflekovanú“ sadzbu za každých 10 bodov (sem sa počítajú aj ďalšie hlášky), ktoré uhrá. Ak záväzok nesplní, musí rovnakú čiastku uhradiť protistrane za každých 10 bodov, ktoré mu chýbajú do 100.

Tichých sedem – V prípade hry alebo stovky je možné tiež uhrať tichých (nehlásených) sedem. Tichá sedma je za polovicu sadzby tej obyčajnej a v prípade, že sa tento pokus v poslednom zdvihu nepodarí, musí sa zaplatiť sadzba protistrane, aj keď takúto sedmu zabije spoluhráč.

Tichých sto – Pokiaľ ide o varianty sto alebo hra, je možné v prípade aktéra alebo obrany uhrať variant tzv. tichých (nehlásených) sto. V tomto prípade sa počítajú všetky hlášky bez rozdielu. Pri uhratí tichej stovky sa hodnota hry zdvojnásobuje. Víťaznej strane naviac prináleží hodnota tejto tichej stovky za každých 10 bodov nad sto.

Stosedem – Kombinácia sto a sedem. Oba záväzky sa vyhodnocujú samostatne.

Ďalšie dva varianty sú tzv. netromfové, teda neurčuje sa pri nich tromf ani sa nepočítajú body. Desiatka sa radí hodnotovo za dolníka.

Betl(malý) – hráč sa zaväzuje neuhrať ani jeden zdvih.

Durch(veľký) – hráč sa zaväzuje získať všetky štichy.

Mariáš je hazardná kartová, a preto sa hrá takmer výlučne o peniaze, pričom peňažný základ býva maximálne niekoľko korún českých. Najväčšia možná sadzba je 500-násobok peňažného základu. Aktér získava výhernú hodnotu od oboch obrancov, ale v prípade prehry musí obom zaplatiť „vyflekovanú“ čiastku. Pokiaľ je tromf červený, sadzba sa zdvojnásobuje.

Sadzby:

Uvedené sadzby sú násobiteľom peňažného základu.

Hra – 1

Sedem – 2

Sto – 4

Betl (malý) – 15

Durch (veľký) - 30

Karty eso a desať sa nazývajú aj mastné karty. Hráč, ktorý berie zdvih s takýmito kartami, si automaticky pripočíta hodnotu 10 bodov. Okrem toho existujú aj ďalšie spôsoby zbierania bodov uvedené nižšie. Body brániacich hráčov sa sčítavajú.

Body:

-eso a desať = 10 bodov

-posledný zdvih (tzv. Ultimo) = 10 bodov

-hláška (mariáš) = 20 bodov

-tromfová hláška = 40 bodov

Pozn.: Hláška znamená dvojicu kráľ a horník v rovnakej farbe na ruke.

3 Použité technológie

3.1 Java

Táto podkapitola bola prevzatá z [6].

Programovací jazyk Java predstavila firma Sun Microsystems 23. mája 1995. Java je považovaná za objektovo orientovaný programovací jazyk, hoci presnejšie by bolo zaradiť ho medzi hybridné programovacie jazyky. Dôvodom je to, že tento jazyk je založený na imperatívnom programovaní a objektovo orientovaný prístup implementuje iba do určitej miery. Ďalej sa programovací jazyk Java radí medzi triedne orientované jazyky.

Sprvoti bola Java známa ako nástroj na tvorbu appletov pre webové stránky. Dnes je jedným z najpopulárnejších a najpoužívanejších jazykov na svete. Je to predovšetkým kvôli jeho jednoduchosti a prenositeľnosti. Syntax jazyka je zjednodušenou a upravenou verziou syntaxe jazykov C a C++. Mnoho problematických programátorských konštrukcií oproti spomínaným jazykom odpadlo. Jazyk obsahuje primitívne dátové typy, ktoré zdedil po svojich predchodcoch, avšak všetky ostatné dátové typy sú objektové.

Java je obzvlášť užitočná pre distribuované sieťové prostredia ako web. Podporované sú tiež rôzne úrovne sieťového spojenia a práce so vzdialenými súbormi.

Namiesto skutočného strojového kódu sa vytvára iba tzv. „medzikód“(bajtkód), čo umožňuje danej aplikácii fungovať na ľubovoľnom počítači alebo zariadení, ktoré má k dispozícii interpret Javy, tzv. virtuálny stroj Javy (Java Virtual Machine). Pre rôzne platformy je potom možné upraviť vzhľad a správanie aplikácie. V neskorších verziách bola použitá optimalizácia (just in time compilation – JIT), ktorá pred prvým prevedením dynamicky skompilovala medzikód do strojového kódu počítača, čo zásadne zrýchlilo fungovanie programov, ale na druhej strane sa spomalil ich štart. Dnes sa zväčša používajú technológie HotSpot compiler, ktoré medzikód najskôr interpretujú a na základe štatistík z tejto interpretácie neskôr prevedú preklad často používaných častí a ďalších dynamických optimalizácii do strojového kódu.

Programovací jazyk Java je často používaný na tvorbu náročných projektov. Jedným z dôvodov je jeho vysoká spoľahlivosť. Na dosiahnutie takejto spoľahlivosti bolo potrebné vynechať isté programátorské konštrukcie, ktoré boli pri predchodcoch Javy častým zdrojom množstva chýb (napr. správa pamäte, príkaz „goto“, používanie ukazovateľov a pod.). Jazyk ďalej používa silnú typovú kontrolu. To znamená, že každá premenná musí mať definovaný svoj dátový typ.

Správa pamäte je realizovaná prostredníctvom tzv. Garbage collectoru (doslova zberateľ odpadu – smetiar), ktorý vyhľadáva nepoužívané časti pamäte a uvoľňuje ich na ďalšie použitie. Tento spôsob správy pamäte zo začiatku spôsoboval pomalší beh programov, ale vďaka novým algoritmom a tzv. generačnej správe pamäte sa podarilo tento problém z veľkej časti eliminovať.

Napriek tomu, že Java je jazyk interpretovaný, nie je strata výkonu významná, pretože prekladače fungujú v režime „práve včas“ a do strojového kódu sa prekladá iba kód, ktorý je skutočne potrebný.

Java ďalej podporuje spracovanie viacerých vlákien a tiež poskytuje za chodu dynamické rozširovanie knižnice o nové triedy. Tento programovací jazyk je veľmi intuitívny a elegantný, teda sa v ňom veľmi dobre pracuje, a kód je prehľadný a ľahko čitateľný. Java vyžaduje ošetrenie výnimiek a typovú kontrolu. Pri používaní Javy môžeme povedať, že ide o veľmi komfortné a užívateľsky prijateľné programovanie.

Medzi hlavné nevýhody tohto programovacieho jazyka patrí pomalší štart programov a väčšia pamäťová náročnosť oproti jazykom s tzv. statickou kompiláciou (napr. C++). Ďalšia nevýhoda sa skrýva v návrhu Javy, kde je cítiť snaha znemožniť programátorovi používať niektoré programátorské konštrukcie (napr. príkaz „goto“ alebo neznamienkové číselné dátové typy), aj keď tieto majú v istých prípadoch odôvodnené uplatnenie.

3.1.1 Knižnica Swing

Na tvorbu podkapitoly „Knižnica Swing“ bola použitá [4].

Swing je súborom tried používaných na vytváranie elementov užívateľského rozhrania. V počiatkoch Javy však neexistoval a bol až odpoveďou na nedostatky pôvodného podsystému pre grafické užívateľské rozhrania, tzv. AWT (Abstract Window Toolkit). AWT definuje základný súbor ovládacích prvkov, okien a dialógových komponentov, ktorý podporuje použiteľné, ale obmedzené grafické rozhranie. Jedným z dôvodov tohto obmedzenia je preklad jednotlivých komponentov do korešpondujúcich ekvivalentov v závislosti od platformy. To znamená, že vzhľad akéhokoľvek komponentu závisel od danej platformy a nie od samotnej Javy. Pretože AWT komponenty používajú natívne zdroje, sú tiež nazývané ako ťažké (heavyweight).

Použitie takýchto natívnych prvkov malo za následok vznik niekoľkých problémov. Po prvé, rozdielnosti operačných systémov spôsobili odlišný vzhľad (look and feel) a dokonca aj správanie takýchto komponentov na rozdielnych platformách. Po druhé, tento vzhľad neurčovala Java. To znamená, že každý prvok mal stabilný vzhľad a nebolo jednoduché ho zmeniť podľa vôle programátora. Po tretie, používanie ťažkých prvkov spôsobovalo rôzne ďalšie obmedzenia (napr. museli byť nepriehľadné a pravouhlé).

Zakrátko po vydaní Javy sa ukázali problémy AWT ako dostatočne závažné na to, aby sa zvolil nový prístup. Riešením bolo vytvorenie Swingu, ktorý bol predstavený v roku 1997 ako súčasť JFC (Java Foundation Classes). Zo začiatku bol k dispozícii ako stiahnuteľná knižnica a od verzie Java 1.2 sa Swing, ako aj zvyšok JFC, stali plne zabudovanými súčasťami Javy.

Je nutné podotknúť, že Swing síce vyriešil mnohé problémy AWT, nie je však jeho náhradou, ale iba akousi nadstavbou. AWT je teda stále nevyhnutnou súčasťou programovacieho jazyka Java. Swing vylepšuje AWT prostredníctvom dvoch kľúčových znakov. Jednak sú to ľahké (lightweight) komponenty, a potom zásuvný vzhľad týchto komponentov. Tieto dva znaky sú pre Swing kľúčové, riešia dôležité nedostatky AWT a poskytujú ľahko používateľné a elegantné riešenie.

Ľahké komponenty sú omnoho flexibilnejšie. Sú napísané v Jave a nemapujú sa priamo na špecifické natívne komponenty platformy. Dokážu byť aj nepravouhlého tvaru a priehľadné. Každý komponent je určený Swingom, čo znamená, že na každej platforme sa správa konzistentne.

Zásuvný vzhľad komponentov znamená, že každý prvok je renderovaný Javou a nie natívnymi korešpondujúcimi prvkami. To umožňuje oddeliť dizajn komponentu od jeho logiky. Je teda možné meniť vzhľad prvku a zároveň zachovať ostatné aspekty tohto prvku. Ďalej je možné definovať celé sety vzhľadov a dizajnov, ktoré reprezentujú rozličné GUI štýly. Ak teda napríklad vieme, že aplikácia bude bežať výlučne v prostredí Windows, potom môžeme nastaviť pre všetky komponenty štýl Windows alebo nastaviť vlastný štýl, prípadne tento štýl dynamicky meniť počas chodu programu.

Swing GUI obsahuje dve kľúčové skupiny: komponenty a kontajnery. Toto rozdelenie je iba pojmové, pretože všetky kontajnery sú zároveň komponenty, rozdiel je iba v ich uplatnení.

Komponent je všeobecne definovaný ako nezávislí grafický ovládací prvok, ako napríklad tlačidlo (push button) alebo behúň (slider). Vo všeobecnosti sú komponenty potomkami triedy JComponent. Táto trieda poskytuje spoločnú funkcionalitu pre všetky komponenty (napr. JButton, JSlider, JTextField).

Kontajner môže obsahovať skupinu komponentov. Je to teda špeciálny komponent určený na zachytávanie a zobrazovanie jednotlivých komponentov. Ďalej platí, že komponent musí byť súčasťou nejakého kontajneru, ak sa má zobraziť. Existujú dva typy kontajnerov.

Prvá skupina nededí z triedy JComponent, ale z AWT tried Component a Container. Tieto kontajnery najvyššej úrovne sú ťažké a sú výnimkou v knižnici Swing. Medzi tieto kontajnery sa radia JFrame, JApplet, JWindow a JDialog. Tieto kontajnery sú vždy na vrchu hierarchie kontajnerov, a teda nie sú ani obsiahnuté v žiadnom inom kontajneri.

Druhou skupinou sú ľahké kontajnery, ktoré opäť dedia z triedy JComponent. Príkladom takéhoto kontajneru je JPanel. Ľahké kontajnery sa obvykle používajú na zoskupovanie a organizovanie skupín komponentov, ktoré majú medzi sebou nejaké vzťahy. Takto sa dá vytvoriť niekoľko prehľadných podskupín komponentov obsiahnutých v spoločnom kontajneri, keďže ľahké kontajnery môžu byť vložené do iných kontajnerov.

3.1.2 Balíčky pre I/O operácie

Na tvorbu tejto podkapitoly bola použitá predovšetkým [2]. Niektoré doplňujúce údaje boli čerpané z [4].

Väčšina programov sa dnes nezaobíde bez prístupu k externým dátam. Dáta sú získavané zo vstupných zdrojov a zasielané na výstupné ciele. Java ponúka niekoľko balíčkov, ktoré sa zaoberajú pohybom dát do našich programov a z nich.

Balík **java.io** definuje vstupno-výstupné operácie ako prúdy (streams) dát. Prúd je usporiadaná postupnosť dát, ktorá má zdroj (vstupné prúdy) alebo cieľ (výstupné prúdy), a dalo by sa povedať, že buď produkuje, alebo konzumuje dáta. Zdroje a ciele môžu byť rôzne (napr. sieťové pripojenie, pamäťový zásobník alebo súbor na disku), ale všetky prúdy sa správajú rovnako, bez ohľadu na fyzické zariadenie. Triedy balíčka java.io oddeľujú programátorské postupy od špecifických detailov používaného operačného systému a umožňujú prístup k systémovým zdrojom prostredníctvom súborov a pod.

Balík **java.nio** a súvisiace balíčky definujú vstupno-výstupné operácie ako zásobníky (buffers) a kanály (channels). Zásobník obsahuje dáta (podobne ako napr. polia) a je možné z neho čítať alebo naopak zapisovať dáta. Kanály reprezentujú spojenia medzi entitami vykonávajúcimi vstupno-výstupné operácie, vrátane zásobníkov, súborov a soketov. Písmeno „n“ v názve balíku java.nio znamená nový (new). Zámerom autorov však nebolo nahradiť pôvodný java.io balík, ale poskytnúť odlišný kanálovo založený prístup, ktorý môže mať za určitých okolností výhody. Hlavným rozdielom oproti spomínanému prúdovo založenému prístupu je to, že kanály umožňujú neblokujúce vstupno-výstupné operácie a prerušiteľné blokujúce operácie. Táto vlastnosť je veľmi užitočná, napríklad pri serverových aplikáciách.

Balík **java.net** poskytuje špecifickú podporu pre sieťovo orientované vstupno-výstupné operácie, ktoré využívajú sokety spoločne s prúdovým alebo kanálovým modelom I/O operácii. Java podporuje TCP/IP rozšírením zaužívaného prúdového I/O rozhrania a pridaním vlastností potrebných na vytvorenie vstupno-výstupných objektov na sieti. Java je kompatibilná s TCP aj UDP protokolom. TCP protokol je využívaný na spoľahlivý prúdový prenos, zatiaľ čo UDP podporuje rýchlejší, dvojbodový (point-to-point), datagramový model.

3.1.3 NetBeans

NetBeans je voľne dostupné, open-source integrované vývojové prostredie (IDE) pre softvérových vývojárov. Obsahuje nástroje, ktoré uľahčujú tvorbu profesionálnych aplikácií v rôznych programovacích jazykoch, ako sú Java, PHP alebo C/C++. Netbeans ďalej obsahuje framework pre Swingové aplikácie, ktorý umožňuje vytvárať grafické užívateľské rozhrania

spôsobom drag&drop a tým uľahčiť programátorovi prácu s písaním rozsiahleho kódu. NetBeans je podobne ako Java multiplatformový. [9]

3.2 UML

Nasledujúca podkapitola bola prevažne prevzatá z [3]. Ďalšie informácie boli čerpané z [8].

Modelovací jazyk UML (Unified Modeling Language) je výsledkom snaženia analytikov a dizajnérov, ktorí v priebehu 80. a 90. rokov minulého storočia vytvárali metódy, ktoré by umožnili popísať objektovo orientovanú analýzu a návrh.

V roku 1995 boli začaté práce na zjednotení rôznych metód na modelovanie pod záštitou spoločnosti Rational. Výsledkom bola prvá verzia modelovacieho jazyka UML v roku 1997. Tento súhrn metód sa stal priemyselným štandardom a ďalej sa postupne vyvíja. Súčasná verzia je 2.3 a verzia 2.4 je v štádiu beta testovania.

Modelovací jazyk UML je súhrnom predovšetkým grafických notácií k vyjadreniu analytických a návrhových modelov. Pomocou rovnakej formálnej syntaxe je možné použiť UML na tvorbu jednoduchých aj zložitých aplikácií a ďalej tieto návrhy zdieľať s ostatnými návrhármi. Pri návrhu každého väčšieho systému je nutná kooperácia užívateľa, aby výsledná aplikácia spĺňala predstavy zadávateľa. Vybrané modely UML sú pochopiteľné aj pre zadávateľa – laika a umožňujú kvalitné vyjasnenie požiadaviek užívateľov na vytváraný systém. Diagramy nezachytávajú navrhovaný systém ako celok, ale sústredia sa vždy práve na jeden pohľad na vyvíjaný systém. UML je tiež jazyk pre vizualizáciu, špecifikáciu, stavbu a dokumentáciu softwarových systémov.

3.2.1 Use Case diagramy

Na zobrazenie vysoko úrovňových funkcií a požiadaviek sa využívajú **diagramy prípadu užitia** (Use Case diagrams). Prípady užitia zachytávajú presnú funkčnosť, ktorá bude budúcim informačným systémom pokrytá, a vymedzujú tak jednoznačne rozsah práce. Každý prípad užitia popisuje práve jeden zo spôsobov použitia systému a teda popisuje jednu jeho požadovanú funkčnosť. Prípady užitia teda určujú, čo presne sa bude programovať, a vyvinutý systém nebude obsahovať nič iné len to, čo popisujú prípady užitia. Prípad užitia sa dá definovať aj ako súbor scenárov, ktoré spája dovedna spoločný cieľ.

V Use Case diagramoch rozlišujeme predovšetkým dve entity. Tou prvou je aktér, ktorý je externou entitou, a nejakým spôsobom komunikuje s daným systémom (môže to byť užívateľ alebo aj napríklad databázový systém). Jeden fyzický aktér môže voči systému vystupovať vo viacerých roliach. Druhou entitou je samotný prípad užitia, ktorý popisuje činnosť, ktorú môže aktér vykonávať. Medzi jednotlivými entitami sú vzťahy (relácie), ktoré sa v diagramoch znázorňujú plnou čiarou. Aktér má tvar postavičky a prípad užitia sa najčastejšie vyskytuje ako elipsa, vo vnútri ktorej je napísaná určitá činnosť. Okrem obyčajných relácií sa v Use Case diagramoch vyskytujú aj ďalšie špeciálne vzťahy medzi prípadmi užitia alebo medzi aktérmi.

Relácia **<<include>>** sa objavuje iba medzi prípadmi užitia tam, kde existuje podobná alebo rovnaká časť sekvencie scenára, opakujúca sa vo viacerých prípadoch užitia. Inými slovami ide o vyčlenenie spoločného správania zo scenárov základných prípadov užitia. Podstatné je, že základný prípad užitia nie je bez rozširujúceho prípadu užitia kompletný. V praxi to znamená, že pokiaľ aktér chce vykonať nejakú činnosť popísanú v prípade užitia, musia sa zároveň vykonať všetky činnosti v prípadoch užitia, ktoré sú v relácii <<include>> s prvotným Use Case, ktorý vykonáva aktér.

Relácia typu **<<extend>>** pridáva rozširujúci prípad užitia, t.j. nové – doplnkové správanie do základného prípadu užitia. Podstatným rozdielom oproti relácii <<include>> je však to, že základný prípad užitia je úplne sebestačný, keďže sa deklarujú iba tzv. body rozšírenia (extension points), ktoré

nie sú nevyhnutnou súčasťou scenára. Vzťah <<extend>> teda ukazuje iba miesta, kde eventuálne môže dôjsť k použitiu činnosti popísanej v rozširujúcom prípade užitia a modeluje tak voliteľné súčasti scenára.

Ďalším typom relácie medzi prípadmi užitia, ale aj medzi aktérmi, je tzv. **zovšeobecnenie (generalizácia)**. Generalizácia prípadov užitia umožňuje správanie spoločné pre viac prípadov užitia previesť do rodičovského prípadu užitia. V prípade generalizácie medzi aktérmi je jej funkcia podobná a používa sa vtedy, ak viacero aktérov vykonáva rovnakú činnosť (napr. zamestnanec a zamestnávateľ sú aktéri, ktorí komunikujú s odlišnými prípadmi užitia, ale existujú aj spoločne používané prípady užitia (napr. prihlásenie do systému) a vtedy je možné ich zovšeobecniť do aktéra užívateľ).

3.2.2 Diagramy tried

Diagramy tried patria do štrukturálneho modelovania navrhovanej aplikácie a sú kľúčovou aktivitou pri modelovaní objektovo orientovaných systémov. Kvalita výsledného systému je predovšetkým odrazom kvality modelu tried. Trieda je definovaná svojimi atribútmi a metódami a predstavuje šablónu pre skupinu inštancií, ktoré nazývame objekty. Model tried teda dáva základ pre funkciu jednotlivých objektov.

Atribút ako nositeľ informácií o objekte je definovaný svojím menom, formátom (v objektových implementačných prostrediach je to dátový typ) a viditeľnosťou. Názov atribútu jednoznačne pomenováva vlastnosť objektu (napr. meno zákazníka, dátum narodenia a pod.). Formát atribútu závisí aj od použitého implementačného prostredia. Vo všeobecnosti reťazec pomenujeme ako string, celé číslo ako integer, prípadne sa používajú užívateľské typy formátov a pod. Viditeľnosť môže byť trojakého druhu. V UML rozlišujeme Public (verejný prístup pre každý element systému), Private (súkromný prístup výlučne pre operácie implementované v danej triede) a Protected (chránený prístup, ktorý k privátnemu prístupu pridáva ešte prístup potomkov triedy, ktorá takýto atribút obsahuje).

Operácie, rovnako ako atribúty, majú svoju charakteristiku, ktorá je daná ich názvom, zoznamom parametrov a návratovými hodnotami. Niektoré z týchto metód vykonávajú operácie s dátami, iné môžu byť typom „interface“, čiže poskytujú rozhranie ostatným objektom, ktoré požadujú služby tohto objektu.

Diagramy tried zobrazujú statickú stránku systému, predovšetkým vzťahy medzi triedami. Vzťahy, ktoré jednotlivé triedy spája, sú asociácia, agregácia, kompozícia a špecializácia (generalizácia).

Podradenosť jedného objektu voči druhému je v návrhu chápaná buď ako agregácia, alebo ako kompozícia. Vzťah typu **agregácia** je jednou z najčastejších väzieb pri modelovaní objektových tried. Väzba typu agregácia hovorí, že jedna trieda je časťou druhej triedy (napr. trieda Motor je časťou triedy Auto, je medzi nimi agregačný vzťah). U agregácie nadradený objekt využíva možnosti podradeného objektu a mal by niesť zodpovednosť za jeho vznik a zánik. **Kompozícia** je špeciálnym typom agregácie a ešte viac umocňuje vzťah medzi objektmi, keďže pri kompozičnom vzťahu nemôže podradený objekt bez nadradeného existovať (príkladom je trieda Riadok dokladu a trieda Doklad, kde trieda Riadok dokladu sama o sebe nemá žiadny význam).

Asociácia znázorňuje vzťahy medzi jednou alebo viacerými triedami, ktoré sú abstrakciou množiny spojení medzi inštanciami týchto tried. Pri asociácii môžeme definovať jej meno, meno role, násobnosť a navigovateľnosť. Použitím asociácie poukazujeme na rovnocenný vzťah medzi triedami. Pokiaľ je zámer asociácie z kontextu jasný, potom nemusí byť pomenovaná, inak sa volí aktívny (slovesný) tvar názvu. Meno role pomenováva rolu na príslušnom konci asociácie a tak nahradzuje alebo doplňuje názov asociácie. Násobnosť určuje počet inštancií jednej triedy vo vzťahu

reprezentovaným danou asociáciou s jednou inštanciou druhej triedy. Asociačný vzťah je v podstate obojsmerný, pokiaľ nie je explicitne špecifikovaný ako jednosmerný. Špeciálny typ asociácie je tzv. reflexná asociácia, kedy má trieda asociačný vzťah sama so sebou.

Ďalším veľmi dôležitým vzťahom, ktorý používame k statickému pohľadu na triedy, je **generalizácia** alebo aj **špecializácia**. Pokiaľ nájdeme v analýze tento typ vzťahu, tak v objektovom prostredí znamená dedičnosť (inheritance). Princíp dedenia umožňuje objektovým triedam zdieľať ich charakteristiky vrátane hierarchie dedenia a zároveň uchováva ich rozdiely. V podstate je špecializácia vzťah medzi všeobecnejšou triedou a ďalšou viac spresnenou objektovou triedou, ktorá nasleduje v hierarchii dedení na nižšej úrovni. Podriadené objekty dedia zo svojho predka všetky vlastnosti, t.j. atribúty, relácie, operácie a obmedzenia. Generalizácia sa zapisuje ako plná čiara s prázdnym trojuholníkom na konci, ktorý znázorňuje smer dedenia a ukazuje na nadradenú triedu.

3.2.3 Diagramy interakcie

Sekvenčný diagram slúži na zobrazenie interakcie medzi užívateľmi, systémami a podsystémami. Zameriava sa predovšetkým na chronologickú postupnosť jednotlivých správ a životnosť zúčastnených objektov. V sekvenčnom diagrame môžeme rozlíšiť dve dimenzie. Prvá je dimenzia účastníkov interakcie (lifelines). Tou druhou je čas. Správy reprezentujú komunikáciu medzi dvomi účastníkmi interakcie, ktorá môže znamenať volanie metódy, vytvorenie alebo zrušenie inštancie (účastníka interakcie), alebo zaslanie signálu.

Komunikačný diagram nahrádza predošlý diagram spolupráce z verzie UML 1.x. Na rozdiel od sekvenčného diagramu zdôrazňuje statickú štruktúru (prepojenie objektov), ktorá sa využije pri interakcii na dosiahnutie požadovaného správania, napríklad definovaného nejakým prípadom užitia. Syntax je podobná so sekvenčným diagramom s tým rozdielom, že nezachytáva druhú časovú dimenziu. Je preto veľmi dôležité číslovať postupnosť jednotlivých správ. V oboch spomínaných diagramoch je možné vyjadriť iterácie i podmienky.

3.2.4 Diagramy balíčkov

Zoskupenia tried (balíčky - packages) sú veľmi dôležitým nástrojom pre väčšie projekty. Jednotlivé balíčky obsahujú triedy, ktoré spolu nejako súvisia a komunikujú medzi sebou. Účelom **diagramov balíčkov** (package diagrams) je dekomponovať zložitý systém na menšie podsystémy a zobraziť ich vzájomné vzťahy na balíčkovej (nie triednej) úrovni. Tu je predovšetkým dôležité, aby nekomunikoval každý balík s každým, pretože v takom prípade je návrh nekorektný a je nutné zmeniť obsah jednotlivých balíčkov a tým upraviť ich vzájomnú komunikáciu. V diagrame balíčkov existuje hierarchické zanorenie, a teda balík môže obsahovať iné balíky a zobrazovať vzťahy medzi nimi.

3.2.5 CASE nástroje

CASE nástroje sú nástroje na podporu analýzy a návrhu aplikácií (Computer Aided Software Engineering). V súčasnosti všetky svetové objektovo orientované CASE nástroje vychádzajú z modelovacieho jazyka UML. Najmä pri tvorbe zložitejších informačných systémov sú tieto nástroje absolútne nevyhnutné.

CASE nástroje ponúkajú oproti bežným kresliacim nástrojom oveľa vyššiu funkčnosť, čo sa odráža aj na ich pomerne vysokej cene. Predovšetkým sú to možnosti prepojenia jednotlivých modelov, zachytenie celého analytického návrhu v spoločnom repozitári, ktorý je zdieľaný tímami vývojárov. Ďalej tieto nástroje obvykle zahŕňajú možnosti generovania a synchronizovania kódov objektových prostredí (napr. prostredie .NET, Java, C++, atď.), vrátane databázových skriptov na

založenie databázy, reverzného inžinierstva a správy dátových modelov. Väčšina dnešných CASE nástrojov umožňuje aj procesné modelovanie ako doplnok k technikám UML.

Medzi bežne používané CASE nástroje patria Rational Rose od firmy Rational, Select Component Architect od Select Business Solutions, PowerDesigner od firmy Sybase alebo Visual Paradigm for UML (VP-UML), ktorý vyvinula spoločnosť Visual Paradigm International.

Pri návrhu sieťovej aplikácie mariáš bude použitá práve skúšobná verzia programu Visual Paradigm. Visual Paradigm je multiplatformový a ľahko používateľný CASE nástroj na vizuálne modelovanie UML. VP-UML poskytuje nástroje na rýchlejšie a kvalitnejšie tvorenie aplikácií. Visual Paradigm for UML tiež poskytuje možnosť spolupráce s inými CASE nástrojmi väčšinou svetovo rozšírených integrovaných vývojových prostredí.[8]

4 Analýza a návrh aplikácie

Zo zadania bakalárskej práce vyplývajú isté požiadavky na sieťovú hru mariáš. Hra je určená pre 3-4 hráčov. Aplikácia musí obsahovať algoritmus na miešanie kariet a ďalšie operácie na správne fungovanie, rešpektujúc pravidlá hry. V neposlednom rade je nutné zabezpečiť korektne implementovaný sieťový protokol podporujúci komunikáciu prostredníctvom TCP/IP.

Pri vytváraní počítačovej hry je veľmi dôležité myslieť na užívateľa a tomu podrobiť vzhľad a funkcionalitu celej aplikácie. Grafické užívateľské rozhranie musí byť prehľadné, komfortné a jednoduché. Nie je vhodné, aby sa užívateľ strácal v množstve nastavení a nelogickom usporiadaní okien, tlačidiel, textových polí a pod. Práve intuitívnosť ovládania je jednou z vlastností, ktorú by mala výsledná aplikácia obsahovať.

Počítačové hry sú aplikácie, ktoré veľmi často menia svoj stav. Je preto nutné na tieto zmeny adekvátne reagovať. Vzhľadom na to, že ide o sieťovú aplikáciu, je o to dôležitejšie tieto zmeny správne distribuovať na vzdialené zariadenia a zachovať konzistenciu behu aplikácie na všetkých zúčastnených počítačoch. Aplikáciu bude preto potrebné podrobiť dôkladnému testovaniu, aby sa predišlo chybám v komunikácii, čo by mohlo spôsobiť neadekvátne správanie alebo haváriu jednotlivých inštancií sieťovej aplikácie mariáš bežiacej na vzdialených počítačoch.

V dnešných programoch sa už iba ťažko zaobídeme bez prístupu k externým zdrojom. Aplikácia mariáš v tomto ohľade nebude výnimkou a bude uchovávať užívateľské nastavenia v XML formáte. Ďalšie externé dáta budú súbory so slovníkmi, ktoré budú uchovávať jednotlivé slová v rôznych jazykoch. Použitie internacionalizácie programu bude ďalším príspevkom ku komfortu užívateľa. Aplikácia bude teda ponúkať štyri rôzne jazyky: slovenčinu, češtinu, angličtinu a nemčinu.

V aplikácii bude možné nastaviť farbu pozadia a tiež rôzne štýly lícnych a rubových strán kariet a okrem bežných užívateľských nastavení bude možné meniť aj nastavenia samotnej hry.

Okrem už spomínaných funkcií bude v programe zahrnutá aj možnosť textovej komunikácie medzi hráčmi (tzv. chat). Beh aplikácie budú sprevádzať vhodné zvukové efekty a hudba.

4.1 Sieť

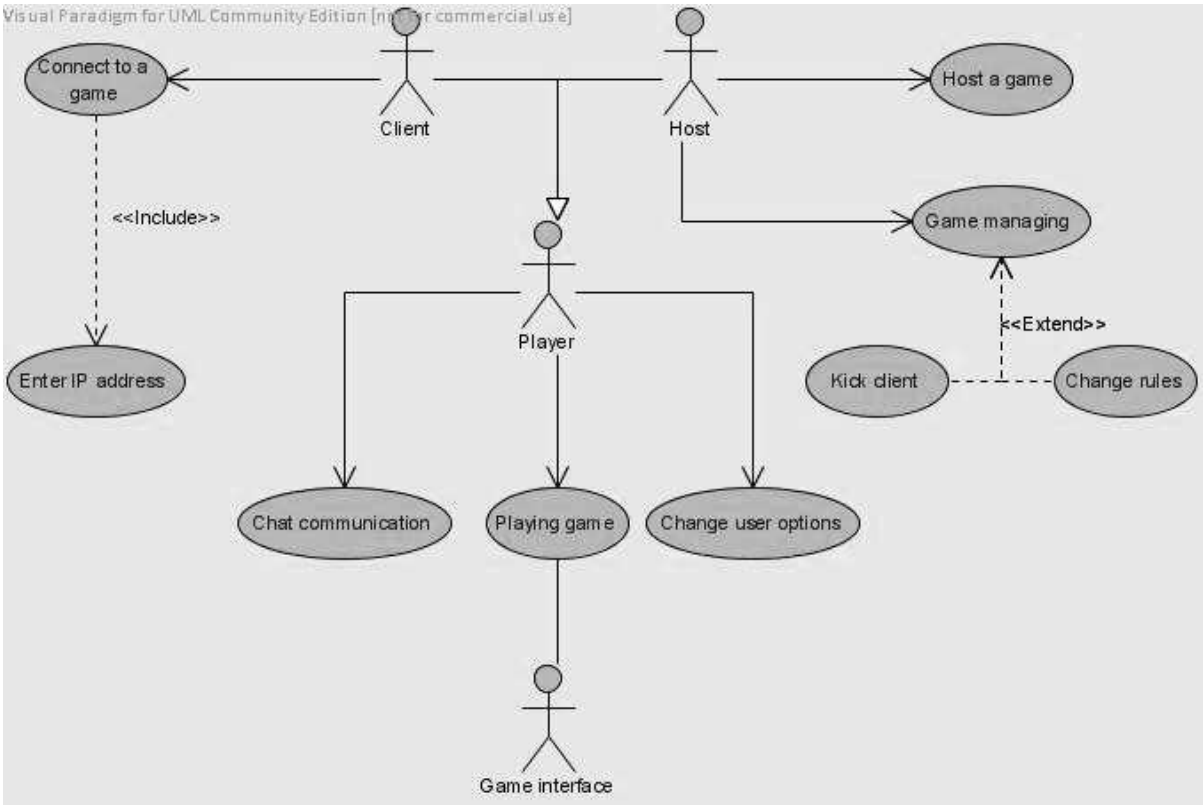
Pri vytváraní sieťovej architektúry prichádzajú do úvahy dve alternatívy. Tou prvou je tzv. peer-to-peer (peer je v preklade seberovný alebo rovesník). Tento variant poskytuje jednoduchý spôsob komunikácie a zdieľania súborov medzi počítačmi. Je pomerne jednoduché ho implementovať, nie sú potrebné zložité nastavenia ani údržba. Peer-to-peer je založený na vzájomnom spojení všetkých uzlov v sieti. Takže v praxi to znamená, že sieťová aplikácia, založená na takomto princípe, odosiela všetky udalosti každému pripojenému počítaču. Tento spôsob komunikácie je vhodný pre menšie siete, kde je toto riešenie lacné a postačujúce. Peer-to-peer nerieši žiadnym spôsobom bezpečnosť, takže každý môže na sieti zdieľať čokoľvek úplne s každým. Okrem problémov s bezpečnosťou môže nastať preťaženie siete, predovšetkým pri sieťach s veľkým množstvom pripojených zariadení.

Druhým variantom je architektúra klient – server. Táto možnosť vyžaduje náročnejšiu implementáciu, keďže je nevyhnutné implementovať zvlášť správanie klienta aj serveru. Pri tejto architektúre je server akýmsi centrálnym bodom. Klienti nekomunikujú priamo medzi sebou, ale iba so serverom, ktorý posiela všetky správy na určené miesto v sieti. Takýmto spôsobom môžeme výrazným spôsobom znížiť zaťaženie siete a zároveň zvýšiť bezpečnosť, pretože na server je možné implementovať rôzne ochranné nástroje (firewall a i.). Server ale vyžaduje náročnejšiu konfiguráciu a tiež je náročnejšie ho spravovať.

Zvolený variant pre sieťovú aplikáciu mariáš je architektúra typu klient – server. Pre sieťové hry je vo všeobecnosti zvyčajnejšou alternatívou. Jedným z problémov pri peer-to-peer architektúre by bolo zisťovanie všetkých pripojených klientov. Ako ďalší problém sa javí spravovanie nastavení a hry samotnej. Toto bude lepšie riešiť architektúra klient – server, pričom server bude klientom iba posielať správy o zmenách. Táto zvolená architektúra bude náročnejšia na implementáciu, ale v konečnom dôsledku prinesie oveľa viac výhod a zjednodušení v ďalšej fáze tvorby aplikácie.

4.2 Funkcionalita

Pri návrhu nejakej aplikácie je veľmi dôležité vymedziť funkcionalitu daného programu. Je veľmi dobré stanoviť si jednotlivé možnosti, ktoré môže užívateľ vykonávať. Keďže ide o sieťovú aplikáciu, je veľmi užitočné rozlíšiť možnosti klienta a hostiteľa (server). Nasledujúci diagram prípadu užitia sa snaží o zachytenie práve týchto možností.



Obrázok 1: Diagram prípadu užitia (Use Case)

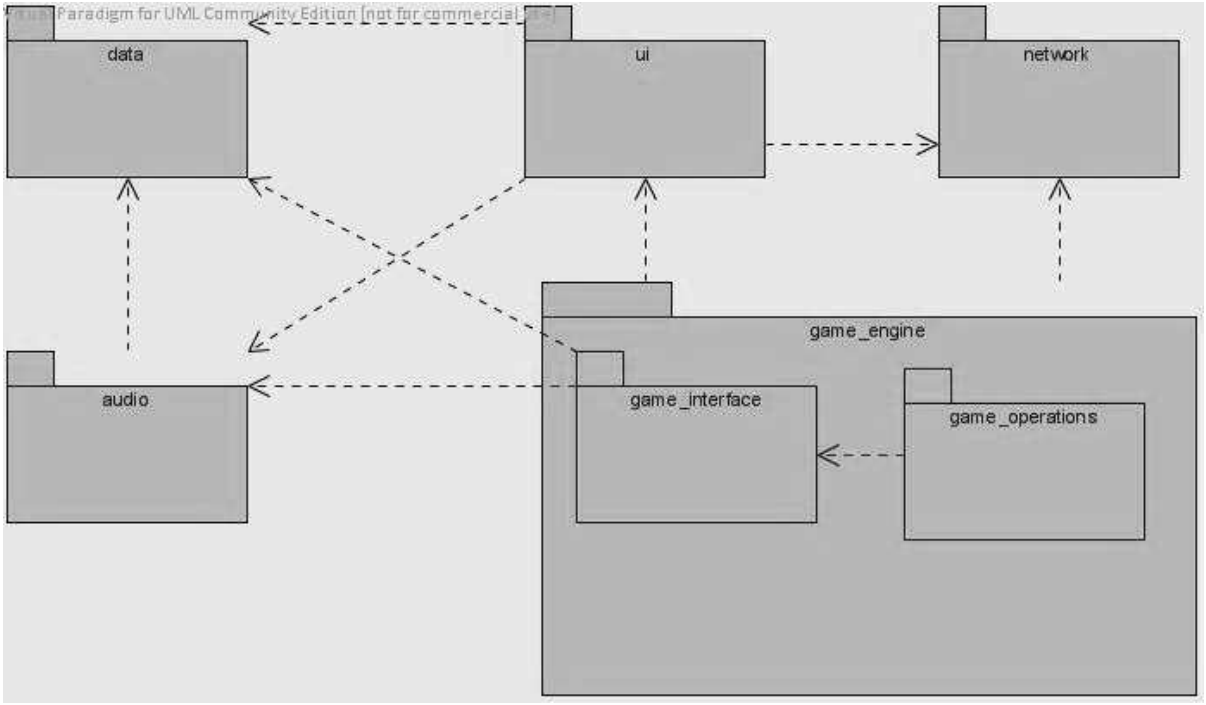
Ako je z obrázku 1 vidieť, klient a hostiteľ majú odlišné možnosti interakcie s aplikáciou. Ich spoločné možnosti sú zahrnuté v aktérovi Player (hráč), ktorý je generalizáciou klienta a hostiteľa. Pri pripájaní klienta je samozrejmosťou zadanie IP adresy, čo naznačuje relácia <<include>>. Server má pod svojou kontrolou správu vytvorenej hry. Môže teda využiť niektorú z možností ako odpojiť klienta alebo zmeniť pravidlá. Počas hrania samotnej hry dochádza ku komunikácii so samotným herným rozhraním, ktoré je tu vyznačené vo forme aktéra. Toto rozhranie je abstrakciou celého herného enginu a v diagrame obstaráva všetky aspekty nevyhnutné pre hranie hry mariáš. Nasleduje konkrétny prípad použitia Use Case diagramu na obrázku 1.

Prípad použitia: PripojenieKlientaKServeru
ID: 2
Stručný popis: Klient sa pripojí k serveru.
Primárni aktéri: Klient
Sekundárni aktéri: Žiadny
Predpoklady: Je vytvorená nová hra a server beží.
Hlavný tok: 1. Prípad použitia sa spustí, keď klient vyberie možnosť „pripojiť k hre“. 2. Pokiaľ nie je zadaná IP adresa platná. 2.1 Systém požaduje korektný formát IP adresy. 2.2 Systém overí zadanú IP adresu. 3. Systém sa pokúša pripojiť klienta k serveru. 3.1 Klient je pripojený k serveru a získava od serveru potrebné dáta.
Nasledujúce podmienky: 1. Boli vytvorení noví hráči v zozname hráčov. 2. Inicializovali sa nastavenia hry.
Alternatívne toky: NeplatnáIPAdresa NemožnoSaPripojiť ServerJePlný Storno

Alternatívny tok: ServerJePlný
ID: 2.3
Stručný popis: Server odošle informáciu o tom, že je plný a klienta odpojí.
Primárny aktéri: Žiadny
Sekundárni aktéri: Klient
Predpoklady: Klient sa pokúša pripojiť k serveru.
Alternatívny tok: 1. Alternatívny tok sa spustí po kroku 3 hlavného toku. 2. Systém na strane serveru registruje nové pripojenie. 2.1 Systém zisťuje, že nemá voľnú kapacitu pre ďalšieho hráča. 2.2 Systém odosiela informáciu klientovi o tom, že je plný a klienta následne odpája. 3. Systém na strane klienta získa informáciu od serveru. 3.1 Systém upozorňuje klienta hlásením, že server je plný.
Následné podmienky: Žiadne

4.3 Architektúra aplikácie

V nasledujúcej kapitole sa budeme venovať návrhu architektúry programu. Tá bude zahŕňať predovšetkým vytvorenie balíčkov, vymedzenie ich funkcie a v neposlednom rade komunikáciu medzi nimi.



Obrázok 2: Diagram balíčkov (Package diagram)

Balíček *data* bude obsahovať triedy pracujúce s externými dátami a zaisťovať ich ukladanie a načítavanie. Medzi užívateľské dáta sa bude radiť meno hráča, naposledy použitá IP adresa a názov sieťového rozhrania. Ďalšie ukladané dáta sú zvolený jazyk, hlasitosť hudby a zvukov, farba pozadia a štýl lícnej a rubovej strany hracích kariet. V tomto balíku sa teda rieši predovšetkým perzistencia dát.

Balíček *audio* zodpovedá za zvukové efekty a hudbu. Ďalej tu bude figurovať ovládací prvok na kontrolu hlasitosti. Úroveň hlasitosti sa bude získavať z balíčku *data*.

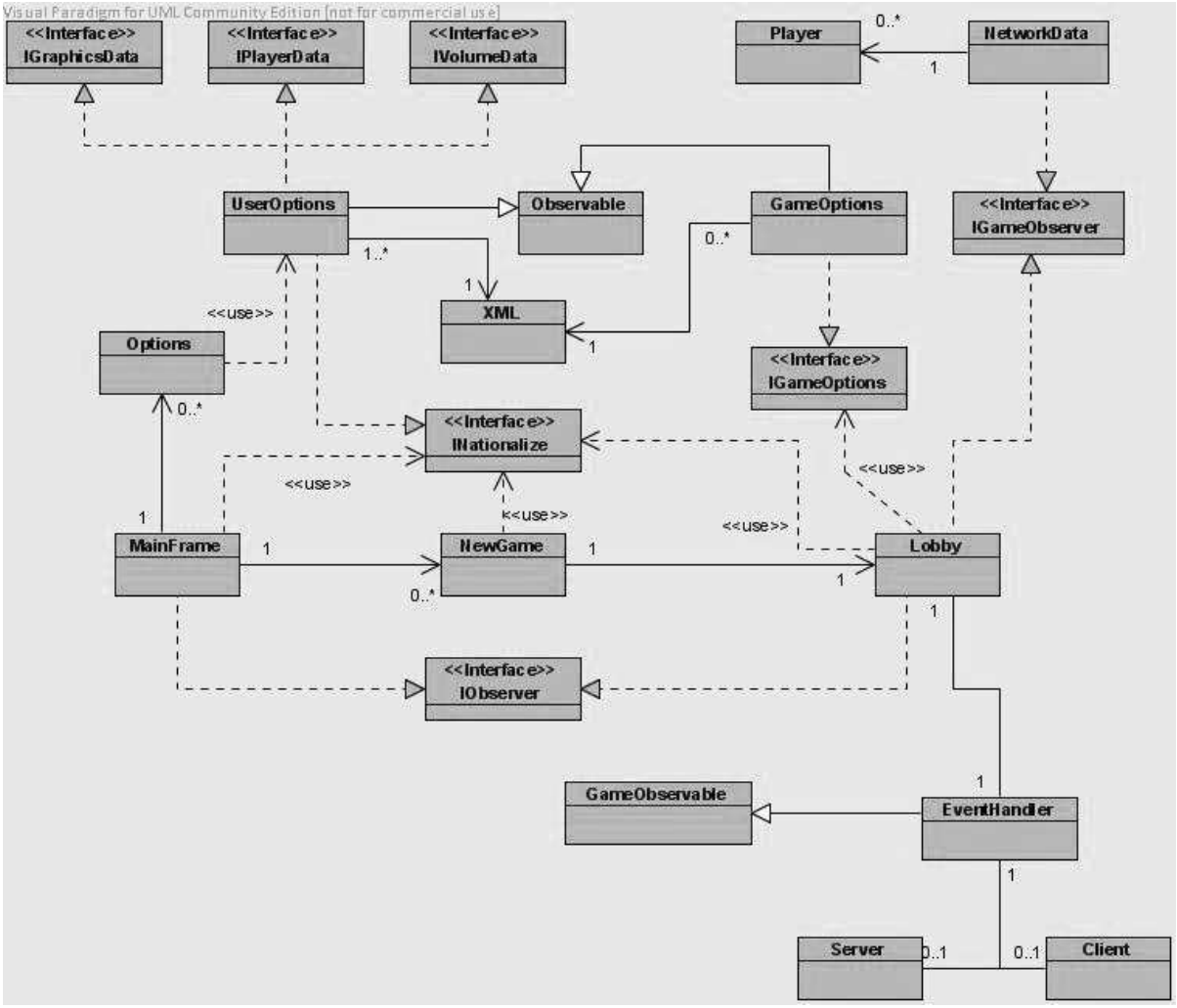
Balík *network* zahŕňa sieťovú komunikáciu. Budú sa tu nachádzať triedy implementujúce architektúru typu klient – server. V tomto balíčku sa budú ďalej ukladať informácie o všetkých hráčoch (meno, ID, stav financií a pod.).

Balíček *ui* zahŕňa triedy pracujúce s grafickým užívateľským rozhraním. Tieto budú vytvorené pomocou knižnice Swing. Každá trieda bude schopná zistiť používaný jazyk a tým preložiť všetok zobrazovaný text do požadovaného jazyka. Bude tu použitý návrhový vzor observer, ktorého úlohou je upozorniť na zmeny všetky naslúchajúce triedy. V tomto prípade budú triedy upozorňované pri zmene v balíčku *data*. Balík *ui* ďalej využíva balík *audio* pri prehrávaní zvukových efektov a hudby a tiež balík *network* na získanie dát od ostatných pripojených hráčov, ktoré budú zobrazované v užívateľskom rozhraní.

Balíček *game_engine* obsahuje triedy priamo ovládajúce samotnú kartovú hru mariáš, rešpektujúc všetky jej pravidlá. Tento balík obsahuje balík *game_interface*, ktorý bude obsahovať grafické užívateľské rozhranie samotnej hry (vzhľad kariet, Swingové panely, layouty a pod.) a bude čerpať z balíkov *audio* a *data*. Balík *game_operations* bude obsahovať predovšetkým operácie súvisiace s kartami (vyťahovanie z balíka, miešanie, rozdávanie a pod.)

4.4 Štrukturálny návrh

Štrukturálny návrh vychádza predovšetkým z diagramu tried, ktorý zobrazuje interakciu jednotlivých tried v objektovo orientovanom návrhu. Z dôvodu obmedzeného rozsahu tejto práce, nebudú v nasledujúcich UML diagramoch zahrnuté všetky triedy a ich rozhrania.



Obrázok 3: Prehľadový diagram tried (Class diagram)

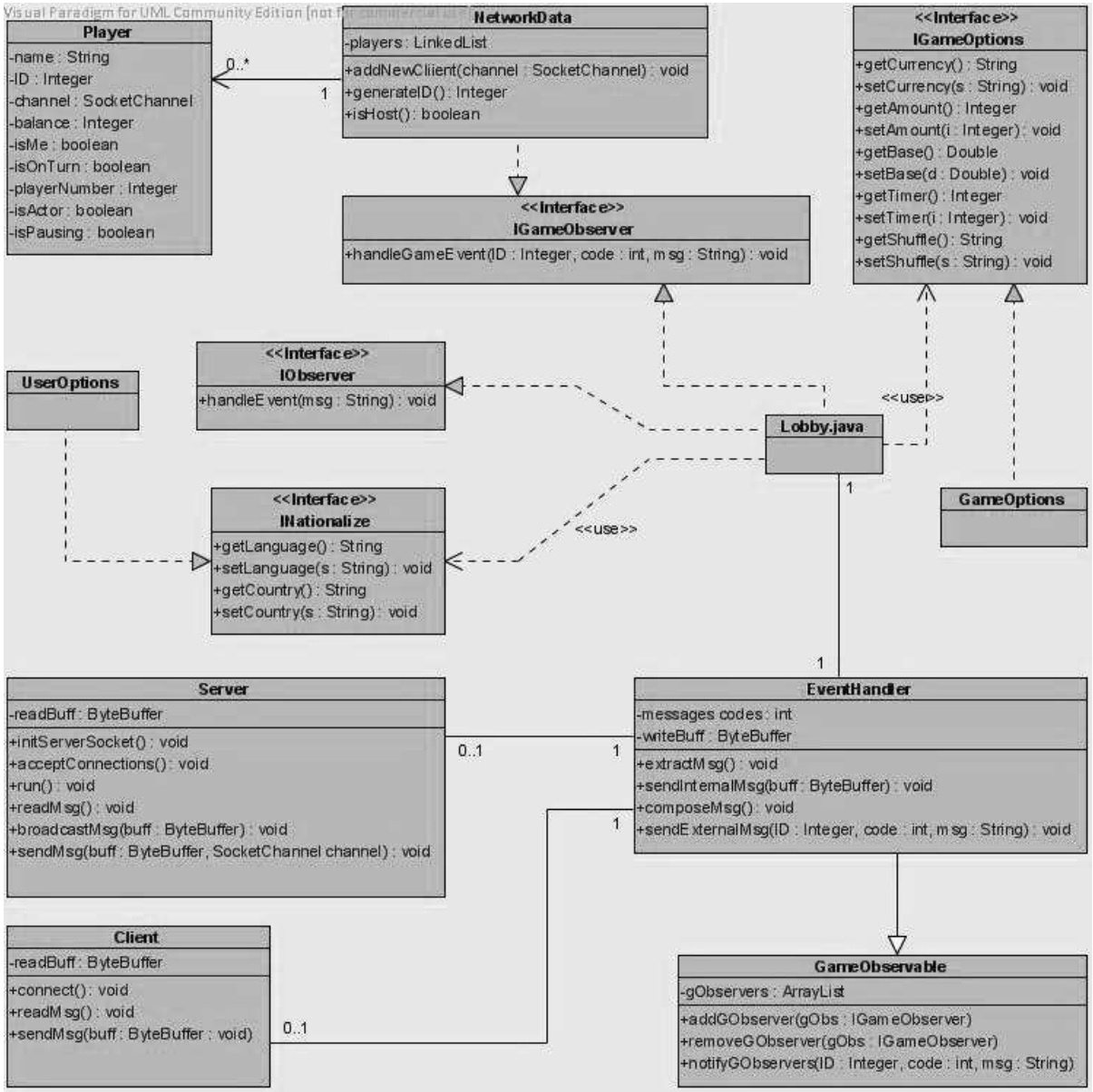
Diagram na obrázku 3 ukazuje komunikáciu niektorých tried. Nerieši, aké operácie a atribúty tieto triedy obsahujú, keďže bola snaha zahrnúť čo najväčšie množstvo tried a prehľadne tak načrtnúť ich interakciu.

Najskôr si rozoberieme triedy a rozhrania patriace do balíka *data*. Trieda *UserOptions* realizuje štyri rozhrania: *IGraphicData*, *IPlayerData*, *IVolumeData a INationalize*. Ďalej táto trieda dedí z triedy *Observable*, ktorá implementuje návrhový vzor observer. Týmto spôsobom bude možné upozorniť každý observer v prípade zmeny v užívateľských dátach. Trieda *XML* poskytuje metódy na vstupno-výstupné operácie so súborom typu xml a poskytuje tieto metódy triedam *UserOptions* a *GameOptions*. Trieda *GameOptions* realizuje rozhranie *IGameOptions* a funguje podobne ako trieda *UserOptions*.

Triedy *Options*, *MainFrame*, *NewGame* a *Lobby* patria do balíku *ui* a budú dediť z niektorého zo Swingových kontajnerov. Na zobrazenie textu v správnom jazyku musia všetky tieto triedy

využívať metódy poskytované rozhraním *INationalize*. Niektoré z týchto tried implementujú rozhranie *IObserver* a to poskytuje možnosť okamžite reagovať na každú zmenu užívateľských dát.

Do balíku *network* sa radia triedy *Player*, *NetworkData*, *EventHandler*, *GameObservable*, *Client* a *Server*. Trieda *NetworkData* uchováva údaje o jednotlivých hráčoch v zozname. Na tento účel využíva triedu *Player*. Trieda *EventHandler* dedí z *GameObservable*, čím vlastne vytvára druhý typ observeru – sieťový. Funguje rovnako ako spomínaný observer vyššie, ale narábanie s jednotlivými správami na strane observerov bude odlišné. Primárnym účelom triedy *EventHandler* je zbierať zachytené správy od tried *Server* a *Client* a odosielať ich observerom (v tomto prípade triedam *NetworkData* a *Lobby*), aby dokázali reagovať na zmenu ostatných pripojených užívateľov. *Eventhandler* túto funkciu vykonáva aj opačne, a teda zbiera informácie od observerov a zasiela ich triedam *Client*, resp. *Server*.



Obrázok 4: Diagram tried (Class diagram)

Diagram tried na obrázku 4 nám už ukazuje aj niektoré atribúty a operácie, ktoré jednotlivé triedy poskytujú. Ako je z uvedeného diagramu poznať, trieda *Player* sa bude dať prirovnať k užívateľskému dátovému typu a bude poskytovať metódy na prácu s dátami (budú to „get“ a „set“

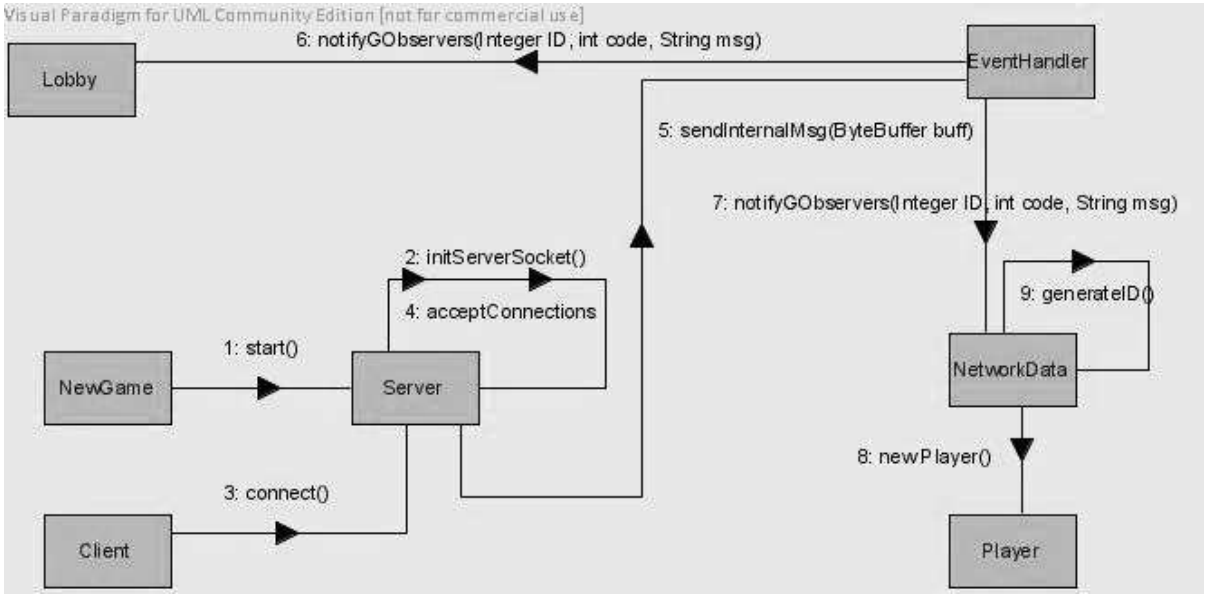
metódy, ktoré v diagrame nie sú uvedené z kapacitných dôvodov), ktoré uchováva. Rozhrania *IObserver* a *IGameObserver*, ktoré poskytujú jednotlivým triedam možnosť reagovať na jednotlivé udalosti, sú pomerne jednoduché a obsahujú iba jednu operáciu.

Trieda *EventHandler* bude okrem upozorňovania jednotlivých tried na nejakú hernú zmenu pracovať aj so zásobníkmi. Prichádzajúce dáta od tried *Server* resp. *Client* vo formáte typu zásobník bude musieť „rozoberať“ a následne pomocou triedy *GameObservable*, ktorú rozširuje, upozorňovať jednotlivé observery.

Triedy *UserOptions* a *GameOptions* iba implementujú korešpondujúce rozhrania a ďalšie operácie neponúkajú. Trieda *Lobby* sa stará predovšetkým o grafické užívateľské rozhranie a korektné reakcie na správy získané prostredníctvom observerov. Taktiež komunikuje s triedou *EventHandler* pri zmene vykonanej užívateľom.

4.5 Návrh správania

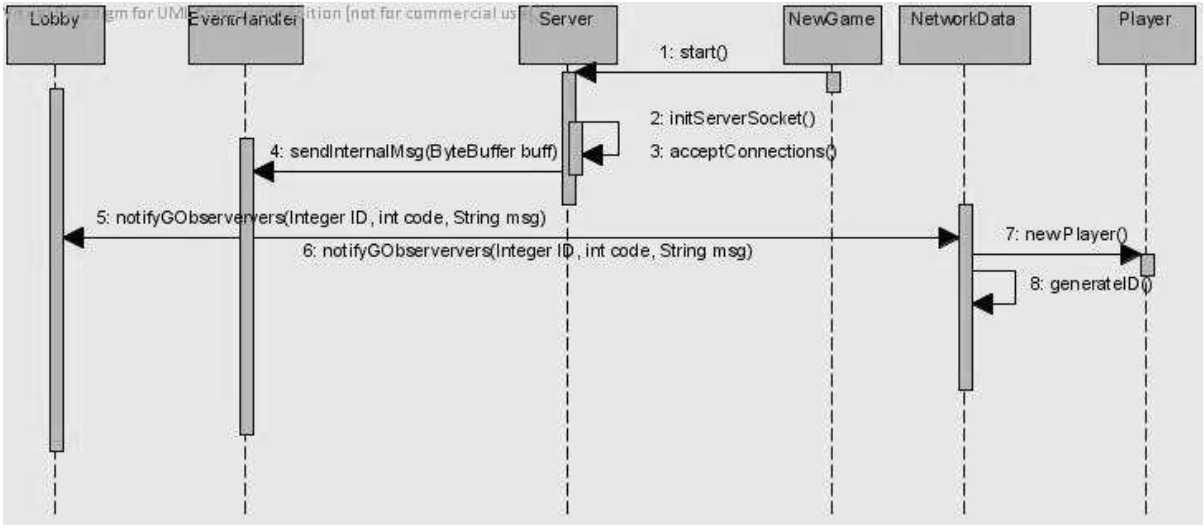
Ukážka správania aplikácie bude demonštrovaná na dvoch UML diagramoch. Tým základným je komunikačný diagram, ktorý bude popisovať sled jednotlivých operácií, ktorými je zabezpečená interakcia medzi triedami. Nasledujúce diagramy budú z rôzneho hľadiska zobrazovať pripojenie klienta k serveru a následný sled správ v systéme.



Obrázok 5: Diagram komunikácie (Communication diagram)

Trieda *NewGame* patrí do užívateľského rozhrania a užívateľ pomocou nej spustí server. Nasleduje inicializácia serveru. Od triedy *Client* prichádza žiadosť o pripojenie. *Server* upozorní *EventHandler*, ktorý prostredníctvom observeru upozorní príslušné triedy. V triede *NetworkData* sa vytvorí nový hráč. Trieda *Lobby* je na túto skutočnosť upozornená a rezervuje novému hráčovi slot v miestnosti.

K behaviorálnym modelom patrí aj sekvenčný diagram. Ten nám ponúka druhú časovú dimenziu a môžeme prehľadnejšie sledovať jednotlivé operácie tej istej činnosti.



Obrázok 6: Sekvenčný diagram (Sequence diagram)

Pre väčšiu prehľadnosť sa v diagrame na obrázku 6 nenachádza trieda *Client*. Predpokladáme však, že k pripojeniu dôjde pri tretej operácii, podobne ako v hornom komunikačnom diagrame.

5 Implementácia

Implementácia prebiehala už v spomínanom jazyku Java a v integrovanom vývojovom prostredí NetBeans. Oproti návrhu došlo k niekoľkým zmenám. Pribudlo zopár balíčkov a tried, kvôli väčšej prehľadnosti a korektnosti v rámci objektovo orientovaného prístupu. Táto kapitola bude rozdelená na niekoľko logických celkov s ohľadom na implementovanú aplikáciu.

5.1 Dáta

Najskôr sa pozrieme na zabezpečenie perzistencie dát a na to, ako vo výsledku fungujú jednotlivé triedy. Užívateľské dáta sú rozdelené do logických celkov. Rozhranie *IPlayerData* poskytuje metódy na nastavenie a ukladanie mena hráča, IP adresy a sieťového rozhrania. Je to preto, že tieto tri údaje nie sú využívané v triede užívateľského rozhrania *Options*, ale v triede *NewGame*. Rozhranie *INationalize* je používané v každej triede užívateľského rozhrania, keďže pracuje s informáciami obsahujúcimi aktuálny jazyk a krajinu. Rozhranie *IGraphicsData* je zasa používané v balíku *game_engine* a jeho podbalíčkoch. *IVolume* pracuje s nastavením hlasitosti a zvukových efektov a ako také sa využíva najmä v balíku *audio*. Všetky tieto rozhrania sú realizované triedou *UserOptions*, ktoré pri každej zmene týchto nastavení upozorňuje naslúchajúce triedy prostredníctvom návrhového vzoru observer.

Rozhranie *IGameOptions* už poskytuje metódy na nastavovanie herných nastavení. Konkrétne ide o ukladanie a získavanie množstva peňazí pre každého hráča, peňažného základu, meny, spôsobu miešania a časovača. Toto rozhranie je realizované triedou *GameOptions*.

Trieda *XML* obsahuje predovšetkým statické metódy *getElem* a *setElem*. Názvy týchto metód napovedajú, že budú vykonávať I/O operácie práve s XML súborom, v ktorom budú uložené jednotlivé nastavenia. Triedy *UserOptions* a *GameOptions* budú práve tieto dve statické metódy využívať pri práci s jednotlivými dátami. Trieda *XML* ďalej obsahuje metódu, ktorá dokáže vygenerovať predvolený XML súbor. K tejto situácii dochádza pri zachytení výnimky, ktorá indikuje neexistujúci alebo poškodený XML súbor.

5.2 Grafické užívateľské rozhranie

Na vytvorenie grafického užívateľského rozhrania je použitá už spomínaná knižnica Swing. Ide teda o aplikáciu fungujúcu v okne a používajúcu Swingové komponenty a kontajnery. Swingový framework poskytovaný integrovaným vývojovým prostredím NetBeans nie je pri tejto práci použitý, pretože obmedzuje programátorské zásahy do generovaného kódu.

Prvou triedou, ktorá sa po spustení aplikácie vytvorí, je *MainFrame*. Táto trieda obsahuje jednoduché kontextové menu v ľavom hornom rohu, odkiaľ užívateľ vyberá jednotlivé možnosti a ovláda tak aplikáciu. *MainFrame* dedí z triedy *JFrame*.

Všetky triedy v balíčku *ui*, ktoré dedia z niektorého zo Swingových kontajnerov, majú ako rodiča (parent) práve triedu *MainFrame* a vzhľadom na ňu určujú svoju polohu. Tieto triedy majú niektoré spoločné názvy metód, ktoré rozdielne implementujú. V konštruktore si trieda vždy vytvorí potrebné inštancie z balíčku *data* a tým získa užívateľské a herné dáta. Niektoré triedy dediace zo Swingových kontajnerov implementujú rozhranie *IObserver* a práve v konštruktore sa pridávajú k naslúchajúcim triedam. Metóda *initDialog()* nastavuje veľkosť okna, predvolenú operáciu pri zatvorení okna, pozíciu a viditeľnosť okna. Metóda *initComponents()* inicializuje jednotlivé Swingové komponenty a nastavuje niektoré ich vlastnosti. Metóda *internationalize()* pridáva ku

komponentom text v príslušnom jazyku. Tento text získava pomocou tried integrovaných v Jave (*Locale*, *ResourceBundle*). Táto metóda nie je volaná iba pri inicializácii triedy a jej okna, ale vždy, keď užívateľ zmení jazyk v nastaveniach (trieda *Options*). Reakciu na túto zmenu zabezpečuje rozhranie *IObserver*. Ďalšia metóda, ktorú obsahuje väčšina Swingových tried, je *setLayout()*. Účelom tejto operácie je prehľadne zoradiť jednotlivé Swingové komponenty a zobraziť ich v okne. Metóda *addListener()* tiež patrí k obligatórnym pri Swingových triedach. Jej funkciou je pridávať k jednotlivým komponentom inštancie tried (listeners), ktoré umožňujú naslúchať a zachytávať určité akcie na týchto komponentoch a vykonávať tak reakcie na rozličné podnety.

Jednou z najrozsiahlejších tried v celej aplikácii je trieda *Lobby*. Je to preto, že okrem operácií, ktoré sú spomínané v predošlom odseku, musí implementovať pomerne zložité reakcie na udalosti prichádzajúce od klienta, resp. serveru. Táto trieda obsahuje veľa listenerov, pretože sa mnohé komponenty môžu meniť. *Lobby* ďalej musí odlíšiť klienta od serveru a pre každého zaručiť prístup k potrebným operáciám a komponentom a tiež ich prístup k niektorým možnostiam obmedziť. Bolo teda nutné vytvoriť metódy, ktoré využíva výlučne server, a potom také, ktoré sú zasa určené iba pre klienta. Napr. mnohé listenery nie sú pridávané ku komponentom na strane klienta, pretože k nim nemá prístup, a tento prístup má iba server. Jednou zo „serverových“ metód je aj *serverComboAction(int n)*, kde „n“ je číslo combo boxu, kde sa môže nachádzať pripojený klient. Každý zo štyroch combo boxov uchováva stav (otvorený, zatvorený, obsadený) a pokiaľ sa v ňom nachádza meno hráča, tak je box obsadený, inak je prístupný, resp. neprístupný novým pripojeniam v závislosti od toho, či je otvorený alebo zatvorený. Užívateľ tak dokáže jednoducho odpojiť hráča kliknutím na combo box a vybratím možnosti „Otvorený“ resp. „Zatvorený“ (tento systém pripájania a odpájania hráčov bol inšpirovaný hrou Warcraft III). Tento celý proces odpájania spracúva práve metóda *serverComboAction(int n)*. Na druhej strane metóda výlučne používaná klientom *clientInit()* sa spúšťa po pripojení k serveru a prostredníctvom získaných údajov pridáva hráčov do miestnosti, mení pravidlá hry a pod.(inicializuje miestnosť).

Oproti pôvodnému návrhu sa zmenilo použitie návrhového vzoru observer. Bol vytvorený zvlášť balík pre s názvom *observer*, kde sú triedy aj rozhrania potrebné na správne fungovanie observeru. Neexistujú už dva typy observerov, ale boli zlúčené do jedného, poskytujúceho reakcie na užívateľské zmeny (*handleEvent(String msg)*), ako aj na zmeny prichádzajúce od sieťových tried (*handleGameEvent(int ID, int code, String msg)*).

5.3 Sieť

Počas implementácie sieťovej časti aplikácie pribudli k pôvodnému návrhu ďalšie dva balíčky. Balík *network* ostal, ale nachádzajú sa v ňom iba triedy, ktoré používajú spoločne server aj klient. Triedy, ktoré spoločné nemajú, sú rozdelené do samostatných balíkov *network.server* a *network.client*.

Najskôr si rozoberieme triedy obsiahnuté v spoločnom balíčku *network*. Trieda *Events* neobsahuje žiadne metódy. Jej obsah je obmedzený len na statické konštanty celočíselného typu. Tieto konštanty sú kódy udalostí, ktoré každú udalosť jednoznačne identifikujú a prijímateľ takejto udalosti vie, ako naložiť so správu uloženou vo forme reťazca. Trieda *InternalMsg* je určená predovšetkým na prácu so zásobníkmi. Každý zásobník, ktorý získa od triedy *Client* alebo *Server*, je extrahovaný na niekoľko elementov. Jednotlivé elementy sú dĺžka správy, ID hráča, ktorý správu odoslal, kód udalosti a textová správa. Po „rozobratí“ zásobníka informuje trieda *InternalMsg* všetky naslúchajúce triedy prostredníctvom metódy *notifyGameObservers(int ID, int code, String msg)*, ktorú dedí z triedy *Observable*. Trieda *NetworkData* uchováva všetky potrebné dáta, a to predovšetkým prostredníctvom zoznamu hráčov (každý hráč je inštanciou triedy *Player*). *NetworkData* ďalej poskytuje metódy na vyhľadanie hráčov podľa rôznych kritérií a ukladanie

pravidiel hry, ktoré nastavuje server. Rozhranie *IPlayer* poskytuje metódy na získavanie a ukladanie informácií do objektov typu *Player* ako *getID(),setID(int ID), getChannel(), setChannel(SocketChannel channel)* a pod. Posledná trieda *Chat* sa stará o komunikáciu medzi klientmi. Dedí zo Swingového rozhrania JDialog a umožňuje písať a posielať užívateľovi správy ostatným hráčom v rozsahu do 400 znakov.

Balík *network.server* obsahuje tri triedy. Tou prvou je *Server*. Táto trieda je ďalším vláknom aplikácie a funguje na jej pozadí. Jej úlohou je v potenciálne nekonečnom cykle prijímať správy a nové pripojenia. Po prijatí akejkoľvek správy vytvorí novú inštanciu *InternalMsg* a ďalej odošle túto správu všetkým klientom broadcastovým spôsobom. Trieda *ServerData* dedí z triedy *NetworkData* a reaguje prostredníctvom observeru na správy typu *InternalMsg*. Dokáže tak pridávať nových hráčov do zoznamu a meniť ich atribúty, alebo aj reagovať na prichádzajúce správy odoslaním novej správy prostredníctvom triedy *ServerExternalMsg*. Práve *ServerExternalMsg* sa stará o naplnenie každého zásobníka tak, aby bol pripravený na odoslanie, a následne volá príslušnú metódu v triede *Server*, čím sa správa odošle.

Balík *network.client* funguje podobne, ale jednoduchšie ako *network.server*. Trieda *Client* neodosiela broadcastové správy a *ClientData* neukladá toľko informácií (napr. klient nemá uložené kanály všetkých hráčov, ale iba ten, prostredníctvom ktorého komunikuje so serverom). Reakcie klienta sú na jednotlivé správy odlišné a vo všeobecnosti ide o jednoduchšiu implementáciu oproti implementácii serveru.

5.4 Herná logika

Herná logika (herný engine) pozostáva z troch balíkov. Ide o *game_engine* a jeho podbalíky *game_engine.ui* a *game_engine.cards*. V balíku *engine* väčšinu operácií spravujú triedy *ServerGameCore* a *ClientGameCore*, ktoré rozširujú triedu *GameCore*. *GameCore* sa predovšetkým stará o pravidlá hry a manipuláciu s jednotlivými panelmi, kde budú zobrazené mariášové karty. Dalo by sa povedať, že tu sú všetky metódy, ktoré majú klient a server spoločné. *ServerGameCore* volá metódy, ktoré pracujú s balíčkom hracích kariet, odosiela informácie užívateľskému rozhraniu a tiež informuje o zmenách stavov hry balík *network*, spomínaný v predošlej kapitole. *ClientGameCore* funguje podobne, ale predovšetkým reaguje na udalosti od serveru.

Balíček *game_engine.cards* obsahuje operácie na prácu s kartami. Tam spadá vytvorenie balíčka kariet, miešanie, rozdávanie a ukladanie kariet na kôpky. Sú tu implementované dva rôzne algoritmy na miešanie. Prvý z nich náhodne prehádže jednotlivé karty, druhý sa snaží simulovať skutočné ručné miešanie. Jednotlivé karty, ktoré sú reprezentované triedou *Card*, obsahujú okrem farby a hodnoty aj cestu k obrázku, ktorý danej karte zodpovedá.

Balík *game_engine.ui* obsahuje triedu *GameMainPane*, ktorá je koreňovým Swingovým kontajnerom pre všetky komponenty súvisiace s hrou mariáš. Sem sa zaraďuje aj trieda *MyPlayerPan*, ktorá obsahuje hráčove karty. Tie sú zobrazované pomocou triedy *CardPane*, ktorej každá inštancia reprezentuje práve jednu zobrazovanú kartu. V grafickom užívateľskom rozhraní sú ešte zahrnuté aj ďalšie komponenty, ktoré zobrazujú zakryté karty súperov. V balíku *game_engine* nájdeme aj triedy, ktoré zobrazujú dialógové okná pri voľbe variantu hry. Taktiež sú tu triedy, ktoré sa zaoberajú zobrazením štatistík hry po každom kole, ako aj na konci hry.

6 Testovanie

Testovanie prebiehalo predovšetkým lokálne, ale komunikácia medzi serverom a klientmi bola odskúšaná aj na viacerých zariadeniach prostredníctvom ethernetového pripojenia. Hlavná pozornosť pri testovaní bola venovaná práve komunikácii serveru a klienta. Odhalilo sa niekoľko chýb pri odosielaní jednotlivých udalostí a bolo treba zabezpečiť predovšetkým reakcie serverovej strany na rôzne výnimky. Išlo najmä o výnimku v prípade náhleho a neočakávaného ukončenia pripojenia zo strany klienta (napr. odpojením sieťového kábla). Server pri tejto výnimke správne reaguje a upozorňuje na túto udalosť aj ostatných pripojených hráčov. Rovnako aj klient pri neočakávanom páde serveru korektne reaguje a zruší zoznam so všetkými uloženými hráčmi.

Testovanie ďalej ukázalo nutnosť reagovať na možnosť poškodenia či zmazania externých súborov, ktoré obsahujú nastavenia alebo obsahujú niektorý zo slovníkov. Táto výnimka bola tiež ošetrená a to tak, že boli vytvorené triedy, ktoré obsahujú dáta s predvolenými údajmi.

6.1 Softvérové a hardvérové požiadavky

Aplikácia bola testovaná na operačných systémoch Windows 7 a Windows XP SP2. Aby správne fungovala v unixových systémoch, muselo by pravdepodobne dôjsť k niekoľkým drobným úpravám. Rovnako ako všetky aplikácie naprogramované v Jave aj sieťová hra mariáš potrebuje k svojmu spusteniu JRE (Java Runtime Environment) alebo JDK (Java Development Kit).

Hardvérové požiadavky nie sú na dnešné pomery obzvlášť vysoké. Aplikácia potrebuje na svoj bezproblémový beh iba okolo 30 MB operačnej pamäte. Samotná hra zaberá na disku iba necelý megabajt plus externé dáta, ako sú zvuky a obrázky hracích kariet. Náročnosť na grafickú kartu je minimálna. Aplikácia je teda prístupná aj starším typom počítačov.

7 Záver

V tejto práci boli vysvetlené pravidlá hry mariáš, použité technológie na vypracovanie písomnej aj programovej časti tejto bakalárskej práce a jednotlivé návrhy a postupy pri implementácii kartovej hry mariáš pre 3-4 hráčov.

Samotná sieťová aplikácia mariáš plní svoju úlohu. Je to prehľadná a jednoduchá aplikácia, ktorá si určite nájde svoje uplatnenie. Grafická stránka programu je nenáročná, ale poskytuje hráčovi intuitívne užívateľské rozhranie, aby si mohol vychutnať mariáš so všetkým, čo táto geniálna kartová hra ponúka. Dôležitým cieľom tejto aplikácie bolo demonštrovať sieťovú komunikáciu typu klient – server, ktorá je pri počítačových hrách veľmi častou voľbou. Pre bežného užívateľa je ale najdôležitejšie, aby samotná hra fungovala bez problémov, rešpektujúc pravidlá hry. Práve detaily v pravidlách hry, ktoré sa trochu líšili na rôznych webových stránkach, mohli spôsobiť, že by hra bola prakticky nehrateľná.

Napriek tomu, že sieťová aplikácia mariáš funguje správne a obsahuje všetko, čo potrebuje, existujú ešte možnosti ako tento program vylepšiť. Sem sa radí napríklad aj implementácia licitovaného, prípadne krížového mariášu, ktorá by ešte rozšírila možnosti samotnej aplikácie. Výborným vylepšením programu by bolo pridanie tzv. profilového systému, ktorý by vytváral pre každého užívateľa jednoznačný profil s jeho menom a štatistikami obsahujúcimi napr. finančný stav, výhry, prehry, najväčšie získané množstvo peňazí v jednom kole a pod. Do rozšírení by sa dalo zaradiť aj ukladanie rozohranej hry a jej opätovné neskoršie nahranie tak, aby hráči mohli pokračovať tam, kde skončili.

Literatúra

[1] W. Richard Stevens, Bill Fenner, Andrew M. Rudoff: UNIX Network Programming, Volume 1, Third Edition: The Sockets Networking API, Addison-Wesley, 2004, ISBN: 0-13-141155-1

[2] Ken Arnold, James Gosling, David Holmes: The Java™ Programming Language, Fourth Edition, Addison-Wesley, 2005, ISBN: 0-321-34980-6

[3] Hana Kanisová, Miroslav Müller: UML srozumitelně, Computer Press, 2004, ISBN: 80-251-0231-9

[4] Herbert Schildt: Java™: The Complete Reference, Seventh Edition, The McGraw-Hill Companies, 2007, ISBN: 978-0-07-163177-8

[5] David Brackeen, Bret Barker, Laurence Vanhelsuwe: Vývoj her v jazyku Java, Grada, 2004, ISBN: 80-247-0874-4

[6] Dokument dosupný na URL (základné informácie o programovacom jazyku Java): http://cs.wikipedia.org/wiki/Java_(programovac%C3%AD_jazyk)

[7] Dokument dostupný na URL (stránka českého zväzu mariášu) : http://www.narinx.com/ceskymarias/index.asp?page=12

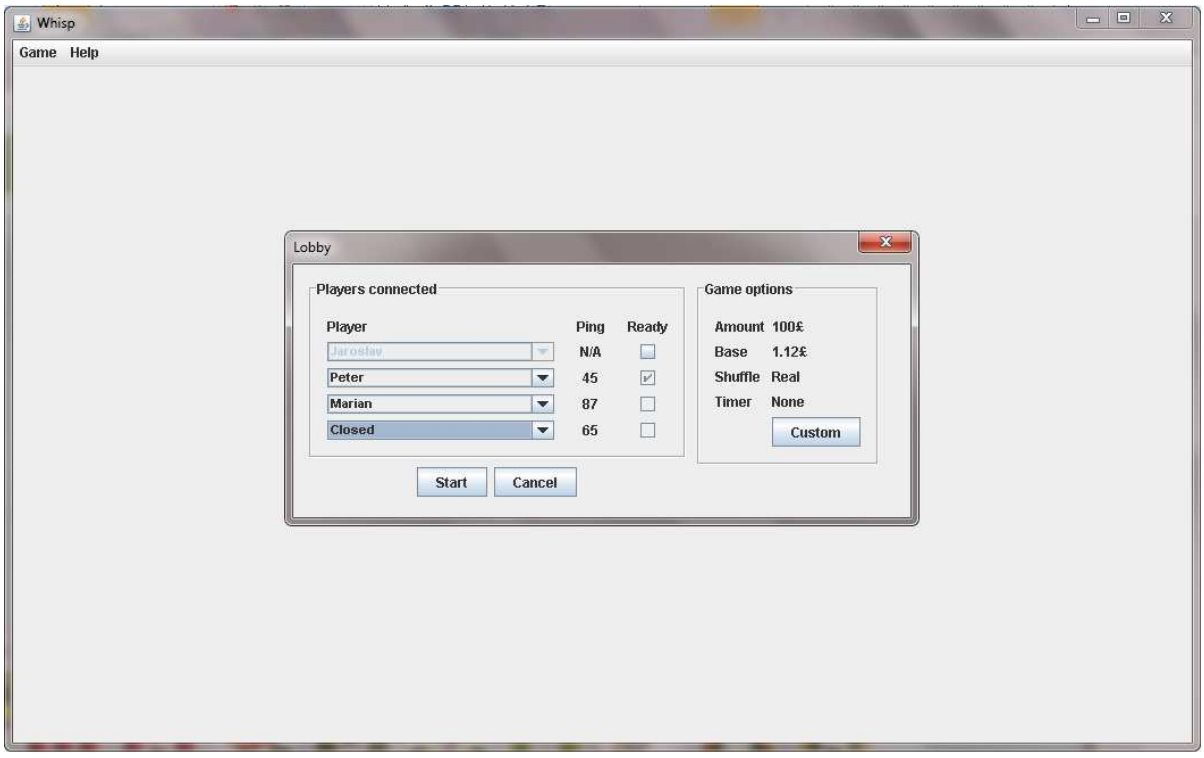
[8] Dokument dostupný na URL (základné informácie o UML diagramoch a VP-UML): http://www.visual-paradigm.com/product/vpuml/provides/umlmodeling.jsp

[9] Dokument dostupný na URL (domovská stránka NetBeans): http://netbeans.org/features/index.html

[10] Dokument dostupný na URL (stránka zaoberajúca sa rôznymi kartovými hrami): http://www.pagat.com/marriage/marias.html

Zoznam príloh

Príloha 1: Obrázky z hry



Príloha 2: Ukážka zdrojového kódu

Nasledujúci zdrojový kód popisuje algoritmus „ručného“ miešania kariet použitom v sieťovej aplikácii mariáš.

```
LinkedList<ICard> tempHeap = new LinkedList<ICard>(); //aux array of cards
Random rand = new Random(); //random generator
int numberOfShuff = rand.nextInt(4) + 4; //number of shuffles
int heapAmount; //heap of cards we take from deck
int groupAmount; //group within the heap, that we put at top of the deck
int i,l; //iterators

for (i = 0; i < numberOfShuff; i++){
	heapAmount = rand.nextInt(16)+7; //starting index of deck
	for (l = heapAmount; l < 32; l++ ){
		tempHeap.add(deck.get(heapAmount)); //add cards to heap
		deck.remove(heapAmount); //card is in heap, remove it from deck
	}
	heapAmount = 31 - heapAmount; //number of cards we have
	while (heapAmount >= 0){
		groupAmount = rand.nextInt(9); //number of cards we take from heap
		if (groupAmount > heapAmount) //group can't be larger then heap
			groupAmount = heapAmount;
		for (l = groupAmount; l >= 0; l--){
			deck.addFirst(tempHeap.get(l)); //return cards to deck
			tempHeap.remove(l); //remove from heap
			heapAmount--; //decrease amount of cards from heap
		}
	  }
}
```