

Vysoká škola ekonomická v Praze

Fakulta informatiky a statistiky

Katedra informačních technologií

Studijní program: Aplikovaná informatika

Obor: Informatika

Ukázkový příklad vývoje Java SE aplikace dle návrhového vzoru MVC

BAKALÁŘSKÁ PRÁCE

Student : Josef Draslar

Vedoucí : doc. Ing. Alena Buchalcenová, Ph.D.

Oponent : Ing. Rudolf Pecinovský, CSc.

2016

Prohlášení:

Prohlašuji, že jsem bakalářskou práci zpracoval samostatně a že jsem uvedl všechny použité prameny a literaturu, ze které jsem čerpal.

V Praze dne 04.05.2016

.....

Jméno a příjmení

Poděkování

Tímto bych chtěl poděkovat doc. Ing. Aleně Buchalcevé, Ph.D., vedoucí bakalářské práce, za všechny podněty, vstřícnost a ochotu při vypracování této bakalářské práce.

Dále přítelkyni za podporu nejen při psaní této práce, ale i během celého studia.

Abstrakt

Cíl práce: Hlavním cílem bakalářské práce je na praktickém příkladě ukázat vývoj Java SE aplikace dle návrhového vzoru MVC, aplikace je dále založená na JavaFX a OOP principech. Dílčím cílem, který přímo vede k dosažení cíle hlavního, je vysvětlit logiku návrhového vzoru MVC. Dále je cílem stručně charakterizovat jednotlivé základní varianty vzoru MVC dané jeho vývojem.

Výstupy: Hlavním výstupem je slovně okomentovaný zdrojový kód (v programovacím jazyce Java) postavený na principech OOP a především vybrané variantě MVC. Dílčím výstupem je stručný popis tohoto návrhového vzoru a jeho základních variací.

Přínosy bakalářské práce: Práce je koncipována jako příručka pro studenty kurzu softwarového inženýrství, hlavním přínosem je tedy zpracování návodu na vývoj Java SE aplikací dle vybrané varianty návrhového vzoru MVC. Již existující tutoriály jsou zpravidla nedostačující, mezi tutoriály se nachází značné nekonzistence v podobě různého pojmenování stejných věcí, či prosazení subjektivních názorů autora, a v anglickém jazyce.

Struktura práce: Práce je rozdělená na dvě části, v části první se zaměřím na objasnění návrhového vzoru jako takového, v části druhé krok za krokem popíšu tvorbu Java SE MVC aplikace za použití knihovny JavaFX – od návrhu a specifikace požadavků, přes stručný popis použitých technologií, vzorů a konstrukcí, až po konečnou implementaci.

Klíčová slova

Java SE, JavaFX, MVC, návrhový vzor, aplikace.

Abstract

The goal of thesis: The main goal of the thesis is to show practical example of development of Java SE application according to MVC design pattern, the application is going to be based on JavaFX and OOP principles. The partial goal, which strictly leads to achievement of the main goal, is to explain the logic of MVC design pattern. Other goal is to briefly describe basic MVC variants (according to its own development).

Outputs: The main output is textually commented source code (in Java programming language) based on OOP principles and primarily on chosen MVC representation. Partial output is brief description of the design pattern and its variants.

Thesis contribution: The thesis is designed to be a guide for students of software engineering course. Thus the main contribution is processing of guide for Java SE app development according to MVC design pattern. Already available tutorials are usually insufficient, there is lots of inconsistency between these tutorials based on other naming conventions of the same things, subjectivity, last but not least there are nearly all in English.

Thesis Structure: The thesis is divided into two sections, in the first section I am going to focus on clarification of the design pattern, in the second section I am going to describe (step by step) the development of Java SE MVC app with JavaFX – from design and specifying of requirements, through brief description of used technologies, patterns and constructions, to final implementation.

Keywords

Java SE, JavaFX, MVC, design pattern, application.

Obsah

1	Úvod	3
1.1	Vymezení tématu práce a důvod výběru tématu	3
1.2	Charakteristika současného stavu problémové oblasti	3
1.3	Cíle práce.....	4
1.4	Metody dosažení cíle	4
1.5	Předpoklady a omezení práce.....	5
1.6	Struktura práce	5
1.7	Přínosy práce	5
1.8	Přehled současné literatury pokrývající problémovou oblast	6
2	Návrhový vzor MVC	8
2.1	Vznik a vývoj MVC	10
2.2	Varianty MVC.....	15
2.2.1	View push model.....	15
2.2.2	View pull model	18
2.2.3	Model-Delegate.....	18
2.2.4	Model-View-Presenter.....	19
2.3	Architektura MVC.....	19
2.4	Přínosy implementace dle návrhového vzoru MVC	22
3	Postup tvorby aplikace piškvorky dle MVC	23
3.1	Úvodní studie	24
3.1.1	Zadání aplikace	24
3.1.2	Funkční požadavky	24
3.1.3	Nefunkční požadavky	25
3.1.4	Diagram případů užití	25
3.2	Analytická studie	27
3.2.1	Varianta MVC a další upřesnění - Návrh architektury systému.....	27
3.2.2	Diagram balíčků	28
3.2.3	Datový model	29
3.2.4	Obecná definice aplikace	30
3.2.5	Diagram komunikace	32
3.2.6	Sekvenční diagram	42
3.2.7	Diagram tříd.....	50
3.3	Návrh uživatelského rozhraní	55
3.4	Implementace aplikace.....	57
3.4.1	Komunikační protokol (spi a commons).....	57
3.4.2	Model.....	59
3.4.3	Model – dao – xml	78
3.4.4	Delegate.....	83

3.4.5 Hlavní třída aplikace	103
Závěr 105	
Seznam literatury.....	107

1 Úvod

Samotná implementace zdrojového kódu při vývoji dané aplikace tvoří velmi malé procento z celkového životního cyklu aplikace. Zpravidla nejvíce času pak připadá na pozdější modifikace a případné rozšiřování aplikace. Málo kdy je aplikace vytvořena, prodána a následně ukončena veškerá komunikace s kupujícím. V dnešní době je mnohem snazší si zákazníka udržet, nežli získávat zákazníka nového, a tak tomu je úplně stejně i v odvětví vývoje informačních systémů, či jiných softwarů. Společnosti daný statek (aplikaci, software, informační systém, ...) zpravidla neustále rozvíjejí a často poskytují podporu pro své zákazníky. Proto je velmi důležité vytvořit kvalitní a robustní návrh architektury systému, aby byly následné úpravy a rozšíření co nejjednodušeji proveditelné.

Jedním ze způsobů jak tohoto docílit, je využít vzor Model-View-Controller, který právě snižuje závislosti v rámci aplikace, vytvořením zpravidla tří nezávislých částí (záleží na konkrétní variantě), především pak zajišťuje oddělení věcné logiky aplikace od vizuálního zobrazení uživateli. Následné úpravy v jedné z těchto částí, poté nemají dopad na celý systém.

1.1 Vymezení tématu práce a důvod výběru tématu

Programováním v Javě se zabývám již tři roky a je to jedním z mých hlavních koníčků. Z toho důvodu jsem se rozhodl vybrat si pro svoji bakalářskou práci téma co nejbližší tomuto oboru. Čtenářům bych rád poskytl ucelený náhled na návrhový vzor MVC a především jeho použití pro vývoj aplikací v Javě SE za použití knihovny Java FX.

1.2 Charakteristika současného stavu problémové oblasti

Vzor Model-View-Controller (MVC) není žádnou uznávanou institucí definován, proto v jednotlivých řešeních hraje velkou roli uživatelská dosavadní zkušenost a jeho návyky. Taktéž mnoho volně dostupných materiálů vysvětluje tento vzor s určitými odlišnostmi, často jsou používána různá označení pro tytéž věci, a proto je pro začátečníka těžké si tento vzor osvojit. Návrhový vzor však stojí na jednoduchých principech, základní pochopení tak trvá řádově minuty, nicméně hloubější pochopení nezbytné pro použití při tvorbě kvalitního kódu, již vyžaduje mnohem více času i úsilí. Drtivá většina volně dostupných

materiálů na internetu je zaměřena právě na vysvětlení těchto jednoduchých principů a jsou zpravidla v anglickém jazyce.

Mnoho děl je adresováno na návrhový vzor Model-View-Controller, nicméně ze samotného textu poté často vyplývá, že se jedná spíše nějakou specifickou variantu (viz kapitola 2.2 Varianty MVC).

1.3 Cíle práce

Hlavním cílem této bakalářské práce je na praktickém příkladě ukázat vývoj Java SE aplikace dle návrhového vzoru MVC, aplikace je dále založená na JavaFX a OOP principech. Dílčím cílem, který přímo vede k dosažení cíle hlavního, je vysvětlit logiku návrhového vzoru MVC. Dále je cílem stručně charakterizovat jednotlivé základní varianty vzoru MVC dané jeho vývojem. Tato práce si neklade za cíl vysvětlovat programování v Javě.

1.4 Metody dosažení cíle

Struktura bakalářské práce je postavena na normě IMRAD (I – úvod, M – metody, Ra – výsledky, D – diskuze/závěr).

Pro dosažení hlavního cíle jsem nejdříve provedl lineární rešerši, kterou stručně zmiňuji v kapitole 1.8 – Přehled současné literatury pokrývající problémovou oblast. Následně stručně teoreticky rozebírám návrhový vzor MVC, poté představuji základní vybrané varianty relevantní pro newbový vývoj.

Pokračuji provedením výběru příslušné varianty vyhovující vývoji v Javě SE a knihovně Java FX. Následně se již přesouvám k naplnění hlavního cíle a provedu vývoj jednoduché ukázkové aplikace piškvorky. Zde stavím na metodě objektově orientované analýzy, návrhu i implementace. Nejdříve jsem sestavil UML diagramy, které následně používám k vygenerování základní struktury zdrojového kódu. Ten následně postupně implementuji a vysvětluji.

1.5 Předpoklady a omezení práce

Tato práce si neklade za cíl vysvětlovat programování v Javě, ani s tím spojené dovednosti. Jsou tedy vyžadovány základní znalosti v programovacím jazyce Java a ve vývojovém prostředí Eclipse. Dále tato práce vyžaduje znalosti UML diagramů, které jsou v práci ve značné míře použity, pro demonstraci aplikace (konkrétně diagram komponent, diagram balíčků a komunikací, sekvenční diagram a diagram tříd). V neposlední řadě tato práce nevysvětluje objektově orientované programování ani tvorbu vizuálních obrazovek v programu SceneBuilder.

1.6 Struktura práce

Rozhodl jsem se tuto práci koncipovat jako příručku pro studenty kurzu softwarového inženýrství na Vysoké Škole Ekonomické, a poskytnout jim, i všem ostatním čtenářům, ucelené informace o návrhovém vzoru MVC, od jeho historie, přes vybrané odnože, až po podrobný návrh a implementaci ukázkové aplikace za užití Java SE. (Dále je při implementaci použita nová vizuální knihovna JavaFX.)

Práce je rozdělená na dvě části, v části první se zaměřuji na objasnění návrhového vzoru jako takového, v části druhé krok za krokem popisuji tvorbu Java SE MVC aplikace za použití knihovny JavaFX – od návrhu a specifikace požadavků, přes stručný popis použitých technologií, vzorů a konstrukcí, až po konečnou implementaci – kde průběžně po částech vkládám zdrojový kód a stručně ho vysvětluji. Zdrojový kód tedy není uveden v celku, ani není z důvodu přehlednosti komentován (ve zdrojovém kódu). Zdrojový kód je vkládán po částech, tak aby se dal snadno aplikovat při výuce.

1.7 Přínosy práce

Práce je koncipována jako příručka pro studenty kurzu softwarového inženýrství, hlavním přínosem je tedy zpracování návodu na vývoj Java SE aplikací dle vybrané varianty návrhového vzoru MVC. Dalším přínosem práce je ucelené shrnutí principů návrhového vzoru MVC a jeho vybraných variant. Práce identifikuje problémy použití návrhového vzoru Model-View-Controller při implementaci Java SE aplikace za použití knihovny JavaFX, a následně podává ucelený návrh s příkladem, jak tento vzor použít. Cílovou skupinou této bakalářské práce jsou nejen studenti kurzu softwarové inženýrství, pro které je hlavním přínosem popis vývoje aplikace, nýbrž všichni kdo mají zkušenosti z vývojem

v Javě a prozatím nedisponují úplnou znalostí tohoto návrhového vzoru pro použití při vývoji Java SE aplikace.

1.8 Přehled současné literatury pokrývající problémovou oblast

Existuje velkém množství publikací a článků, které se zabírají návrhovým vzorem MVC. Pro tuto práci jsou však relevantní zpravidla materiály, které zpracovávají návrhový vzor MVC v Javě, a ještě k tomu ve verzi SE, kterýchžto je malé množství a drtivá většina z nich je postavena na starší knihovně Swing. Opravdu velmi malé množství zpracovává návrhový vzor MVC s ohledem na vývoj v Java SE za použití knihovny Java FX.

Jedním z těchto materiálů je tutoriál od Marca Jakoba zvaný JavaFX 8 Tutorial [16], kde Jakob v sérii krátkých článků jednoduše, stručně a srozumitelně přibližuje čtenářům tvorbu JavaFX aplikace na reálném příkladě. Pro tvorbu obrazovek zde užívá souborů `.fxml`. Strukturu jednoduché aplikace sice staví na zjednodušené formě MVC, nicméně tento návrhový vzor zde popisuje velmi zběžně, především se odkazuje na ostatní literaturu obecně pojednávající o tomto návrhovém vzoru.

Dalším materiálem, který spojuje problematiku návrhového vzoru Model-View-Controller a novější knihovny JavaFX, je článek od Scotta Hommela, publikovaný pod záštitou společnosti Oracle, s názvem Implementing JavaFX Best Practices [17]. Kde Hommela v rámci jednoho odstavce rozebírá návrhový vzor MVC, následně prezentuje stručný příklad zobrazující propojení View s Controllerem.

Dále je dostupný článek Designing JavaFX Business Applications (Part 2) od Hendrika Ebberse, kde autor poskytuje tutoriál pro tvorbu architektury byznys aplikací, nicméně tutoriál je mířen především na dependency injection (vkládání závislostí). Dokonce i autor sám v jedné části tvrdí, že jeho pojetí MVC, tak úplně pravidlům tohoto návrhového vzoru nevyhovuje.

Ještě existuje několik podobných článků/tutoriálů dostupných online, nicméně všechny jsou psané v anglickém jazyce a navíc návrhový vzor MVC popisují příliš stručně.

Literárních zdrojů obecně rozebírající návrhový vzor Model-View-Controller existuje nepřeberné množství. Pro účely teoretické části této práce jsem vybral pár přehledných zdrojů, zpravidla od uznávaných autorů.

Jedním z nich je Martin Fowler a jeho dílo Patterns of Enterprise Application Architecture [10], kde prezentuje Model View Controller v rámci kapitoly 14 – Web Presentation Patterns.

Zakladatel tohoto návrhového vzoru, Reenskaug Trygve, a jeho několik odborných článků [1] [3] [4] [5].

Dále Frank Buschmann, který s kolegy sepsal knihu o návrhových a architektonických vzorech - Pattern-oriented software architecture [12], kde věnuje kapitolu návrhovému vzoru MVC.

V neposlední řadě jsem čepal z článků: Chrise Adamsona - Beyond MVC: A New Look at the Servlet Infrastructure Blog [2], kde Chris nejdříve rozebírá historii návrhového vzoru MVC, pokračuje popsáním špatně implementovaného MVC v knihovně Servletů a v závěru demonstruje potenciální nový design této knihovny. Scotta Stanchfielda - Applying MVC in VisualAge for Java [8], Stanchfield zde nejdříve teoreticky popisuje návrhový vzor MVC, rozebírá pár vybraných variant (jako Model-Delegate) a následně krok za krokem popisuje tvorbu MVC aplikace ve vývojovém prostředí VisualAge. A dalších.

2 Návrhový vzor MVC

V této kapitole provádím stručný rozbor návrhového vzoru MVC, jeho počátků a vybraných variant. Čímž specifikuji potřebné znalosti pro následující kapitolu této práce.

MVC je někdy označován jako architektonický vzor a jindy jako vzor návrhový. Vzor, obecně co se týče programování, představuje, ukazuje, popisuje či vysvětluje best practice pro řešení dané problematiky, ať už na návrhové či implementační bázi. Architektonický vzor je takový vzor, který slouží k uspořádání celkové architektury aplikace. Naopak návrhový vzor je abstraktním vzorem aplikovatelným při implementaci/návrhu aplikace. MVC tak právem může být označován jako vzorem architektonickým, pokud udává celou strukturu aplikace, či vzorem návrhovým pokud „pouze“ zajišťuje lehčí údržbu či měnitelnost, znovupoužitelnost a lepší čitelnost zdrojového kódu.

MVC je vzor rozdělující daný informační systém, či jiný software (potažmo zdrojový kód), na vzájemně nezávislé komponenty, čímž snižuje celkové závislosti v softwaru. Tento vzor není žádnou novinkou, nicméně nikdy nebyl žádnou uznávanou institucí standardizován, proto existuje mnoho různých MVC vzorů, které se zpravidla drobně liší. Vzor našel své využití především na webu, byť byl původně navržen pro desktop. V dnešní době již existuje obrovské množství implementací MVC vzoru, které se více či méně liší. Mezi jedny z nejznámějších implementací patří:

- Ruby On Rails – populární webový framework v jazyce Ruby,
- Apple Cocoa – Apple framework pro vývoj Mac OS a iOS aplikací,
- ASP.Net Framework: framework od Microsoftu pro implementaci webových aplikací,
- Apache Struts: webový framework v jazyce Java,
- Zend framework,
- Smalltalk.

Scott Stanchfield v článku Applying MVC in VisualAge [8] for Java popisuje tento vzor následovně. MVC je návrhový vzor umožňující především oddělení byznys logiky od uživatelského rozhraní (GUI), poté je mnohem jednodušší modifikovat kód v jedné z částí. Protože části jsou na sobě nezávislé, tak změna v části jedné neovlivní kód v části druhé. Na začátku je vyžadováno více úsilí na vytvoření takovéto MVC struktury, avšak tento „ztracený“ čas se velmi rychle vrátí, když je třeba systém následně upravovat a rozšiřovat. MVC model předpokládá implementaci dle objektově orientovaného principu.

Dean Helman ve svém článku Model-View-Controller [7] definuje MVC, jako způsob rozbití aplikace na věcně a logicky související části, konkrétně tři. Původně byl MVC navržen na: vstup – zpracování – výstup. Uživatelský vstup, model externího světa a vizuální zobrazení uživateli. Tyto části jsou od sebe odděleny. Controller přebírá vstupy od uživatele a spouští logiku, která následně vyústí v zajištění požadovaných změn v modelu a následné upravení View. Model obsluhuje data a upozorňuje Observery na změnu těchto dat. Obsahuje logiku věcně a logicky spjatou s daty. Model představuje výpočetní zjednodušení či abstrakci nějaké části reálného světa. A Controller je zodpovědný za to, jak aplikace reaguje na konkrétní uživatelskou akci.

Všeobecně vzato vzor MVC definuje následující tři základní komponenty (jak již název napovídá):

- Model – představuje data a pravidla, kterými jsou ukládány a měněny. Často poskytuje abstrakci procesu z reálného světa kolem nás (např.: vizuální kontrola zda nikdo doposud nevyhrál stolní hru). Obsahuje byznys logiku.
- View – zobrazuje data z Modelu a udává, jakým způsobem mají být tyto data prezentována uživateli.
- Controller – překládá akce uživatele (např.: kliknutí na tlačítko) nad daným View do akcí, které se provedou v daném Modelu. [6]

Vysvětlení základní MVC architektury na příkladě v Excelu [11]:

- Model – datová struktura držící/uchovávající jednotlivá čísla. Zde také nalezneme průměr jednotlivých hodnot, aneb byznys logiku (ta však nemusí být přítomna).
- View – jednotlivé grafy zobrazující data z tabulek. View je nezávislé, tedy funguje na jakákoli data jemu poskytnutá (musí však být stejné struktury). Třeba název grafu, či konkrétní typ grafu (koláčový, sloupcový, atd.) je věcí pouze vizuálního vzhledu, proto o tom nemusí vědět další vrstvy.
- Controller – zde není příliš vidět, projeví se po změně dat v tabulce, kdy zajistí aktualizaci grafů. Jedná se o jakéhosi prostředníka spojujícího vrstvy dohromady. A to jakým způsobem už záleží na konkrétní variaci tohoto vzoru.

Nejobecnějšími a základními principy MVC jsou [9]:

- Oddělení prezentační vrstvy od Modelu – což umožňuje implementaci odlišných uživatelských rozhraní a lepší testovatelnost.
- Oddělení prezentační vrstvy od Controlleru – nejužitečnější ve webových rozhraních.

Nicméně ne všechny MVC implementace tyto dva základní principy dodržují, dokonce jedny z úvodních implementací toho vzoru, ve Smalltalku, jak Martin Fowler ve své knize

Patterns of Enterprise Application Architecture [10] zmiňuje, nedodržují druhý princip, tedy oddělení View od Controlleru – „*The irony is that almost every version of Smalltalk didn't actually make a view/controller separation.*“ [10].

Nakonec otec MVC, Reenskaug Trygve, ve svém díle MVC XEROX PARC 1978-79 [3] specifikuje MVC následovně. Základním přínosem vzoru MVC je propojit reálný model s počítačovým modelem, který představuje abstraktní reprezentaci dané relevantní části světa reálného. Myšlenkou MVC řešení je vytvořit iluzi uživatele, takovým způsobem, že vidí a manipuluje s doménovými daty na přímo. Takováto struktura je nejužitečnější za předpokladu, že uživatel potřebuje vidět či manipulovat s modelovými daty nezávisle v různém kontextu či různých úhlech pohledu. Všeobecně řečeno MVC představuje řešení pro problematiku, kdy uživatelé manipulují s velkým a komplexním datovým skladem.

2.1 Vznik a vývoj MVC

Úplné počátky MVC sahají až do sedmdesátých let minulého století, které jsou spjaty s následným použitím jako vzoru pro tlustého klienta – uživatelská rozhraní komponentových frameworků.

Historicky první dokument zvaný Administrative Control in the Shipyard zabývající se vzorem Model-View-Controller pochází ze srpna roku 1973, byl vytvořen profesorem Trygvem Reenskaugem při práci pro Central Institute for Industrial Research v Oslu (Norsko) a představen na první International Conference on Computing Applications konané na podzim roku 1973. V tomto dokumentu Reenskaug zanalyzoval moderní loděnici do podoby informačního systému, identifikoval mnoho technických a sociálních vztahů v rámci loděnice, a navrhl několik technik vhodných pro popis systému. Jeho cílem bylo snížit celkovou složitost, tak aby se dala loděnice snáze namodelovat do počítačové aplikace. Hned odpočátku se Reenskaug pohybuje okolo – tehdy neznámého – principu dnes zvaného „*distributed, communication components*“¹, v dokumentu tuto strategii popisuje následovně „*total system will be decomposed into a number of components*“ [1]. Dekompozice velkého, komplexního systému na modulární komponenty hrála hlavní roli v jeho strategii. Dnes je tato strategie nazývána „*Separation of Concerns*“². Reenskaug

¹ Oddělené vzájemně komunikující komponenty

² Oddělení zodpovědností

v dokumentu popisuje obecný framework umožňující vytvoření nového informačního systému dle pravidel fyzického systému, tak aby byl nový systém dostatečně flexibilní. [3] Definoval hlavní předpoklady umožňující vývoj pomocí tohoto frameworku [3]:

- zjednodušený ruční přepis,
- rozdělitelné do modulárních podsystémů,
- transparentní chování podsystémů,
- vzájemná propojitelnost podsystémů,
- systém jako celek se schopný kontinuálního rozvoje.

Poté byl jeho projekt neoficiálně pozastaven. Vymyšlený systém pro vývoj frameworku, nebyl dostatečný pro jeho implementaci, zde mu chyběly potřebné prostředky jako ucelený počítač, či objektově orientované programování. [3]

V roce 1978 se Reenskaug dostává k prvnímu osobnímu počítači ALTO pod záštitou Xerox Palo Alto Research Center (PARC) v Kalifornii. Kde začíná experimentovat s interakcí koncových uživatelů. Reenskaug a jeho kolegové vynalézají MVC jako řešení pro zjednodušení přenosu dat mezi reálným a počítačovým modelem, stejně jak tomu bylo v roce 1973. Tentokrát jedním podsystémem bylo uživatelské rozhraní. Definují tedy nový framework zvaný Model-View-Editor popsany v dokumentu Thing-Model-View-Editor. [3]

Tento dokument definuje následující komponenty:

- Thing – část reálného světa o kterou se uživatel zajímá (jako např.: dům, kolo). („*Something that is of interest to the user.*“ [4]).
- Model – abstraktní reprezentace části reálného světa ve výpočetním systému. („*A Model is an active representation of na abstraction in the form of data in a computing system.*“ [4]) (Abstractin of the Thing)
- View – jeden či více obrazových náhledů na model. Každé View je schopné poskytnout jeden či více reprezentací daného modelu. View také zajišťuje takové operace na Modelu, které se objektivně týkají daného View. („*To any given Model there is attached one or more Views, each View being capable of showing one or more pictorial representations of the Model on the screen and on hardcopy. A View*

is also able to perform such operations upon the Model that is reasonably associated with that View.“ [4])

- Editor – rozhraní stojící mezi uživatelem a jedním či více náhledů, poskytuje uživateli příkazový systém – např.: prostřednictvím menu, které je schopné se dynamicky generovat dle aktuálního kontextu. Obohacuje View o nezbytnou navigaci a texty. (*„An Editor is an interface between a user and one or more views. It provides the user with a suitable command system, for example in the form of menus that may change dynamically according to the current context. It provides the Views with the necessary coordination and command messages.*“ [4])

V roce 1978 Reenskaug vytvořil první implementaci MVC v Xerox PARC, založenou na komponentách: Model, View, Controller a Editor.

Jednotlivé komponenty

- Model – jednoduchý objekt či struktura objektů reprezentující znalost [5] (shodný výklad s předchozí specifikací).
- View – vizuální reprezentace přidruženého modelu, vybrané atributy modelu zobrazí a ostatní nechá „skrytá“, chová se jako prezentační filtr. View je propojeno s modelem či částí modelu a data získává užitím „otázek“. Otázky musí být v terminologii modelu, View tedy musí znát sémantiku modelu [5] (shodný výklad s předchozí specifikací).
- Controller – tvoří propojení mezi uživatelem a systémem, poskytuje uživateli uspořádané View. Nikdy by neměl nahrazovat View a prezentovat data uživateli. Controller je napojen na jeden či více View. [5]
- Editor – speciální Controller, který umožňuje uživateli upravovat informace prezentované skrze dané View. [5]

Krátce po vytvoření této první implementace Reenskaug odešel z laboratoří společnosti PARC, v jeho práci pokračoval Jim Althoff a Dan Ingalls.

V roce 1983 Jim Althoff a další, vytváří novou verzi MVC v rámci Smalltalk-80.

Jednotlivé komponenty

- Model – abstrakce části reálného světa (A real-world concept.).

- View – vizuální prezentace modelu.
- Controller – zodpovědný za vytváření a koordinaci podřízených („*Controller was responsible for creating and coordinating its subordinate views*“ [3]).

Každá komponenta měla umožněno komunikovat s ostatními dvěma, žádné nezávislé vrstvy či abstrakce zde není použita. [3]

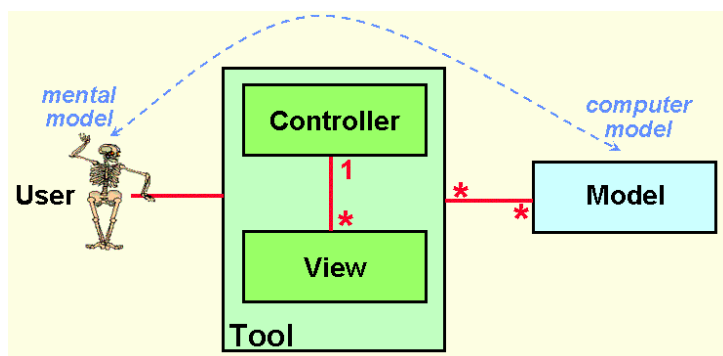
Hlavním myšlenkou celého tohoto procesu tedy je snížit složitost uživatelského rozhraní ve velkých a komplexních informačních systémech („*to reduce the complexity of user interfaces for a large and complex informatin system*“ [2]).

Reenskaug pokračoval ve své práci ohledně MVC i nadále. Vytváří další, novou, verzi MVC. [3]

Jednotlivé komponenty (viz Obr. 1):

- Model – abstrakce části reálného světa (A real-world concept.).
- Tool
 - View – přijímá vstup uživatele a provádí základní zpracování vstupů relevantních právě pro toto aktuální view („*a view accepts and handles user input relevant to itself*“ [3]),
 - Controller – přijímá a zpracovává vstup relevantní právě pro aktuální Controller/View („*The Controller accepts and handles input relevant to the Controller/View*“ [3]).

Obr. 1: Model-Tool (View-Controller)-User



Zdroj: [3]

Následně dokonce vytvořil nový jazyk založený na vzoru MVC, který však své široké uplatnění nenašel. [3]

2.2 Varianty MVC

Historii i systém fungování návrhového vzoru MVC jsem již vysvětlil, teď se tedy přesunu k popsání vybraných variant tohoto vzoru.

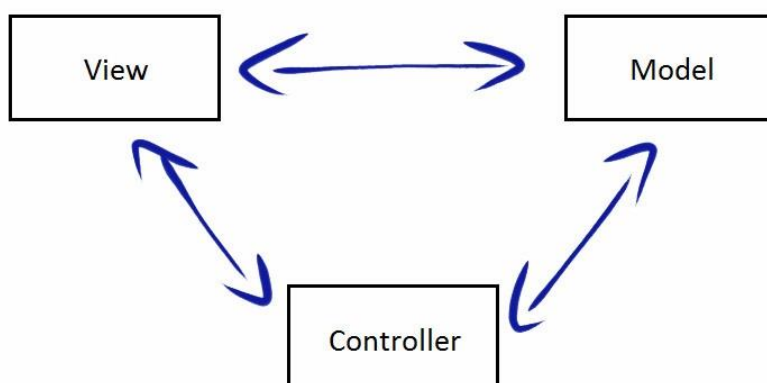
2.2.1 View push model

Push model představuje takové rozestavení a propojení komponent, kde se View registruje u Modelu pro příjem notifikací, pro případ relevantní změny v Modelu (viz Obr. 3). V případě aktualizace daných dat Model notifikuje dané View, aby se aktualizovalo. Controller je úzce spojený s View, jakákoli uživatelská akce vyvolá registrované metody v Controlleru. Controller drží referenci na Model. [6]

Řetězec činností je následující: View zjistí uživatelskou akci (např.: kliknutí na tlačítko), automaticky spustí zaregistrovaný listener, což zajistí zavolání správné metody v Controlleru. Controller upraví Model (dle uživatelské akce). Případná změna dat v Modelu vyvolá spuštění listenerů, které zajistí provedení potřebné aktualizace View. [6]

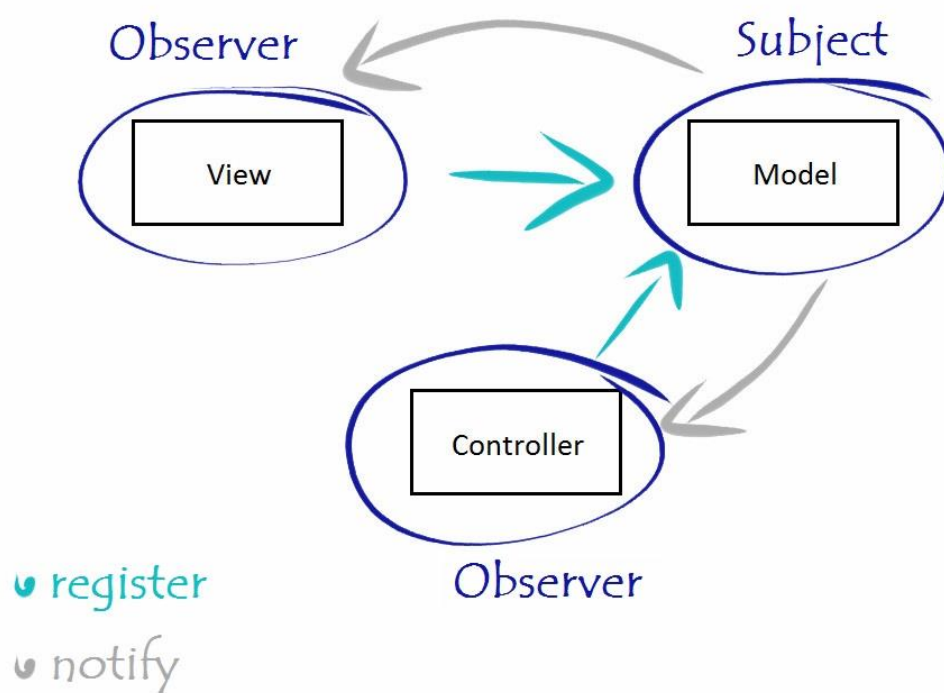
Obrázek 2 znázorňuje vzájemné vztahy jednotlivých komponent. Kooperace komponent z perspektivy návrhového vzoru Observer, probíhá tak, že se View (případně i Controller), registrují u Subjektu, v tomto případě Modelu, jako Observery, Subjekt je poté při změně relevantních dat upozorní (viz Obr. 3). Pokud uživatel provede nad View akci týkající se přímo daného View (jako např.: poposunutí, přechod na další stránku), přichází do akce Controller, který přímo zajistí změny na daném View (viz Obr. 4). Předtím než k takovéto situaci může dojít, tak se musí Controller registrovat jako listener u View, nad který pak uživatel svými akcemi spouští dané registrované metody Controlleru (viz Obr. 5).

Obr. 2: View push model rozložení



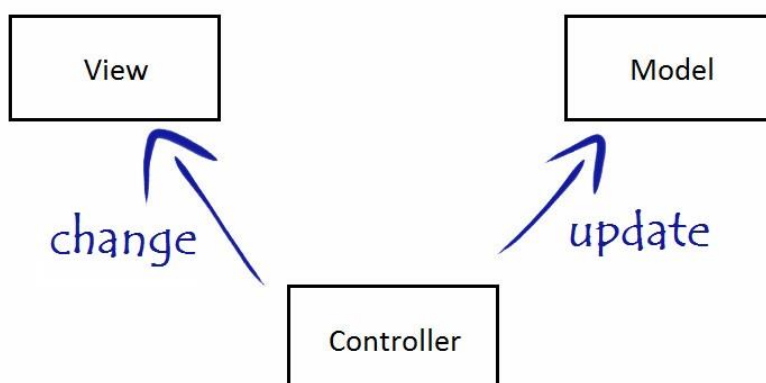
Zdroj: vlastní tvorba (v onemotion.com)

Obr. 3: View push model rozložení – návrhový vzor Observer



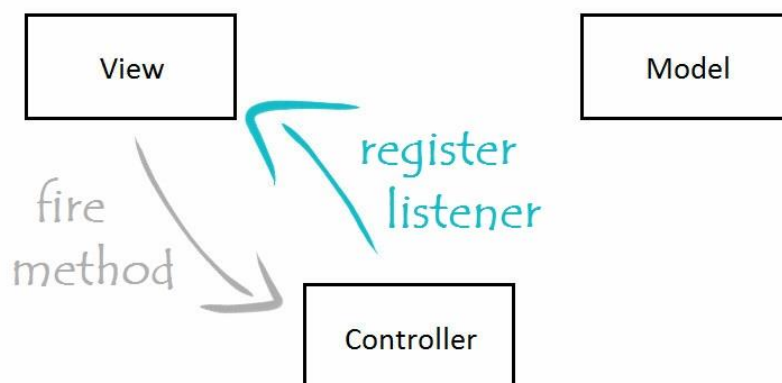
Zdroj: vlastní tvorba (v onemotion.com)

Obr. 4: View push model rozložení (akce Controlleru)



Zdroj: vlastní tvorba (v onemotion.com)

Obr. 5: View push model rozložení – vztah Controlleru a View



Zdroj: vlastní tvorba (v onemotion.com)

2.2.2 View pull model

V případě pull modelu je View stoprocentně zodpovědné za aktualizaci dat, musí samo ve správný okamžik požádat Model o aktuální data. [6]

Controller jako prostředník (Application Controller / Cocoa MVC)

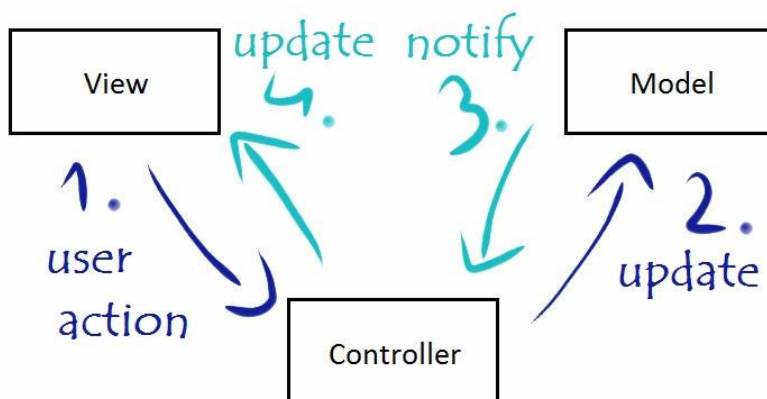
Hlavním rozdílem oproti předchozím modelům je, že Controller propaguje aktualizovaná data z Modelu do View. Controller drží referenci jak na Model tak i na View. Tento model se vyznačuje velkou separací Modelu od View. [6]

Řetězec činností odpovídá View push modelu, s tím rozdílem, že tentokrát Controller má zaregistrované listenery u Modelu na změnu dat, ta pak vyvolá danou metodu v Controlleru, která zajistí aktualizaci View (viz Obr. 6). [6]

Controller, resp. Application Controller, je zodpovědný za dvě základní věci [10]:

- Rozhoduje jakou doménovou logiku spustit v návaznosti na akce uživatele.
- Rozhoduje jaké View daná data zobrazí.

Obr. 6: Application Controller a jeho základní funkce



Zdroj: vlastní tvorba (v onemotion.com)

2.2.3 Model-Delegate

Model-Delegate je modifikací MVC vzoru, kde View a Controller jsou takovým způsobem provázáni, že dohromady tvoří jednu vrstvu, zvanou Delegate. I v rámci této varianty je

chtěné separovat View od Controlleru co nejvíce, avšak klíčovou myšlenkou je separace celkově Delegate od Modelu.

2.2.4 Model-View-Presenter

MVP je vzor, kde View zobrazuje výstup uživateli a zároveň přebírá od něj vstup, který je již zpracováván Controllerem. Tento vzor je často označován stále jako MVC, nicméně View v úplně původním MVC zastávalo funkci pouze zobrazování výstupu uživateli a vstup od uživatele přebíral Controller sám. Tato varianta je v dnešní době k vidění v klientských aplikacích, jak již bylo řečeno, zpravidla se k tomuto vzoru přistupuje s označím MVC, správné označení MVP je často opomíjeno.

2.3 Architektura MVC

Eric Freeman, a další, v knize Head First Design Patterns [11] označují MVC jako „compound pattern“, z hlediska toho, že vlastní struktura MVC je v podstatě tvořena vhodným uspořádáním jiných návrhových vzorů.

MVC jako jako uskupení jiných návrhových vzorů

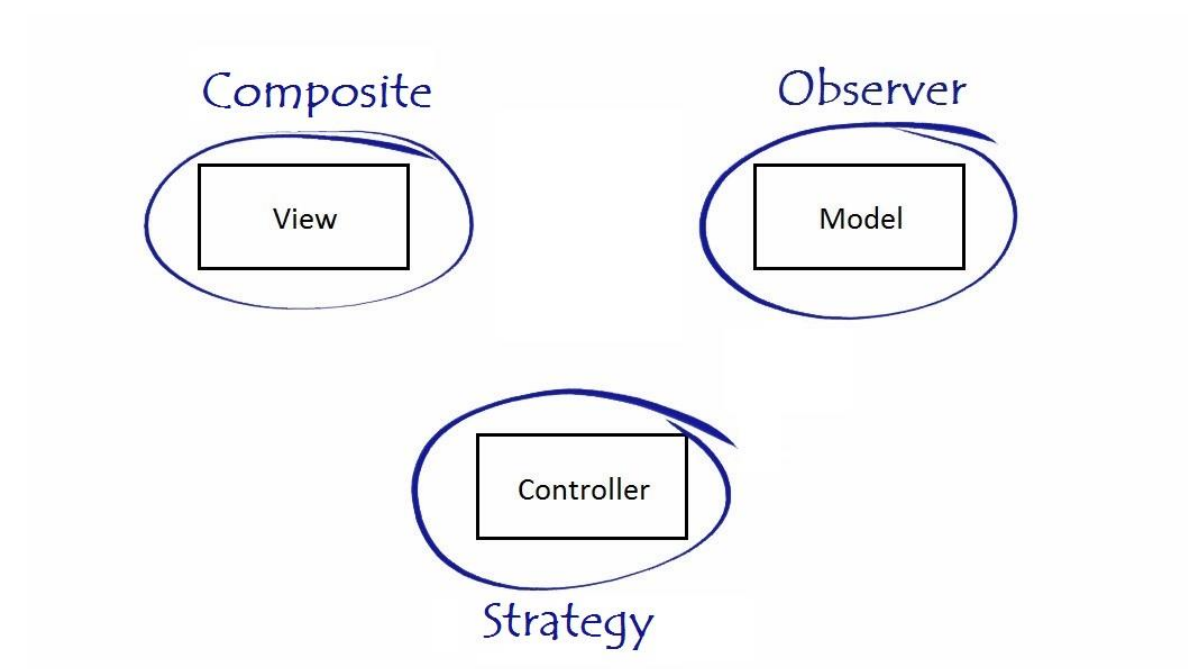
Každá komponenta je navržena dle návrhového vzoru (viz Obr. 7).

Model je navržen dle návrhového vzoru Observer (viz Obr. 8), aby udržel data napříč aplikací konzistentní. Při změně dat upozorní všechny objekty, kterých se změna dotkne, resp. které se o upozornění přihlásili. Tím je zajištěna kompletní nezávislost Modelu na View a Controlleru, zároveň to umožňuje užít rozdílná View pro stejný model – a to dokonce i najednou. [11]

Controller je navržen dle návrhového vzoru Strategy (viz Obr. 9), kde View je objektem, kterému Controller poskytuje strategii. Jinak řečeno, View se stará pouze o vizuální zobrazení, a veškeré uživatelské akce deleguje na Controller. Tímto je zajištěno oddělení View od Modelu, protože Controller je zodpovědný za komunikaci s Modelem nikoli View, ve skutečnosti View o tom vůbec nic neví, pouze upozorní Controller na danou uživatelskou akci. [11]

View je navrženo dle návrhového vzoru Composite (viz Obr. 10), kde se výsledná obrazovka skládá z jednotlivých komponent, které jsou buď „composite“ (jako frame), či „leaf“ (jako button). Pokud Controller požádá View o aktualizaci, View vyvolá aktualizaci na nejobecnější komponentě a Composite se postará o zbytek. [11]

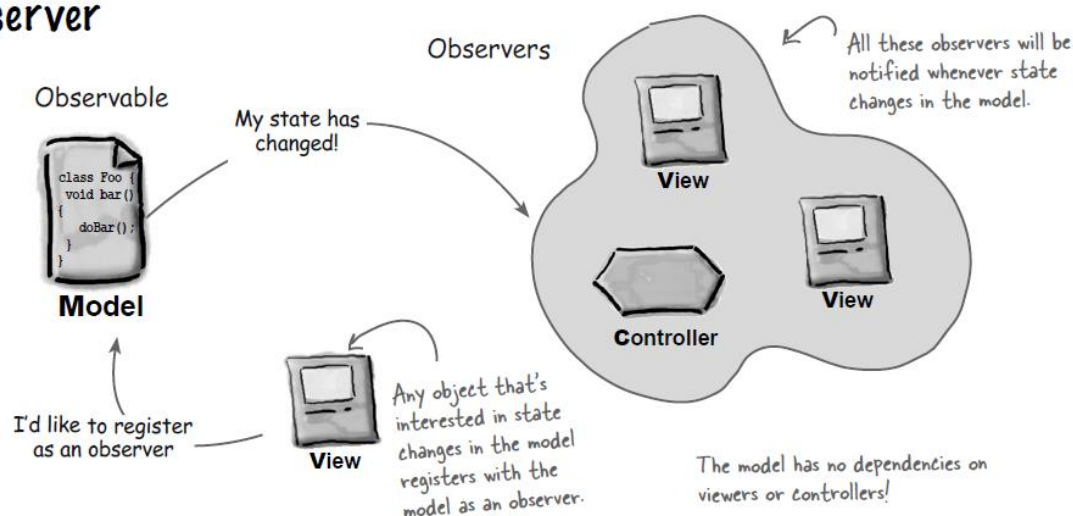
Obr. 7: MVC jako compound pattern



Zdroj: vlastní tvorba

Obr. 8: Model jako návrhový vzor Observer

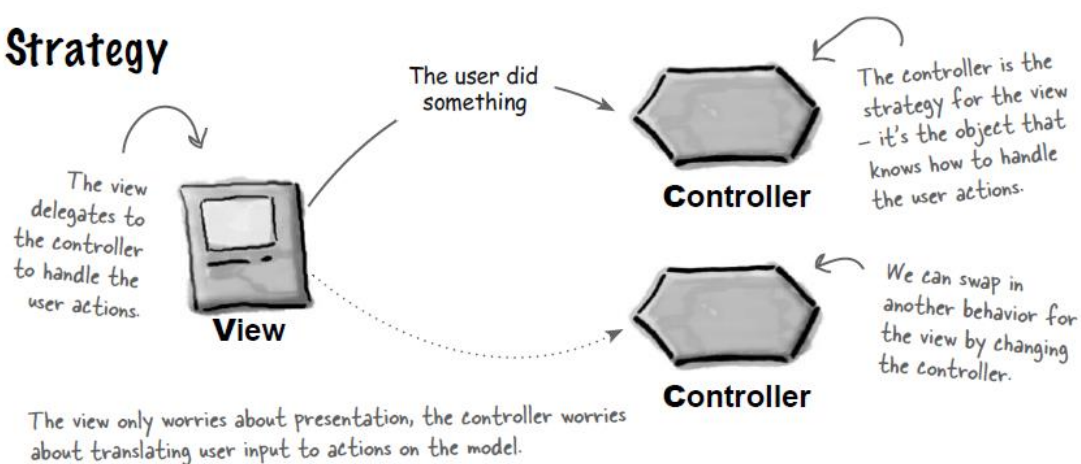
Observer



Zdroj: [11, s. 533]

Obr. 9: Controller jako návrhový vzor Strategy

Strategy



Zdroj: [11, s. 533]

Obr. 10: View jako návrhový vzor Composite

Composite



Zdroj: [11, s. 533]

2.4 Přínosy implementace dle návrhového vzoru MVC

Jak Scott Stanchfield v článku *Applying MVC in VisualAge for Java* [8] popisuje, samotný vývoj tvoří pouze 10 % či dokonce méně z celkového životního cyklu aplikace, obrovskou část totiž tvoří následné změny a modifikace. Z tohoto důvodu by na začátku každé implementace měl stát kvalitní design, který především odděluje byznys logiku od uživatelského rozhraní. Změny totiž přichází zpravidla v byznys logice a výsledné podobě aplikace separátně, pokud jsou tyto vrstvy tedy odděleny, následné změny nemají takový dopad a celkově je aplikace mnohem snáze udržitelná.

Jak Frank Buschmann, a další, píší v knize *Pattern Oriented Software Architecture* [12], změna uživatelského rozhraní nesmí ovlivnit základní funkcionalitu aplikace, která je zpravidla nezávislá na prezentaci, a také se mění s mnohem menší frekvencí. Dále změny prováděné v uživatelském rozhraní musí být jednoduše a pouze lokálně (pro danou obrazovku) proveditelné, a musí zobrazovat aktuální/relevantní data. [12, s. 188]

Z těchto a dalších důvodu je rozdělení aplikace do tří vzájemně nezávislých částí výhodné. Aplikací MVC dále zajistíme konzistenci mezi těmito třemi částmi. [12, s. 188]

3 Postup tvorby aplikace piškvorky dle MVC

V této kapitole postupně krok za krokem rozebírám návrh, vývoj a na závěr implementaci zvolené aplikace, čímž naplňuji hlavní cíl této práce.

Pro zpřehlednění následujícího oddílu vyznačuji balíčky kurzívou (příklad: balíček *model*) a popisy jednotlivých diagramů či zdrojových kódů probíhají zpravidla v rámci jednoho odstavce, který je zakončen „odkazem“ na obrazec, který popisují. Uvnitř odstavce pak jsou veškerá označení, korespondující zpravidla z odkazovaným obrazcem, zvýrazněny speciálním písmem (takto označené jsou: komponenty, třídy, metody, proměnné, konstanty, .fxml soubory, a případně další).

Příklad: Třída jménem `MojeTrida`. Viz obr. 999.

Pro popis postupu tvorby aplikace jsem použil, s drobnými úpravami, strukturu vytvořenou Stanislavem Kuznetsovem v práci Použití technologie Model View Controller (MVC) [14]. Tato struktura definuje tři základní části, ke kterým ještě přidávám část čtvrtou:

- Úvodní studie – studie popisující základní vlastnosti aplikace, definuji zde zadání aplikace, funkční i nefunkční požadavky a diagram užití.
- Analytická studie – studie pokrývající analýzu a návrh aplikace. Nejdříve specifikuji architekturu aplikace, pokračuji diagramem balíčků, datovým modelem. Následně udávám obecnou definici aplikace, která je nezbytná pro tvorbu následujících diagramů: diagram komunikace, sekvenční diagram a na závěr diagram tříd.
- Návrh uživatelského rozhraní – obsahuje jednotlivé obrazovky aplikace.
- Implementace aplikace – zde čerpám ze všech předchozích částí a postupně vysvětluji tvorbu zdrojového kódu aplikace.

3.1 Úvodní studie

Studie popisující základní vlastnosti aplikace, definují zde zadání aplikace, funkční i nefunkční požadavky a diagram užití.

3.1.1 Zadání aplikace

Pro ukázkou návrhového vzoru MVC jsem zvolil aplikaci piškvorky. Jelikož piškvorky jsou hrou světoznámou, tak existuje mnoho variant a upravených pravidel, pro udržení jednoduchosti, ale zároveň „použitelnosti“ aplikace omezují zadání následovně. Hru hrají vždy dva hráči proti sobě, jelikož hra probíhá v jednom okně na jednom stroji, pak není vyloučeno, že za dva uživatele nehraje reálně jedna osoba. Hrací pole je o velikosti 10x10 políček a pět schodných znaků v řadě vítězí. Základními znaky hráčů je křížek a kolečko. Oba hráči si mohou před spuštěním hry zvolit své jméno/přezdívku, pod kterou budou hrát, a hra uživateli poskytuje základní statistiku – konkrétně počet výher, proher a remíz. Jednu hru nesmí hrát dva hráči se stejnou přezdívkou. Uživateli je umožněno v průběhu hry přepnout na hru časovanou – kdy se jednotliví hráči střídají nikoli po tahu, nýbrž po specifikovaném čase, či hru eskalovaně časovanou – kdy se interval pro střídání hráčů ještě zkracuje. Tedy pokud hráč A, který je aktuálně na tahu, neprovede v daném intervalu tah, pak se automaticky dostává na tah hráč B, ten tah provede a tím se dostává na tah opět hráč A. Samozřejmě tyto časové varianty musí jít během hry opět vypnout. Jakýkoli z hráčů může kdykoli během hry vyhlásit remízu. Aplikace je napsána v Javě SE s užitím knihovny JavaFX.

3.1.2 Funkční požadavky

V Tabulce 1 poskytují výčet všech funkčních požadavků.

Tabulka 1: Funkční požadavky

ID požadavku	Název požadavku	Podrobný popis	Priorita (1 – nejvyšší, 5 – nejnižší)	Případ užití
F01	Spuštění hry	Actor nejdříve spustí aplikaci, zadá své hráčské jméno, poté čeká na zadání druhého hráčského jména, poté se hra spouští.	1	UC1, UC2, UC3
F02	Hraj hru	Actor po úspěšném spuštění, hry provádí svůj tah kliknutím na prázdné políčko v rámci hracího plánu o velikosti 10x10 políček. Následně se hráči střídají, pokud nedojde k výhře (5 výherních políček v řadě), či remíze (zaplnění všech hracích políček) tímto tahem. V případě remízy dochází k zobrazení konečné obrazovky.	1	UC4
F03	Kontrola duplicity hráčských jmen	Hru nesmí hrát dva hráči se stejným herním jménem zároveň	1	UC2

F04	Vyznačení výherních políček	V případě výhry dochází k vyznačení výherních políček a zobrazení konečné obrazovky.	3	UC4
F04	Uchovávání statistik	Uchovávání základní statistiky pro jednotlivé hráče, resp. hrací jména – počet výher, proher a remíz.	2	UC4
F05	Zobrazení statistik	Pokud hráč již pod stejným hracím jménem hrál, pak je hráčům statistika zobrazena hned po spuštění hry.	2	UC3
F06	Hraj časovanou hru	Actor může de/aktivovat časovanou hru, kdykoli během hry. Poté střídání hráčů probíhá na základě zvoleného časového intervalu.	2	UC5
F07	Hraj eskalovaně časovanou hru	Actor může de/aktivovat eskalovaně časovanou hru, kdykoli během hry. Jedná se o je hru časovanou plus se ještě interval zkracuje	2	UC6
F08	Vyhlaš remízu	Actor může kdykoli v průběhu hry vyhlásit remízu	1	UC7
F09	Ukonči aplikaci	Actor může kdykoli aplikaci ukončit, bez ukládání jakýchkoli údajů.	1	UC8

Zdroj: vlastní tvorba

3.1.3 Nefunkční požadavky

Požadavky vycházející přímo ze zadání a zároveň kladoucí omezení na design či požadavky. Výčet nefunkčních požadavků je zobrazen v Tabule 2.

Tabulka 2: Nefunkční požadavky

ID požadavku	Název požadavku	Podrobný popis	Priorita (1 – nejvyšší, 5 – nejnižší)
NP01	Java SDK 8	Aplikace je realizována v programovacím jazyce Java Standard Edition 8.	1
NP02	JavaFX knihovna	Je použita nová knihovna JavaFX.	1
NP03	MVC architektura	Architektura aplikace je založena na principech MVC.	1

Zdroj: vlastní tvorba

3.1.4 Diagram případů užití

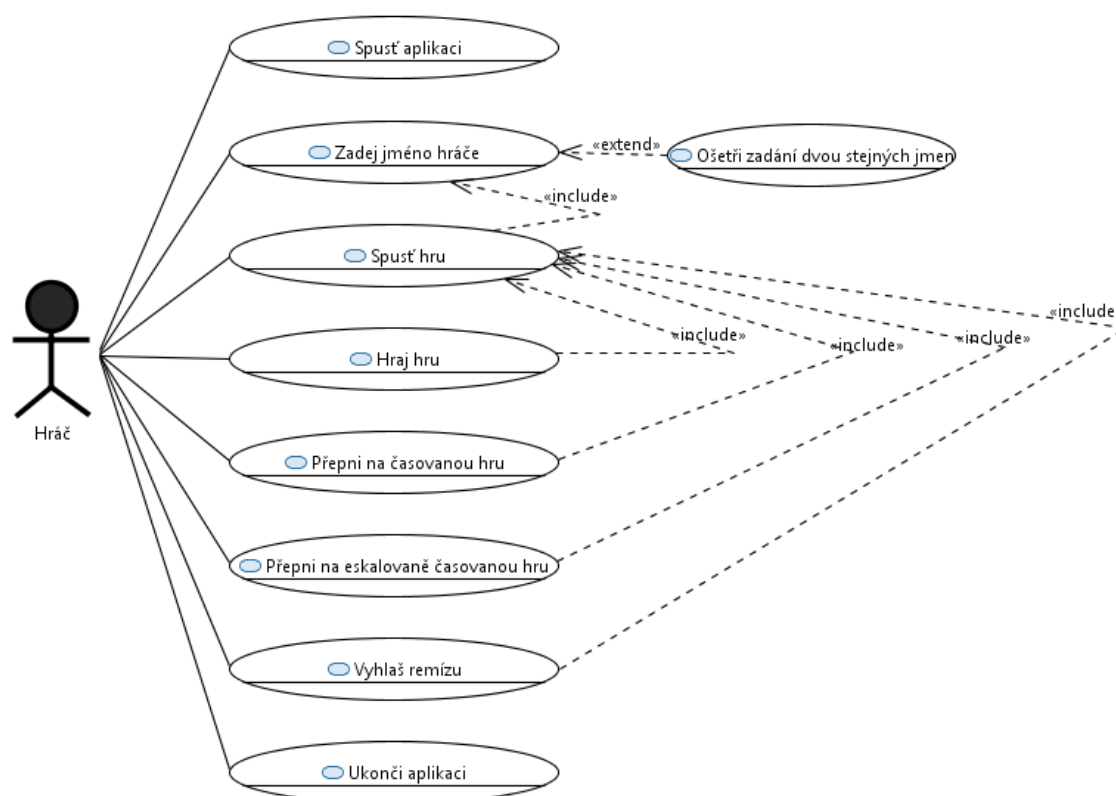
Diagram užití představuje komunikaci uživatele se systémem. „Umožňuje graficky znázornit vztah mezi aktéry a případy užití.“ [15]

Hlavním a jediným aktérem v diagramu užití je Hráč, který má možnost spouštět příslušné případy užití (viz Obr. 11). Jednotlivé případy užití vychází z požadavků:

- UC1 – Spust' aplikaci,
- UC2 – Zadej jméno hráče,

- UC3 – Spust' hru,
- UC4 – Hraj hru,
- UC5 – Přepni na časovanou hru,
- UC6 – Přepni na eskalovaně časovanou hru,
- UC7 – Vyhláš remízu,
- UC8 – Ukonči aplikaci.

Obr. 11: Příklad užití aplikace piškvorky



Zdroj: vlastní tvorba (v programu Papyrus)

3.2 Analytická studie

Cílem této studie je provést ukázkovou analýzu a návrh aplikace piškvorky.

3.2.1 Varianta MVC a další upřesnění - Návrh architektury systému

Aplikace je postavena jako klientská aplikace v Javě SE. Jelikož standardní edice Javy definuje komponenty uživatelského rozhraní takovým způsobem, že zpravidla pokrývají vstup i výstup (např.: na Fieldu můžeme definovat tzv. Placeholder, který prezentuje informace uživateli, tedy výstup, zároveň Field slouží k přijmutí údajů od uživatele tedy vstup). Z toho vyplývá varianta MVP – Model View Presenter (viz kapitola 1.2 Varianty MVC).

Dále je použita nová knihovna JavaFX, která nabízí dva způsoby vytvoření uživatelského rozhraní:

- Užití jazyka JavaFX FXML založeného na XML, který poskytuje strukturu právě pro tvorbu uživatelských rozhraní odděleně od aplikační logiky. [13]
- Klasickým programováním v Javě.

Pro tvorbu View jsem vybral možnost JavaFX FXML, z důvodu přehledného oddělení uživatelského rozhraní od aplikační logiky. Pouze v případě potřeby použití runtime kontextových informací, či jednoduššího použití hromadného generování komponent použiji klasické programování v Javě.

JavaFX FXML je založen na principech vytvoření XML souboru, utvářejícího uživatelské rozhraní (View), a vytvoření Controlleru, ve stejném balíčku (package), a následné provázání obou komponent. Jakákoli abstrakce, či užití rozhraní je zde vyloučeno. Takovéto nedostatečné oddělení by odpovídalo variaci MVC zvanou Model-Delegate a zároveň Model-View-Presenter (viz kapitola 1.2 Varianty MVC).

Pro účely této aplikace je nezbytné užít kombinaci dvou variant, Model-View-Presenter a Model-Delegate. Z důvodu jednoduchosti a srozumitelnosti však dále označuji vzor zpravidla jako Model-View-Controller. Komponenty View a Controller jsou však pevně provázány, že v podstatě tvoří vrstvu Delegate. A Controller představuje spíše Presenter, jelikož vstup přenechává na View.

View (MVC)

Uživatelské rozhraní je zpravidla definováno v XML souborech s koncovkou „.fxml“. Takovéto soubory jsou umístěny v balíčku *delegate*. Ke každé uživatelské obrazovce existuje Controller, který je napevno z XML souboru odkazován. View je představitelem návrhového vzoru Composite (viz kapitola 1.3 Architektura).

Vybrané stylování obrazovek přesunu do externího CSS souboru, částečně tím vznikne oddělení struktury uživatelské obrazovky od jejích konkrétních stylů zobrazení.

Controller (MVC)

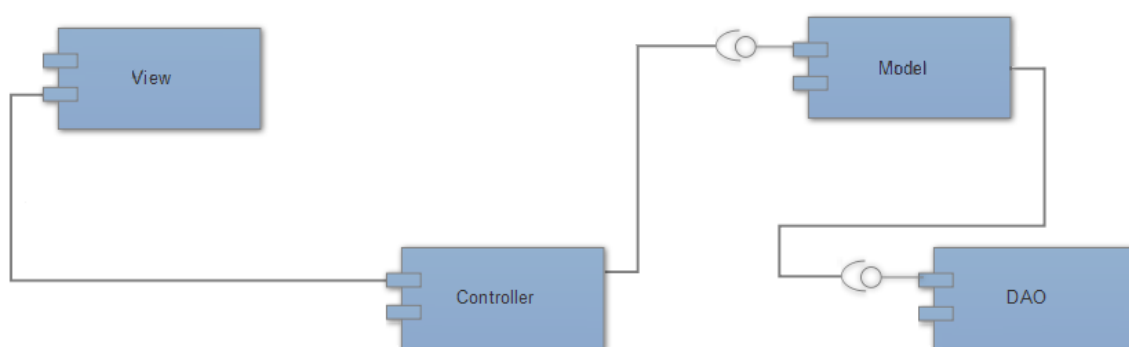
Controller je umístěn taktéž v balíčku *delegate*. Registruje se jako Observer u daného Modelu (Subjektu). Controller pevně drží referenci na View a v abstraktní rovině drží referenci i na Model (vyžaduje rozhraní Modelu) (viz Obr. 12).

Model (MVC)

Model zde představuje vlastní logiku piškvorek, uchovává informace o hráčích a stav rozehrané hry (pouze v rámci kontextu jedné hry). Model je Subjektem upozorňující registrované Observery na vykonaný tah ve hře, je tedy maximálně oddělený od View a Controlleru. Dále vystavuje rozhraní pro Controller. Model je umístěn v balíčku *model*. Model vyžaduje rozhraní DAO (viz Obr. 12).

DAO představuje komponentu zajišťující ukládání dat o uživateli, je umístěn v podbalíčku *model.dao*. A vystavuje rozhraní pro Model (viz Obr. 12).

Obr. 12: Komponenty aplikace piškvorky



Zdroj: vlastní tvorba (v online aplikaci cloud.smartdraw.com)

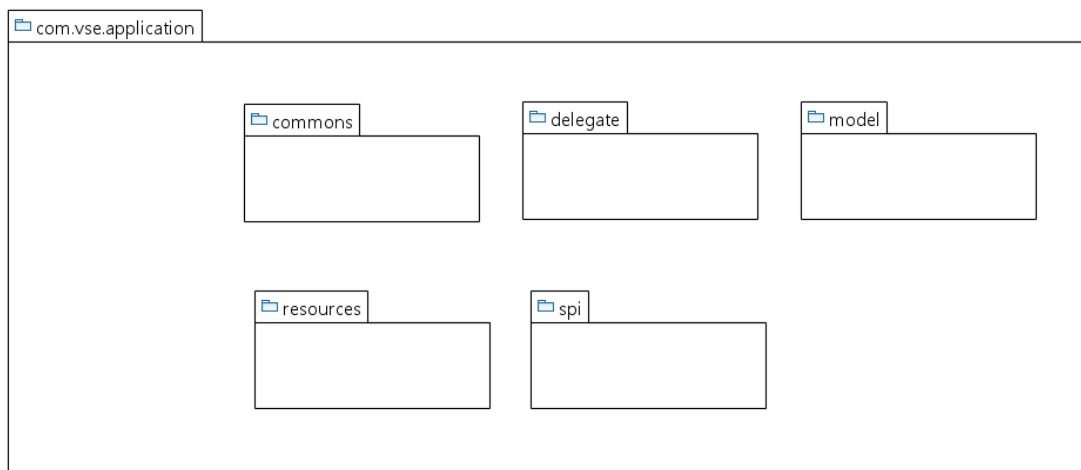
3.2.2 Diagram balíčků

Diagram balíčků zobrazuje hierarchickou strukturu elementů modelu a jejich závislosti. „Balíček sdružuje libovolné elementy UML, které spolu souvisí a vytváří z nich skupiny, mezi kterými můžou existovat vztahy.“ [15]

Všechny balíčky jsou obsaženy v kořenových balíčcích – *com.vse.application*, kde také nalezneme konstrukci pro spouštění aplikace. Hlavní balíčky vypadají následovně (viz Obr 13):

- *delegate* – obsahuje View, Controllery a další konstrukce týkající se prezentace,
- *model* – obsahuje Model a další konstrukce týkající se logiky piškvorek a uchovávání dat,
- *commons* – obsahuje konstrukce společné pro ostatní balíčky,
- *spi* – obsahuje jednotlivá rozhraní,
- *resources* – obsahuje obrázky, a další soubory.

Obr. 13: Základní balíčky aplikace piškvorky

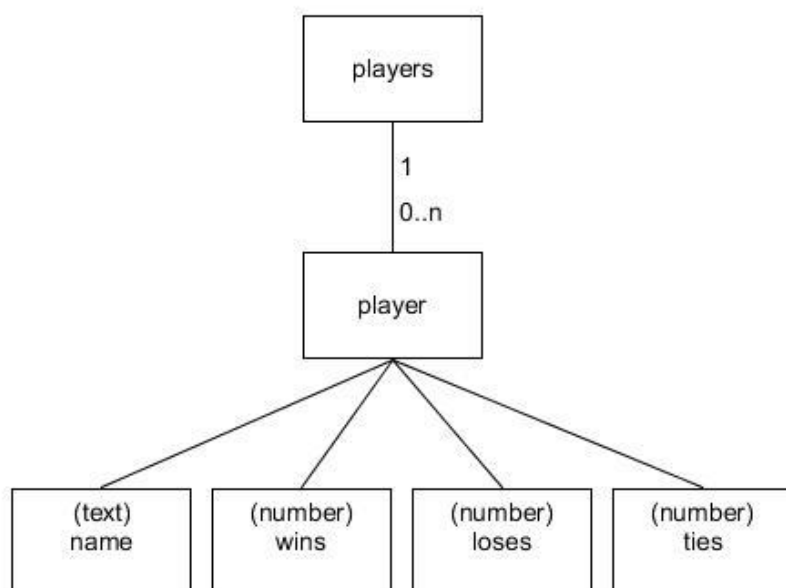


Zdroj: vlastní tvorba (v pluginu Papyrus)

3.2.3 Datový model

Stav hry je uchováván pouze v rámci kontextu dané hry, není tedy umožněno hru uložit a poté v ní později pokračovat. Ukládat je tedy potřeba pouze data uživatelů, konkrétněji: hráčské jméno, počet výher, proher a remíz. Z důvodu jednoduchosti a předpokládaného malého počtu uložených uživatelů, použijí ukládání v podobě XML do souboru. XML struktura je znázorněna na Obrázku 14.

Obr. 14: Datový model - XML



Zdroj: vlastní tvorba (v programu UMLet verze 14.1.1)

3.2.4 Obecná definice aplikace

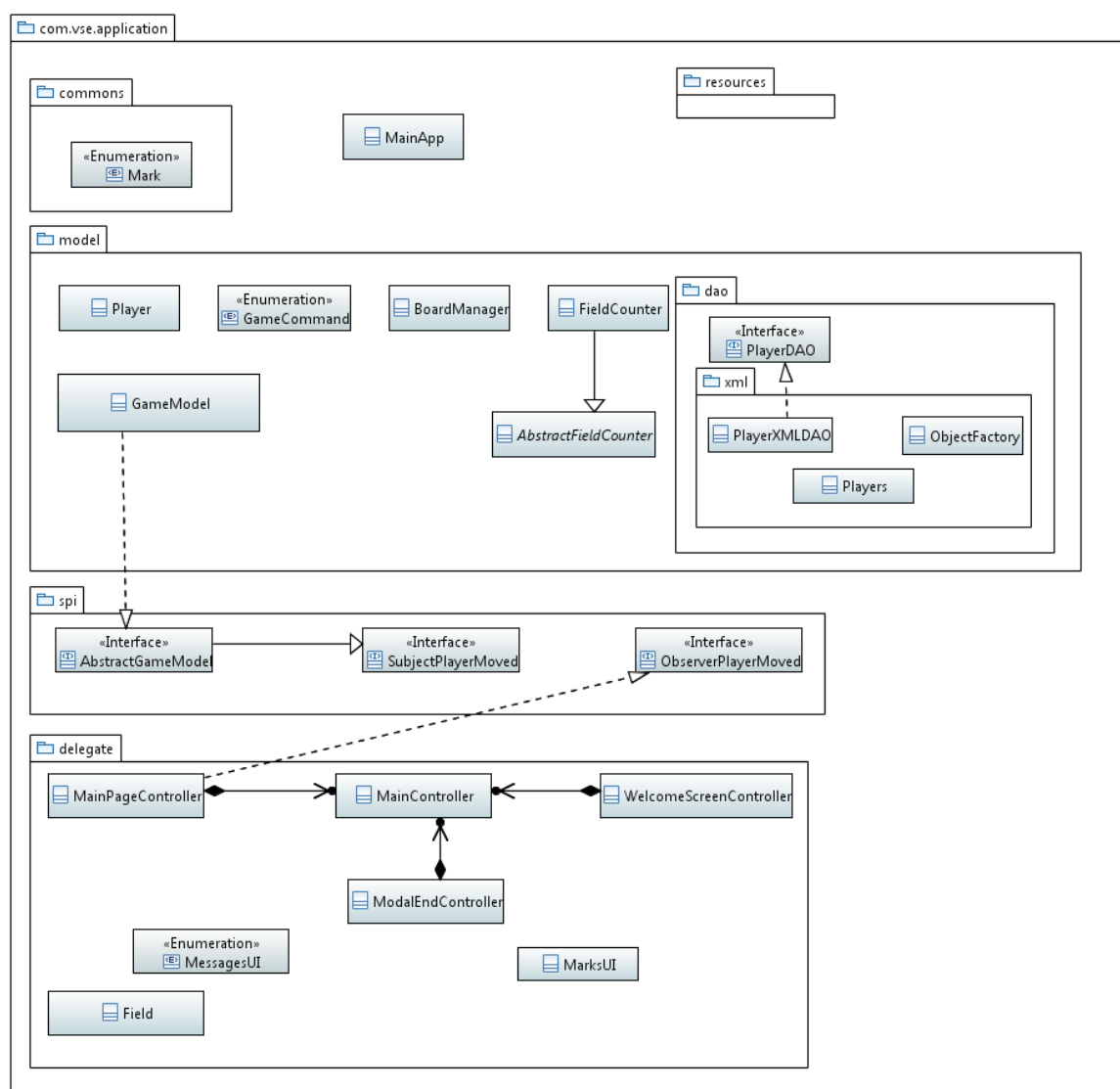
Nejdříve nežli začnu představovat jednotlivé diagramy, je třeba v rychlosti zmínit základní obrazovky a umístění klíčových komponent v aplikaci.

První obrazovkou, která je hráčům zobrazena při řádném spuštění aplikace, je obrazovka pro zadání hracích jmen, dále jen welcome screen. Při úspěšném zadání hracích jmen se zobrazí hra samotná, dále jen main page. Při jakémkoli ukončení/dokončení hry (nikoli aplikace jako takové) je zobrazena modální obrazovka, zobrazující výsledek hry a dotaz zda chtějí hrát ještě jednou, dále jen modal end.

Jednotlivé třídy aplikace dle balíčků (kořenovým balíčkem je *com.vse.application*) (viz Obr. 15):

-
- **MainApp** – třída pro spuštění aplikace
 - *Spi*
 - **AbstractGameModel** – rozhraní zajišťující abstrakci Modelu
 - **SubjectPlayerMoved, ObserverPlayerMoved** – rozhraní pro návrhový vzor Observer
 - *model*
 - **GameModel** – hlavní třída logiky aplikace (Model)
 - **AbstractFieldCounter** – třída zajišťující abstrakci FieldCounteru
 - **FieldCounter** – třída zajišťující iterace nad hracím plánkem
 - **BoardManager** – správce hracího plánu
 - **GameCommand** – výčet standardizující interní komunikaci v balíčku
 - **Player** – třída reprezentující entitu hráče v aplikaci
 - *dao*
 - **PlayerDAO** – rozhraní zajišťující abstrakci ukládání dat
 - *xml*
 - **PlayerXMLDAO** – třída zodpovědná za ukládání dat do XML
 - **Players** – entita hráče reprezentující uložení v XML
 - **ObjectFactory** – továrna pro třídu Players
 - *delegate*
 - **MainController** – hlavní třída vrstvy Delegate
 - **MainPageController, WelcomeScreenController, ModalEndController** – konkrétní Controllery svázané se svými View
 - **Field** – třída reprezentující vizuální políčko na plánu
 - **MarksUI** – třída držící všechny obrázky políček
 - **MessageUI** – výčet obsahující texty pro aplikaci
 - *commons*
 - **Mark** – výčet obsahující základní znaky hráčů (X, O)

Obr. 15: class diagram aplikace



Zdroj: vlastní tvorba (v pluginu Papyrus)

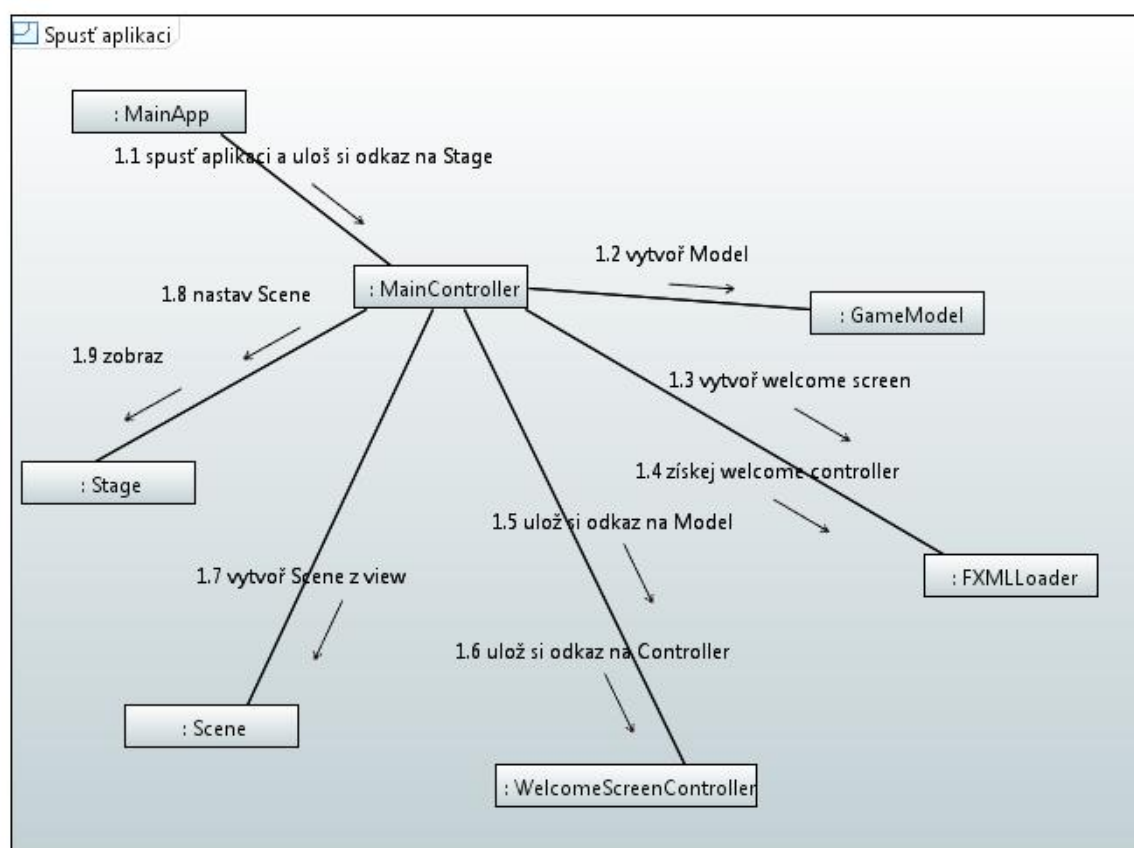
3.2.5 Diagram komunikace

Diagram komunikace prezentuje interakci objektů s důrazem na spojení. [15]

V této kapitole jsem postupně vytvořil diagramy komunikací pro jednotlivé případy užití, které obecně rozebírají tok zpráv napříč aplikací (vyjma UC ukončí aplikaci, který je příliš jednoduchý, pro to aby byl zobrazen). Uživatel vždy začíná spuštěním aplikace (viz Obr. 16), následuje postupné zadání jména oběma hráči (viz Obr. 17), poté uživatel spouští hru samotnou (viz Obr. 18). Následně hráč může provést tah (viz Obr. 19 a 20), přepnout na časovanou hru – resp. (de)aktivovat časovanou hru (viz Obr. 21 a 20), (de)aktivovat eskalovaně časovanou hru (viz Obr. 22 a 20) nebo vyhlásit remízu (viz Obr. 23 a 20).

Use Case – spust' aplikaci (viz Obr. 11): Vše začíná spuštěním java archivu, kdy je spuštěna hlavní metoda v aplikaci, která je umístěna v třídě `MainApp` umístěné v adresáři `com.vse.application`. Hlavní metoda prvotně inicializuje FX platformu, následně obdrží primární Stage a vyžádá spuštění aplikace na `MainControlleru`, kterému zároveň získanou Stage předá. `MainController` je hlavní třídou ve vrstvě Delegate řídící tok aplikace a je umístěn v balíčku `delegate`³. `MainPageController` vytvoří instanci `GameModelu`, která je následně v abstraktní podobě (`AbstractMainController`) předávána jednotlivým `Controllerům` zodpovědným za jednotlivá View. Následně si vytvoří instanci `FXMLLoaderu`, po kterém vyžádá vytvoření welcome screen – View, a získání `Controlleru` pro toto View. Získanému `Controlleru` (`WelcomeScreenController`) předá odkaz na instanci `AbstractGameModelu` a hlavního uzlu aplikace (`MainControlleru`, tedy sama sebe). Zbývá již vytvořit `Scene` ze získaného View, `Scene` nastavit na primární Stage a Stage zobrazit. Viz Obr. 16.

Obr. 16: Diagram komunikace (UC spust' aplikaci)

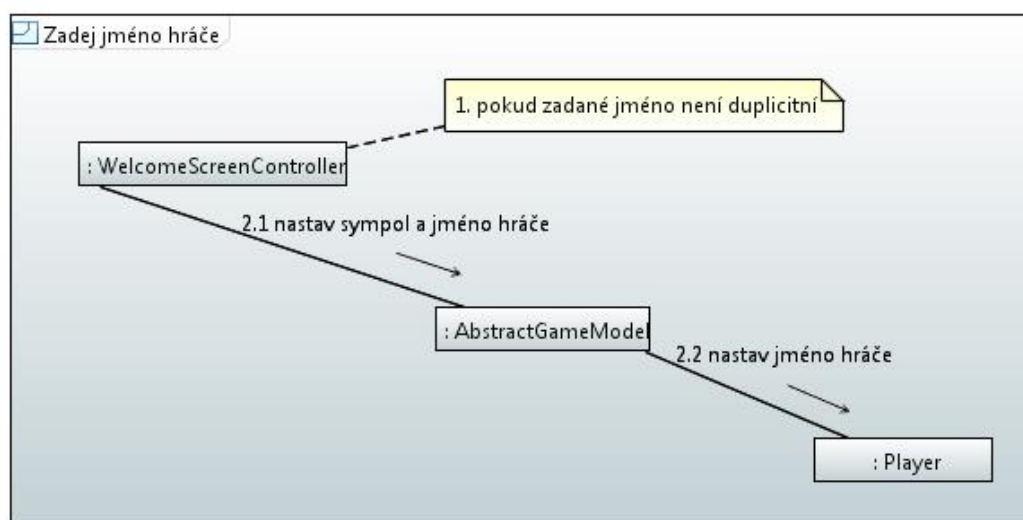


Zdroj: vlastní tvorba (v pluginu Papyrus)

³ `com.vse.application.delegate`

Use Case – zadej jméno hráče (viz Obr. 11): Komunikace se spustí zadáním hracího jména uživatelem, `WelcomeScreenController` nejdříve zkontroluje zda jméno není duplicitní, a pokud není, pak spustí nastavení symbolu a hracího jména uživateli v `AbstractGameModelu`, a ten nastaví hrací jméno konkrétní entitě hráče. Viz Obr. 17.

Obr. 17: Diagram komunikace (UC zadej jméno hráče)

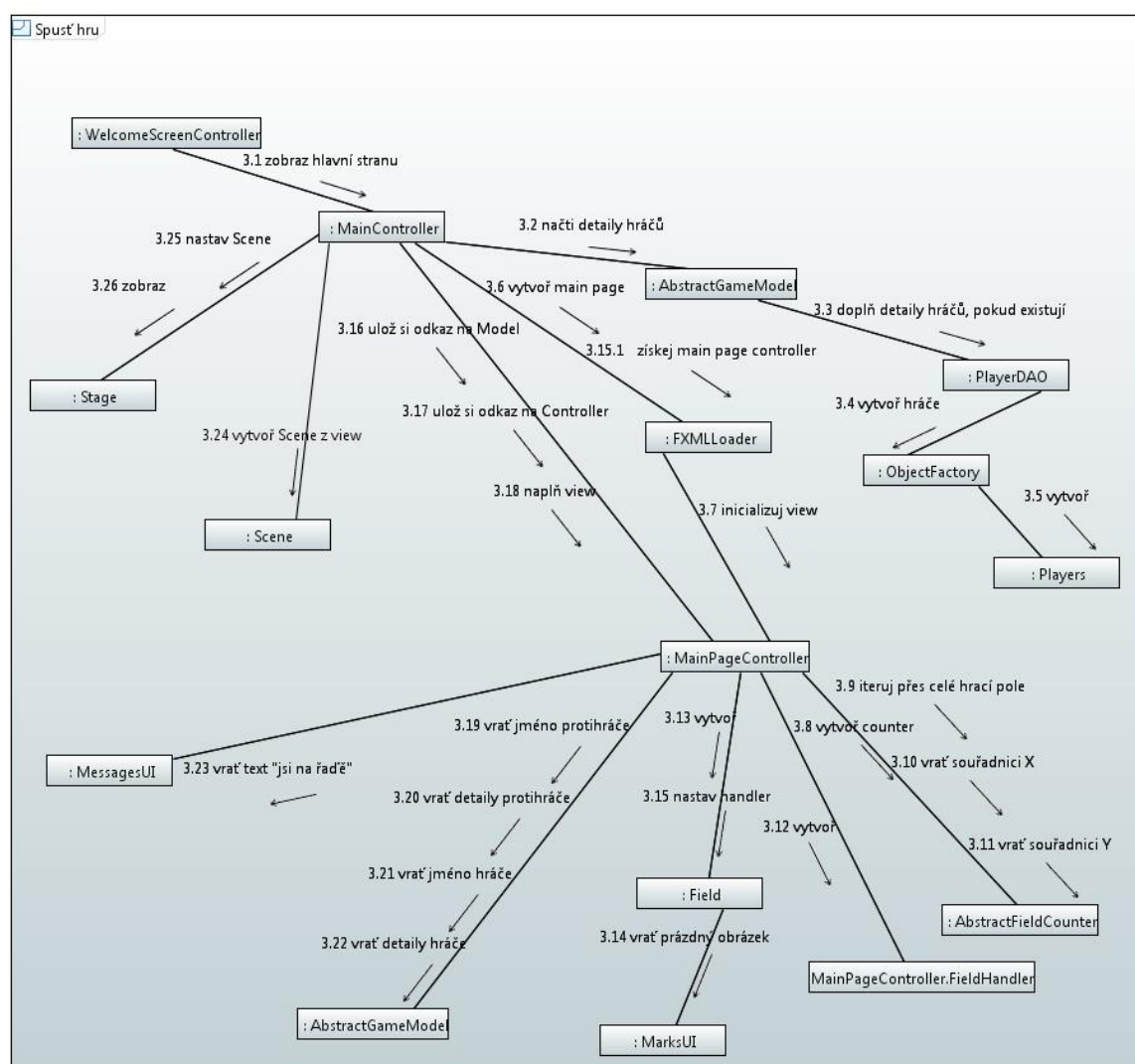


Zdroj: vlastní tvorba (v pluginu Papyrus)

Use Case – spust' hru (viz Obr. 11): tato komunikace probíhá po úspěšném nastavení druhého hracího jména. `WelcomeScreenController` požaduje po `MainControlleru` načtení a zobrazení main page (již s hracím plánkem). `MainController` požaduje načtení detailů hráčů po `AbstractGameModelu`, který předává oba hráče do `PlayerDAO`, které doplní detaily hráčů (počet výher, proher a remíz), pokud již hru někdy předtím hráli pod stejným hracím jménem. Jelikož jsem se rozhodl pro ukládání těchto dat použít ukládání do souboru XML pomocí JAXB knihovny, tak knihovna pomocí `ObjectFactory` vytvoří třídu `Players`, reprezentující strukturu XML úložiště. Tímto jsou detaily uživatelů načteny (pokud vůbec existují) a `MainController` pokračuje vytvořením `MainPage` a získáním odpovídajícího `Controlleru` (`MainPageController`) pomocí `FXMLLoaderu`. Ten při vytváření `View` spouští základní inicializaci v `MainPageControlleru`. `MainPageController` v této inicializaci spouští iteraci skrz celý hrací plánec pomocí nově vytvořeného `AbstractFieldCounteru`, v jednotlivých iteracích získáním souřadnice X a Y hracího plánu, vytvořením `FileHandleru` (pro zachycení kliknutí na dané políčko),

vytvořením políčka a následným nastavením FileHandleru a příslušného obrázku na dané políčko, vytváří vizuální zobrazení hracího plánu v rámci MainPage. MainController následně předá MainPageControlleru odkaz na AbstractGameModel a odkaz na hlavní Controller (tedy sám sebe) a spustí finální inicializaci MainPageControlleru, která příslušnými dotazy na AbstractGameModel získá hrací jména a detaily hráčů, které nastaví do MainPage a taktéž tam nastaví úvodní hlášku, že je hráč na řadě. Poté již MainController z kompletně nastaveného MainPage vytvoří Scene, kterou nastaví na Stage a tu zobrazí. Viz Obr. 18.

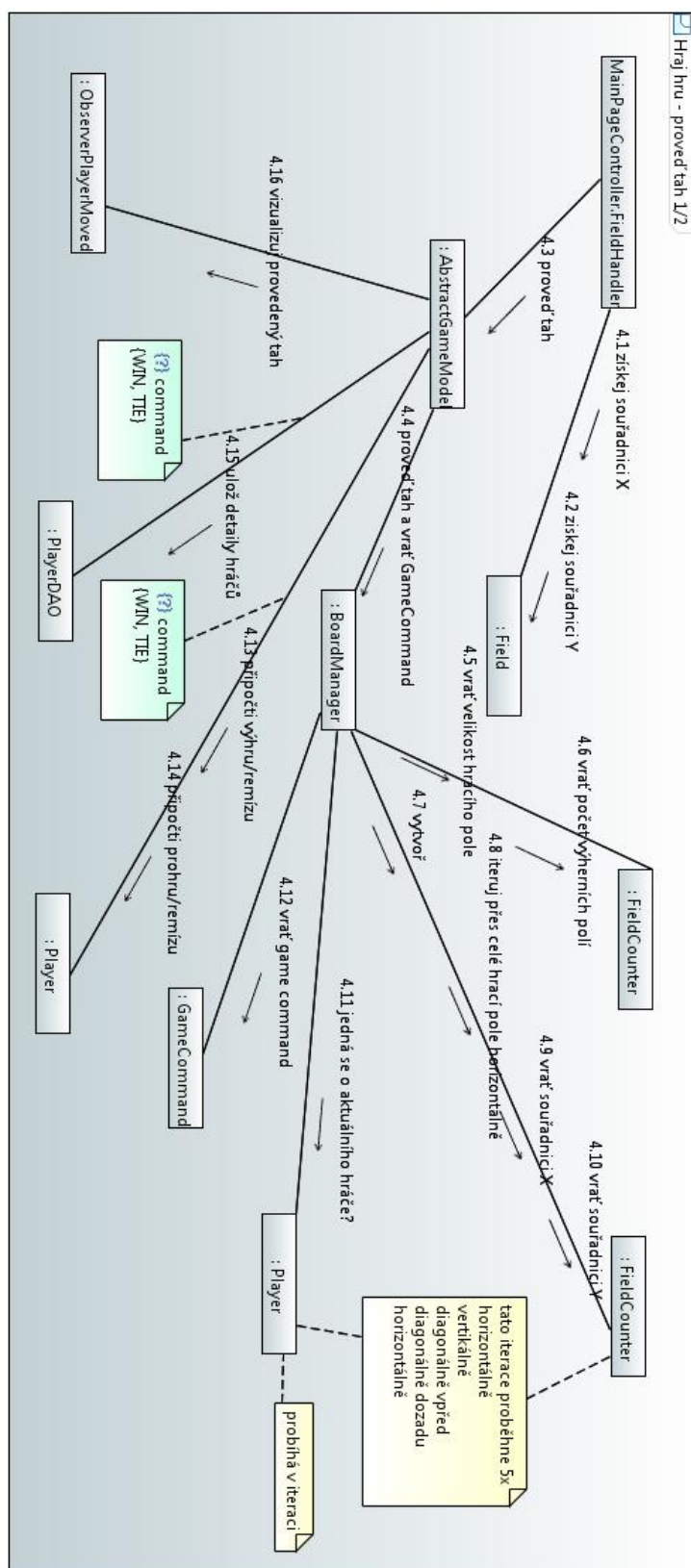
Obr. 18: Diagram komunikace (UC spustí hru)



Zdroj: vlastní tvorba (v pluginu Papyrus)

Use Case – hraj hru 1/2 (viz Obr. 11): tento diagram komunikace představuje proces spuštěný úspěšným kliknutím hráče na dané herní políčko (provedení tahu). Na začátku stojí `FieldHandler`, který zaregistruje zvolení daného hracího políčka, získá souřadnici X a Y hracího políčka a vyžádá provedení tahu na `AbstractGameModelu`, který spustí provedení tahu v `BoardManageru`. `BoardManager` provede pět obdobných iterací, čtyři pro kontrolu zda hráč aktuálním tahem nevyhrál (nejdříve iteruje zda nevznikla vodorovná, poté vertikální, následně dopředně šikmá a na závěr zpětně šikmá pěťice), pátou iterací kontroluje zda nejsou všechny pole zabrána (v tom případě je třeba vyhlásit remízu). Před iterací se vždy získá velikost hracího pole, počet výherních polí a vytvoří `FieldCounter`, pomocí kterého pak iteruje daným směrem napříč hracím plánkem, v každém kroku se pak dotáže `FieldCounteru` na příslušnou souřadnici X a Y, pomocí kterých zjistí zda se na dané pozici v plánu nachází hráč, který je na tahu. Po dokončení iterací vrátí `BoardManager` `AbstractGameModelu` v jakém stavu je hra, zda se očekává další tah, hráč vyhrál, či způsobil remízu. `AbstractGameModel` zajistí povýšení počtu výher u hráče na tahu a u druhého hráče povýšení počtu proher (resp. povýšení počtu remíz v případě remízy u obou) a uložení těchto změn pomocí `PlayerDAO`. `AbstractGameModel` vše zakončí vyžádáním odpovídající vizualizace provedeního tahu a stavu hry na `ObserverPlayerMoved` (registrovaném `Observeru`). Tím je spuštěna část, zobrazena na Obrázku 20, a popsaná v následujícím odstavci. Viz Obr. 19.

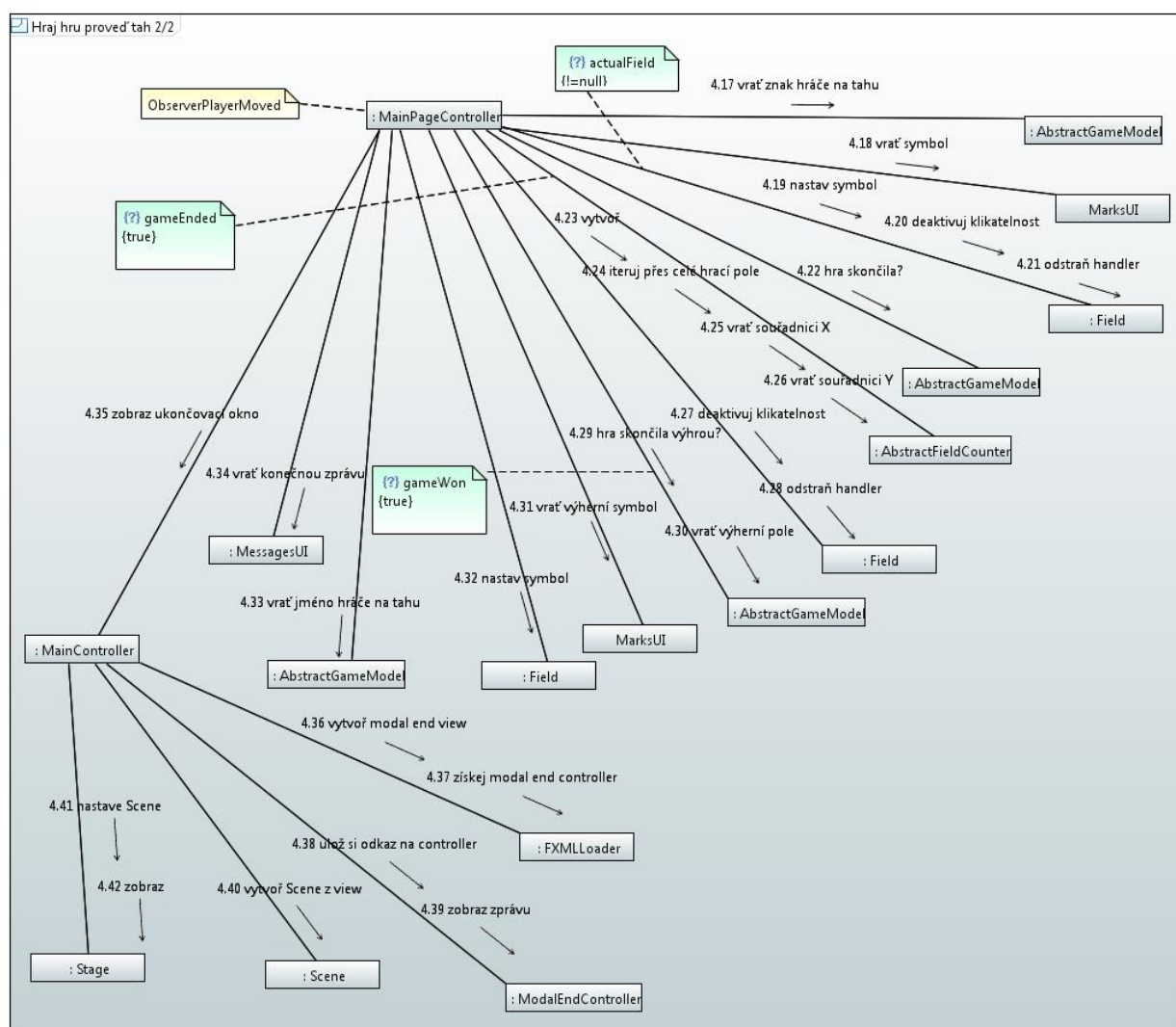
Obr. 19: Diagram komunikace (UC hraj hru 1/2)



Zdroj: vlastní tvorba (v pluginu Papyrus)

Vizualizace tahu hráče a stavu hry: `MainPageController` je jako `Observer` registrovaný u `AbstractGameModelu`, vizualizace tahu hráče a stavu hry vypadá následovně. Nejprve – pokud je zvoleno nějaké aktuální políčko (tedy hráč provedl nějaký tah) - tak si od `AbstractGameModelu` zjistí znak hráče, který je na tahu, získá si jemu odpovídající symbol z `MarksUI` a ten nastaví danému políčku, zároveň políčko (`Field`) deaktivuje a odstraní z něho `FieldHandler` (aby nemohlo být znovu použito). Následně zjistí z `AbstractGameModelu` zda hra skončila, pokud ano, získá instanci `AbstractFieldCounteru`, a „proiteruje“ celé hrací pole, v jednotlivých cyklech získá souřadnice X a Y a pomocí nich postupně deaktivuje a odstraní `FieldHandler` celého hracího pole (z všech `Fieldů`). `MainPageController` pokračuje dotazem na `AbstractGameModel` zda hra skončila výhrou, v případě kladné odpovědi, si rovnou od `AbstractGameModelu` získá seznam pěti výherních polí. Získá si z `MarksUI` výherní symbol odpovídající hráči na tahu a symbol nastaví na daných pět hracích políček (`Fieldů`). `MainPageController` si dále zjistí jméno hráče na tahu od `AbstractGameModel` s pomocí `MessageUI` sestaví zprávu, kterou předá `MainControlleru` s požadavkem na zobrazení modal end. `MainController` si pomocí nově vytvořeného `FXMLLoaderu` získá `ModalEnd` a příslušný `Controller` (`ModalEndController`), kterému hned vzápětí předá odkaz na hlavní `Controller` (tedy sám sebe) a nastaví mu získanou zprávu. Z `ModalEnd` vytvoří `Scene`, kterou nastaví na `Stage` a tu zobrazí. Viz Obr. 20.

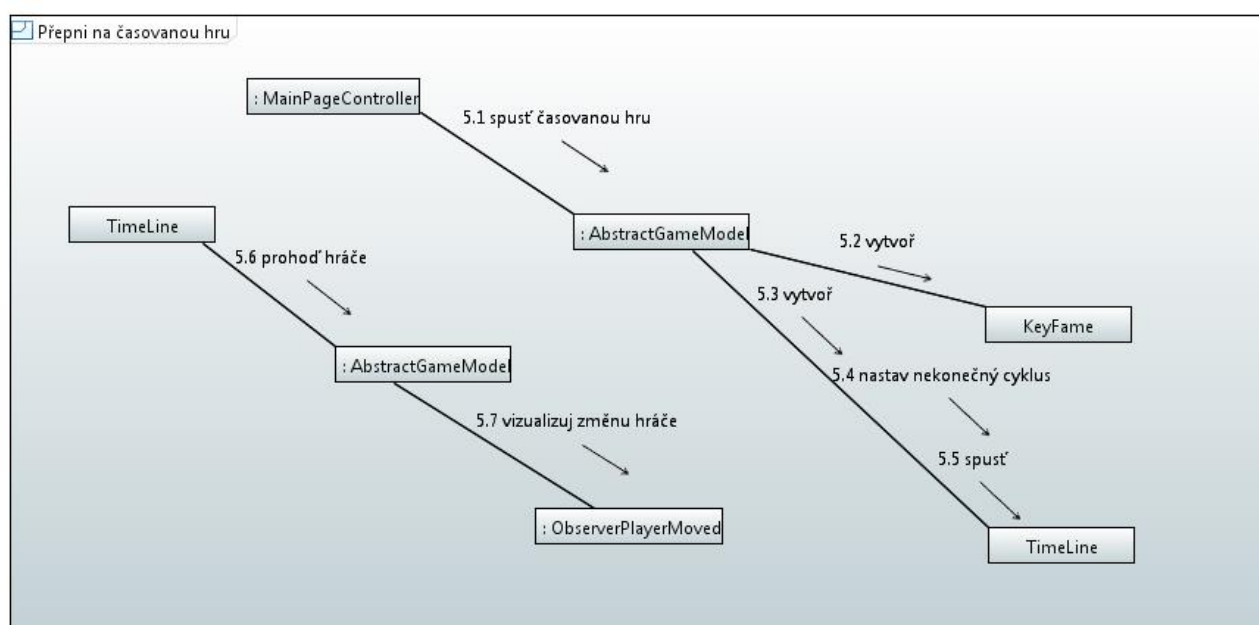
Obr. 20: Diagram komunikace (Vizualizace tahu hráče)



Zdroj: vlastní tvorba (v pluginu Papyrus)

Use Case – přepni na časovanou hru (viz Obr. 11): Komunikace je spuštěna vybráním intervalu a stisknutím příslušného tlačítka na main page (MainPage). MainPageController to automaticky zaregistruje a zavolá spuštění časované hry na AbstractGameModelu, který vytvoří nový KeyFrame dle zvoleného intervalu, z kterého vytvoří Timeline, které ještě nastaví nekonečný cyklus a spustí ji. Timeline teď v pravidelných zvolených intervalech spouští prohození hráčů na AbstractGameModelu, který následně vyžádá vizualizaci změny hráče na Observerech (ObserverPlayerMoved). Viz Obr. 21. Následná komunikace (požadovaná vizualizace) je již popsána a zobrazena na Obrázku 20.

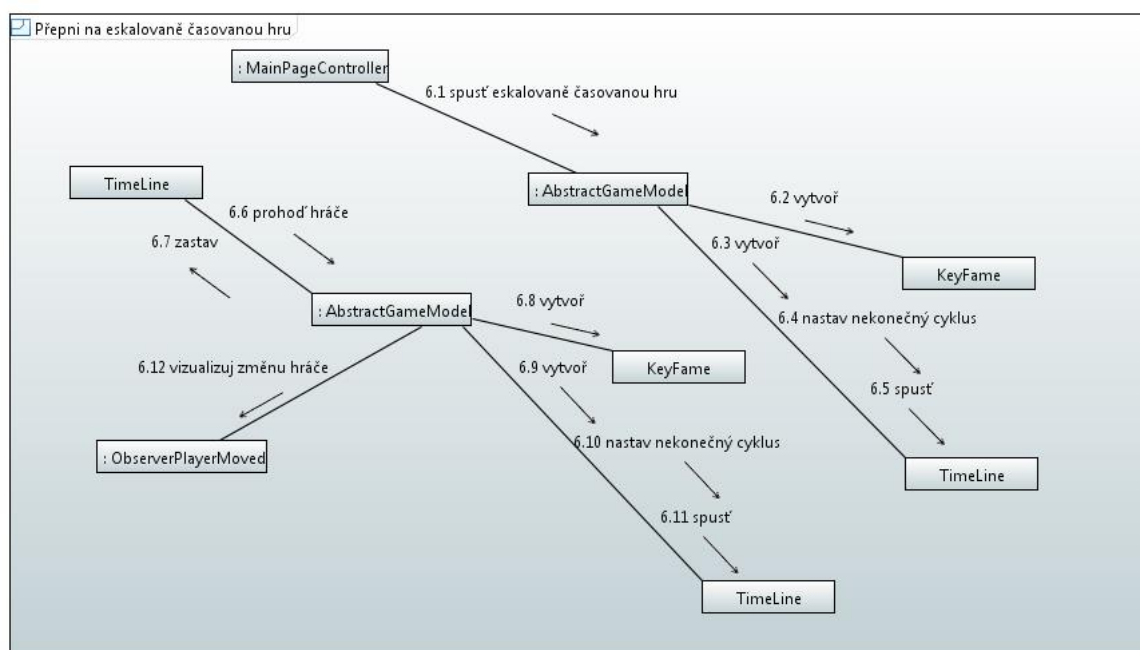
Obr. 21: Diagram komunikace (UC Přepni na časovanou hru)



Zdroj: vlastní tvorba (v pluginu Papyrus)

Use Case – přepni na eskalovaně časovanou hru (viz Obr. 11): Komunikace je velmi podobná té předešlé (viz obr. 21), je spuštěna vybráním intervalu, zaškrtnutím checkboxu (pro povolení zkracování intervalu pro přepínání hráčů) a stisknutím příslušného tlačítka na main page (MainPage). MainPageController to automaticky zaregistruje a zavolá spuštění eskalovaně časované hry na AbstractGameModelu, který vytvoří nový KeyFrame dle zvoleného intervalu, z kterého vytvoří TimeLine, které ještě nastaví nekonečný cyklus a spustí ji. TimeLine po uplynutí daného časového intervalu spustí prohození hráče a spojenou logiku se zkracováním intervalu na AbstractGameModelu, který po zastavení aktuální TimeLine, znovu vytvoří nový KeyFrame, nyní již z kratšího časového úseku, z kterého opět vytvoří nový TimeLine, nastaví ji nekonečný cyklus a spustí ji. (TimeLine za daný časový úsek opět znovu spustí tuto logiku, která ji zastaví a vytvoří a spustí novou.) Na závěr AbstractGameModel vyžádá vizualizaci po ObserverPlayerMoved. Viz Obr. 22. Následná komunikace (požadovaná vizualizace) je již popsána a zobrazena na Obrázku 20.

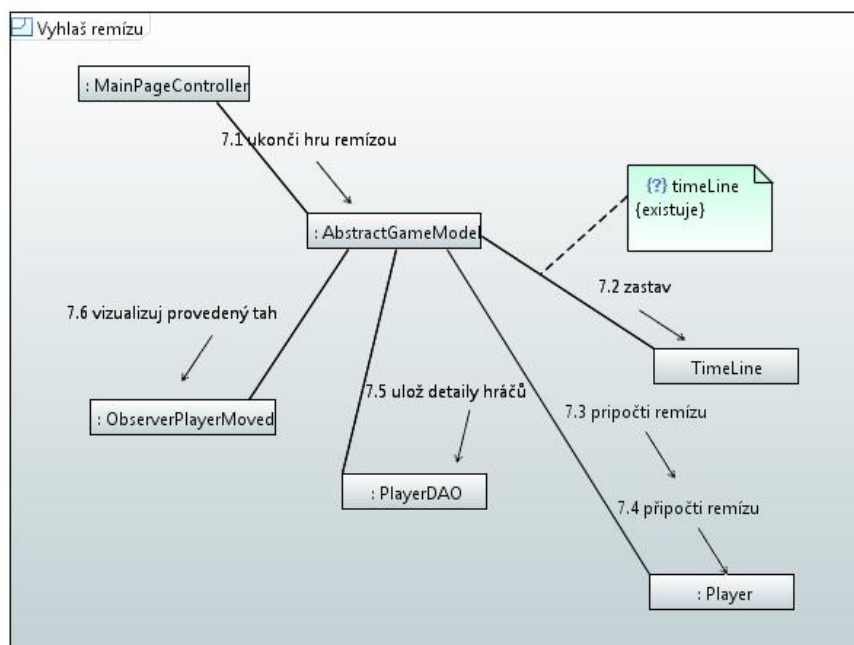
Obr. 22: Diagram komunikace (UC Přepni na eskalovaně časovanou hru)



Zdroj: vlastní tvorba (v pluginu Papyrus)

Use Case – vyhláš remízu (viz Obr. 11): Komunikace začíná stisknutím příslušného tlačítka na MainPage, tato akce je hned odchycena MainPageContollerem, který požaduje po AbstractGameModelu ukončení hry remízou. Ten ukončí TimeLine, však pouze v případě, že existuje. Aktualizuje počet remíz (připočtením jedné) u obou hráčů (Player) a pomocí PlayerDAO zajistí perzistentní uchování této aktualizace. Nakonec AbstractGameModel vyžádá vizualizace po ObserverPlayerMoved, zobrazené na Obr. 20. Viz Obr. 23.

Obr. 23: Diagram komunikace (UC Vyhlaš remízu)



Zdroj: vlastní tvorba (v pluginu Papyrus)

3.2.6 Sekvenční diagram

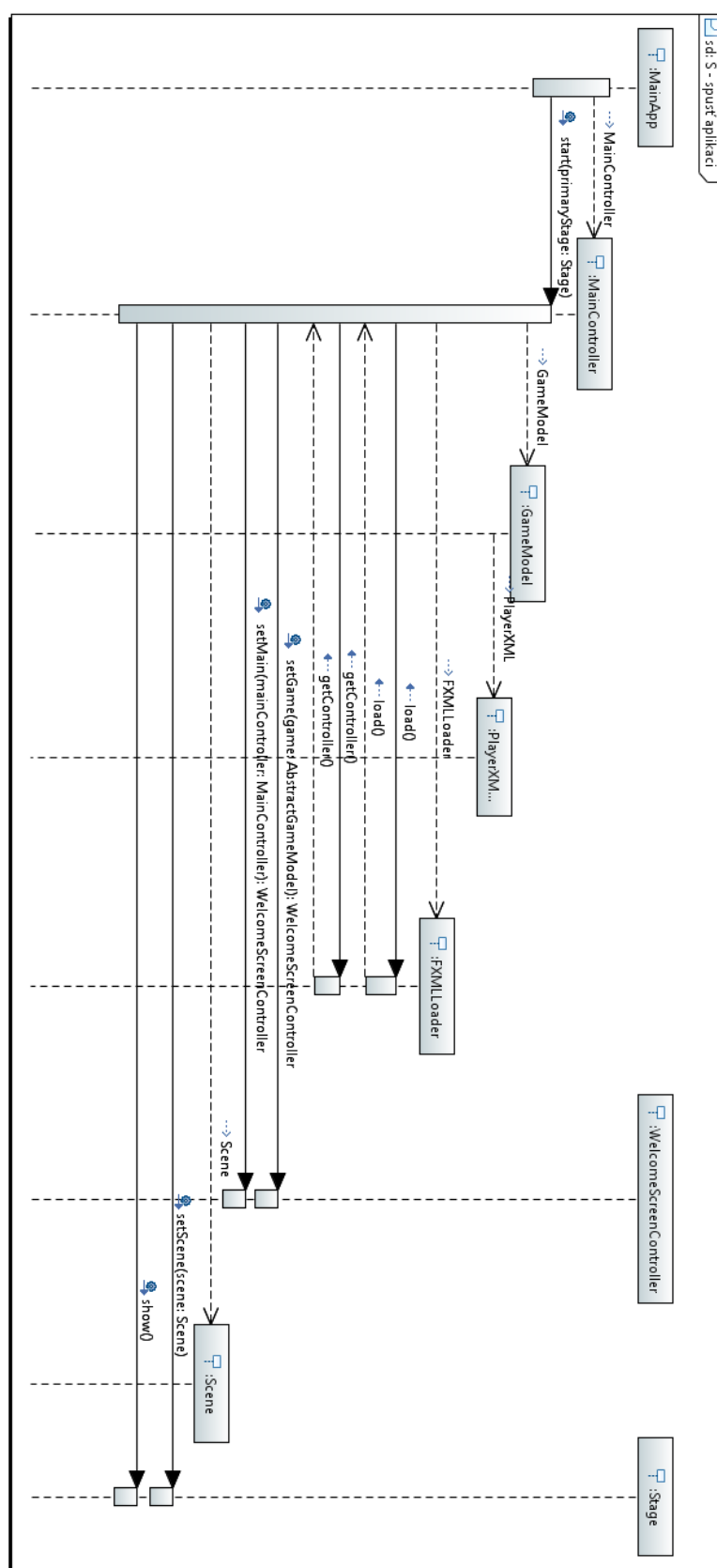
Sekvenční diagram představuje časovou posloupnost zasílání zpráv mezi objekty. „Sekvenční diagram umožňuje vyzkoušet si průchod scénáři případu užití.“ [15]

Jednotlivé sekvenční diagramy korespondují s diagramy komunikací. Sekvenční diagramy jsou definovány pro jednotlivé případy užití (s výjimkou případu užití – Ukonči aplikaci, který je velmi jednoduchý a nevyžaduje bližší rozkreslení v podobě diagramu). Tyto sekvenční diagramy jsou samovysvětlující, a nepotřebují další hlubší popis, než-li ten, který je uveden u odpovídajících diagramů komunikací. Zde uvádím přehled diagramů (vždy v závorce je odpovídající diagram komunikací):

- UC spusť aplikaci, viz Obr. 24 (diagram komunikací - Obr. 16)
- UC zadej jméno hráče, viz Obr. 25 (diagram komunikací – Obr. 17)
- UC spusť hru, viz Obr. 26 (diagram komunikací – Obr. 18)
- UC hraj hru – část 1., viz Obr. 27 (diagram komunikací – Obr. 19)
- Vizualizace tahu hráče, viz Obr. 28 (diagram komunikací – Obr. 20)
- UC přepni na časovanou hru, viz Obr. 29 (diagram komunikací – Obr. 21)

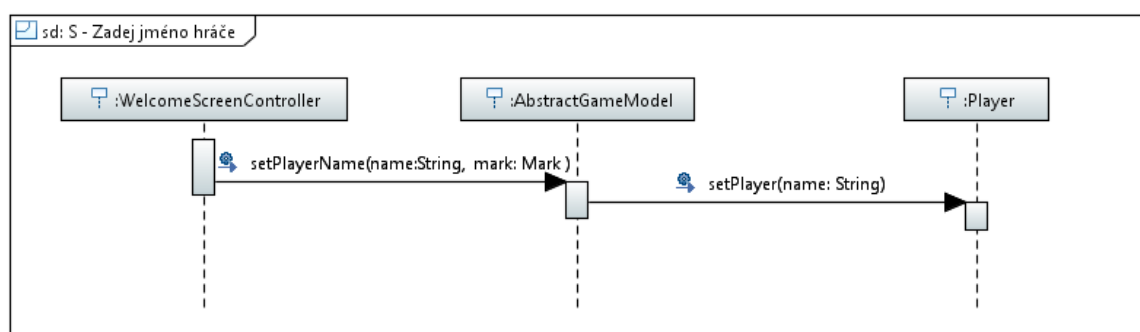
- UC přepni na eskalovaně časovanou hru, viz Obr. 30 (diagram komunikací – Obr. 22)
- UC vyhláš remízu, viz Obr. 31 (diagram komunikací – Obr. 23)

Obr. 24: Sekvenční diagram – UC spust' aplikaci



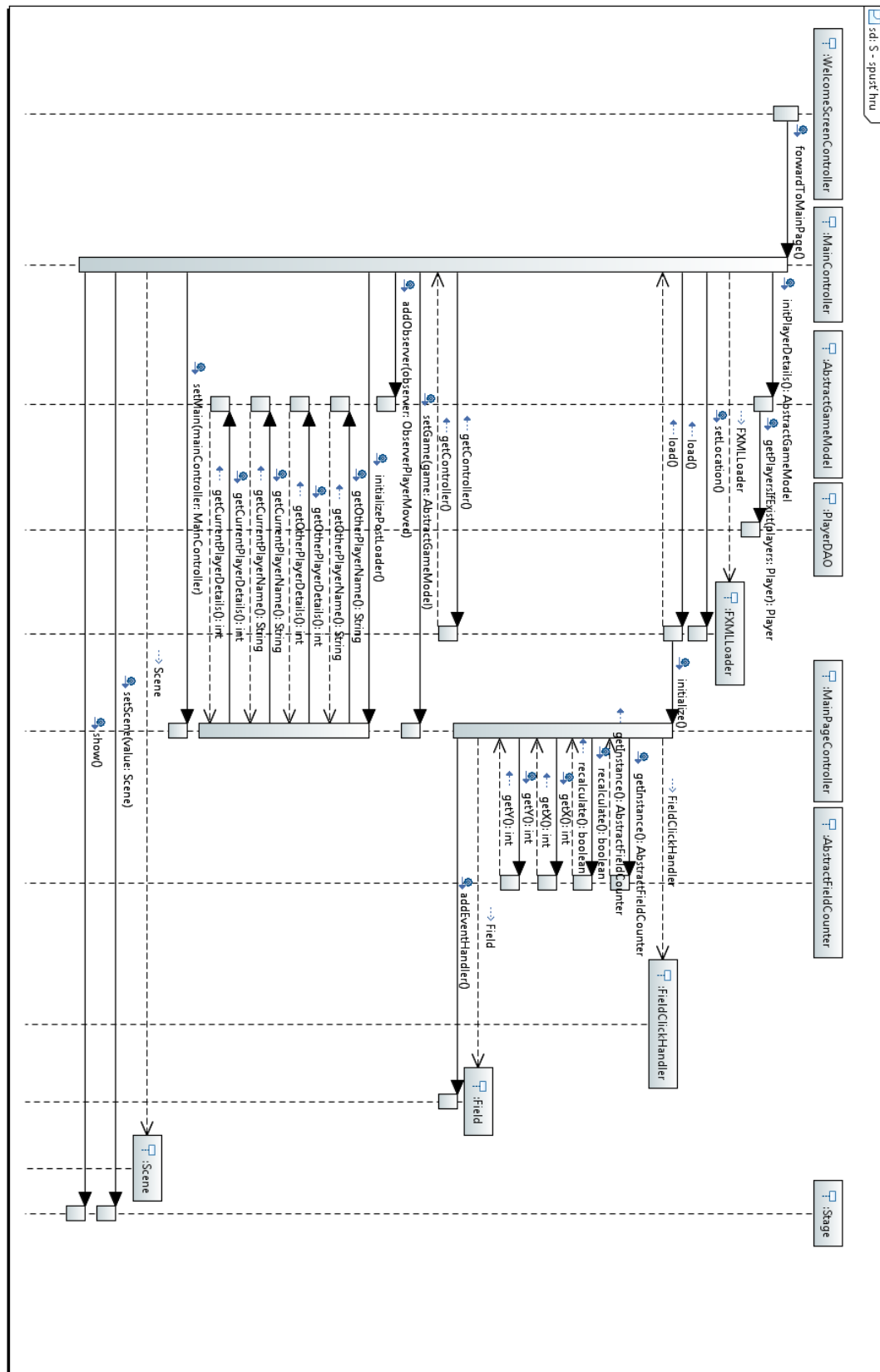
Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 25: Sekvenční diagram – UC zadej jméno hráče



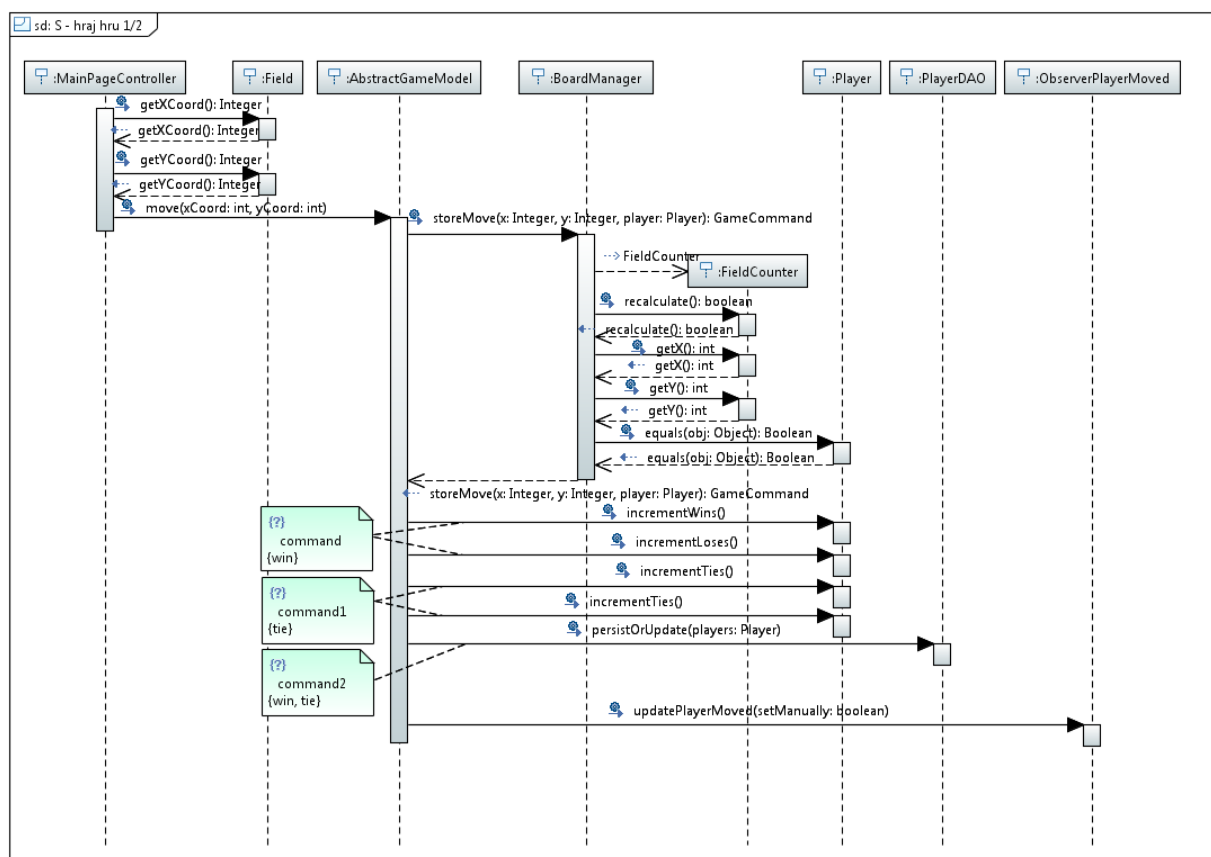
Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 26: Sekvenční diagram – UC spust' hru



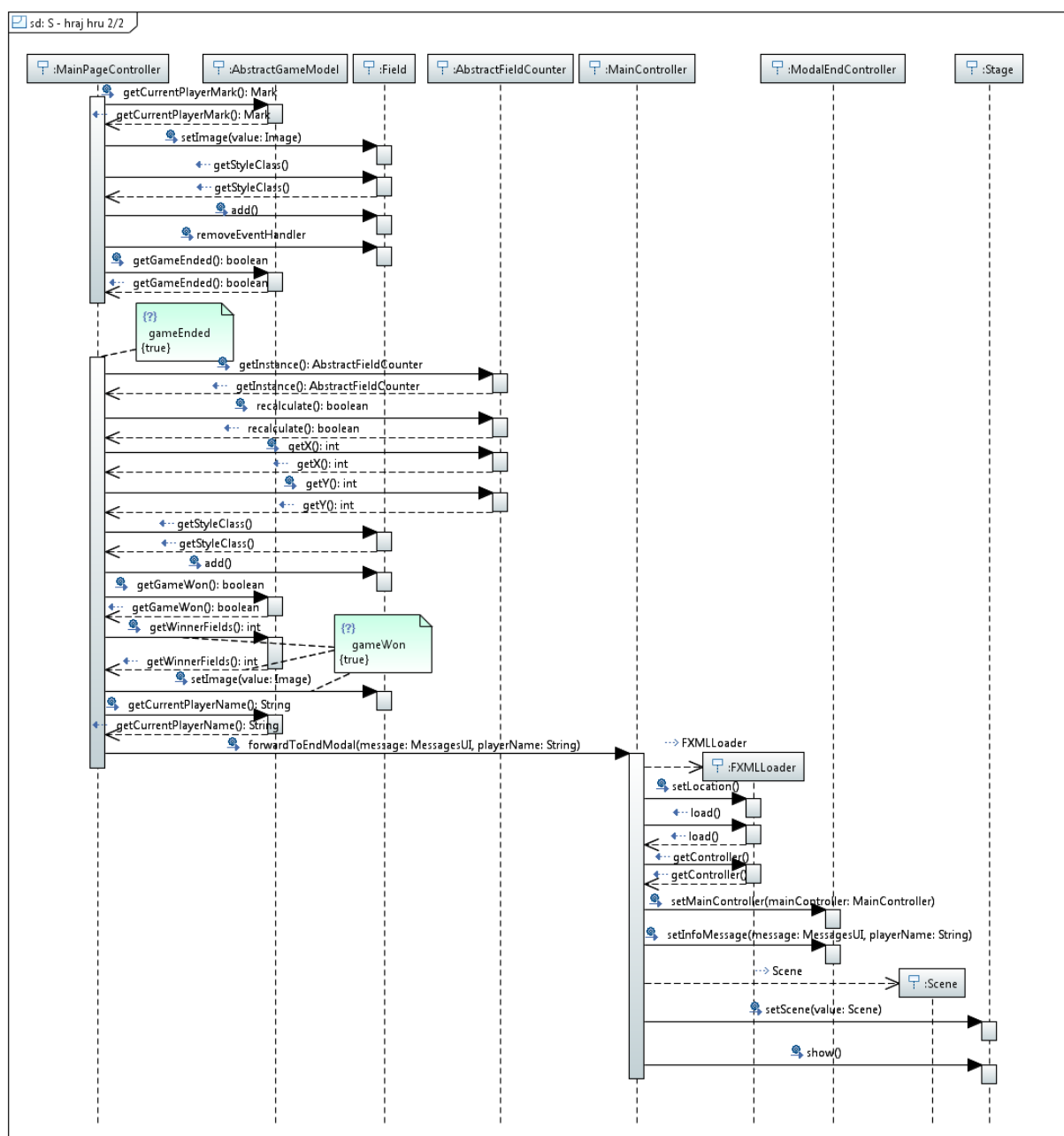
Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 27: Sekvenční diagram – UC hraj hru 1/2



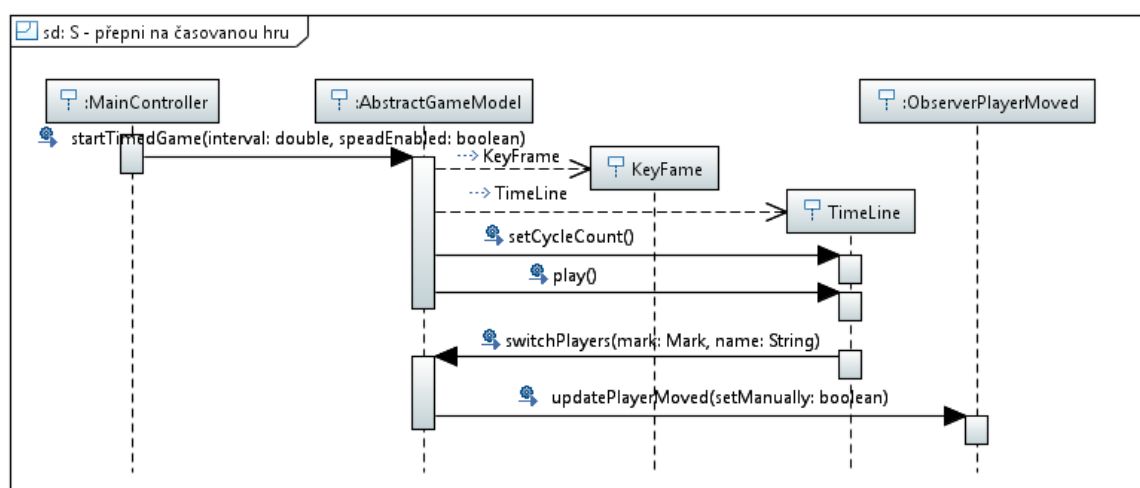
Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 28: Sekvenční diagram – Vizualizace tahu hráče



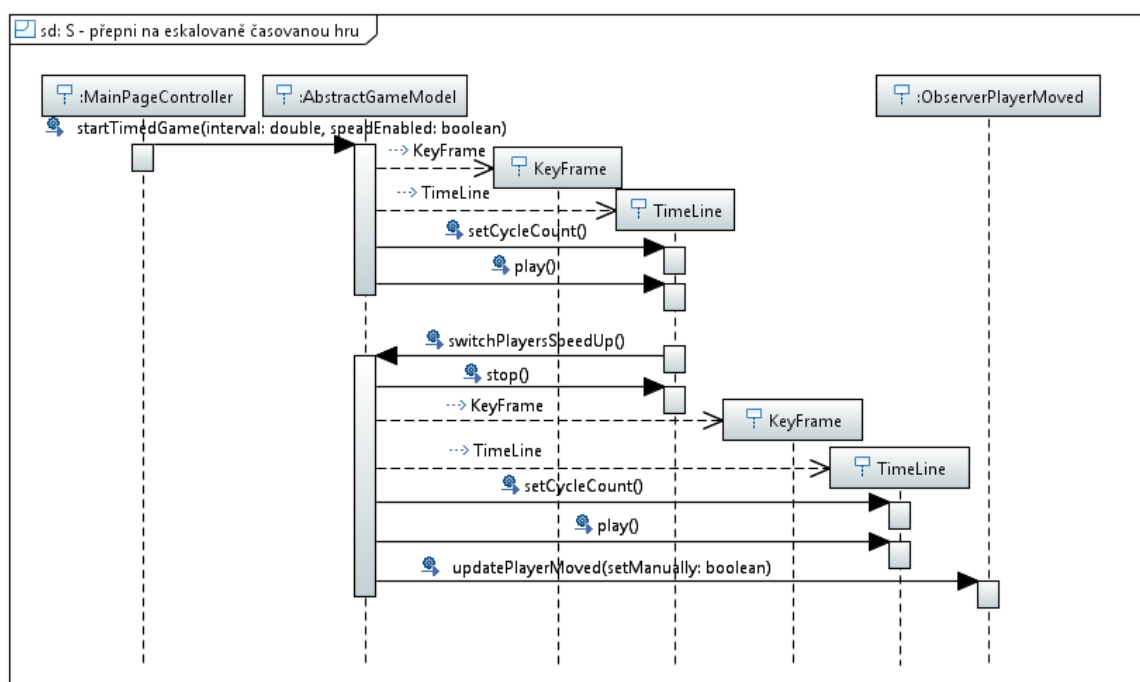
Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 29: Sekvenční diagram – UC Přepni na časovanou hru



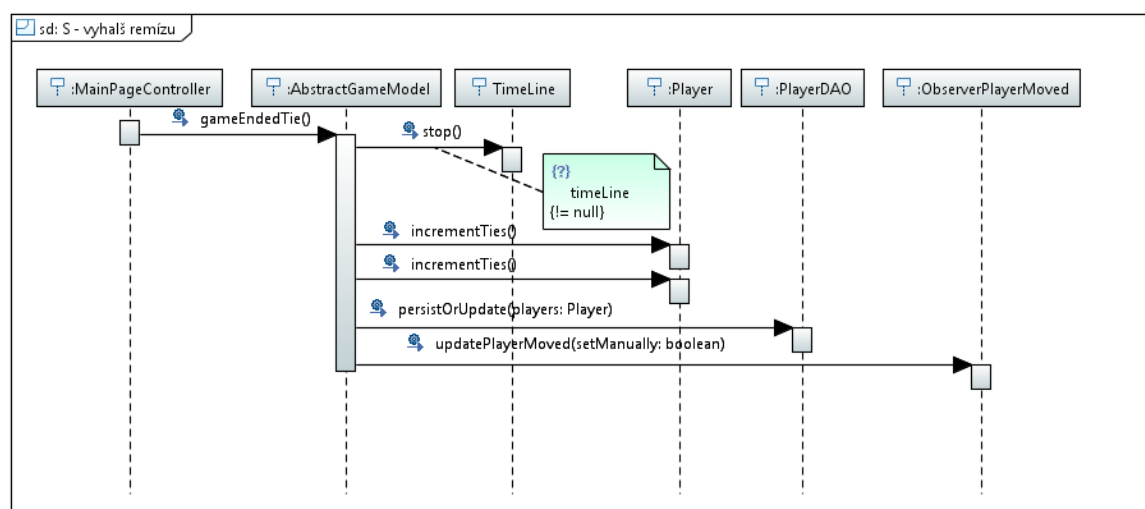
Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 30: Sekvenční diagram – UC Přepni na eskalovaně časovanou hru



Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 31: Sekvenční diagram – UC Vyhlaš remízu



Zdroj: vlastní tvorba (v pluginu Papyrus)

3.2.7 Diagram tříd

Diagram tříd zobrazuje třídy modelu, jejich vlastnosti a vztahy mezi třídami. [15]

V diagramu tříd nejdříve popisují obecné balíčky a balíčky tvořící rozhraní (konkrétně *commons* a *spi*) společně s třídou spouštějící aplikaci. Následně definují obsah balíčku *model* a až nakonec balíček *delegate*.

Nejdříve tedy definují výčet (Enumeration) v balíčku *commons*, který představuje jednotlivé přípustné znaky hráčů, tedy X a O. Viz Obr. 32.

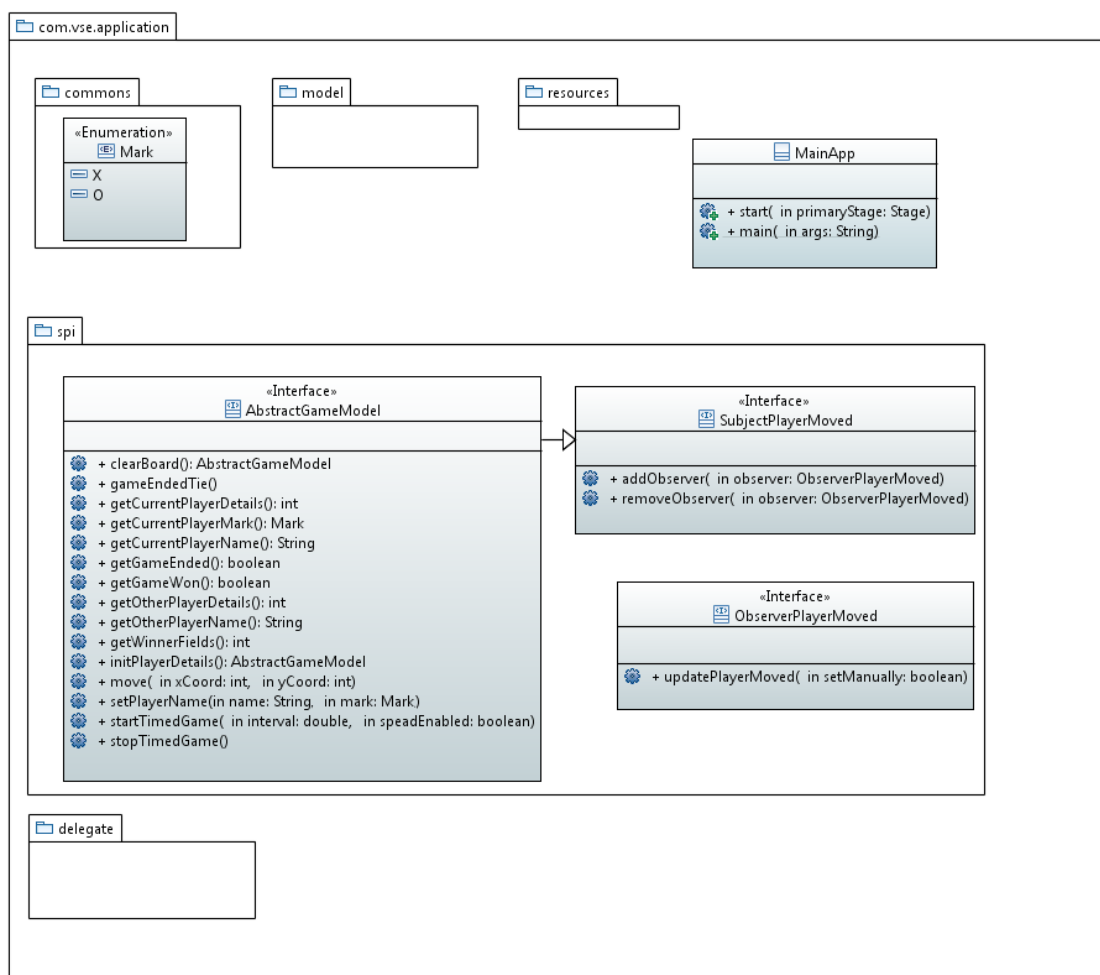
V obecném balíčku *com.vse.application* vytvářím třídu *MainApp*, která obsahuje metody *main* a *start*. Metoda zajišťuje spustitelnost aplikace, inicializuje FX framework, který následně volá metodu *start*, která spouští aplikaci samotnou. Viz Obr. 32.

Balíček *spi* představuje přípustnou komunikaci vrstvy *Delegate* s vrstvou *Model*, v abstraktní rovině. Obsahuje rozhraní *SubjectPlayerMoved* a *ObserverPlayerMoved*, které jsem později použil pro implementaci návrhového vzoru *Observer*, který slouží k upozornění *Controlleru* Modelem, že byl dokončen tah hráče. *SubjectPlayerMoved* tedy definuje metodu *addObserver* a *removeObserver*, aby umožnil registraci a zrušení registrace *Observerům*, které voláním metody *updatePlayerMoved* (definované v *ObserverPlayerMoved*) notifikuje. Viz Obr. 32.

AbstractGameModel představuje rozhraní modelu, s kterým může vrstva *Delegate* komunikovat, zároveň dědí z rozhraní *SubjectPlayerMoved*, tedy zároveň se zde

registrují Observery pro upozornění nově uloženého tahu ve hře. Metoda `clearBoard`, slouží pro vyčištění hracího pole, to je použito pokud po konci hry se hráči rozhodnout hrát znovu. Použití zbylých metod je již patrné ze sekvenčních diagramů (viz kapitola 2.2.6 Sekvenční diagramy) . Viz Obr. 32.

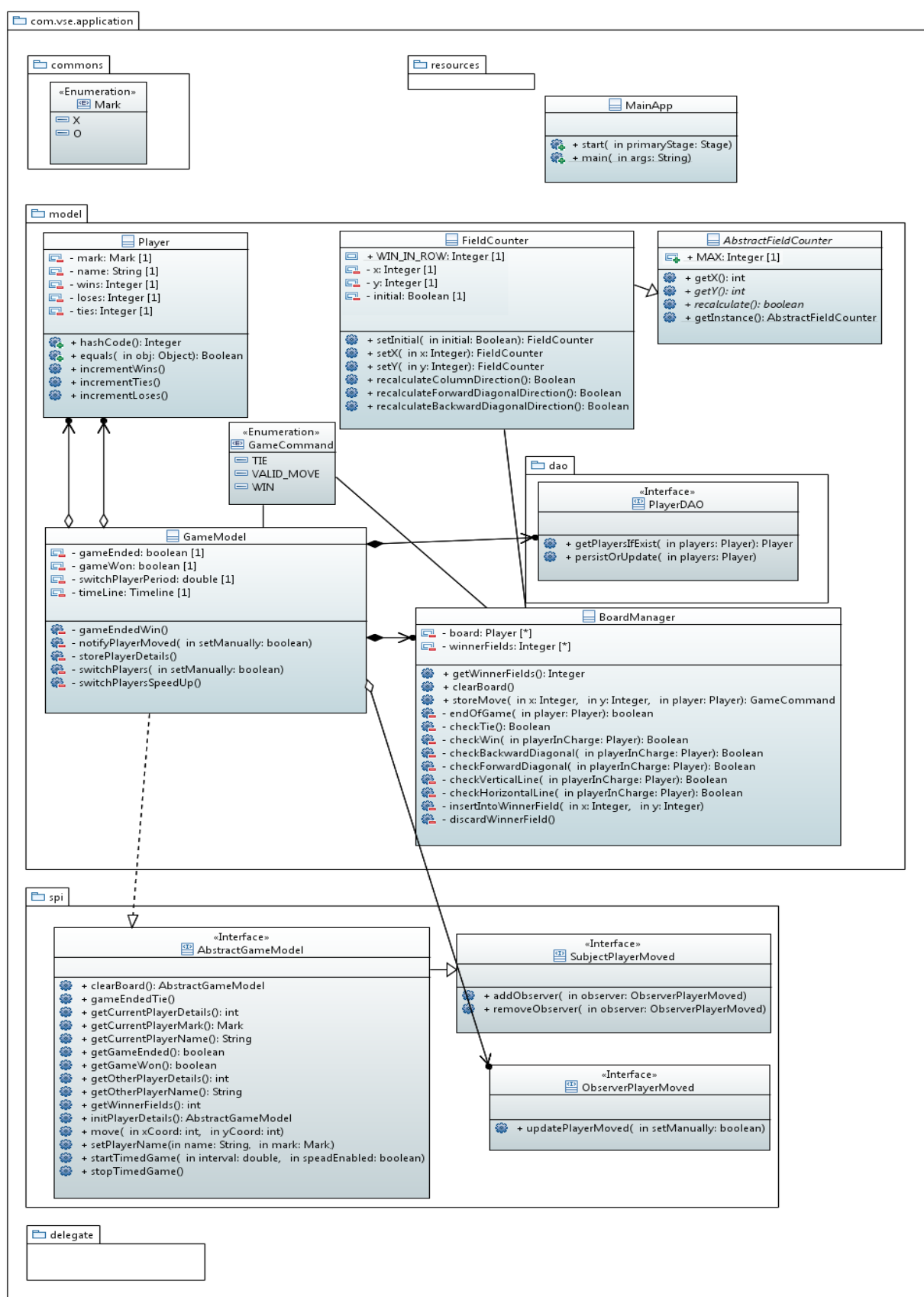
Obr. 32: Diagram tříd *spi* a *commons*



Zdroj: vlastní tvorba (v pluginu Papyrus)

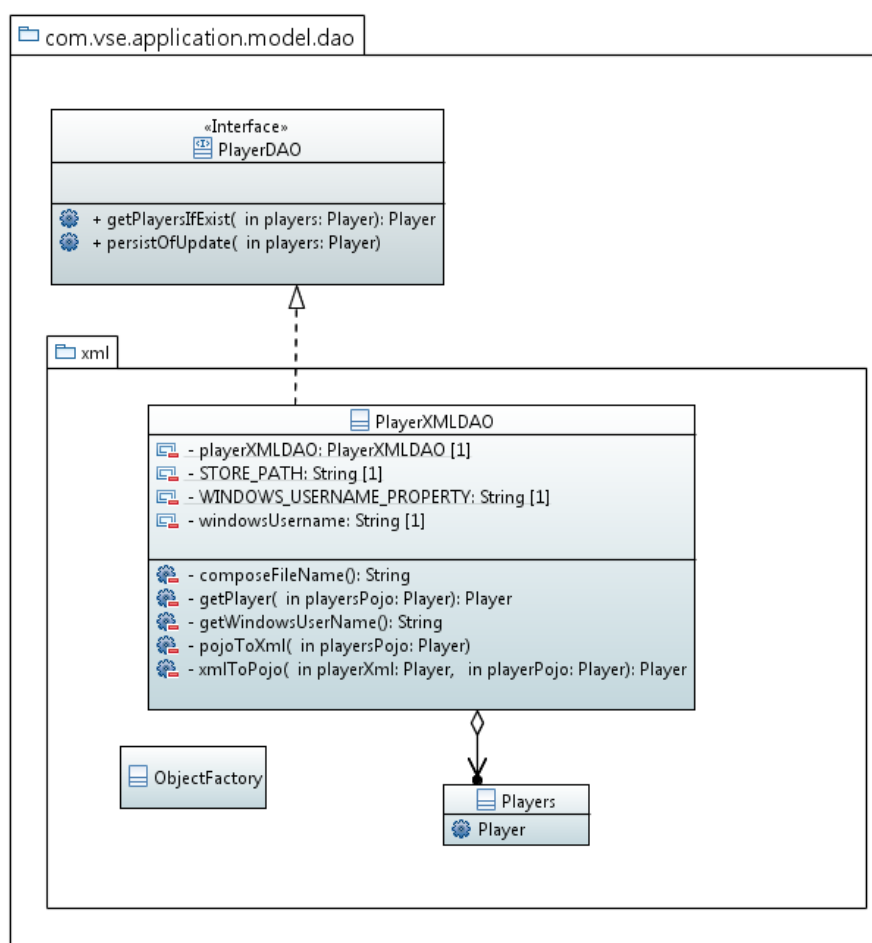
Komunikační cesty mezi komponentami a sdílená data jsou tedy definovány, teď vytvořím Model (viz Obr. 33, kde není, pro přehlednost, zobrazen kompletní balíček „DAO“) a sekci DAO pro ukládání údajů o hráčích (viz Obr. 34).

Obr. 33: Diagram tříd spi, commons a model



Zdroj: vlastní tvorba (v pluginu Papyrus)

Obr. 34: Diagram tříd dao

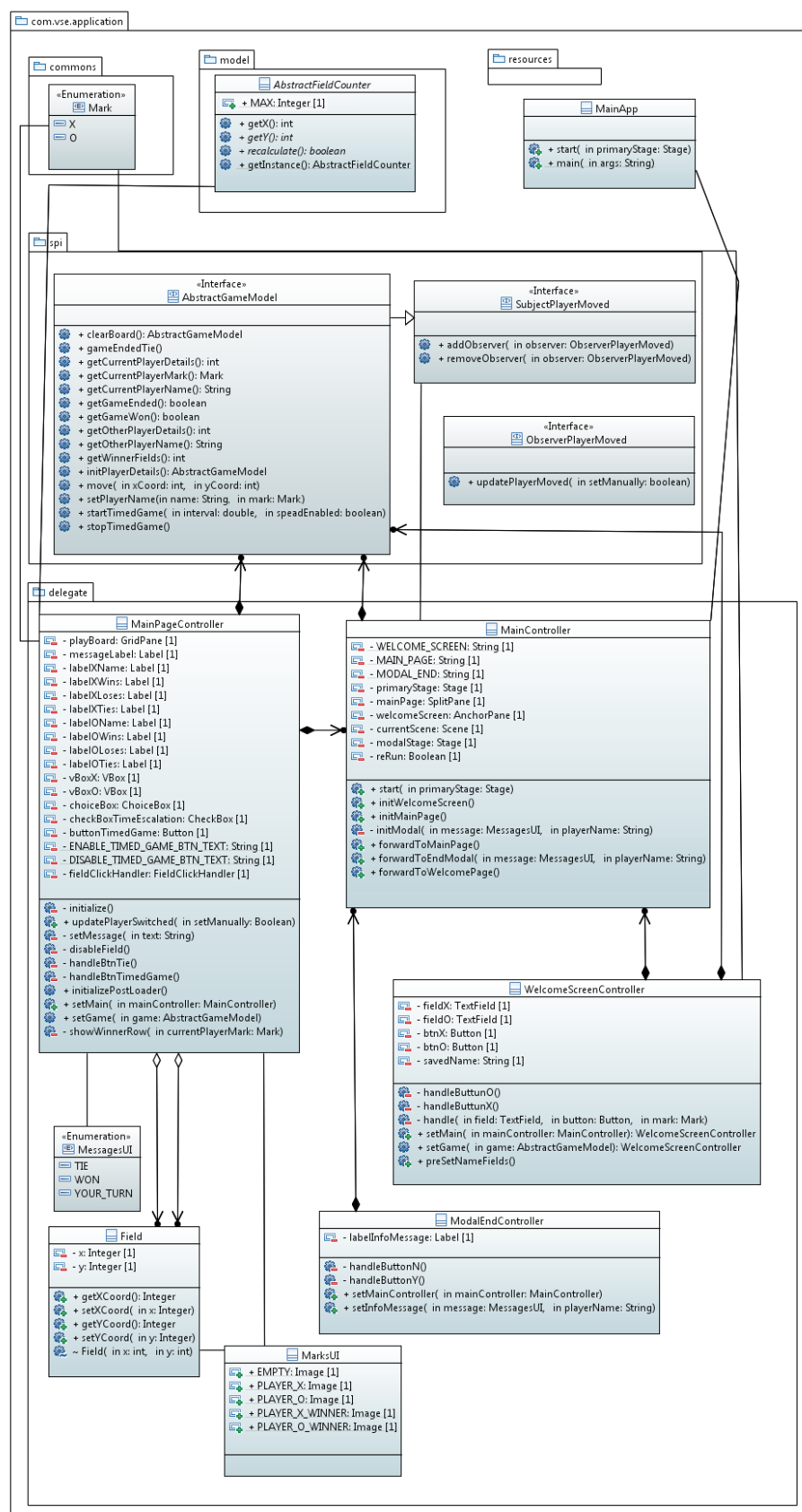


Zdroj: vlastní tvorba (v pluginu Papyrus)

Tímto je hotová část Model, tedy vlastní logika aplikace s uchováváním herních dat o hráčích.

Pokračuji sestavením vrstvy Delegate (viz Obr. 35), definujeme zde pouze Controllery, některé z nich však již obsahují datové vlastnosti odpovídající komponentám ve View, View jako také zde nejsou zobrazeny, protože je jsou později vytvořeny v JavaFX FXML. V class diagramu není, pro přehlednost, zobrazen kompletní balíček *model*.

Obr. 35: Diagram tříd spi, commons, delegate

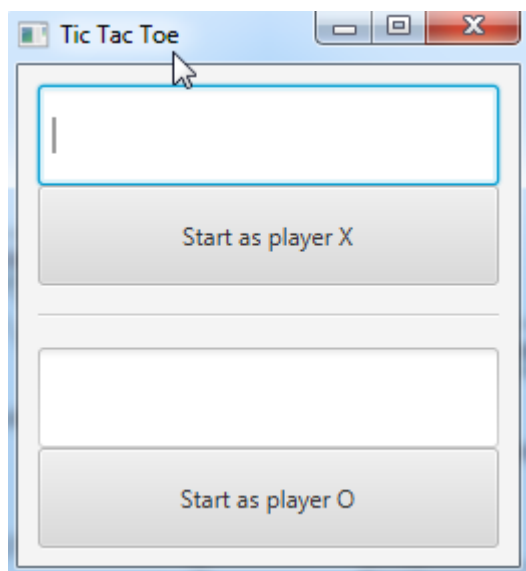


Zdroj: vlastní tvorba (v pluginu Papyrus)

3.3 Návrh uživatelského rozhraní

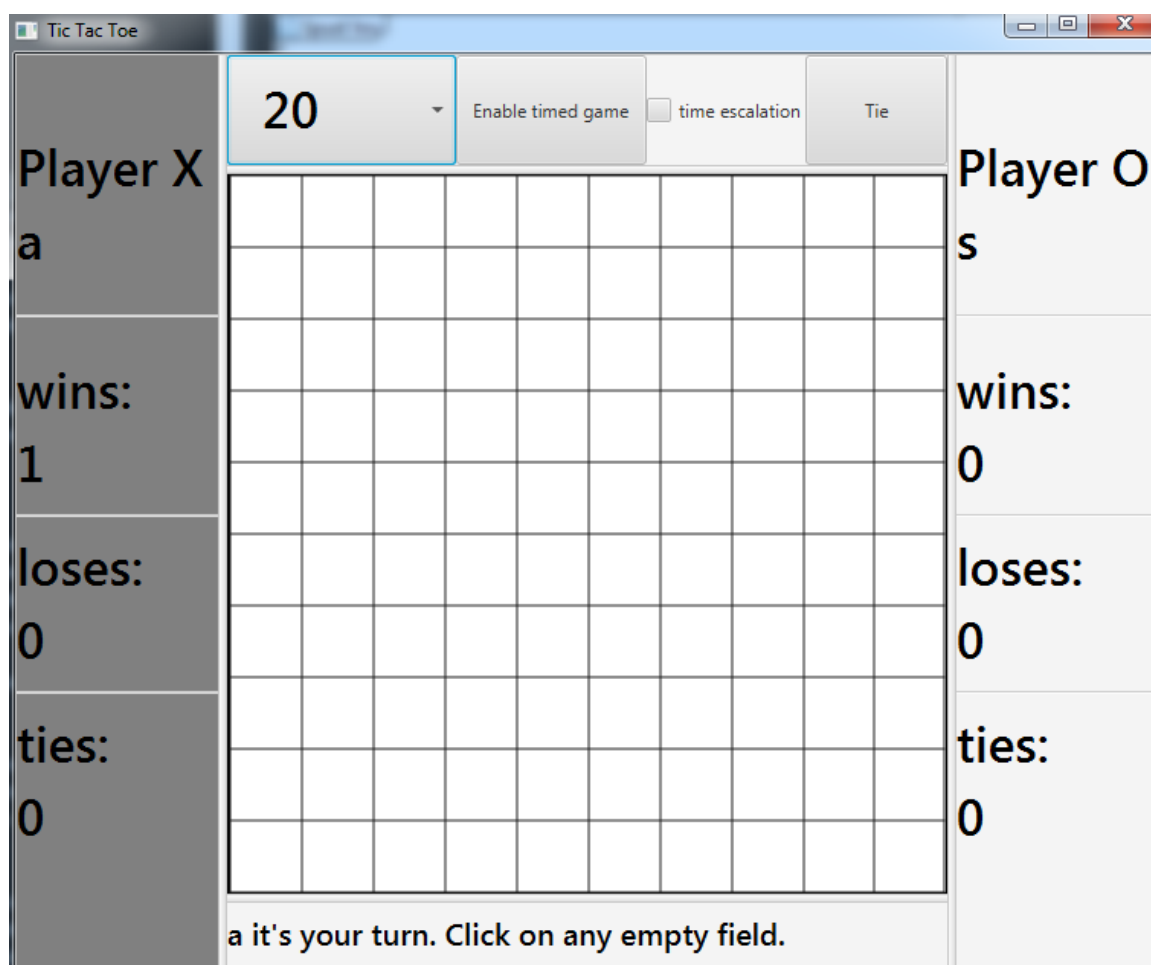
Základní obrazovky aplikace jsou následující: zadání jmen hráče (viz Obr. 36), hlavní hrací obrazovka (viz Obr. 37) a obrazovka při ukončení hry (viz Obr. 38).

Obr. 36: Welcome screen



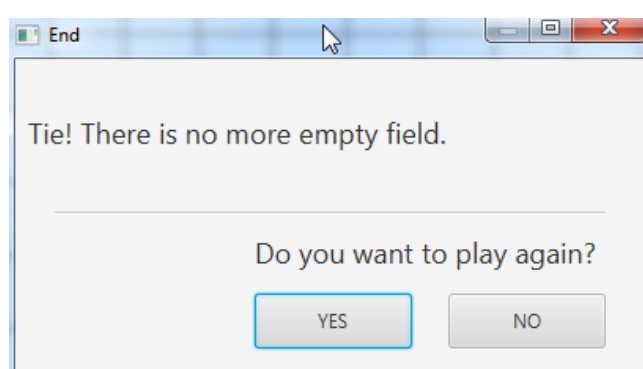
Zdroj: vlastní tvorba

Obr. 37: Main page



Zdroj: vlastní tvorba

Obr. 38: Modal end



Zdroj: vlastní tvorba

3.4 Implementace aplikace

Následující implementaci krok za krokem procházím. Potřebným nástrojem je Java SDK 8, vývojové prostředí Eclipse Mars, JavaSceneBuilder (propojení s Eclipse značně zjednoduší práci), nainstalovaný QVT Operational a Papyrus plugin v Eclipse. Vyjmenované nástroje jsem použil již pro tvorbu podkladů pro kapitulu 2.2 Analytická studie, kde veškeré UML diagramy jsem cíleně tvořil v Papyrus pluginu v Eclipse, teď mohu použít funkci generování Java zdrojového kódu. Čímž jsou vytvořeny třídy, výčty, rozhraní jejich metody, instanční proměnné a statické proměnné a jejich vzájemné vztahy. Zbývá tedy „doimplementovat“ těla metod.

Pro snazší vysvětlení rozdělím konstrukci aplikace dle MVC do čtyřech fází:

- Tvorba komunikačního protokolu – definuje způsob komunikace mezi vrstvami model a delegate užitím rozhraní (interface). Především zde je definováno, jakým způsobem model upozorní vrstvu delegate, že data byla aktualizována. Jaká data může delegate získat z modelu a jakým způsobem. Jaká data může delegate aktualizovat a jakým způsobem.
- Tvorba Modelu – vytvořím model implementující rozhraní, je zde implementace věčné logiky aplikace.
- Tvorba Delegate – následně konstruuji vrstvu delegate (tedy kombinaci View a Controlleru), která obsahuje instance variables typu vytvořeného rozhraní.
- Hlavní třída aplikace.

Architektura aplikace je navržena dle MVC tak, aby tvorba Modelu a Delegate mohla probíhat nezávisle (paralelně).

V následujících podkapitolách krok za krokem rozebírám tvorbu zdrojového kódu. Vždy využívám již vytvořených UML diagramů, ze kterých vygeneruji kostru zdrojového kódu. Po vygenerování vždy automaticky měním vygenerované objekty Boolean na primitivní datový typ boolean, poté není třeba počítat z prázdnou hodnotou null, která by mohla v rámci objektu Boolean přijít. Zároveň, pro udržení přehlednosti, odstraňuji modifikátory přístupu u metod v rozhraních, které nejsou potřeba, jelikož veškeré objekty obsažené v Interface, jsou defaultně označeny modifikátory public a abstract. Ve vyobrazeném zdrojovém kódu také nejsou uvedeny komentáře, či JavaDoc, opět pro dosažení vyšší čitelnosti.

3.4.1 Komunikační protokol (spi a commons)

Tato fáze je vyřešena generací z UML diagramů, až na drobné úpravy. Vygenerován byl výčet `Mark`, obsahující základní znaky hráčů (viz Zdrojový kód 1), rozhraní

AbstractGameModel (viz Zdrojový kód 2), ObserverPlayerMoved (viz Zdrojový kód 3) a SubjectPlayerMoved (viz Zdrojový kód 4).

Zdrojový kód 1: Mark

```
package com.vse.application.common;
```

```
public enum Mark {  
    X,O;  
}
```

Zdroj: vlastní tvorba

Zdrojový kód 2: AbstractGameModel

```
package com.vse.application.spi;
```

```
import java.util.List;
```

```
import com.vse.application.common.Mark;
```

```
public interface AbstractGameModel extends SubjectPlayerMoved {
```

```
    void setPlayerName(Mark mark, String name);
```

```
    void move(int xCoord, int yCoord);
```

```
    String getCurrentPlayerName();
```

```
    Mark getCurrentPlayerMark();
```

```
    String getOtherPlayerName();
```

```
    Integer[] getCurrentPlayerDetails();
```

```
    Integer[] getOtherPlayerDetails();
```

```
    AbstractGameModel clearBoard();
```

```
    AbstractGameModel initPlayerDetails();
```

```
    void gameEndedTie();
```

```
    void startTimedGame(Double interval, boolean speedEnabled);
```

```
    void stopTimedGame();
```

```
    boolean getGameEnded();
```

```
    boolean getGameWon();
```

```
    List<Integer[]> getWinnerFields();
```

```
}
```

Zdroj: vlastní tvorba

Zdrojový kód 3: ObserverPlayerMoved

```
package com.vse.application.spi;

public interface ObserverPlayerMoved {
    void updatePlayerSwitched(boolean setManually);
}
```

Zdroj: vlastní tvorba

Zdrojový kód 4: SubjectPlayerMoved

```
package com.vse.application.spi;

public interface SubjectPlayerMoved {
    void addObserver(ObserverPlayerMoved observer);
    void removeObserver(ObserverPlayerMoved observer);
}
```

Zdroj: vlastní tvorba

3.4.2 Model

GameModel

Hlavní třídou Modelu je `GameModel`. Využívám tzv. „instance variable initializers“ (inicializace instančních proměnných) a naplňuji list `Observerů` (`observersPlayerMoved`) klasickým prázdným array listem (není třeba opakovat generický typ). Vytvářím si instanci hráče na tahu (`currentPlayer`) se znakem X, podobně dalšího hráče (`otherPlayer`) se znakem O. Vytvářím novou a jedinou instanci `BoardManageru` a `PlayerXMLDAO`. Viz Zdrojový kód 5.

Zdrojový kód 5: *GameModel*

```
package com.vse.application.model;

public final class GameModel implements AbstractGameModel {
    private List<ObserverPlayerMoved> observersPlayerMoved = new ArrayList<>();
    private Player currentPlayer = new Player(Mark.X);
    private Player otherPlayer = new Player(Mark.O);
    private BoardManager boardManager = new BoardManager();
    private PlayerDAO playerDao = new PlayerXMLDAO();
    private Timeline timeline;
    private Double switchPlayerPeriod;
    private boolean gameEnded;
    private boolean gameWon;

    @Override
    public void move(int xCoord, int yCoord) {}
    @Override
    public AbstractGameModel initPlayerDetails() { return null;}
    @Override
    public AbstractGameModel clearBoard() { return null;}
    @Override
    public void startTimedGame(Double selectedItem, boolean speedEnabled) {}
    @Override
    public void stopTimedGame() {}
    @Override
    public void setPlayerName(Mark mark, String name) {}
    @Override
    public void gameEndedTie() {}
    @Override
    public String getCurrentPlayerName() { return null;}
    @Override
    public String getOtherPlayerName() { return null;}
    @Override
    public Integer[] getCurrentPlayerDetails() { return null;}
    @Override
    public Integer[] getOtherPlayerDetails() { return null;}
    @Override
    public Mark getCurrentPlayerMark() { return null;}
    @Override
    public void addObserver(ObserverPlayerMoved observer) {}
    @Override
    public void removeObserver(ObserverPlayerMoved observer) {}
    @Override
    public boolean getGameEnded() { return true;}
    @Override
    public boolean getGameWon() { return true;}
    @Override
    public ArrayList<Integer[]> getWinnerFields() { return null;}

    private void switchPlayers(boolean setManually) {}
    private void gameEndedWin() {}
    private void storePlayerDetails() {}
    private void switchPlayersSpeedUp() {}
    private void notifyPlayerMoved(boolean setManually) {}
}
```

Zdroj: vlastní tvorba

V této třídě jsou potřeba následující importy: `Mark` (pro znakovou identifikaci hráčů), `PlayerDAO` (pro ukládání/čtení hráčských detailů do/z perzistentní vrstvy), `PlayerXMLDAO` (reálnou implementaci zajišťující přístup k perzistentním datům – pouze pro naplnění hodnoty pomocí instance `variable initializers`), `AbstractGameModel` (rozhraní), `ObserverPlayerMoved` (rozhraní pro komunikaci s `Observery`), `List` a `ArrayList` (pro implementaci kolekce pro `Observery`) a zbytek importů pro zajištění časování hry. Viz Zdrojový kód 6.

Zdrojový kód 6: *GameModel* – metoda *clearBoard*

```
import com.vse.application.commons.Mark;
import com.vse.application.model.dao.PlayerDAO;
import com.vse.application.model.dao.xml.PlayerXMLDAO;
import com.vse.application.spi.AbstractGameModel;
import com.vse.application.spi.ObserverPlayerMoved;
import java.util.ArrayList;
import java.util.List;
import javafx.animation.Animation;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.util.Duration;
```

Zdroj: vlastní tvorba

Nejdůležitější metodou je `move`, která je vždy zavolána vrstvou `Delegate` při zadání tahu hráčem (viz Obr. 27). Metoda zavolá uložení kroku ve správci hracího plánu, který po uložení tahu vrátí v jakém je hra stavu a podle toho zavolá příslušnou privátní metodu. V případě výhry zavolá metodu `gameEndedWin`, pokud hra skončila remízou – zavolá metodu `gameEndedTie`, a jinak, tedy pokud hra pořád běží, zavolá metodu `switchPlayers`. Viz Zdrojový kód 7.

Zdrojový kód 7: *GameModel* – metoda *move*

```
@Override
public void move(int xCoord, int yCoord) {
    GameCommand command = boardManager.storeMove(xCoord, yCoord,
currentPlayer);
    if (command == GameCommand.WIN) {
        gameEndedWin();
    } else if (command == GameCommand.TIE) {
        gameEndedTie();
    } else {
        switchPlayers(true);
    }
}
```

Zdroj: vlastní tvorba

Metoda `initPlayerDetails` zavolá příslušnou metodu v `PlayerDAO` pro naplnění detailů obou hráčů (počet výher, proher a remíz). Metoda využívá prvky návrhového vzoru `Builder`, což umožňuje zřetězení. Viz Zdrojový kód 8.

Zdrojový kód 8: *GameModel* – metoda *initPlayerDetails*

```

@Override
public AbstractGameModel initPlayerDetails() {
    playerDao.getPlayersIfExists(currentPlayer, otherPlayer);
    return this;
}

```

Zdroj: vlastní tvorba

Metoda `clearBoard` je volána po ukončení a následovném spouštění nové hry (viz Obr. 38). Zajistí připravení hracího pole na další hru. Odstraní registrované observery a zrestartuje flagy do původního nastavení. Viz Zdrojový kód 9.

Zdrojový kód 9: *GameModel* – metoda *clearBoard*

```

@Override
public AbstractGameModel clearBoard() {
    boardManager.clearBoard();
    observersPlayerMoved.clear();
    gameEnded = false;
    gameWon = false;
    return this;
}

```

Zdroj: vlastní tvorba

Pokračuji metodou `startTimedGame`, která zajišťuje hlavní logiku v UC přepni na časovanou hru a UC přepni na eskalovaně časovanou hru (viz Obr. 30). Metoda ukládá interval do instanční proměnné v podobě milisekund, dle kterého vytváří `KeyFrame`. Vytvoří `Timeline`, které předává jako argument vytvořený `KeyFrame` a anonymní třídu spouštějící příslušnou metodu. Pro rozhodnutí jakou metodu spustí je použit tzv. „ternární operátor“ (podmínka ? vyhovělo podmínce : nevyhovělo podmínce) a flag `speedEnabled`. Pro dosažení jednoduchosti jsem pro vytvoření anonymních tříd použil tzv. „lambda výraz“. Po vytvoření, nastavuji `timeLine` nekonečný cyklus a spouštím ji. Viz Zdrojový kód 10.

Zdrojový kód 10: *GameModel* – metoda *startTimedGame*

```

@Override
public void startTimedGame(Double selectedItem, boolean speedEnabled) {
    switchPlayerPeriod = selectedItem * 1000;
    timeline = new Timeline(new
        KeyFrame(Duration.millis(switchPlayerPeriod),
            speedEnabled ? ae -> switchPlayersSpeedUp() : ae ->
            switchPlayers(false)));
    timeline.setCycleCount(Animation.INDEFINITE);
    timeline.play();
}

```

Zdroj: vlastní tvorba

Metoda pro spuštění samostatného vlákna, které periodicky spouští příslušnou metodu, je již hotova. Jelikož hra musí umožňovat časování také vypnout, musím definovat metodu na

zastavení `timeLine` – `stopTimedGame`, která `timeLine` zastaví, pokud vůbec existuje. Viz Zdrojový kód 11.

Zdrojový kód 11: GameModel – metoda stopTimedGame

```
@Override
public void stopTimedGame() {
    if (timeLine != null) {
        timeLine.stop();
    }
}
```

Zdroj: vlastní tvorba

Následující metoda je `setPlayerName`, která nastavuje jména hráčům dle příslušného znaménka. Viz Zdrojový kód 12.

Zdrojový kód 12: GameModel – metoda setPlayerName

```
@Override
public void setPlayerName(Mark mark, String name) {
    if (Mark.X.equals(mark)) {
        currentPlayer.setName(name);
    } else {
        otherPlayer.setName(name);
    }
}
```

Zdroj: vlastní tvorba

Metoda `gameEndedTie` je volána jak interně (z metody `move`, viz Zdrojový kód 7), tak z vrstvy `Delegate` (pokud hráči na main page [viz Obr. 37] stisknou příslušné tlačítko, které spustí UC vyhlás remízu [viz Obr. 31]). Nejdříve volá metodu `stopTimedGame`, náležitě upravuje flag `gameEnded`, připočítá každému hráči jednu remízu, zajišťuje uložení této změny a na konec její zobrazení (předává flag nepravda, naznačující, že zdrojem změny není validní tah hráče). Viz Zdrojový kód 13.

Zdrojový kód 13: GameModel – metoda gameEndedTie

```
@Override
public void gameEndedTie() {
    stopTimedGame();
    gameEnded = true;
    currentPlayer.incrementTies();
    otherPlayer.incrementTies();
    storePlayerDetails();
    notifyPlayerMoved(false);
}
```

Zdroj: vlastní tvorba

Následně implementuji všechny gettery vyžadované rozhraním, které jsou používány vrstvou `Delegate`. Viz Zdrojový kód 14.

Zdrojový kód 14: *GameModel* – ostatní gettersy

```

@Override
public String getCurrentPlayerName() {
    return currentPlayer.getName();
}

@Override
public String getOtherPlayerName() {
    return otherPlayer.getName();
}

@Override
public Integer[] getCurrentPlayerDetails() {
    Integer[] details = { currentPlayer.getWins(),
        currentPlayer.getTies(), currentPlayer.getLoses() };
    return details;
}

@Override
public Integer[] getOtherPlayerDetails() {
    Integer[] details = { otherPlayer.getWins(),
        otherPlayer.getTies(), otherPlayer.getLoses() };
    return details;
}

@Override
public Mark getCurrentPlayerMark() {
    return currentPlayer.getMark();
}

@Override
public boolean getGameEnded() {
    return gameEnded;
}

@Override
public boolean getGameWon() {
    return gameWon;
}

@Override
public ArrayList<Integer[]> getWinnerFields() {
    return boardManager.getWinnerFields();
}

```

Zdroj: vlastní tvorba

Ještě musím implementovat metody definované rozhraním *SubjectPlayerMoved*, jehož metody podědilo rozhraní *AbstractGameModel*. Konkrétně metody na přidání a odebrání *Observeru*, který je upozorňován na provedený tah (viz Zdrojový kód 16). Viz Zdrojový kód 15.

Zdrojový kód 15: *GameModel* – metody *Observer*

```
@Override
public void addObserver(ObserverPlayerMoved observer) {
    observersPlayerMoved.add(observer);
}

@Override
public void removeObserver(ObserverPlayerMoved observer) {
    observersPlayerMoved.remove(observer);
}
```

Zdroj: vlastní tvorba

Na závěr metody soukromé. Viz Zdrojový kód 16.

Zdrojový kód 16: *GameModel* – soukromé metody

```
private void switchPlayers(boolean setManually) {
    notifyPlayerMoved(setManually);
    Player temporaryPlayer = currentPlayer;
    currentPlayer = otherPlayer;
    otherPlayer = temporaryPlayer;
}

private void gameEndedWin() {
    stopTimedGame();
    gameEnded = true;
    gameWon = true;
    currentPlayer.incrementWins();
    otherPlayer.incrementLoses();
    storePlayerDetails();
    notifyPlayerMoved(true);
}

private void storePlayerDetails() {
    Player[] players = { currentPlayer, otherPlayer };
    playerDao.persistOrUpdate(players);
}

private void switchPlayersSpeedUp() {
    if (Double.compare(switchPlayerPeriod, new Double(850)) > 0) {
        timeline.stop();
        timeline = new Timeline(new
            KeyFrame(Duration.millis(switchPlayerPeriod =
                switchPlayerPeriod * 0.95),
                ae -> switchPlayersSpeedUp()));
        timeline.setCycleCount(Animation.INDEFINITE);
    }
    timeline.play();
    switchPlayers(false);
}

private void notifyPlayerMoved(boolean setManually) {
    for (ObserverPlayerMoved observer : observersPlayerMoved) {
        observer.updatePlayerSwitched(setManually);
    }
}
```

Zdroj: vlastní tvorba

BoardManager

Třída `BoardManager` představuje správce plátna. První instanční proměnou, je ve skutečnosti dvojrozměrné pole hráčů o velikosti 10x10 (konstanta `max` má hodnotu 9, tedy je třeba přičíst jedničku, aby to dalo výsledných deset). Dále zde je instanční proměnná v podobě listu obsahující pole čísel – `winnerFields`, sem se dočasně ukládají jednotlivá políčka pro případ, že by řada byla nakonec vyhodnocena jako vítězná, pak by se toto pole předalo vrstvě `Delegate` pro zobrazení. Jedná se o třídu pouze pro vnitřní použití v `Modelu`, jak již napovídá přístupový modifikátor, který není uveden na třídě, ta tedy může být adresovaná pouze ze stejného balíčku. Viz Zdrojový kód 17.

Zdrojový kód 17: BoardManager

```
package com.vse.application.model;

import java.util.ArrayList;

final class BoardManager {

    private Player[][] board = new Player[FieldCounter.MAX+1][FieldCounter.MAX+1];
    private ArrayList<Integer[]> winnerFields = new ArrayList<Integer[]>();

    void clearBoard() {}
    GameCommand storeMove(int x, int y, Player player) { return null;}
    ArrayList<Integer[]> getWinnerFields() { return null;}
    private GameCommand endOfGame(Player player) { return null;}
    private boolean checkTie() { return true;}
    private boolean checkWin(Player playerInCharge) { return true;}
    private boolean checkBackwardDiagonal(Player playerInCharge) { return true;}
    private boolean checkForwardDiagonal(Player playerInCharge) { return true;}
    private boolean checkVerticalLine(Player playerInCharge) { return true;}
    private boolean checkHorizontalLine(Player playerInCharge) { return true;}
    private void insertIntoWinnerField(int x, int y) {}
    private void discardWinnerField() {}
}
```

Zdroj: vlastní tvorba

Metoda `clearBoard` znovu inicializuje dvojrozměrné instanční pole hráčů (představující hrací plánec), je použita třídou `GameModel`, pro vyčištění hracího plánu (viz Zdrojový kód 9). Viz Zdrojový kód 18.

Zdrojový kód 18: BoardManager – metoda clearBoard

```
void clearBoard() {
    board = new Player[FieldCounter.MAX + 1][FieldCounter.MAX + 1];
}
```

Zdroj: vlastní tvorba

Metoda `storeMove`, jak již název napovídá, ukládá tah hráče do hracího plánu. Předtím však kontroluje zda se nesnaží uložit hodnotu mimo rozměry hracího plánu, a případně vyhazuje `unchecked` (nehlídanou) výjimku – při implementaci nám to usnadní hledání případných chyb. Následně volá kontrolu stavu hry a vrací v jakém stavu se hra nachází (viz Zdrojový kód 21). Viz Zdrojový kód 19.

Zdrojový kód 19: BoardManager – metoda storeMove

```
GameCommand storeMove(int x, int y, Player player) {
    if (x > FieldCounter.MAX || x < 0 || y > FieldCounter.MAX || y < 0) {
        throw new Error("An attempt to access field out of bound of play
        board!");
    }
    board[x][y] = player;
    return endOfGame(player);
}
```

Zdroj: vlastní tvorba

Následuje jednoduchý getter na výherní pole, který je volaný GameModelem (viz Zdrojový kód 14). Viz Zdrojový kód 20.

Zdrojový kód 20: BoardManager –getter

```
ArrayList<Integer[]> getWinnerFields() {
    return winnerFields;
}
```

Zdroj: vlastní tvorba

Privátní metoda endOfGame, volá kontrolu zda hráč nevyhrál či zda nevznikla remíza.

Zdrojový kód 21: BoardManager –privátní metoda endOfGame

```
private GameCommand endOfGame(Player player) {
    boolean won = checkWin(player);
    boolean tie = checkTie();
    return won == true ? GameCommand.WIN : tie == true ? GameCommand.TIE :
    GameCommand.VALID_MOVE;
}
```

Zdroj: vlastní tvorba

Následující metody kontrolují zda nejsou všechna pole obsazena (checkTie), či zda hráč tímto tahem nevyhrál (checkWin), kde se postupně herní pláněk kontroluje zda neexistuje horizontální, poté vertikální pěťice, nebo pěťice v podobě dopředného lomítka či zpětného lomítka (viz Zdrojový kód 23-26). Viz Zdrojový kód 22.

Zdrojový kód 22: BoardManager –privátní metoda checkWin a checkTie

```
private boolean checkTie() {
    int count = 0;
    FieldCounter fieldCounter = new FieldCounter();
    while (fieldCounter.recalculate()) {
        if (count == FieldCounter.MAX * FieldCounter.MAX) {
            return true;
        }
        Player player = board[fieldCounter.getX()][fieldCounter.getY()];
        if (player != null) {
            count++;
        } else {
            return false;
        }
    }
    return false;
}

private boolean checkWin(Player playerInCharge) {
    if (checkHorizontalLine(playerInCharge))
        return true;

    if (checkVerticalLine(playerInCharge))
        return true;

    if (checkForwardDiagonal(playerInCharge))
        return true;

    if (checkBackwardDiagonal(playerInCharge))
        return true;

    return false;
}
```

Zdroj: vlastní tvorba

Zdrojový kód 23: BoardManager –privátní metoda checkBackwardDiagonal

```

private boolean checkBackwardDiagonal(Player playerInCharge) {
    FieldCounter fieldCounter;
    int count;
    for (int pos = 0; pos <= FieldCounter.MAX; pos++) {
        if (pos >= FieldCounter.WIN_IN_ROW - 1) {
            count = 0;
            fieldCounter = new FieldCounter().setX(0).setY(pos);
            while(fieldCounter.recalculateBackwardDiagonalDirection())
            {
                Player player =
                    board[fieldCounter.getX()][fieldCounter.getY()];
                if (player != null && player.equals(playerInCharge))
                {
                    count++;
                    insertIntoWinnerField(fieldCounter.getX(),
                        fieldCounter.getY());
                } else {
                    count = 0;
                    discardWinnerField();
                }
                if (count == FieldCounter.WIN_IN_ROW) {
                    return true;
                }
            }
        }
        if (pos <= FieldCounter.WIN_IN_ROW - 1) {
            count = 0;
            fieldCounter = new
                FieldCounter().setX(pos).setY(FieldCounter.MAX);
            while(fieldCounter.recalculateBackwardDiagonalDirection())
            {
                Player player =
                    board[fieldCounter.getX()][fieldCounter.getY()];
                if (player != null && player.equals(playerInCharge))
                {
                    count++;
                    insertIntoWinnerField(fieldCounter.getX(),
                        fieldCounter.getY());
                } else {
                    count = 0;
                    discardWinnerField();
                }
                if (count == FieldCounter.WIN_IN_ROW) {
                    return true;
                }
            }
        }
    }
    return false;
}

```

Zdroj: vlastní tvorba

Zdrojový kód 24: BoardManager –privátní metoda checkForwardDiagonal

```

private boolean checkForwardDiagonal(Player playerInCharge) {
    FieldCounter fieldCounter;
    int count;
    final int border = (FieldCounter.MAX - (FieldCounter.WIN_IN_ROW - 1));
    for (int pos = 0; pos <= border; pos++) {
        if (pos > 0) {
            count = 0;
            fieldCounter = new FieldCounter().setX(0).setY(pos);
            while (fieldCounter.recalculateForwardDiagonalDirection()) {
                Player player =
                    board[fieldCounter.getX()][fieldCounter.getY()];
                if (player != null && player.equals(playerInCharge)) {
                    count++;
                    insertIntoWinnerField(fieldCounter.getX(),
                                            fieldCounter.getY());
                } else {
                    count = 0;
                    discardWinnerField();
                }
                if (count == FieldCounter.WIN_IN_ROW) {
                    return true;
                }
            }
        }
        count = 0;
        fieldCounter = new FieldCounter().setX(pos).setY(0);
        while (fieldCounter.recalculateForwardDiagonalDirection()) {
            Player player =
                board[fieldCounter.getX()][fieldCounter.getY()];
            if (player != null && player.equals(playerInCharge)) {
                count++;
                insertIntoWinnerField(fieldCounter.getX(),
                                        fieldCounter.getY());
            } else {
                count = 0;
                discardWinnerField();
            }
            if (count == FieldCounter.WIN_IN_ROW) {
                return true;
            }
        }
    }
    return false;
}

```

Zdroj: vlastní tvorba

Zdrojový kód 25: BoardManager –privátní metoda checkVerticalLine

```
private boolean checkVerticalLine(Player playerInCharge) {
    FieldCounter fieldCounter = new FieldCounter();
    int count = 0;
    while (fieldCounter.recalculateColumnDirection()) {
        if (fieldCounter.getY() == 0) {
            count = 0;
        }
        Player player = board[fieldCounter.getX()][fieldCounter.getY()];
        if (player != null && player.equals(playerInCharge)) {
            count++;
            insertIntoWinnerField(fieldCounter.getX(),
                                fieldCounter.getY());
        } else {
            count = 0;
            discardWinnerField();
        }
        if (count == FieldCounter.WIN_IN_ROW) {
            return true;
        }
    }
    return false;
}
```

Zdroj: vlastní tvorba

Zdrojový kód 26: BoardManager –privátní metoda checkHorizontalLine

```
private boolean checkHorizontalLine(Player playerInCharge) {
    FieldCounter fieldCounter = new FieldCounter();
    int count = 0;
    while (fieldCounter.recalculate()) {
        if (fieldCounter.getX() == 0) {
            count = 0;
        }
        Player player = board[fieldCounter.getX()][fieldCounter.getY()];
        if (player != null && player.equals(playerInCharge)) {
            count++;
            insertIntoWinnerField(fieldCounter.getX(),
                                fieldCounter.getY());
        } else {
            count = 0;
            discardWinnerField();
        }
        if (count == FieldCounter.WIN_IN_ROW) {
            return true;
        }
    }
    return false;
}
```

Zdroj: vlastní tvorba

Zbylé soukromé metody pro vkládání do instančního pole vítězných herních políček a vyčištění tohoto pole, jsou zobrazeny ve Zdrojovém kódu 27.

Zdrojový kód 27: BoardManager –zbylé privátní metody

```
private void insertIntoWinnerField(int x, int y) {
    Integer[] field = { x, y };
    winnerFields.add(field);
}

private void discardWinnerField() {
    winnerFields = new ArrayList<Integer[]>();
}
```

Zdroj: vlastní tvorba

FieldCounter

Třída `FieldCounter` představuje iterátor, který pomocí příslušných metod virtuálně prochází správným způsobem hrací pole a vždy vrátí souřadnici X a Y. Je to třída jak pro interní použití v Modelu, tak i v omezeném měřítku (definovaném pomocí `AbstractFieldCounter` – abstraktní třídy) pro vrstvu Delegate, která counter používá pro vytvoření hracího pole ve vizuálním prostředí. Obsahuje konstantu s počtem výherních políček v řadě. Soukromé instanční proměnné `x` a `y` představují souřadnice, které jsou pomocí příslušných metod použity. Instanční proměnná `initial` určuje zda je instance `FieldCounteru` čerstvě vytvořená, nebo na ní již došlo k přepočtení souřadnic. Ještě zde definuji jednu konstantu, přístupnou pouze v rámci balíčku Model, a to počet výherních polí. Viz Zdrojový kód 28.

Zdrojový kód 28: `FieldCounter`

```
package com.vse.application.model;

public final class FieldCounter extends AbstractFieldCounter {
    static final int WIN_IN_ROW = 5;
    private int x = 0;
    private int y = 0;
    private boolean initial = true;

    FieldCounter() {}
    @Override
    public boolean recalculate() { return true;}
    @Override
    public int getX() { return 0;}
    @Override
    public int getY() { return 0;}
    FieldCounter setInitial(boolean initial) { return null;}
    boolean recalculateColumnDirection() { return true;}
    boolean recalculateForwardDiagonalDirection() { return true;}
    boolean recalculateBackwardDiagonalDirection() { return true;}
    FieldCounter setX(int x) { return null;}
    FieldCounter setY(int y) { return null;}
}
```

Zdroj: vlastní tvorba

Nejdříve začínám implementací konstrukturu metod definovaných v abstraktním předkovi. Jediný důvod proč explicitně definuji konstruktor, a nenechávám to tak na bedrech

kompilátoru, je snížení viditelnosti, tak aby vrstva delegate neměla možnost si přímo vytvořit instanci `FieldCounter` a tak přistupovat ke všem metodám. Metoda `recalculate` postupně „prochází hrací plánec“ ve směru řádků, následují dva gettery pro instanční proměnné typu `int`. Viz Zdrojový kód 29.

Zdrojový kód 29: FieldCounter – povinné metody

```
FieldCounter() {
}

@Override
public boolean recalculate() {
    if (!initial) {
        if (x == MAX) {
            if (y == MAX) {
                return false;
            } else {
                x = 0;
                y++;
                return true;
            }
        } else {
            x++;
            return true;
        }
    } else {
        initial = false;
        return true;
    }
}

@Override
public int getX() {
    return x;
}

@Override
public int getY() {
    return y;
}
```

Zdroj: vlastní tvorba

Následně implementuji metody pro vnitřní použití v balíčku `Model`, a to metody přepočítávající instanční proměnné typu `int` – tedy souřadnice hracího políčka, ve směru řádků, dopředného a zpětného lomítka. Viz Zdrojový kód 30.

Zdrojový kód 30: *FieldCounter* –package metody

```

    boolean recalculateColumnDirection() {
        if (!initial) {
            if (y == MAX) {
                if (x == MAX) {
                    return false;
                } else {
                    y = 0;
                    x++;
                    return true;
                }
            } else {
                y++;
                return true;
            }
        } else {
            initial = false;
            return true;
        }
    }

    boolean recalculateForwardDiagonalDirection() {
        if (!initial) {
            if (y == MAX || x == MAX) {
                return false;
            } else {
                y++;
                x++;
                return true;
            }
        } else {
            initial = false;
            return true;
        }
    }

    boolean recalculateBackwardDiagonalDirection() {
        if (!initial) {
            if (y == 0 || x == MAX) {
                return false;
            } else {
                y--;
                x++;
                return true;
            }
        } else {
            initial = false;
            return true;
        }
    }
}

```

Zdroj: vlastní tvorba

Na závěr této třídy implementuji settery pro instanční proměnné typu int, které opět vrací instanci této třídy, aby se jejich volání dalo zřetěžit. Viz Zdrojový kód 31.

Zdrojový kód 31: *FieldCounter* –package settery

```
FieldCounter setX(int x) {
    this.x = x;
    return this;
}

FieldCounter setY(int y) {
    this.y = y;
    return this;
}
```

Zdroj: vlastní tvorba

AbstractFieldCounter

Třída *AbstractFieldCounter* je abstraktním předkem třídy *FieldCounter* a mimo definice abstraktních metod, které může vrstva *Delegate* použít, definuje také metodu pro získání instance *FieldCounteru* v abstraktní rovině, tedy jako *AbstractFieldCounter*. Dále obsahuje konstantu *MAX* představující rozměr pole, počínaje nulou (číslo 9 tedy představuje hrací pole o velikosti 10x10). Viz Zdrojový kód 32.

Zdrojový kód 32: *AbstractFieldCounter*

```
package com.vse.application.model;

public abstract class AbstractFieldCounter {
    public static final int MAX = 9;

    public abstract int getY();
    public abstract int getX();
    public abstract boolean recalculate();

    public static AbstractFieldCounter getInstance() {
        return new FieldCounter();
    }
}
```

Zdroj: vlastní tvorba

Player

Třída *Player* představuje v podstatě pojo (Plain Old Java Object), tedy jednoduchou třídu obsahující instanční proměnné a k nim gettery a settery. V tomto případě však nejsou použity všechny settery, protože znaménko hráče (*mark*), nelze u hráče změnit, znaménko hráč získá při vytvoření instance. A ještě rozšiřuji třídu o tři jednoduché metody, na zvýšení počtu výher, proher a remíz (viz Zdrojový kód 34). Viz Zdrojový kód 33.

Zdrojový kód 33: *Player*

```
package com.vse.application.model;
import com.vse.application.commons.Mark;

public class Player {
    private Mark mark;
    private String name;
    private int wins;
    private int loses;
    private int ties;

    public Player(Mark mark) {
        this.mark = mark;
    }

    @Override
    public int hashCode() {
        return (mark == null) ? 0 : mark.hashCode();
    }

    @Override
    public boolean equals(Object obj) {
        if (obj instanceof Player) {
            return ((Player) obj).mark.equals(this.mark);
        } else {
            return false;
        }
    }

    public Mark getMark() {
        return mark;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getWins() {
        return wins;
    }

    public void setWins(int wins) {
        this.wins = wins;
    }

    public int getLoses() {
        return loses;
    }

    public void setLoses(int loses) {
        this.loses = loses;
    }

    public int getTies() {
        return ties;
    }

    public void setTies(int ties) {
        this.ties = ties;
    }
}
```

Zdroj: vlastní tvorba

Zdrojový kód 34: *Player* – rozšiřující metody

```

    void incrementWins() {
        wins++;
    }

    void incrementTies() {
        ties++;
    }

    void incrementLoses() {
        loses++;
    }

```

Zdroj: vlastní tvorba

GameCommand

`GameCommand` je jednoduchý výčet, který je již vygenerován z UML diagramů. Obsahuje tři konstanty, představující stav, v jakém se hra nachází. Hra byla vyhrána, došlo k remíze, či hra pokračuje (`VALID_MOVE`). Viz Zdrojový kód 35.

Zdrojový kód 35: *GameCommand*

```

package com.vse.application.model;

enum GameCommand {
    WIN, TIE, VALID_MOVE;
}

```

Zdroj: vlastní tvorba

dao.PlayerDAO

`PlayerDAO` představuje rozhraní, poslední tři písmenka značí, že se jedná o jednu z částí návrhového vzoru DAO (Data Access Object), který poskytuje abstrakci nad perzistentní vrstvou. Potažmo je tedy aplikaci jedno, kde jsou data uloženy (v databázi, xml,..). Rozhraní definuje metodu pro získání pole hráčů, obsahující oba dva hráče, z perzistentní vrstvy, tedy pokud hráči mají nějaké detaily uloženy, pak jsou touto metodou načteny do jednotlivých entit hráčů, které jsou v poli vráceny. Ještě definuje jednu metodu, která slouží k uložení, případně aktualizace hráče v perzistentní vrstvě. Viz Zdrojový kód 36.

Zdrojový kód 36: *PlayerDAO*

```

package com.vse.application.model.dao;

import com.vse.application.model.Player;

public interface PlayerDAO {
    Player[] getPlayersIfExist(Player... players);

    void persistOrUpdate(Player... players);
}

```

Zdroj: vlastní tvorba

3.4.3 Model – dao – xml

Implementace tohoto balíčku může opět probíhat nezávislá na ostatních, jediné co musí být před implementací hotovo je rozhraní `PlayerDAO` (viz Zdrojový kód 36).

PlayerXMLDAO

Třída `PlayerXMLDAO` je implementací rozhraní `PlayerDAO` a další částí návrhového vzoru DAO. Prvních pět importů využívá třída pro ukládání a získávání hodnoty z/do souboru v podobě xml. Následně využívá třídu `Player` z balíčku *model* a samozřejmě rozhraní `PlayerDAO`, které implementuje. Třída obsahuje tři soukromé konstanty, první obsahuje cestu k souboru na disku (pro ukládání detailů hráčů), do které je dynamicky doplněn Windows uživatel (struktura uložení souboru tedy odpovídá rozložení Windows) získáním příslušné hodnoty ze systému, definované druhou konstantou. Poslední konstanta představuje instanci této třídy, a spolu se statickým inicializačním blokem (který se provádí při načítání třídy class loaderem) a metodou `getInstance` představují jednu z možných implementací návrhového vzoru Singleton. Třída `PlayerXMLDAO` je tedy jedináčkem, pro každý class loader existuje právě jedna instance této třídy. Viz Zdrojový kód 37.

Zdrojový kód 37: *PlayerXMLDAO*

```
package com.vse.application.model.dao.xml;

import java.io.File;
import javax.xml.bind.JAXBContext;
import javax.xml.bind.JAXBException;
import javax.xml.bind.Marshaller;
import javax.xml.bind.Unmarshaller;
import com.vse.application.model.Player;
import com.vse.application.model.dao.PlayerDAO;

public class PlayerXMLDAO implements PlayerDAO {

    private static final String STORE_PATH =
        "C:\\Users\\%s\\AppData\\Local\\Temp\\playersTicTacToe.xml";
    private static final String WINDOWS_USERNAME_PROPERTY = "user.name";
    private static final PlayerXMLDAO playerXMLDAO;

    private String windowsUsername;
    private Players players;

    static{
        playerXMLDAO = new PlayerXMLDAO();
    }

    public static PlayerXMLDAO getInstance(){
        return playerXMLDAO;
    }

    @Override
    public Player[] getPlayersIfExist(Player... player) { return null;}
    @Override
    public void persistOrUpdate(Player... player) {}
    private String composeFileName() { return null;}
    private String getWindowsUserName(){return null;}
    private Player[] getPlayer(Player[] playersPojo){ return null;}
    private void pojoToXml(Player[] playersPojo){}
    private Player xmlToPojo(Players.Player playerXml, Player playerPojo){
        return null;}
}
```

Zdroj: vlastní tvorba

Metoda `getPlayersIfExist`, nejdříve volá vytvoření cesty k souboru, pomocí kterého pak soubor získá. A poté pokud získaný soubor existuje a zároveň není složkou (tedy je souborem), pak získává uložená data pomocí JAXB frameworku (ten využívá třídu `Players` a `ObjectFactory`). Na závěr pokud byli načtení nějakí hráči, tak volá metodu pro převedení relevantních `Players` (třída reprezentující XML úložiště) na `Player` (rozšířené pojo) a výsledek vrací. Viz Zdrojový kód 38.

Zdrojový kód 38: *PlayerXMLDAO – metoda getPlayersIfExist*

```
@Override
public Player[] getPlayersIfExist(Player... player) {
    File file = new File(composeFileName());
    if(file.exists() && !file.isDirectory()) {
        try {
            JAXBContext jaxbContext =
                JAXBContext.newInstance(Players.class);
            Unmarshaller jaxbUnmarshaller =
                jaxbContext.createUnmarshaller();
            players = (Players) jaxbUnmarshaller.unmarshal(file);
            if(player!=null){
                return getPlayer(player);
            }
        } catch (JAXBException e) {
            //do nothing, just continue and null will be returned
        }
    }
    return null;
}
```

Zdroj: vlastní tvorba

Metoda `persistOrUpdate`, podobně jako předchozí metoda, nejdříve volá vytvoření cesty k souboru, pomocí kterého pak soubor získává, případně vytváří pokud neexistuje. Následně převádí instance `Player` a `Players`, které reprezentují strukturu XML uložení, a předává je frameworku JAXB k validace dle xsd, převedení objektů do XML a uložení do souboru, pokud vznikne v JAXB frameworku problém tak se aplikace ukončí, není tak umožněno automatické spuštění nové hry, protože data nejdou uložit. Viz Zdrojový kód 39.

Zdrojový kód 39: *PlayerXMLDAO – metoda persistOrUpdate*

```
@Override
public void persistOrUpdate(Player... player) {
    File file = new File(composeFileName());
    if(players==null){
        players = new Players();
    }
    pojoToXml(player);
    try {
        JAXBContext jaxbContext = JAXBContext.newInstance(Players.class);
        Marshaller jaxbMarshaller;
        jaxbMarshaller = jaxbContext.createMarshaller();
        jaxbMarshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT,
            true);
        jaxbMarshaller.marshal(players, file);
    } catch (JAXBException e) {
        throw new Error("Problem with storing");
        //terminate application, problem with storing
    }
}
```

Zdroj: vlastní tvorba

Následující soukromé metody slouží k převedení pojo objektu (Player) na objekt reprezentující XML (Players). Viz Zdrojový kód 40.

Zdrojový kód 40: PlayerXMLDAO – soukromé metody pro konverzi objektů

```
private void pojoToXml(Player[] playersPojo){
    OUTER:
    for(Player playerPojo : playersPojo){
        for(Players.Player playerXml : players.getPlayer()){
            if(playerXml.getName().equals(playerPojo.getName())){
                playerXml.setWins(playerPojo.getWins());
                playerXml.setLoses(playerPojo.getLoses());
                playerXml.setTies(playerPojo.getTies());
                playerXml.setName(playerPojo.getName());
                continue OUTER;
            }
        }
        Players.Player playerXml = new Players.Player();
        playerXml.setWins(playerPojo.getWins());
        playerXml.setLoses(playerPojo.getLoses());
        playerXml.setTies(playerPojo.getTies());
        playerXml.setName(playerPojo.getName());
        players.getPlayer().add(playerXml);
    }
}

private Player xmlToPojo(Players.Player playerXml, Player playerPojo){
    playerPojo.setWins(playerXml.getWins());
    playerPojo.setLoses(playerXml.getLoses());
    playerPojo.setTies(playerXml.getTies());
    return playerPojo;
}
```

Zdroj: vlastní tvorba

Na závěr této třídy implementuji dvě soukromé metody, jednu pro získání cesty k souboru a jednu pro získání relevantních dvou hráčů z množiny získaných hráčů a zavolání konverze na pojo Player. Viz Zdrojový kód 41.

Zdrojový kód 41: *PlayerXMLDAO* – ostatní soukromé metody

```

private String composeFileName() {
    return String.format(STORE_PATH, getWindowsUserName());
}

private String getWindowsUserName(){
    if(windowsUsername == null){
        windowsUsername = System.getProperty(WINDOWS_USERNAME_PROPERTY);
    }
    return windowsUsername;
}

private Player[] getPlayer(Player[] playersPojo){
    int count = 0;
    for(Players.Player playerXml : players.getPlayer()){
        if(playerXml.getName().equals(playersPojo[0].getName())){
            count++;
            xmlToPojo(playerXml, playersPojo[0]);
        }else if(playerXml.getName().equals(playersPojo[1].getName())){
            count++;
            xmlToPojo(playerXml, playersPojo[1]);
        }
    }
    if(count==2)
        return playersPojo;

    return null;
}

```

Zdroj: vlastní tvorba

Players a ObjectFactory

Tyto dvě třídy generuji pomocí Eclipse, abych tak mohl udělat nejdříve vytvářím ukázkový soubor uložení dat (sampleXml.xml v balíčku *resources*, viz Zdrojový kód 42). Z obsahu tohoto xml souboru definuji xsd schéma pro validaci (pomocí XmlGrid.net/xml2xsd.html), toto xsd ukládám do souboru *playerSchema.xsd*, taktéž v balíčku *resources*. V Eclipse v Project Exploreru, pravým kliknutím na *playerSchema.xsd* rozbaluji kontextové menu, v které vybírám *Generate, JAXB Classes...* Následně vybírám svůj projekt, dále balíček do kterého to chci uložit (tedy *com.vse.application.model.dao.xml*) a kliknu na *Finish*. Tímto jsem vygeneroval potřebné třídy *Players* a *ObjectFactory* pro ukládání a načítání dat z xml souboru pomocí JAXB frameworku.

Zdrojový kód 42: *sampleXml.xml*

```
<?xml version="1.0" encoding="UTF-8"?>
<players>
  <player>
    <name>Karel</name>
    <wins>3</wins>
    <loses>4</loses>
    <ties>1</ties>
  </player>
  <player>
    <name>Tomáš</name>
    <wins>3</wins>
    <loses>42</loses>
    <ties>43</ties>
  </player>
  <player>
    <name>Roman</name>
    <wins>54</wins>
    <loses>33</loses>
    <ties>22</ties>
  </player>
</players>
```

Zdroj: vlastní tvorba

3.4.4 Delegate

Tento balíček/vrstva obsahuje především Controllery, View (definované v.fxml souboru) a soubor definující css styly pro tyto View. Implementace tohoto balíčku může opět probíhat paralelně a nezávisle na ostatních (po definování *spi*, *commons*, a abstraktní třídy *AbstractFieldCounter* – viz Zdrojový kód 32, v balíčku *model*).

MainController

Hlavní třídou této vrstvy je *MainController*, který inicializuje aplikaci a řídí tok mezi jednotlivými Controllery. Třída importuje *AbstractGameModel* a její reálnou implementaci *GameModel*, kterou v abstraktní podobě předává jednotlivým Controllery. Dále *IOException*, kontrolovanou výjimku, která potenciálně může přijít z FX frameworku. Následují nezbytné prvky FX frameworku. Třída obsahuje tři konstanty, uchovávající relativní cestu k .FXML souborům – které představují jednotlivé obrazovky. Dále jsou zde soukromé instanční proměnné: *primaryStage* (představující Stage, která je *MainControlleru* předána při spouštění od hlavní třídy aplikace), *currentScene* (dočasné uložení aktuální scény, dané obrazovky), *modalStage* (představuje modální Stage pro konečnou obrazovku [viz Obr. 38]), *mainPage* (*SplitPane* představující hlavní obrazovku [viz Obr. 37]), *welcomeScreen* (*AnchorPain* představující úvodní obrazovku [viz Obr. 36]) a flag *reRun* (identifikující zda je hra znovu spuštěna v rámci jednoho běhu). Viz Zdrojový kód 43.

Zdrojový kód 43: MainController

```
package com.vse.application.delegate;

import com.vse.application.model.GameModel;
import com.vse.application.spi.AbstractGameModel;
import java.io.IOException;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.scene.control.SplitPane;
import javafx.scene.layout.AnchorPane;
import javafx.stage.Modality;
import javafx.stage.Stage;

public class MainController {
    private static final String WELCOME_SCREEN = "WelcomeScreen.fxml";
    private static final String MAIN_PAGE = "MainPage.fxml";
    private static final String MODAL_END = "ModalEnd.fxml";

    private Stage primaryStage;
    private Scene currentScene;
    private Stage modalStage;
    private SplitPane mainPage;
    private AnchorPane welcomeScreen;
    private AbstractGameModel game;
    private boolean reRun;

    public void start(Stage primaryStage) {}
    public void initWelcomeScreen() {}
    public void initMainPage() {}
    public void forwardToMainPage() {}
    public void forwardToEndModal(MessagesUI message, String playerName) {}
    public void forwardToWelcomePage() {}
    private void initModal(MessagesUI message, String playerName) {}
}
```

Zdroj: vlastní tvorba

Velmi důležitou metodou je metoda `start`, která je spouštěna hlavní třídou aplikace. Uloží si získanou `Stage`, instancuje `GameModel` a volá metodu `initWelcomeScreen` pro zobrazení úvodní obrazovky (viz Obr. 24). Viz Zdrojový kód 44.

Zdrojový kód 44: MainController – metody pro spuštění aplikace

```

public void start(Stage primaryStage) {
    this.primaryStage = primaryStage;
    this.primaryStage.setTitle("Tic Tac Toe");
    this.game = new GameModel();
    initWelcomeScreen();
}

public void initWelcomeScreen() {
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(MainController.class
            .getResource("WELCOME_SCREEN"));
        welcomeScreen = (AnchorPane) loader.load();
        WelcomeScreenController welcomeScreenController =
            (WelcomeScreenController) loader.getController();
        welcomeScreenController.setGame(game).setMain(this);
        currentScene = new Scene(welcomeScreen);
        primaryStage.setScene(currentScene);
        if (reRun) {
            welcomeScreenController.preSetNameFields();
        }
        primaryStage.show();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Zdroj: vlastní tvorba

Následují metody zajišťující spuštění main page (viz Obr. 26) a spuštění končené obrazovky (viz Obr. 28). Viz Zdrojový kód 45.

Zdrojový kód 45: MainController – metody pro spuštění MainPage a ModalEnd

```

public void initMainPage() {
    try {
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(MainController.class.getResource(MAIN_PAGE));
        MainPage = (SplitPane) loader.load();
        MainPageController mainPageController = (MainPageController)
            loader.getController();
        mainPageController.setGame(game);
        game.addObserver(mainPageController);
        mainPageController.initializePostLoader();
        mainPageController.setMain(this);
        currentScene = new Scene(mainPage);
        primaryStage.setScene(currentScene);
        primaryStage.show();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

private void initModal(MessagesUI message, String playerName) {
    try {
        modalStage = new Stage();
        FXMLLoader loader = new FXMLLoader();
        loader.setLocation(MainController.class.getResource(MODAL_END));
        Parent root = loader.load();
        ModalEndController modalEndController = (ModalEndController)
            loader.getController();
        modalEndController.setMainController(this);
        modalEndController.setInfoMessage(message, playerName);
        modalStage.setScene(new Scene(root));
        modalStage.setTitle("End");
        modalStage.initModality(Modality.WINDOW_MODAL);
        modalStage.initOwner(currentScene.getWindow());
        modalStage.show();
    } catch (IOException e) {
        e.printStackTrace();
    }
}

```

Zdroj: vlastní tvorba

Následující metody slouží pro přepínání obrazovek. Zobrazení main page (forwardToMainPage), zobrazení modal end (forwardToEndModal) a zobrazení welcome screen (forwardToWelcomePage). Viz Zdrojový kód 46.

Zdrojový kód 46: *MainController* – metody pro přepínání obrazovek

```

public void forwardToMainPage() {
    game.initPlayerDetails();
    initMainPage();
}

public void forwardToEndModal(MessagesUI message, String playerName) {
    initModal(message, playerName);
}

public void forwardToWelcomePage() {
    modalStage.close();
    game.clearBoard().initPlayerDetails();
    reRun = true;
    initWelcomeScreen();
}

```

Zdroj: vlastní tvorba

WelcomeScreenController

WelcomeScreenController využívá *Mark*, pro identifikaci znaku hráče, *AbstractGameModel* pro komunikaci s logikou aplikace a příslušné prvky *FX* frameworku. Obsahuje skupinu instančních proměnných s anotací *@FXML*, což znamená že tento field přímo odpovídá danému prvku v *.fxml* souboru, ze kterého je tento *Controller* odkázán. Z následující třídy tedy vyplývá, že *.fxml* obsahuje alespoň dva *TextFieldy* (pojmenované *fieldX* a *fieldO*) a dva *Buttony* (pojmenované *btnX* a *btnO*). Dále obsahuje dvě metody anotované taktéž pomocí *@FXML*, které představují *handlers* a v tomto případě spuštěny při stisknutí příslušného tlačítka. Také obsahuje *settery* pro nastavení *mainControlleru* a *game*, které jsou použity *MainControllerem* (viz Zdrojový kód 44). A metodu pro nastavení hráčských jmen (v případě, že aplikace není čerstvě spuštěna). Na závěr privátní metodu *handle* volanou oběma *handlerama* (viz Zdrojový kód 48). Viz Zdrojový kód 47.

Zdrojový kód 47: *WelcomeScreenController*

```
package com.vse.application.delegate;

import com.vse.application.commons.Mark;
import com.vse.application.spi.AbstractGameModel;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;

public class WelcomeScreenController {

    @FXML
    private TextField fieldX;
    @FXML
    private TextField fieldO;
    @FXML
    private Button btnX;
    @FXML
    private Button btnO;

    private MainController mainController;
    private AbstractGameModel game;
    private String savedName;

    @FXML
    private void handleButtunO(){
        handle(fieldO, btnO, Mark.O);
    }

    @FXML
    private void handleButtunX(){
        handle(fieldX, btnX, Mark.X);
    }

    public WelcomeScreenController setMain(MainController mainController) {
        this.mainController = mainController;
        return this;
    }

    public WelcomeScreenController setGame(AbstractGameModel game) {
        this.game = game;
        return this;
    }

    public void preSetNameFields(){
        fieldX.setText(game.getCurrentPlayerName());
        fieldO.setText(game.getOtherPlayerName());
    }

    private void handle(TextField field, Button button, Mark mark){}
```

Zdroj: vlastní tvorba

Zdrojový kód 48: *WelcomeScreenController* – soukromá metoda *handle*

```
private void handle(TextField field, Button button, Mark mark){
    String name = field.getText();
    if(savedName != null && savedName.equals(name)){
        field.setText(null);
        field.setPromptText("Choose other name!");
        return;
    }
    if(name != null && !name.isEmpty()){
        savedName = name;
        game.setPlayerName(mark, name);
        button.setDisable(true);
        field.setDisable(true);
        if(btnX.isDisable() && btnO.isDisable())
            mainController.forwardToMainPage();
    }else{
        field.setPromptText("Fill your name here!");
    }
}
```

Zdroj: vlastní tvorba

MainPageController

MainPageController je *Controller* pro hlavní stranu aplikace (*MainPage*) a využívá *Mark* pro identifikaci znaku hráče, *AbstractFieldCounter* pro vytvoření vizuální reprezentace hracího plátnu, *AbstractGameModel* pro komunikaci s logikou aplikace, rozhraní *ObserverPlayerMoved*, které implementuje a stává se tak *Observerem* pro příjem notifikací na úspěšně provedený tah ve hře, nakonec využívá příslušné komponenty z FX knihovny. Viz Zdrojový kód 49.

Zdrojový kód 49: *MainPageController*

```
package com.vse.application.delegate;

import com.vse.application.commons.Mark;
import com.vse.application.model.AbstractFieldCounter;
import com.vse.application.spi.AbstractGameModel;
import com.vse.application.spi.ObserverPlayerMoved;
import java.util.List;
import javafx.collections.FXCollections;
import javafx.event.EventHandler;
import javafx.fxml.FXML;
import javafx.scene.control.Button;
import javafx.scene.control.CheckBox;
import javafx.scene.control.ChoiceBox;
import javafx.scene.control.Label;
import javafx.scene.input.MouseEvent;
import javafx.scene.layout.GridPane;
import javafx.scene.layout.VBox;

public class MainPageController implements ObserverPlayerMoved{
}
```

Zdroj: vlastní tvorba

V *MainPageControlleru* opět definuji sadu instančních fieldů anotovaných pomocí `@FXML`, které korespondují s `.fxml` souborem, z kterého je tento *Controller* odkázán. Dále zde vytvářím dvě soukromé konstanty představující textace pro tlačítko na spouštění a vypínání časované hry. Stejně jako ve většině *Controllerů*, tak i zde jsou instanční proměnné pro uchování odkazu na *AbstractGameModel* a *MainController*. Dále je zde `actualField` (představující vizuální reprezentaci zvoleného políčka), `fieldClickHandler` (handler pro zpracování kliknutí na dané políčko, je zde uložen z toho důvodu, aby se dal od *Fieldu* následně i odebrat). Nakonec obsahuje dvojrozměrné pole *Fieldů* reprezentující hrací plánec, který při načtení i po každé tahu je v absolutním souladu s plánkem v *BoardManageru*. Viz Zdrojový kód 50.

Zdrojový kód 50: *MainPageController* – instanční proměnné / konstanty

```

@FXML
private GridPane playBoard;
@FXML
private Label messageLabel;
@FXML
private Label labelXName;
@FXML
private Label labelXWins;
@FXML
private Label labelXLosses;
@FXML
private Label labelXTies;
@FXML
private Label labelOName;
@FXML
private Label labelOWins;
@FXML
private Label labelOLosses;
@FXML
private Label labelOTies;
@FXML
private VBox vBoxX;
@FXML
private VBox vBoxO;
@FXML
private ChoiceBox<Integer> choiceBox;
@FXML
private CheckBox checkBoxTimeEscalation;
@FXML
private Button buttonTimedGame;

private static final String ENABLE_TIMED_GAME_BTN_TEXT = "Enable timed game";
private static final String DISABLE_TIMED_GAME_BTN_TEXT = "Disable timed game";

private AbstractGameModel game;
private MainController mainController;
private Field actualField;
private FieldHandler fieldClickHandler;
private Field[][] allFields = new Field[AbstractFieldCounter.MAX+1][
    AbstractFieldCounter.MAX+1];

```

Zdroj: vlastní tvorba

Nejdříve implementuji metodu definovanou v rozhraní, tedy metodu `updatePlayerSwitched` (viz obr. 28). Viz Zdrojový kód 51.

Zdrojový kód 51: *MainPageController* – povinné metody

```

@Override
public void updatePlayerSwitched(boolean setManually) {
    Mark currentPlayerMark = game.getCurrentPlayerMark();
    if (actualField != null && setManually == true) {
        if (Mark.X.equals(currentPlayerMark)) {
            actualField.setImage(MarksUI.PLAYER_X);

        } else {
            actualField.setImage(MarksUI.PLAYER_O);
        }
        disableField();
    }
    if (game.getGameEnded()) {
        AbstractFieldCounter fieldCounter =
            AbstractFieldCounter.getInstance();
        while (fieldCounter.recalculate()) {
            actualField =
                allFields[fieldCounter.getX()][fieldCounter.getY()];
            disableField();
        }
        MessagesUI message;
        if (game.getGameWon()) {
            message = MessagesUI.WON;
            showWinnerRow(currentPlayerMark);
        } else {
            message = MessagesUI.TIE;
        }
        mainController.forwardToEndModal(message,
            game.getCurrentPlayerName());
    } else {
        if (Mark.X.equals(game.getCurrentPlayerMark())) {
            vBoxX.getStyleClass().remove("active-player");
            vBoxO.getStyleClass().add("active-player");
            setMessage(String.format(MessagesUI.YOUR_TURN.getValue(),
                game.getOtherPlayerName()));
        } else {
            vBoxX.getStyleClass().add("active-player");
            vBoxO.getStyleClass().remove("active-player");
            setMessage(String.format(MessagesUI.YOUR_TURN.getValue(),
                game.getOtherPlayerName()));
        }
    }
}

```

Zdroj: vlastní tvorba

Pokračuji implementací metod, které jsou anotovány pomocí `@FXML`, a jsou odkazovány z `.fxml` souborů. Metoda `initialize`, spouštěná FX frameworkem po zavolání metody `load` (viz Zdrojový kód 45), tok metody je zobrazen na Obrázku 26. Metoda `handleBtnTimedGame`, nejdříve zjišťuje zda se uživatel snaží časovanou hru zapnout či vypnout, poté získává hodnotu zvolenou v `choiceBox`, zjišťuje zda je `checkBoxTimeEscalation` zvolen a volá příslušnou metodu v `AbstractGameControlleru`. Viz Zdrojový kód 52.

```

@FXML
private void initialize() {
    AbstractFieldCounter fieldCounter = AbstractFieldCounter.getInstance();
    fieldClickHandler = new FieldClickHandler();
    while (fieldCounter.recalculate()) {
        final int x = fieldCounter.getX();
        final int y = fieldCounter.getY();
        final Field field = new Field(x, y);
        allFields[x][y] = field;
        field.addEventHandler(MouseEvent.MOUSE_CLICKED,
            fieldClickHandler);
        playBoard.add(field, x, y);
    }
    choiceBox.setItems(FXCollections.observableArrayList(1, 2, 3, 4, 5, 6,
        7, 8, 9, 10, 15, 20));
    choiceBox.getSelectionModel().selectLast();
    buttonTimedGame.setText(ENABLE_TIMED_GAME_BTN_TEXT);
    vboxX.getStyleClass().add("active-player");
}

@FXML
private void handleBtnTie() {
    game.gameEndedTie();
}

@FXML
private void handleBtnTimedGame() {
    if (buttonTimedGame.getText().equals(ENABLE_TIMED_GAME_BTN_TEXT)) {
        game.startTimedGame(choiceBox.getSelectionModel()
            .getSelectedItem().doubleValue(),
            checkBoxTimeEscalation.isSelected());
        buttonTimedGame.setText(DISABLE_TIMED_GAME_BTN_TEXT);
    } else {
        game.stopTimedGame();
        buttonTimedGame.setText(ENABLE_TIMED_GAME_BTN_TEXT);
    }
}
}

```

Zdroj: vlastní tvorba

Pokračuji implementací zbylých metod. Metoda initializePostLoader je rozebrána na Obrázku 26. Dále definuji setter pro mainController a AbstractGameModel používaný MainContollerem (viz Zdrojový kód 45). Nakonec soukromé metody pro nastavení textového výstupu do messageLabelu, deaktivování použitého políčka a zobrazení výherních políček při konci hry výhrou. Viz Zdrojový kód 53.

Zdrojový kód 53: *MainPageController* – ostatní metody

```

    void initializePostLoader() {
        labelOName.setText(game.getOtherPlayerName());
        Integer[] otherPlayerDetails = game.getOtherPlayerDetails();
        labelOWins.setText(otherPlayerDetails[0].toString());
        labelOTies.setText(otherPlayerDetails[1].toString());
        labelOLoses.setText(otherPlayerDetails[2].toString());
        labelXName.setText(game.getCurrentPlayerName());
        Integer[] currentPlayerDetails = game.getCurrentPlayerDetails();
        labelXWins.setText(currentPlayerDetails[0].toString());
        labelXTies.setText(currentPlayerDetails[1].toString());
        labelXLoses.setText(currentPlayerDetails[2].toString());
        setMessage(String.format(MessagesUI.YOUR_TURN.getValue(),
            game.getCurrentPlayerName()));
    }

    void setGame(AbstractGameModel game) {
        this.game = game;
    }

    public void setMain(MainController mainController) {
        this.mainController = mainController;
    }

    private void setMessage(String text) {
        messageLabel.setText(text);
    }

    private void disableField() {
        actualField.getStyleClass().add("imageViewDisabled");
        actualField.removeEventHandler(MouseEvent.MOUSE_CLICKED,
            fieldClickHandler);
    }

    private void showWinnerRow(Mark currentPlayerMark) {
        List<Integer[]> winnerFields = game.getWinnerFields();
        for (Integer[] winnerField : winnerFields) {
            allFields[winnerField[0]][winnerField[1]]
                .setImage(Mark.X.equals(currentPlayerMark) ?
                    MarksUI.PLAYER_X_WINNER : MarksUI.PLAYER_O_WINNER);
        }
    }
}

```

Zdroj: vlastní tvorba

Do *MainPageController*u ještě zbývá přidat třídu *FieldHandler*, která zpracovává kliknutí na ještě nepoužitá herní políčka – resp. metoda *handle* v ní obsažená. Která zjišťuje koordinace políčka a předává je do metody *move* na *AbstractGameModelu*. Viz Zdrojový kód 54.

Zdrojový kód 54: *MainPageController* – vnitřní třída *FieldHandler*

```
class FieldHandler implements EventHandler<MouseEvent> {
    @Override
    public void handle(MouseEvent event) {
        vboxX.getStyleClass().remove("active-player");
        actualField = (Field) event.getSource();
        game.move(actualField.getXCoord(), actualField.getYCoord());
    }
}
```

Zdroj: vlastní tvorba

ModalEndController

Jednoduchý Controller, pojící se k modal end (viz Obr 38). Drží odkaz pouze na *MainController* ve stejnojmenné instanční proměnné a dále definuje instanční proměnnou představující label pro zobrazení zprávy uživateli, k čemuž slouží metoda *setInfoMessage*. Dále obsahuje dva handlers, opět anotované pomocí *@FXML*, pro tlačítko ANO (volá přesměrování v *MainControlleru* na welcome screen) a NE (ukončuje aplikaci). Viz Zdrojový kód 55.

Zdrojový kód 55: *ModalEndController*

```
package com.vse.application.delegate;

import javafx.fxml.FXML;
import javafx.scene.control.Label;

public class ModalEndController {
    private MainController mainController;
    @FXML
    private Label labelInfoMessage;

    @FXML
    private void handleButtonN(){
        System.exit(0);
    }

    @FXML
    private void handleButtonY(){
        mainController.forwardToWelcomePage();
    }

    public void setMainController(MainController mainController) {
        this.mainController = mainController;
    }

    public void setInfoMessage(MessagesUI message, String playerName) {
        labelInfoMessage.setText(String.format(message.getValue(), playerName));
    }
}
```

Zdroj: vlastní tvorba

Field

Třída `Field` je potomkem `ImageView` z FX knihovny a rozšiřuje tuto třídu o instanční proměnné, konstrukturu a gettery a settery, což umožňuje přidat souřadnice, tak aby bylo jednoznačné jaké herní políčko dané `ImageView` představuje. Viz Zdrojový kód 56.

Zdrojový kód 56: Field

```
package com.vse.application.delegate;

import javafx.scene.image.ImageView;

public class Field extends ImageView {
    private int x;
    private int y;

    Field(int x, int y) {
        super(MarksUI.EMPTY);
        this.x = x;
        this.y = y;
    }

    public int getXCoord() {
        return x;
    }

    public void setXCoord(int x) {
        this.x = x;
    }

    public int getYCoord() {
        return y;
    }

    public void setYCoord(int y) {
        this.y = y;
    }
}
```

Zdroj: vlastní tvorba

MarksUI

Třída uchovávající odkazy na jednotlivé obrázky pro jednotlivá hrací políčka v podobě konstant. Viz Zdrojový kód 57.

Zdrojový kód 57: MarksUI

```
package com.vse.application.delegate;

import javafx.scene.image.Image;

public final class MarksUI {

    public static final Image EMPTY = new
        Image(MarksUI.class.getResourceAsStream("../resources/empty.png"));

    public static final Image PLAYER_X = new
        Image(MarksUI.class.getResourceAsStream("../resources/cross.png"));

    public static final Image PLAYER_O = new
        Image(MarksUI.class.getResourceAsStream("../resources/circle.png"));

    public static final Image PLAYER_X_WINNER = new
        Image(MarksUI.class.getResourceAsStream("../resources/cross-
        winner.png"));

    public static final Image PLAYER_O_WINNER = new
        Image(MarksUI.class.getResourceAsStream("../resources/circle-
        winner.png"));

}
```

Zdroj: vlastní tvorba

MessageUI

Jedná se o výčet obsahující textace, které jsou uživateli zobrazovány v průběhu hry. Viz Zdrojový kód 58.

Zdrojový kód 58: MarksUI

```
package com.vse.application.delegate;

import javafx.scene.image.Image;

public final class MarksUI {

    public static final Image EMPTY = new
        Image(MarksUI.class.getResourceAsStream("../resources/empty.png"));

    public static final Image PLAYER_X = new
        Image(MarksUI.class.getResourceAsStream("../resources/cross.png"));

    public static final Image PLAYER_O = new
        Image(MarksUI.class.getResourceAsStream("../resources/circle.png"));

    public static final Image PLAYER_X_WINNER = new
        Image(MarksUI.class.getResourceAsStream("../resources/cross-
        winner.png"));

    public static final Image PLAYER_O_WINNER = new
        Image(MarksUI.class.getResourceAsStream("../resources/circle-
        winner.png"));

}
```

Zdroj: vlastní tvorba

View

Jednotlivé obrazovky byly definovány v XML v .fxml souborech v program SceneBuilder – což je v podstatě WYSIWYG (What You See Is What You Get) editor, ovládání je tedy intuitivní a velmi jednoduché, ještě k tomu existuje mnoho návodů dostupných na internetu, proto tedy nerozebírám tvorbu těchto obrazovek. Zdrojový kód vytvořený, v tomto editoru, je vidět zde - Zdrojový kód 59-62.

Zdrojový kód 59: WelcomeScreen

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Separator?>
<?import javafx.scene.control.TextField?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.VBox?>

    <AnchorPane prefHeight="250.0" prefWidth="250.0"
xmlns="http://javafx.com/javafx/8.0.65" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.vse.application.delegate.WelcomeScreenController">
        <children>
            <VBox layoutX="156.0" layoutY="60.0" prefHeight="200.0" prefWidth="100.0"
AnchorPane.bottomAnchor="10.0" AnchorPane.leftAnchor="10.0"
AnchorPane.rightAnchor="10.0" AnchorPane.topAnchor="10.0">
                <children>
                    <TextField fx:id="fieldX" prefHeight="50.0" prefWidth="230.0" />
                    <Button fx:id="btnX" mnemonicParsing="false"
onAction="#handleButtunX" prefHeight="50.0" prefWidth="234.0" text="Start as player
X" />

                    <Separator prefHeight="31.0" prefWidth="230.0" />
                    <TextField fx:id="field0" prefHeight="50.0" prefWidth="230.0" />
                    <Button fx:id="btn0" mnemonicParsing="false"
onAction="#handleButtun0" prefHeight="50.0" prefWidth="234.0" text="Start as player
0" />
                </children>
            </VBox>
        </children>
    </AnchorPane>
```

Zdroj: vlastní tvorba

Zdrojový kód 60: MainPage (1/2)

```
<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.geometry.Insets?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.control.CheckBox?>
<?import javafx.scene.control.ChoiceBox?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Separator?>
<?import javafx.scene.control.SplitPane?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.layout.ColumnConstraints?>
<?import javafx.scene.layout.GridPane?>
<?import javafx.scene.layout.HBox?>
<?import javafx.scene.layout.RowConstraints?>
<?import javafx.scene.layout.VBox?>

<SplitPane dividerPositions="0.18, 0.82" prefHeight="600.0" prefWidth="750.0"
stylesheets="@PlayField.css" xmlns="http://javafx.com/javafx/8.0.65"
xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.vse.application.delegate.MainPageController">
    <items>
        <VBox fx:id="vBoxX" prefHeight="200.0" prefWidth="100.0">
            <children>
                <Label prefHeight="48.0" prefWidth="138.0" text="Player X" />
                <Label fx:id="labelXName" prefHeight="48.0" prefWidth="179.0" />
                <Separator prefHeight="50.0" prefWidth="132.0" />
                <Label prefHeight="48.0" prefWidth="252.0" text="wins: " />
                <Label fx:id="labelXWins" prefHeight="48.0" prefWidth="190.0" />
                <Separator prefHeight="20.0" prefWidth="200.0" />
                <Label prefHeight="48.0" prefWidth="256.0" text="Loses: " />
                <Label fx:id="labelXLoses" prefHeight="48.0" prefWidth="187.0" />
                <Separator prefHeight="20.0" prefWidth="200.0" />
                <Label prefHeight="48.0" prefWidth="429.0" text="ties: " />
                <Label fx:id="labelXTies" prefHeight="48.0" prefWidth="158.0" />
            </children>
            <padding>
                <Insets top="50.0" />
            </padding>
        </VBox>
        <SplitPane dividerPositions="0.18, 0.82" layoutX="140.0" layoutY="146.0"
orientation="VERTICAL" AnchorPane.bottomAnchor="0.0" AnchorPane.leftAnchor="0.0"
AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
            <items>
                <HBox prefHeight="100.0" prefWidth="200.0">
                    <children>
                        <ChoiceBox fx:id="choiceBox" prefWidth="150.0" />
                        <Button fx:id="buttonTimedGame" mnemonicParsing="false"
onAction="#handleBtnTimedGame" prefHeight="216.0" prefWidth="125.0" text="" />
                        <CheckBox fx:id="checkBoxTimeEscalation" mnemonicParsing="false"
prefHeight="72.0" prefWidth="104.0" text="time escalation" />
                        <Button fx:id="buttonTie" mnemonicParsing="false"
onAction="#handleBtnTie" prefHeight="99.0" prefWidth="93.0" text="Tie" />
                    </children>
                </HBox>
            </items>
        </SplitPane>
    </items>
</SplitPane>
```

Zdroj: vlastní tvorba

Zdrojový kód 61: MainPage (2/2)

```

        <GridPane fx:id="playBoard" layoutX="76.0" layoutY="185.0"
maxHeight="472.0" maxWidth="472.0" minHeight="472.0" minWidth="472.0"
prefHeight="472.0" prefWidth="472.0" styleClass="gridPain"
stylesheets="@PlayField.css" AnchorPane.bottomAnchor="0.0"
AnchorPane.leftAnchor="0.0" AnchorPane.rightAnchor="0.0" AnchorPane.topAnchor="0.0">
    <columnConstraints>
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
        <ColumnConstraints hgrow="SOMETIMES" prefWidth="4.72" />
    </columnConstraints>
    <rowConstraints>
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
        <RowConstraints maxHeight="47.2" vgrow="SOMETIMES" />
    </rowConstraints>
    <children/>
</GridPane>
<AnchorPane>
    <children>
        <Label fx:id="messageLabel" layoutY="6.0" styleClass="messageLabel"/>
    </children>
</AnchorPane>
</items>
</SplitPane>
<VBox fx:id="vBox0" prefHeight="200.0" prefWidth="100.0">
    <children>
        <Label prefHeight="48.0" prefWidth="282.0" text="Player 0" />
        <Label fx:id="labelOName" prefHeight="48.0" prefWidth="245.0" />
        <Separator prefHeight="50.0" prefWidth="132.0" />
        <Label prefHeight="48.0" prefWidth="279.0" text="wins: " />
        <Label fx:id="labelOWins" prefHeight="48.0" prefWidth="199.0" />
        <Separator prefHeight="20.0" prefWidth="200.0" />
        <Label prefHeight="48.0" prefWidth="223.0" text="loses: " />
        <Label fx:id="labelLOloses" prefHeight="48.0" prefWidth="275.0" />
        <Separator prefHeight="20.0" prefWidth="200.0" />
        <Label prefHeight="48.0" prefWidth="211.0" text="ties: " />
        <Label fx:id="labelOTies" prefHeight="48.0" prefWidth="269.0" />
    </children>
    <padding>
        <Insets top="50.0" />
    </padding>
</VBox></items></SplitPane>

```

Zdroj: vlastní tvorba

Zdrojový kód 62: ModalEnd

```

<?xml version="1.0" encoding="UTF-8"?>

<?import javafx.scene.control.Button?>
<?import javafx.scene.control.Label?>
<?import javafx.scene.control.Separator?>
<?import javafx.scene.layout.AnchorPane?>
<?import javafx.scene.text.Font?>

<AnchorPane prefHeight="200.0" prefWidth="400.0"
xmlns="http://javafx.com/javafx/8.0.65" xmlns:fx="http://javafx.com/fxml/1"
fx:controller="com.vse.application.delegate.ModalEndController">
    <children>
        <Label layoutX="152.0" layoutY="106.0" prefHeight="35.0" prefWidth="223.0"
text="Do you want to play again?">
            <font>
                <Font size="18.0" />
            </font>
        </Label>
        <Button layoutX="152.0" layoutY="149.0" mnemonicParsing="false"
onAction="#handleButtonY" prefHeight="35.0" prefWidth="100.0" text="YES" />
        <Button fx:id="btnN" layoutX="275.0" layoutY="149.0" mnemonicParsing="false"
onAction="#handleButtonN" prefHeight="35.0" prefWidth="100.0" text="NO" />
        <Label fx:id="labelInfoMessage" layoutX="8.0" layoutY="30.0" prefHeight="35.0"
prefWidth="388.0">
            <font>
                <Font size="18.0" />
            </font>
        </Label>
        <Separator layoutX="25.0" layoutY="97.0" prefWidth="200.0"
AnchorPane.leftAnchor="25.0" AnchorPane.rightAnchor="25.0" />
    </children>
</AnchorPane>

```

Zdroj: vlastní tvorba

Na závěr ještě definuji, z View odkázané styly, v podobě css. Viz Zdrojový kód 62.

Zdrojový kód 63: *PlayField.css*

```
.label {
    -fx-font-size: 25pt;
    -fx-font-family: "Segoe UI Semibold";
    -fx-text-fill: black;
}
.gridPain {
    -fx-alignment: center;
    -fx-grid-lines-visible:true;
    -fx-border-width: 2;
    -fx-border-color: black;
    -fx-padding: 0;
}
#messageLabel{
    -fx-font-size: 15pt;
}
.image-view{
    -fx-cursor: hand;
}
.imageViewDisabled{
    -fx-cursor: default;
}
.active-player{
    -fx-background-color: gray;
}
```

Zdroj: vlastní tvorba

3.4.5 Hlavní třída aplikace

V `MainApp` implementuji hlavní třídu, která spouští FX framework, který po své inicializaci spouští metodu `start` definovanou v předkovi `Application`. V této metodě vytvořím novou instanci `MainControlleru`, na které se volá metoda `start` (viz Zdrojový kód 44). Viz Zdrojový kód 63.

Zdrojový kód 63: *MainApp*

```
package com.vse.application;

import com.vse.application.delegate.MainController;
import javafx.application.Application;
import javafx.stage.Stage;

public class MainApp extends Application {

    @Override
    public void start(Stage primaryStage) {
        new MainController().start(primaryStage);
    }

    public static void main(String[] args) {
        launch(args);
    }
}
```

Zdroj: vlastní tvorba

Závěr

V průběhu práce jsem se snažil co nejlépe splnit jednotlivé mnou definované cíle, stanovené na začátku.

Na začátku, v kapitole 2 (Návrhový vzor MVC), jsem představil návrhový vzor jako takový, kde jsem také prezentoval několik definicí a myšlenek známých autorů a také tvůrce samotného, ohledně návrhového vzoru MVC. Také jsem zde prezentoval hlavní principy MVC a na jednoduchém příkladu v Excelu jsem tento vzor přiblížil. V podkapitole jsem popsal samotný vznik návrhového vzoru, a zkráceně jak se dále vyvíjel. V další podkapitole jsem se zaměřil na jednotlivé varianty MVC, jako View push model, View pull model, Model-Delegate a Model-View-Presenter. Poslední dvě dílčí kapitoly jsem věnoval architektuře MVC, především pak rozboru MVC jako složeniny z ostatních návrhových vzorů (především tedy návrhového vzoru Observer, v podobě Modelu, návrhového vzoru Strategy, v podobě Controlleru a nakonec návrhového vzoru Composite, v podobě View), a přínosu implementace aplikace dle návrhového vzoru MVC.

V poslední a zároveň stěžejní kapitole, kapitole 3 – Postup tvorby aplikace piškvorky dle MVC, jsem ve čtyřech dílčích částech popsal vývoj takovéto aplikace. Začal jsem částí zvanou Úvodní studie, v které jsem nejdříve vydefinoval přesné zadání aplikace, ze kterého jsou následně vytvořil funkční i nefunkční požadavky. V této části jsem ještě vytvořil diagram případů užití.

V další části, části zvané Analytická studie, jsem provedl analýzu a návrh aplikace. Z důvodu hlavního zaměření této práce na MVC, jsem nejdříve rozebral návrh architektury systému, právě podle návrhového vzoru Model View Controller, přesněji se však jedná o upravenou verzi odpovídající spíše kombinaci variant Model-Delegate a Model-View-Presenter. Stručně jsem popsal každý, ze třech, stavebních pilířů (Model, View, Controller). Rozložení těchto komponent jsem demonstroval na diagramu komponent. Pokračoval jsem vytvořením diagramu balíčků, kde jsem udal reálnou strukturu aplikace. Dále jsem představil jednoduchý datový model pro uložení detailů hráče v podobě XML. Následovala obecná definice podkladů, nezbytně nutných pro tvorbu dalších podrobnějších diagramů, kde jsem představil obecný diagram tříd aplikace. Následně jsem rozebral

komunikaci v rámci aplikace pro jednotlivé případy užití, především pomocí komunikačních diagramů, na kterých je vidět detailní tok zpráv v rámci celé aplikace. Jednotlivé komunikační diagramy, jsem poté ještě detailněji rozebral pomocí diagramů sekvenčních. Poslední co jsem v této části představil byly diagramy tříd na implementační úrovni, které samozřejmě vycházely z předchozích, již definovaných, diagramů.

V části třetí, Návrh uživatelského rozhraní, jsem, jak již nadpis samotný napovídá, představil jednotlivé obrazovky aplikace – konkrétně tři.

V části poslední, zvané Implementace aplikace, jsem postupně implementoval jednotlivé balíčky aplikace, nejdříve jsem tedy začal společnými balíčky *spi* a *commons*. Poté implementace již mohla probíhat paralelně či na přeskáčku (ještě tedy bylo třeba definovat jednu abstraktní třídu z balíčku *model*, kterou vyžadoval balíček *delegate*), byť jsem v implementaci pokračoval od balíčků s nejmenší závislostí na zbytku, tedy pokračoval jsem balíčkem *model* a na závěr až *delegate*.

Tímto jsem naplnil cíle definované na začátku práce, a tak doufám, že se mi podařilo vytvořit práci přínosnou a přehlednou, která bude plnit svůj účel, pro který byla napsána.

Seznam literatury

1. **REENSKAUG, Trygve.** *ADMINISTRATIVE CONTROL IN THE SHIPYARD*. Oslo: Central Institute for Industrial Research [online]. 1973 [cit. 2016-03-20]. Dostupný z WWW <<http://heim.ifi.uio.no/~trygver/1973/iccas/1973-08-ICCAS.pdf> >
2. **ADAMSON, Chris.** *Beyond MVC: A New Look at the Servlet Infrastructure Blog*. Oracle [online]. 2003, verze 2 [cit. 2016-03-18]. Dostupný z WWW <<https://community.oracle.com/docs/DOC-983320>>
3. **REENSKAUG, Trygve.** *MVC XEROX PARC 1978-79*. [online]. 2010 [cit. 2016-03-18]. Dostupný z WWW <<http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>>
4. **REENSKAUG, Trygve.** *THING-MODEL-VIEW-EDITOR*. LRG [online]. 1979 [cit. 2016-03-18]. Dostupný z WWW <<http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf>>
5. — **REENSKAUG, Trygve.** *MODELS -VIEWS - CONTROLLERS*. [online]. 1979 [cit. 2016-03-18]. Dostupný z WWW <<http://heim.ifi.uio.no/~trygver/1979/mvc-2/1979-12-MVC.pdf>>
6. — **ECKSTEIN, Robert.** *Java SE Application Design With MVC*. Oracle [online]. 2007, [cit. 2016-03-22]. Dostupný z WWW <<http://www.oracle.com/technetwork/articles/javase/index-142890.html>>
7. — **DEAN, Helman.** *Model-View-Controller*. [online]. 1998, [cit. 2016-03-26]. Dostupný z WWW <<http://ootips.org/mvc-pattern.html> >
8. — **SCOTT, Stanchfield.** *Applying MVC in VisualAge for Java*. JavaDude.com [online]. [cit. 2016-03-30]. Dostupný z WWW <<http://javadude.com/articles/vaddmvc1/mvc1.htm>>
9. — **SALIHEFENDIC, Amir.** *Flux vs. MVC (Design Patterns)*. medium.com [online]. 2015 [cit. 2016-04-01]. Dostupný z WWW <<https://medium.com/hacking-and-gonzo/flux-vs-mvc-design-patterns-57b28c0f71b7#.m31gtyvy9>>
10. — **FOWLER, Martin.** *Patterns of Enterprise Application Architecture*. 1st ed. Boston: Addison-Wesley. 2003. [Kap.] 14. ISBN 0321127420
11. — **BERNARD, Borek.** *Úvod do architektury MVC*. zdrojak.cz [online]. 2009 [cit. 2016-04-01]. Dostupný z WWW <<https://www.zdrojak.cz/clanky/uvod-do-architektury-mvc/>>
11. — **FREEMAN, Eric; FREEMAN, Elisabeth; SIERRA, Kathy; Bates, Bert.** *Head First Design Patterns*. O'Reilly Media, Inc. 2004. [Kap.] 12. ISBN 9780596007126
12. — **BUSCHMANN, Frank; HENNEY, Kevlin; SCHMIDT, C. Douglas.** *Pattern-oriented software architecture*. John Wiley & Sons, Ltd. 2007. [Kap.] 9. ISBN 0470059028

-
13. — **Oracle.** *JavaFX: Getting Started with JavaFX - 6 Using FXML to Create a User Interface.* docs.oracle.com [online]. 2014 [cit. 2016-04-03]. Dostupný z WWW <http://docs.oracle.com/javase/8/javafx/get-started-tutorial/fxml_tutorial.htm>
14. — **KUZNETSOV, Stanislav.** *Použití technologie Model View Controller (MVC).* České vysoké učení technické v Praze, Fakulta elektrotechnická. 2009, č 2
15. — **HAJNÍK, Julius.** *Analýza a návrh aplikace s využitím UML.* Vysoká škola ekonomická v Praze, Fakulta informatiky a statistiky. 2010
16. — **MARK, Jacob.** *JavaFX 8 Tutorial.* Code.makery.ch [online]. 2014 [cit. 2016-04-10]. Dostupný z WWW <<http://code.makery.ch/library/javafx-8-tutorial/>>
17. — **HOMMEL, Scott.** *Implementing JavaFX Best Practices.* docs.oracle.com [online]. 2014 [cit. 2016-04-13]. Dostupný z WWW <https://docs.oracle.com/javafx/2/best_practices/jfxpub-best_practices.htm>