PŘÍRODOVĚDECKÁ FAKULTA UNIVERZITY PALACKÉHO
KATEDRA INFORMATIKY

ROČNÍKOVÝ PROJEKT

Projektový seminář

Latrunculi

Červen 2016

Ondřej Grätz
Aplikovaná informatika, III. ročník

Abstrakt

Implementace deskové hry Latrunculi s použitím *.NET Framework* pro operační systém *Windows*.

Obsah

Seznam obrázků

1. Zadání projektu

Cílem projektu je vytvoření hry pro desktopový operační systém s grafickým uživatelským rozhraním (GUI) dle standardů.

Požadavek na přenositelnost spustitelných souborů nebyl stanoven. Uživatelské rozhraní se předpokládá objektově orientované s použitím oken a standardních prvků (hlavní nabídka, nástrojová lišta, tlačítka).

2. Volba technologie

2.1. Výběr operačního systému

Já jsem pro vývoj i běh hry (aplikace) zvolil operační systém *Windows*, jelikož je mi dobře známý a také proto, že je to s podílem 52 % (viz [4]) mezi uživateli i programátory nejpoužívanější operační systém pro osobní počítače.

2.2. Výběr vývojového prostředí

Pro vývoj byl zvolen nástroj *Visual Studio* od firmy *Microsoft*, jazyky *C#* a *F#* a pro vývoj uživatelského rozhraní *Windows Presentation Framework* (WPF).

Pro zvolený operační systém (*Windows*) by s ohledem na zadání bylo výhodné použít jeden z následujících typů aplikací.

- Win32 (Visual C++ s použitím Windows API nebo MFC)
- .NET Framework + WinForms
- .NET Framework + WPF
- Windows Runtime (C++/CX)

Win32 a *Windows Runtime* poskytují nejmenší úroveň abstrakce. Vývoj tohoto typu aplikací vyžaduje větší znalosti a zkušenosti programátora. *Windows Runtime* aplikace je navíc možné spustit pouze pod operačním systémem *Windows 8* (nebo novějším). Výhodou je ale standardní vzhled výsledných aplikací a nejlepší výpočetní výkon.

Použití *WinForms* by bylo výhodné, jelikož vývoj je jednoduchý (událostmi řízený, objektově orientovaný). Výsledná aplikace má vzhled v souladu se standardy operačního systému a očekáváním uživatele. Nevýhodou je strohý design, který se příliš nepřizpůsobuje rozlišení obrazovky a jehož vzhled se může jevit zastaralý. Design je navíc velmi svázán s kódem, který má na starosti výpočty a samotnou logiku aplikace.

Naproti tomu u *WPF* je již při vývoji myšleno na responzivní design. Prvky uživatelského rozhraní mají velmi bohaté a pro programátora snadno použitelné vlastnosti, které umožňují přízpůsobit aplikaci různým velikostem obrazovky a různým způsobům vstupu od uživatele (dotykové obrazovky, přizpůsobení pro zrakově nebo tělesně hendikepované uživatele). Návrh uživatelského rozhraní je navíc velmi dobře oddělen od samotného kódu a umožňuje s použitím nástroje *Microsoft Blend* výrazně zasáhnout do designu aplikace i grafikovi (bez nutnosti znalosti programování a bez zásahu do modelu aplikace).

3. Organizace kódu

3.1. Zvyklosti pro .NET aplikace

Při vývoji pro *.NET Framework* je nutno veškerý kód umisťovat do metod tříd. Třídy musejí být povinně organizovány do jmenných prostorů. Jmenné prostory je vhodné stromově strukturovat dle navržené architektury aplikace.

Pro *.NET* aplikace (s výjimkou aplikací v jazyku F#) je typické rozdělení zdrojového kódu do několika projektů, které jsou přidány do skupiny projektů (Solution). Rozdělení se provádí na základě jmenných prostorů (`namespaces`) tak, aby třídy patřící do stejného prostoru byly ve stejném projektu.

Spustitelný (EXE) soubor obsahje jádro programu a definice hlavních částí uživatelského rozhraní, ale komponenty uživatelského rozhraní i třídy obsahující doménový nebo datový model, rozhodovací logiku a třídy pro práci se soubory, databázemi či síťovou komunikaci jsou umístěny v tzv. knihovnách tříd (DLL soubory). Toto dělení poté představuje výhodu při požadavku na změnu typu výsledné aplikace. Lze například textové rozhraní aplikace nahradit GUI. Nebo nahradit samostatně spustitelnou konzolovou aplikaci službou systému Windows. Společné součásti tak mohou být opětovně využívány (sdíleny).

DLL i EXE soubory po sestavení obsahují překladačem vytvořený *bytecode*, který je poté při spuštění aplikace částečně kompilován do strojového kódu (just-in-time kompilace) a částečně také interpretován běhovým prostředím *.NET*.

3.2. Prevence cyklických závislostí

Abstraktní rozdělení kódu na vrstvy pomáhá při vývoji předcházet cyklickým (rekurzivním) závislostem mezi třídami (respektive mezi jednotlivými knihovnami tříd). Nevhodné závislosti mezi třídami by totiž mohly při dokončování aplikace způsobit komplikace, které by v krajním případě bránily sestavení aplikace a vyžadovaly by refaktorování celého kódu.

Jako prevence se proto činnosti jednotlivých součástí seskupují tak, aby vyšší vrstvy byly závislé na nižších vrstvách. Nevhodné je naopak odkazovat se z nižších vrstev na vyšší, ačkoliv i tato možnost může být potřebná. V případě potřeby se ale namísto reference používají události tak, že nižší vrstva generuje událost a vyšší vrstva událost zachytává. Bližší informace viz [6].

3.3. Soubory projektu

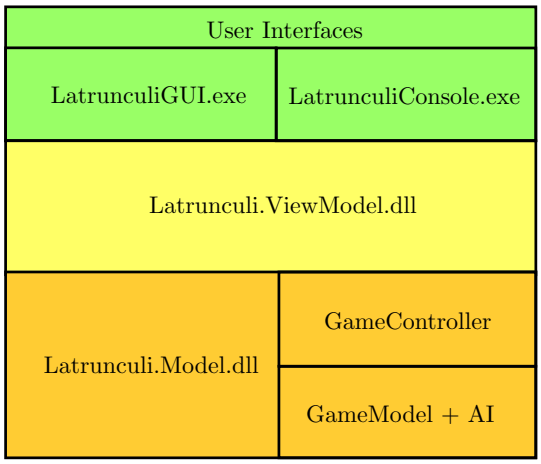
Třídy projektu jsou organizovány v podprostorech jmenného prostoru `Latrunculi`.

Rozvrstvení aplikace je zobrazeno na obr.1. Výsledné soubory sestavení jednotlivých projektů (assemblies) a jejich závislosti jsou zobrazeny na obr.2. Rozdělení do více projektů bylo provedeno s ohledem na výše uvedené zásady.

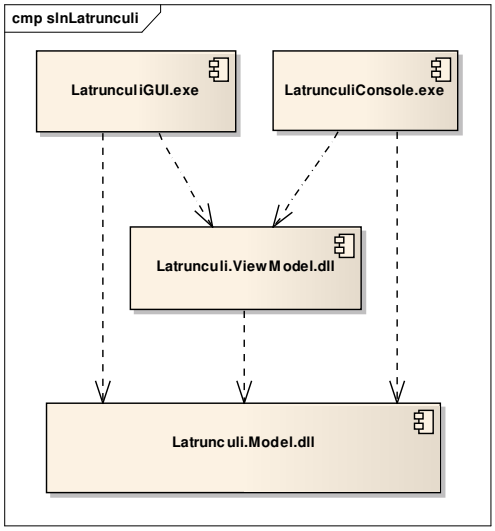
Úlohy jednotlivých částí systému budou popsány v další kapitole.

4. Vrstvy aplikace

Rozvrstvení aplikace vychází z architektury `MVC` (*Model-View-Controller*, viz [5]). Pro vývoj *WPF* aplikace jsem jej ale musel mírně modifikovat. Úlohu pohledu (`View`) zastává konkrétní okno aplikace (instance třídy `System.Windows.Window`) nedělitelně spolu



Obrázek 1. Vrstvy aplikace



Obrázek 2. Závislosti souborů

s odpovídajícím pohledovým modelem (`ViewModel`), jehož instanci okno vytvoří ve svém konstruktoru. Výsledkem je `MWVmC`, tj. *Model-Window-ViewModel-Controller*.

4.1. Uživatelské rozhraní (`Latrunculi*.exe`)

Nejvyšší vrstvou v našem diagramu z obr.1. je *User Interfaces*, tj. uživatelská rozhraní. Vrstva je horizontálně rozdělena, neboť naše aplikace má dva typy samostatně spustitelných uživatelských rozhraní. Konzolové rozhraní (`Latrunculi.ConsoleUI`) a grafické rozhraní (`Latrunculi.GUI`).

Uživatelské rozhraní zobrazuje stav hry uživateli a poskytuje prvky pro ovládání hry. Neobsahuje ale samotnou logiku ani řízení hry. Požadavky od uživatele jsou předávány nižší vrstvě - `GameController`.

Konstruktoru hlavního okna aplikace je nutno předat vytvořené instance tříd z nižších vrstev - `MainWindowViewModel`, `GameController`. Za vytvoření jejich instancí zodpovídá třída, která okno vytváří. V našem případě třída `App` (vstupní bod aplikace).

4.2. Pohledový model (`Latrunculi.ViewModel.dll`)

Použití pohledových modelů je typické pro WPF aplikace. Úlohou pohledového modelu je poskytnout uživatelskému rozhraní třídy, které obsahují vlastnosti (*Properties*), jež mohou být přímo napojeny na jednotlivé grafické komponenty uživatelského rozhraní. Takové třídy musejí implementovat rozhraní `INotifyPropertyChanged`, tj. implementovat událost `PropertyChanged`. Tato událost je zachytávána uživatelským rozhraním.

Při zachycení události dojde automaticky k aktualizaci grafických komponent, které jsou navázány (`Binding`) na vlastnost třídy, jejíž hodnota byla změněna. Dodejme navíc, že je podporováno i zachytávání změn v generických kolekcích typu `ObservableCollection<T>`, což se využívá pro aktualizaci komponent GUI typu List, ComboBox či tabulkového zobrazení.

Podobný princip aktualizace komponent (využití Bindingu) existuje i v aplikacích *WinForms* a také i v jiných vývojových prostředích a jazycích (Delphi, Java). Smyslem je odstra-

nit nutnost psát kód, který nesouvisí přímo s výpočetní logikou aplikace či jejím doménovým modelem.

Příkladem takového kódu je např. v události `Form.OnShow` kód `textBox1.Text = "Test";` , kterým se zaktualizuje konkrétní komponenta. Pokud komponentu `textBox1` přejmenujeme nebo odstraníme, aplikace přestane fungovat a daný řádek musíme upravit. Při použití pohledového modelu ve *WPF* aplikaci nemusí mít komponenta vůbec stanovený název a navíc může být více komponent (i různých typů) navázáno na stejnou vlastnost v třídě pohledového modelu. Jakákoliv změna v návrhu uživatelského rozhraní nerozbije funkčnost pohledového modelu.

V pohledovém modelu se zpravidla implementují třídy, které odpovídají třídám z nižší vrstvy aplikace (z doménového či datového modelu). Avšak implementují se pouze ty třídy, u kterých se předpokládá nutnost grafické reprezentace. Tj. zobrazení uživateli ve formě odpovídajícího grafického prvku. Někdy může být výhodné sloučit vlastnosti z různých tříd datového modelu do jedné třídy pohledového modelu.

Pohledový model si načítá a transformuje data z doménového modelu. Konstruktoru pohledového modelu proto musí být předána platná instance třídy `GameModel`.

4.3. Kontrolér a doménový model (`Latrunculi.Model.dll`)

Tyto součásti jsou implementovány v jazyku F#, který jakožto ryze funkcionální jazyk poskytuje možnosti interaktivního prototypově či doménově řízeného vývoje. Také obsahuje některé prostředky (např. datové typy a struktury), které usnadňují vývoj aplikací, a přitom v jazyku C# nemají odpovídající náhradu. Skriptovací vlastnosti jazyka F# navíc využijeme pro testování jednotlivých součástí aplikace.

Kontrolér má na starosti řízení hry. Jeho metody jsou volány z uživatelského rozhraní. Kontrolér na základě jejich volání provede příslušnou akci a předá požadavek doménovému modelu. Toto zpravidla také povede ke změně stavu modelu a tudíž k nutnosti aktualizovat pohledový model a uživatelské rozhraní.

Doménový model reprezentuje samotnou hru Latrunculi. Jeho úkolem je modelovat reálnou podobu hry, včetně jejích entit (deska, figurka, hráč, pravidla) a jejich atributů a metod. Doménový model musí zajistit, aby se hra neocitla ve stavu který odporuje pravidlům nebo je v rozporu s očekáváným chováním deskové hry. Více obecně o doménově řízeném vývoji viz [7].

Součástí této vrstvy je i implementace mozku počítačového hráče (`AI`).

5. Návrh aplikace

5.1. Případy užití

Případy užití vycházejí ze zadání a jsou zobrazeny v příloze na obr.3. Někteří aktéři poslouží jako základ pro objekty modelu. Jednotlivé případy užití potom poslouží při implementaci metod daných objektů.

5.2. Diagram tříd

Hlavní třídy používané pro uživatelské rozhraní a jejich atributy a relace jsou uvedeny na obr.4.. Třídy doménového modelu nejsou uvedeny v diagramu tříd, protože kvůli kon-

strukci jazyka F#, vypovídající hodnotě kódu v F# a prototypově řízeném vývoji nemá tvoření diagramu tříd při návrhu téměř žádnou přidanou hodnotu.

Třídy pohledového modelu nebudou v textu probírány, protože jsou z hlediska implementace nezajímavé (je to pouze ,,mechanický"kód zapouzdřující data z doménového modelu.

5.3. Unit Testy

Pro testování je použit framework *NUnit 3.2.1*. Soubory s testy doménového modelu jsou umístěny v sestavení `Latrunculi.Model.Test.dll`.

6. Implementace

Po vytvoření projektů *Visual Studia* a nastavení jejich vzájemných referencí jsem započal implementaci. A to od nejnižší vrstvy, tj. doménového modelu. Pro implementaci ve vývojovém prostředí bylo využito okno *F# Interactive*, které zpřístupňuje `REPL` cyklus jazyka F#. Bez nutnosti kompilace, spuštění či ladění nebo vytvoření zdrojového souboru umožňuje přímo zadat požadovanou definici objektu a ihned objekt otestovat (vytvořit instanci), případně doupravit.

6.1. Modul `Common`

Obsahuje definice společné pro celý model a kontrolér. Např. podporu návratových hodnot funkcí a podporu výpočtových funkcí (monáda `Maybe`).

```
[<AutoOpen>]
module Common

    type Result<'TSuccess,'TError> =
        | Success of 'TSuccess
        | Error of 'TError

    let unwrapResultExn c =
        match c with
        | Success c -> c
        | _ -> failwith "Unable␣to␣extract␣object␣instance␣from␣function␣
           result,␣because␣called␣function␣has␣failed."

    let tryChangeError e m =
        match m with
        | Success x -> Success x
        | Error _ -> Error e

    type MaybeBuilder() =
        member this.Bind(m, f) =
            match m with
            | Success s -> f s
            | Error e -> Error e
```

```
        member this.Return(x) =
            Success x

        member this.ReturnFrom(m) =
            m

    let maybe = new MaybeBuilder()
```

6.2. Entita `Piece`

Tato entita je implementována jako strukturovaný typ `Record` jazyka F#. Reprezentuje herní kámen, který bude umísťován na hrací desku. Ve hře Latrunculi je pouze jeden typ kamenů, a proto jediným atributem bude barva (bílá nebo černá). Zdrojový kód:

```
namespace Latrunculi.Model

module Piece =

    type Colors =
        | White = 0
        | Black = 1

    [<StructuralEquality;NoComparison>]
    type T = {
        Color: Colors }

    let create x =
        { Color = x }
```

Pro představu, jak velkou abstrakci umožňuje jazyk F# proti jazykům C# či .NET Basic uvádím i kód v jazyku C#, který implementuje stejnou funkcionalitu (vyžadovali jsme přímo porovnatelný, nemutovatelný strukturovaný objekt).

```
using System;
using System.Collections;

namespace Latrunculi.Model
{
        [Serializable]
        public class Piece : IEquatable<Piece>, IStructuralEquatable
        {
                public Piece(PieceColors color)
                {
                        Color = color;
                }

                internal PieceColors Color;
                public PieceColors Color
                {
                        get
                        {
                                return this.Color;
                        }
                }

                public override int GetHashCode(IEqualityComparer comp)
                {
                        if (this != null)
                        {
                                int num = 0;
                                return (int)(-1640531527 + (this.Color + ((num << 6) + (num >> 2))));
                        }
                        return 0;
                }
```

```
        public override int GetHashCode()
        {
            return this.GetHashCode(LanguagePrimitives.GenericEqualityComparer);
        }

        public override bool Equals(object obj, IEqualityComparer comp)
        {
            if (this == null)
            {
                return obj == null;
            }
            Piece t = obj as Piece;
            if (t != null)
            {
                Piece t2 = t;
                return this.Color == t2.Color;
            }
            return false;
        }

        public override bool Equals(Piece obj)
        {
            if (this != null)
            {
                return obj != null && this.Color == obj.Color;
            }
            return obj == null;
        }

        public override bool Equals(object obj)
        {
            Piece t = obj as Piece;
            return t != null && this.Equals(t);
        }
    }
}
```

Funkčně ekvivalentní kód v jazyku C# je výrazně delší.

6.3. Entita `Square`

Reprezentuje políčko na hrací desce. Je implementováno jako typ `discriminated union`. Políčko obsahuje buďto kámen nebo je prázdné.

```
module Square =

    type T =
        | Piece of Piece.T
        | Nothing

    let isEmpty x =
        match x with
        | Nothing -> true
        | _ -> false

    let createWithPiece piece =
        Piece piece

    let createEmpty =
        Nothing
```

6.4. Entita `Coord`

Reprezentuje souřadnice. Obsahuje logiku pro kontrolu rozsahu souřadnic. Souřadnice je možné vytvořit z řetězce (např. `"A1"`) nebo předání zvlášť písmene označující sloupec a čísla řádku. Umožňuje také iterovat přes všechny možné souřadnice.

6.5. Entita `RemovedPiece`

Zajatý kámen. Musíme si zaznamenat druh i souřadnice kamene, který bude odebrán (resp. přidán) při provedení tahu (resp. inverzního tahu).

```
module RemovedPiece =

    type T = {
        Coord: Coord.T;
        Piece: Piece.T }

    let create x y =
        { Coord = x; Piece = y }
```

6.6. Entita `Move`

Struktura `Record`, která reprezentuje tah a obsahuje počáteční a koncové souřadnice a také nové stavy obou políček. Při vytváření instance se kontroluje, zda počáteční a koncová souřadnice není shodná.

6.7. Entita `BoardMove`

Struktura, která je předávána desce pro provedení tahu. Základem je `Move`, obsahuje ale navíc seznam zajatých kamenů a jejich souřadnic (kameny, které budou odstraněny z hrací desky).

6.8. Třída `Board`

Hrací deska (Board) provádí tahy (bez ověřování pravidel), uchovává rozmístění kamenů a provedné tahy ukládá do historie.

Rozmístění kamenů je dvojrozměrné pole objektů `Square`.

6.9. Modul `Rules`

Obsahuje herní pravidla.

- `getInitialBoardSquares` vrací výchozí rozestavění kamenů na desce
- `getInitialActiveColor` vrací barvu hráče prvního na tahu
- `getValidMoves` vrací seznam platných tahů pro danou barvu hráče

6.10. Třída `Latrunculi.Model.GameModel`

Při vytvoření instance provede vytvoření herní desky a inicializaci rozmístění kamenů dle pravidel.

6.11. Třída `Latrunculi.Controller.GameController`

Provádí řízení hry. Obsahuje odkaz na instanci třídy `Latrunculi.Model.GameModel`. Umožňuje změnit nastavení hráčů. Metodou NewGame se inicializuje herní deska do výchozí polohy.

V metodě GameLoop je nekonečná (tokenem zrušitelná) herní smyčka, která v každém cyklu:

- vyvolává události pro pohledový model (překreselní desky, aktualizace UI)
- získá tah od hráče (PC nebo lidského)
- zkontroluje tah u rozhodčího (modulu `Rules`)
- provede tah na herní desce
- změní hráče na tahu

6.12. Realizace počítačového hráče - modul `Brain`

Pokud je instance třídy reprezentujícího hráče na tahu typu `ComputerPlayer`, je v její metodě `GetMove` volán modul `Brain`, který obsahuje algoritmus pro zjištění nejlepšího tahu.

Použit byl algoritmus `MiniMax` (viz [3]).

7. Systémové požadavky

Pro vývoj i běh aplikace jsou na počítač kladeny tyto nároky:

- osobní počítač, notebook nebo tablet
- 32bitový (x86) nebo 64bitový (x64) procesor s frekvencí 1 GHz nebo vyšší
- operační systém Microsoft Windows 7 (64 bitový nebo 32 bitový)
- 1 GB paměti RAM (32bitová verze) nebo 2 GB paměti RAM (64bitová verze)
- 16 GB volného místa na disku (32bitová verze) nebo 20 GB (64bitová verze)
- Grafická karta s podporou DirectX 9 s ovladačem WDDM 1.0 nebo novějším
- Microsoft .NET Framework verze 4.5.2 (nebo novější)

8. Vytvoření aplikace

8.1. Instalace vývojového prostředí

Pro sestavení aplikace nainstalujeme nejprve vývojové prostředí *Microsoft Visual Studio*.

Pro vývoj byla použita edice *Community 2015*, která je zdarma ke stažení na stránce `https://www.visualstudio.com/cs-cz/downloads/download-visual-studio-vs.aspx`.

Při instalaci zvolíme jazyk C#. Po nainstalování zvolíme v menu File → New → Project... V zobrazeném dialogovém okně zvolíme ve stromu vlevo položku Installed → Templates → Other Languages → Visual F# a poté v seznamu nabídnutých položek zvolíme Install F# Tools a provedeme instalaci nástrojů pro vývoj v jazyku F# (pokud máme již správně nainstalovánu podporu jazyku F#, budou v seznamu položky Console Application, Library a další).

Lze použít i jiné edice, případně starší verze *Visual Studia*. Jediným požadavkem je, aby daná verze uměla cílit na *.NET Framework verze 4.5.2* a aby podporovala jazyk F# verze 4.0.

8.2. Sestavení aplikace

Jestliže nemáme soubory se zdrojovým kódem, tak v nabídce Team zvolíme Manage Connections. V okně Team Explorer - Connect v části Local Git Repositories zvolíme příkaz Clone a jako Repository URL zadáme `https://github.com/ondrejgr/FLatrunculi.git`. Případně změníme název složky, do které chceme zdrojové soubory stáhnout.

Příkazem Build Solution v menu Build provedeme sestavení aplikace.

Pokud budeme chtít spouštět i testy, nainstalujeme *NUnit* z `http://www.nunit.org/index.php?p=download`.

9. Spuštění aplikace

Aplikace se spouští spustitelným souborem `LatrunculiGUI.exe`. Ve složce s aplikací musejí být uloženy také následující soubory knihoven tříd:

- `FSharp.Core.dll`
- `Latrunculi.Model.dll`
- `Latrunculi.ViewModel.dll`

Reference

[1] Mgr. Jan Outrata, Ph.D.: *Projekt - implementace*, `http://outrata.inf.upol.cz/courses/ps/navrh.pdf`, listopad 2008.

[2] Mgr. Jan Outrata, Ph.D.: *Projekt - analýza a návrh*, `http://outrata.inf.upol.cz/courses/ps/implementace.txt`, listopad 2010.

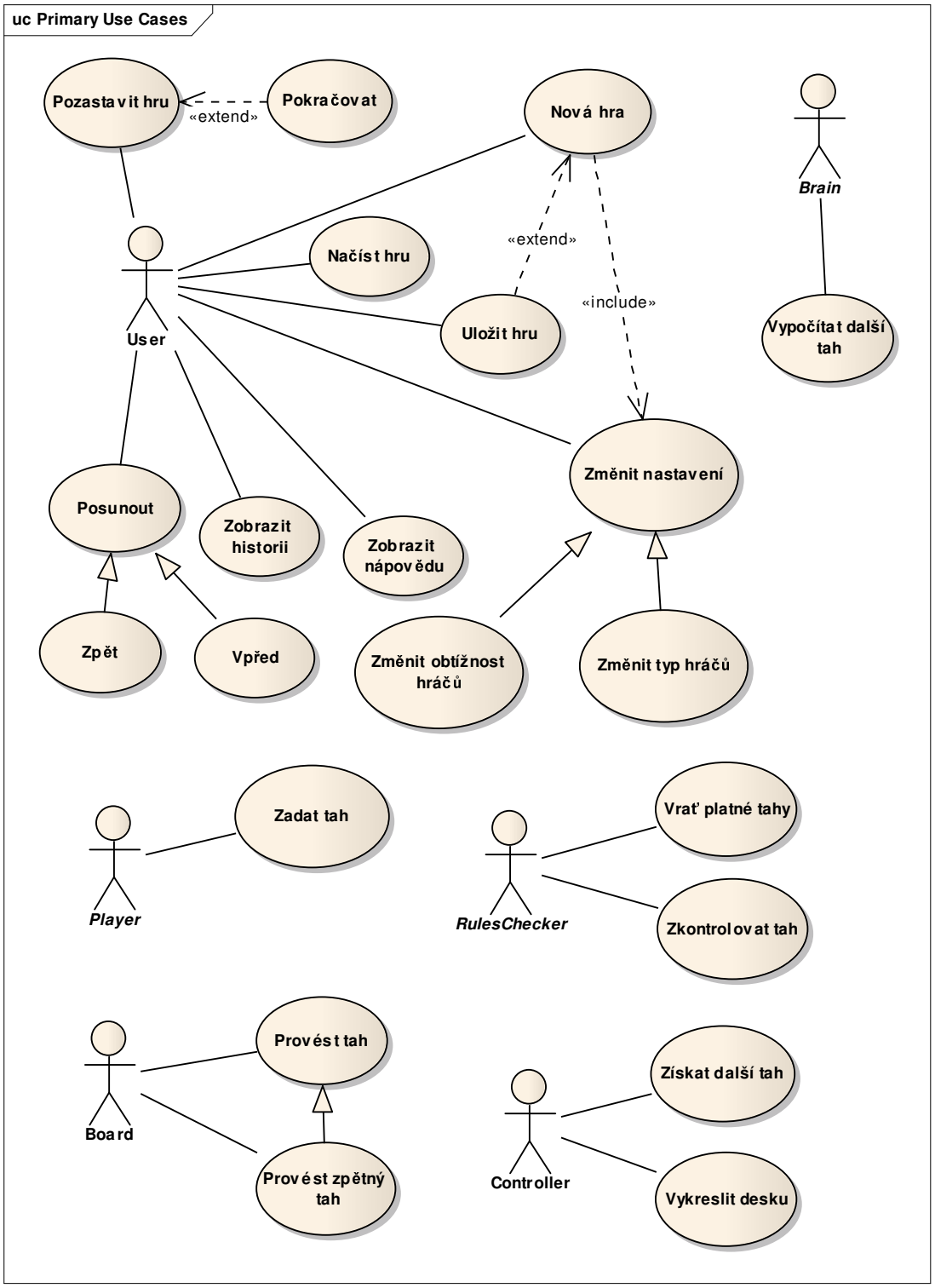
[3] Mgr. Tomáš Kühr, Ph.D.: *Algoritmy realizující počítačového hráče*, `http://www.inf.upol.cz/downloads/studium/PS/algoritmy.pdf`, říjen 2011.

[4] Wikipedia: *Usage share of operating systems*, `https://en.wikipedia.org/wiki/Usage_share_of_operating_systems#Desktop_and_laptop_computers`, květen 2016.

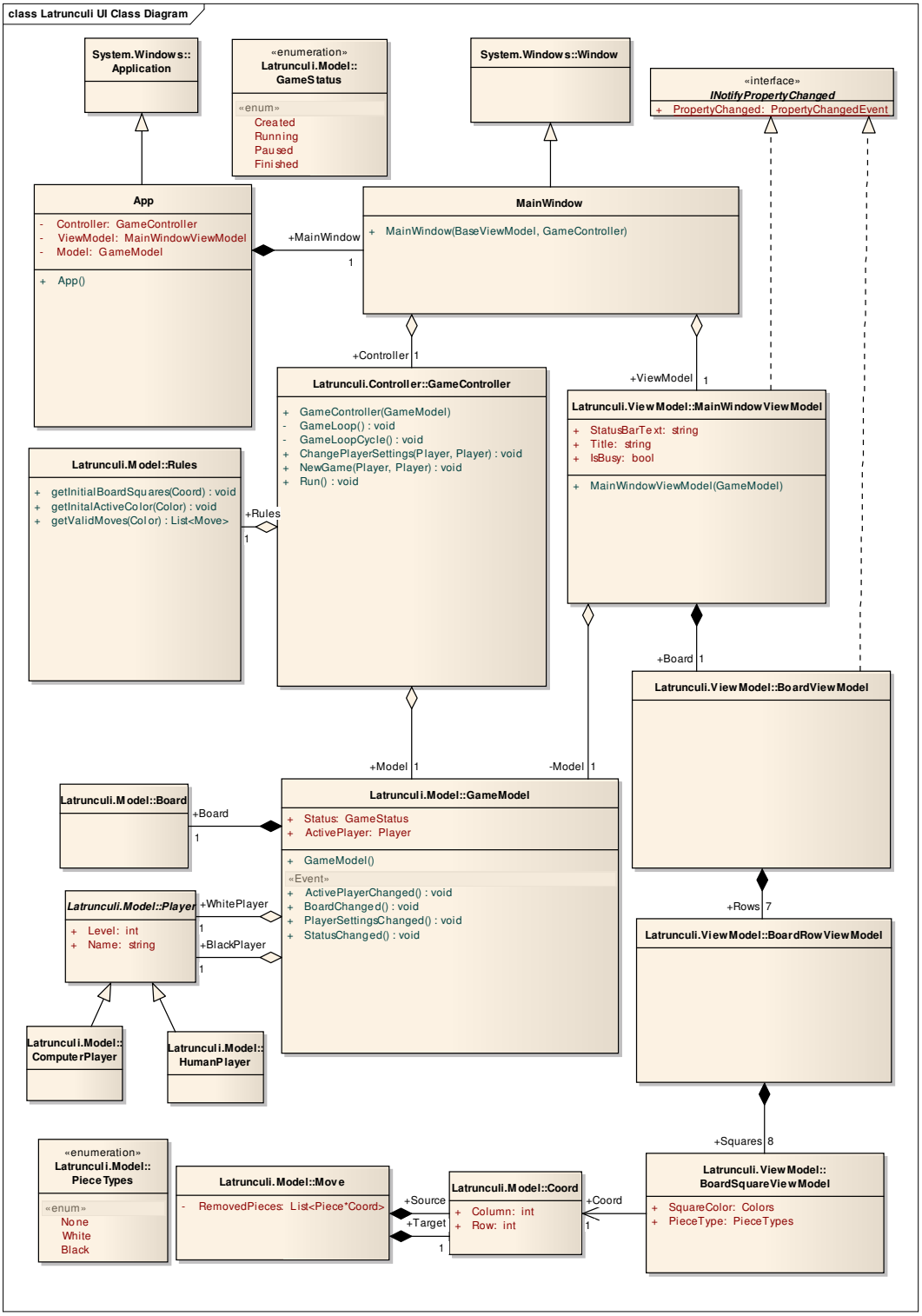
[5] Wikipedia: *Model-view-controller*, `https://en.wikipedia.org/wiki/https://en.wikipedia.org/wiki/Model-view-controller`, květen 2016.

[6] Scott Wlaschin: *Dependency Cycles*, `https://fsharpforfunandprofit.com/series/dependency-cycles.html`, květen 2016.

[7] Scott Wlaschin: *Domain Driven Design*, `https://fsharpforfunandprofit.com/ddd/`, květen 2016.



Obrázek 3. Případy užití



Obrázek 4. Diagram tříd