Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra informačních technologií

Diplomová práce

Software pro podporu výuky objektově orientovaného programování

Bc. Pavel Bory

! ! !

Místo tohoto textu vložte PŘEDNÍ stranu zadání práce, které si můžete vyexportovat do PDF v IS.CZU.cz, pokud již máte schválené zadání i děkanem PEF.

! ! !

! ! !

Místo tohoto textu vložte ZADNÍ stranu zadání práce, které si můžete vyexportovat do PDF v IS.CZU.cz, pokud již máte schválené zadání i děkanem PEF.

V případě, že Vaše zadání je na více než 2 strany, vložte i další strany.

! ! !

Čestné prohlášení

Prohlašuji, že svou diplomovou práci "Software pro podporu výuky objektově orientovaného programování" jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31. 3. 2017 _____

Poděkování

Rád bych touto cestou poděkoval všem, kteří mi jakkoli pomohli při tvorbě mé diplomové práce, ať již morálně, či odbornou radou. Jmenovitě bych pak chtěl poděkovat svému vedoucímu, panu Ing. Martinu Havránkovi, Ph.D. za pomoc a podporu při tvorbě této práce.

Software pro podporu výuky objektově orientovaného programování

Souhrn

Tato práce se zabývá návrhem a vývojem prototypu aplikace pro podporu výuky objektově orientovaného programování. V této aplikaci bude zobrazena konfigurovatelná mřížka obsahující obrázek reprezentující aktéra simulace, kterého bude moci uživatel programovat. Tento aktér se bude moci v rámci mřížky pohybovat a interagovat s jinými objekty reprezentovanými obrázky rozmístěnými v této mřížce. Klíčovými technologiemi zvolenými pro realizaci aplikace jsou programovací jazyk Java, technologie pro tvorbu grafického uživatelského rozhraní JavaFX, programovací jazyk Ruby a technologie JRuby pro integraci Java a Ruby. V práci jsou rovněž použity technologie log4j pro logování a JUnit pro tvorbu unit testů. V teoretické části práce jsou tyto technologie analyzovány a na základě teoretických východisek je vybrána vhodná strategie integrace Ruby a Java pro tuto práci. V úvodu praktické části jsou definovány funkční a nefunkční požadavky na prototyp aplikace kladené. Dále je v praktické části navrženo uživatelské rozhraní, navržena architektura aplikace a jsou navrženy jednotlivé klíčové třídy aplikace. Poté je implementován, zdokumentován a otestován i s použitím unit testů prototyp aplikace. Součástí práce je i ukázkový příklad použití implementovaného prototypu aplikace při výuce programování.

Klíčová slova: Java, Ruby, JRuby, JavaFX, JUnit, programování, výuka programování, objektově orientované programování, návrh software, implementace software.

Software to aid teaching of object oriented programming

Summary

This thesis addresses design and development of the application to aid teaching of object oriented programming. In the thesis's practical part the prototype of the application will be implemented. The application will display a configurable grid containing an image representing the actor of the simulation, that could be programmed by the user. The actor will be able to move within the grid and interact with other objects represented by images, that will be arranged in the grid. Key technologies selected for the implementation of the application are the Java programming language, the GUI framework JavaFX, the Ruby programming language and JRuby, that is the technology for Java and Ruby integration. In the thesis, the log4j for logging and JUnit for unit tests implementation are used. These technologies are analyzed in the theoretical part of the thesis and the appropriate strategy for Java and Ruby is chosen. In the initial part of the practical part, the functional and non-functional requirements for the application's prototype are defined. Then the user interface is designed, the application's architecture is designed as well as the most important classes for the application's prototype. After the design, the application's prototype is implemented, documented and tested manually as well as by the unit tests. The thesis also contains an example of the usage of the prototype in the teaching of object oriented programming.

Keywords: Java, Ruby, JRuby, JavaFX, JUnit, programming, teaching of programming, object oriented programming, software design, software implementation.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

1 Úvod

Programovacích jazyků je nepřeberné množství, přičemž je poměrně obtížné říci, které jsou nejrozšířenější a nejpoužívanější. Je možné se orientovat např. podle TIOBE Index, který ve chvíli, kdy je tato práce psána ukazuje, že nejpopulárnějším programovacím jazykem je programovací jazyk Java (1).

Pro začínající programátory, ať již samouky nebo např. studenty informatiky by se mohl tento jazyk jevit jako vhodná volba. Nicméně v porovnání s některými jinými jazyky má složitější syntaxi a při studiu tohoto jazyka jako prvního programovacího jazyka je v podstatě nevyhnutelné ponechávat v začátcích určité konstrukce ve zdrojovém kódu bez bližšího vysvětlení. Zároveň jsou v dnešní době uživatelé počítačů, mezi něž i ony studenty informatiky řadíme, v podstatě odstíněni od práce v textovém režimu příkazové řádky a jsou zvyklí využívat především graficky bohaté aplikace.

Pokud bychom výuku programování zahájili nad jednoduchými konzolovými aplikace, pak se může stát, že pro studenty nebudou takovéto aplikace příliš přívětivé, neboť jsou zvyklí na úplně jiný typ aplikací. Samozřejmě je možné nalézt důvody, proč jsou pro výuku základů programování takovéto aplikace vhodné, ale v této práci budou uvažování především ti studenti, kterým se konzolové aplikace jeví jako nepopulární. Možným východiskem by bylo začít učit programování na vývoji grafických aplikací, ale v tom případě množství konstrukcí a zdrojového kódu, které bude v začátcích ponechám bez bližšího vysvětlení naroste a orientace ve vývoji takovýchto aplikací může být horší.

Řešením této situace by mohla být kombinace programovacího jazyka, který má jednoduchou a přímočarou syntaxi a prostředí, ve kterým by studenti s pomocí tohoto jazyka např. ovládali určité grafické objekty formou simulací či her. Vhodnou kombinací by mohlo být dosaženo toho, že studenti si osvojí programátorské myšlení a budou studovat jazyk, na kterém lze vysvětlit obecně platné principy objektově orientovaného programování, přičemž programy, které budou tvořit budou atraktivnější než konzolové aplikace a zároveň nebudou složité jako aplikace se standardním grafickým rozhraním.

2 Cíl práce a metodika

2.1 Cíl práce

Cílem této práce je navrhnout a vytvořit prototyp software pro podporu výuky objektově orientovaného programování. Software bude navržen tak, aby poskytl studentům integrované vývojové prostředí, ve kterém budou moci tvořit programy, kterými budou ovládat předpřipravené 2D grafické objekty v různých prostředích na základě pokynů a doporučení lektora. Grafické objekty bude možné do software vkládat v běžně rozšířených formátech jako např. PNG. Software dále poskytne možnost definovat různá prostředí pro výuku jednotlivých témat z OOP. Z hlediska technologií bude jako programovací jazyk pro výuku objektově orientovaného programování zvolen jazyk Ruby zejména pro svou jednoduchou syntaxi a možnost integrace s jazykem Java, ve kterém bude software tvořen.

2.2 Metodika

Ke splnění vytyčeného cíle budou nejdříve analyzovány vhodné technologie a možnosti jejich integrace. Dále budou stanoveny základní oblasti z objektově orientovaného programování, které jsou běžně vyučovány a budou podporovány navrženým software. V rámci návrhu budou popsány funkční a nefunkční požadavky na navržený software a navržena softwarová architektura. Ve výsledku bude implementován prototyp software a provedeno jeho otestování.

3 Teoretická východiska

V této kapitole budou popsána základní teoretická východiska pro návrh a vývoj cílového software. Nejdříve bude stručně popsán programovací jazyk Ruby a důvody jeho volby jako programovacího jazyka pro výuku programování v navrhovaném software. V neposlední řadě bude popsán programovací jazyk Java a související technologie zvolené pro implementaci tohoto software. Dále bude popsána technologie JRuby a její souvislost s touto prací. Praktická část této práce bude primárně vyvíjena a testována v operačním systému Windows 10, konkrétně v edici Pro. Nicméně software by mělo být možné bez větších obtíží provozovat i na jiných operačních systémech, o čemž bude stručně pojednáno na konci této kapitoly, ale cílem této práce není ověřit funkčnost software na jiných operačních systémem než na tom, na kterém bude praktická část práce realizována.

3.1 Programovací jazyk Ruby

Ruby je interpretovaný dynamický objektově orientovaný programovací jazyk, který v porovnání s jinak velmi rozšířenými programovacími jazyky jako jsou např. jazyky C# a Java má v řadě ohledů podstatně jednodušší syntaxi. Cílem této práce není porovnat jej s ostatními programovacími jazyky z hlediska využitelnosti při implementaci různých programů, ale vyzdvihnout jednoduchost jeho syntaxe pro výuku základů objektově orientovaného programování.

Soubory obsahující zdrojový kód v jazyce ruby ukládáme s příponou `.rb`. Aby bylo možné programy v jazyce ruby spustit, je zapotřebí nainstalovat příslušný interpret, který lze získat na oficiálních stránkách https://www.ruby-lang.org/en/. K dispozici jsou instalační zdroje pro nejrozšířenější operační systémy. Pro operační systém Windows je k dispozici standardní instalátor, který nainstaluje jak interpret jazyka Ruby, tak i některé další potřebné prvky, jako např. standardní knihovnu jazyka Ruby. Dokumentaci standardní knihovny a základních tříd jazyka Ruby lze nalézt na oficiálních webových stránkách http://ruby-doc.org/.

Pro ilustraci syntaxe Ruby lze nahlédnout na následující ukázku zdrojového kódu, kde řádky začínající znakem `#` jsou komentáře, které interpret ignoruje a slouží pro zpřehlednění ukázkového zdrojového kódu.

```ruby
# Deklarace promennnych
integer = 100
float = 3.14
name = "Pavel"
arr = [3,4,5]
asoc_arr = {"Pavel" => 25000, "Anna" => 23500 }
# Prazdna "null" hodnota
max_num = nil

# Vypis na konzoli s  odradkovanim
puts "Cyklus s pevne danym poctem opakovani"
3.times do |i|
  puts "#{i}. beh cyklu"
end

# Cyklus s ukoncovaci podminkou
i = 0
while i < arr.length
  # Podminka
  if arr[i] % 2 == 0
    puts "#{i} je sude"
  else
    puts "#{i} je liche"
  end
  i += 1
end

puts "Iterace vsemi prvky pole"
arr.each do |item|
  print "#{item} "
end
puts

puts "Iterace vsemi prvky asociativniho pole"
asoc_arr.each do |key,value|
  puts "#{key} : #{value}"
end
```

Zdrojový kód 1: Ukázka zdrojového kódu v jazyce Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

Pokud by byl výše uvedený zdrojový kód uložen do souboru pojmenovaného `ukazka.rb`, pak je možné jej spustit např. z příkazové řádky následujícím příkazem[1] `ruby ukazka.rb`.

V následující ukázce je možné nahlédnout na to, jakým způsobem lze v Ruby implementovat jednoduchou třídu, vytvořit z ní instanci a na té zavolat metodu. V ukázce je definována třída `Student`, která má dvě instanční proměnné `@first_name` a `@last_name`, které lze z vnějšku třídy pouze číst a dále třídě `Student` obsahuje instanční proměnnou

[1] Předpokladem pro správnou funkčnost příkazu ruby z příkazové řádky je správné nastavení systémové proměnné PATH tak, aby obsahovala cestu k programu ruby.exe, což při standardním průběhu instalace zajistí instalátor.

`@year_of_study`, kterou lze z vnějšku i modifikovat. Třída `Student` zároveň implementuje metodu `full_name`, která vrací celé jméno studenta.

```ruby
# Definice tridy
class Student
  # Definice getteru
  attr_reader :first_name, :last_name
  # Definice getteru/setter
  attr_accessor :year_of_study

  # Konstruktor s parametry
  def initialize(first_name,last_name,year_of_study)
    # Instancni promenne
    @first_name = first_name
    @last_name = last_name
    @year_of_study = year_of_study
  end

  # Definic metody
  def fullname
    # attr_reader zajistit vygenerovani metod first_name a last_name
    # ale samozrejme by bylo mozne vypsat @first_name a @last_name
    return "#{first_name} #{last_name}"
  end
end

student = Student.new("Pavel","Bory",4)
puts student.fullname
```

Zdrojový kód 2: Ukázka implementace třídy v jazyce Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

Z programovacího jazyka Ruby budou ve vyvíjené aplikaci zapotřebí i některé pokročilejší prvky, a proto jsou ty klíčové, zde v teoretické části stručně popsány. Cílem není popsat nepřeberné množství prvků a vlastností jazyka Ruby v úplnosti, za tímto účelem odkazuji na (2). Cílem je pouze stručně shrnout důležité poznatky ke klíčovým prvkům, které budou dále v této práci použity. Těmito prvky jsou moduly, třídní proměnné, modul `Singleton` pro implementaci stejnojmenného návrhového vzoru, metoda `eval` a možnost pomocí metaprogramování dynamicky za běhu programu doplnit na instanci třídy metodu.

Moduly umožňují sdružování metod, tříd a konstant. Moduly mají dva hlavní přínosy. Poskytují jmenný prostor a pomáhají předcházet konfliktu názvů a zároveň podporují použití tzv. mixin. (2 str. 73)

Moduly do jisté míry snižují potřebu využívání dědičnost při implementaci programů v Ruby. Moduly mohou definovat i samostatné metody, které však přímo nenáleží konkrétní

třídy, přičemž z modulů nelze vytvářet instance, ale moduly lze vložit do třídy pomocí metody `include`[1], čímž právě vznikne tzv. mixin, nebo propojení třídy a modulu.

Třídní proměnné napříč celým tělem třídy nebo modulu. Třídní proměnné musí být inicializovány před tím, než jsou použity. Třídní proměnná je sdílena napříč všemi instancemi třídy a je dostupná ze třídy samotné. Třídní proměnné patří nejvnitřnější obklopující třídě nebo modulu. Třídní proměnné použité na nejvyšší úrovni[2] jsou definovány do třídy `Object`[3] a chovají se jako globální proměnné. (2 str. 309)

Singleton jako návrhový vzor má za cíl zajistit, že jde zde pouze jedna instance třídy, která je globálně přístupná. Tento návrhový vzor zajišťuje, že instance třídy je vytvořena nejvýše jednou a všechny požadavky jsou směrovány právě na tento objekt. Navíc by tento objekt neměl být vytvořen dříve, než je skutečně zapotřebí. V tomto návrhovém vzoru je třída samotná zodpovědná za výše popsaná omezení, nikoliv uživatelé této třídy. (3)

V Ruby lze třídu jakožto Singleton implementovat pomocí zavolání metody `include`, kterou bude do třídy vložen Ruby modul `Singleton`[4], který implementaci výše popsaného návrhového vzoru zajistí.

Důležitou metodou, která bude v této práci zapotřebí je metoda `eval`[5], která jako parametr přijímá řetězec, který je následně interpretován jako standardní Ruby kód. Poslední klíčový prvek je z oblasti metaprogramování a jde o možnost v Ruby dynamicky za běhu programu přidat instanci metodu, čehož lze docílit tak, že se napíše klíčové slovo `def` pro definici metody, za které se uveden název proměnné obsahující příslušnou instanci, na níž má být metoda přidána a za ni se tečkou odděleně napíše název přidávané metody. Za tímto následuje tělo metody ukončené klíčovým slovem `end` stejně jako v případě standardní definice metody. Následující ukázka ilustruje použití třídní proměnné, metody `eval` a dynamického přidání metody `makeAction` na instanci třídy `Actor`.

---

[1] https://ruby-doc.org/core-2.4.0/Module.html#method-i-include

[2] Definované přímo v hlavním programu mimo vlastní definovanou třídu.

[3] http://ruby-doc.org/core-2.3.3/Object.html

[4] https://ruby-doc.org/stdlib-1.9.3/libdoc/singleton/rdoc/Singleton.html

[5] https://ruby-doc.org/core-2.4.0/Kernel.html#method-i-eval

```ruby
class World
  def self.actor_code
    @@actor_code
  end

  def initialize
    @actor = Actor.new
  end

  def loadActorActionMethod(code)
    @@actor_code =  code

    def @actor.makeAction
      eval(World.actor_code)
    end
  end

  def actorMakeAction
    @actor.makeAction
  end
end

class Actor
  def initialize
    @positionX = 0
    @positionY = 0
  end
end

world = World.new
world.loadActorActionMethod("puts 'actor makeAction ...' ")
world.actorMakeAction
```

Zdrojový kód 3: Ukázka použití metody eval spolu s přidáním metody do instance třídy Actor. Zdroj: autorem vytvořeno ve vývojém prostředí RubyMine.

3.2 Programovací jazyk Java

Existuje jen několik málo jazyků, kterým se podařilo zásadně přetvořit samotnou podstatu programování. Jeden jazyk z této elitní skupiny vynikl svým rychlým a zároveň širokým dopadem. Samozřejmě se jedná o jazyk Java. Není vůbec přehnané prohlásit, že původní vydání jazyka Java 1.0, se kterým přišla společnost Sun Microsystems, Inc., v roce 1995, způsobilo programátorskou revoluci. Tato revoluce radikálně posílila interaktivitu webového prostředí. Jazyk Java přitom ustavil nový standard návrhu počítačových jazyků. (4 str. 17)

Od vzniku původní verze Java 1.0 prošel tento jazyk řadou aktualizací z nich některé byly poměrně zásadní. Cílem této kapitoly není podrobně popsat vývoj jazyka Java od jeho počátku, ale pouze stručně popsat nejnovější vydání platformy Java SE 8 a související sadu nástrojů pro vývoj aplikací Java Development Kit označený JDK 8.

Sada JDK 8 představuje velmi významnou aktualizaci jazyka Java, protože zahrnuje principiální novou funkci jazyka: výraz lambda. Výrazy lambda budou mít dalekosáhlé dopady a promění, jak způsob návrhu programových řešení, tak i samotné psaní kódu Java. Výrazy lambda přitom mohou zjednodušit a zredukovat rozsah zdrojového kódu, který je potřeba při tvorbě určitých konstrukcí. Spolu s výrazy lambda se do jazyka dostal také nový operátor (`->`) a nový syntaktický prvek. Díky těmto výrazům si jazyk Java i nadále zachovává svou schopnost rychle a přesně formulovat algoritmy, jak to jeho uživatele očekávají. (4 str. 19)

Sada JDK 8 také integruje podporu technologie JavaFX, což je nová architektura grafického uživatelského rozhraní jazyka Java. Předpokládá se, že architektura JavaFX brzy sehraje důležitou roli téměř ve všech aplikacích Java a ve většině projektů s grafickým uživatelským rozhraním nahradí rozhraní Swing. Závěrem lze shrnout, že platforma Java SE 8 zásadně rozšiřuje možnosti jazyka a mění způsob, kterým vzniká kód jazyka Java. Její vliv bude patrný v celém ekosystému Java po dobu mnoha příštích let. (4 str. 19)

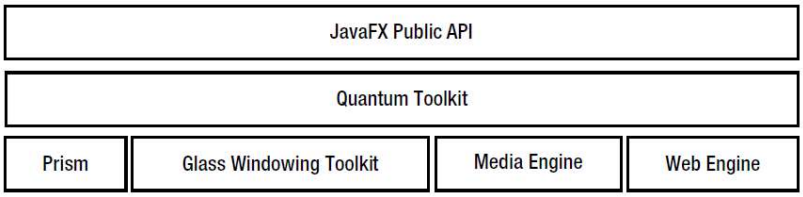
Vzhledem k rozšířenosti, kvalitě a možnosti integrace s Ruby, o které je hovořeno v následujících kapitolách a s přihlédnutím k možnosti implementovat grafické rozhraní moderní technologií JavaFX, která je představena v další kapitole, byl jazyk Java zvolen pro tvorbu aplikace v praktické části této práce. Pro podrobnější informace o jazyku Java odkazuji na (4).

3.3 JavaFX

JavaFX je další generací architektury grafického uživatelského rozhraní pro Jazyk Java. Aktuální verze je JavaFX 8, která je součástí JDK 8 a je uložena v balíčcích s prefixem `javafx`. Pokud bude v této práci hovořeno o JavaFX, pak je tím myšlena právě verze JavaFX 8. V této kapitole bude stručně představena architektura JavaFX a budou zde stručně představeny vybrané základní prvky JavaFX, které budou použity v praktické části. Pro podrobnější informace o JavaFX odkazuji na (5).

3.3.1 Architektura JavaFX

Základní obecný pohled na architekturu JavaFX ilustruje následující obrázek:



Obrázek 1: Základní pohled na architekturu JavaFX. Zdroj: (5 str. 2)

Zde je uveden stručný popis jednotlivých prvků architektury JavaFX: (5 str. 2)

- **JavaFX Public API**

  Grafické uživatelské rozhraní v JavaFX je konstruováno jako tzv. scene graph, což je kolekce vizuálních elementů nazývaných uzly, které jsou hierarchickým způsobem uspořádány. Scene graph je vytvořen za použití veřejného JavaFX API. Uzly ve scene graph mohou obsluhovat uživatelské vstupy. Mohou disponovat efekty, transformacemi a stavy. Typy uzlů v scene graph zahrnují jednoduché prvky uživatelského rozhraní jako např. tlačítka, textová pole, 2D a 3D tvary, obrázky, audio a video, webový obsah a grafy.
- **Prism** je hardwarově akcelerovaná grafická pipeline používaná pro renderování scene graph. Pokud není hardwarově akcelerované renderování k dispozici na dané platformě, pak je použita technologie Java 2D[1] jakožto mechanismus pro renderování.
- **Glass Windowing Toolkit** poskytuje grafické služby např. pro tvorbu oken a časovače za použití nativního operačního systému. Glass Windowing Toolkit je rovněž zodpovědný za správu front událostí. V Java FX jsou fronty událostí spravovány jedním vláknem na úrovni operačního systému. Toto vlákno se nazývá JavaFX Application Thread. JavaFX vyžaduje, aby aktivní scene graph byl modifikován pouze z JavaFX Application Thread. Prism v rámci akcelerace procesu renderování používá samostatné vlákno a synchronizace se scene graph probíhá pomocí tzv. pulse events.
- **Media engine** je zodpovědný za poskytování podpory médií jako je např. přehrávání audio a video v JavaFX. Využívá dostupných kodeků na dané platformě. Media

[1] https://docs.oracle.com/javase/8/docs/technotes/guides/2d/

engine využívá samostatné vlákno, přičemž používá Java FX Application Thread pro synchronizaci se scene graph. Media engine je založen na GStreamer[1], což je open source multimedia framework.

- **Web engine** je zodpovědný za zpracování webového obsahu (HTML) vloženého do scene graph, přičemž Prism zajišťuje renderování webového obsahu. Web engine je založen na open source web browser engine WebKit[2]. Je podporováno HTML, CSS, JavaScript a DOM.
- **Quantum toolkit** je abstrakcí nad komponentami nižší úrovně – Prism, Glass, Media Engine a Web Engine. Zároveň zajišťuje koordinaci mezi těmito komponentami.

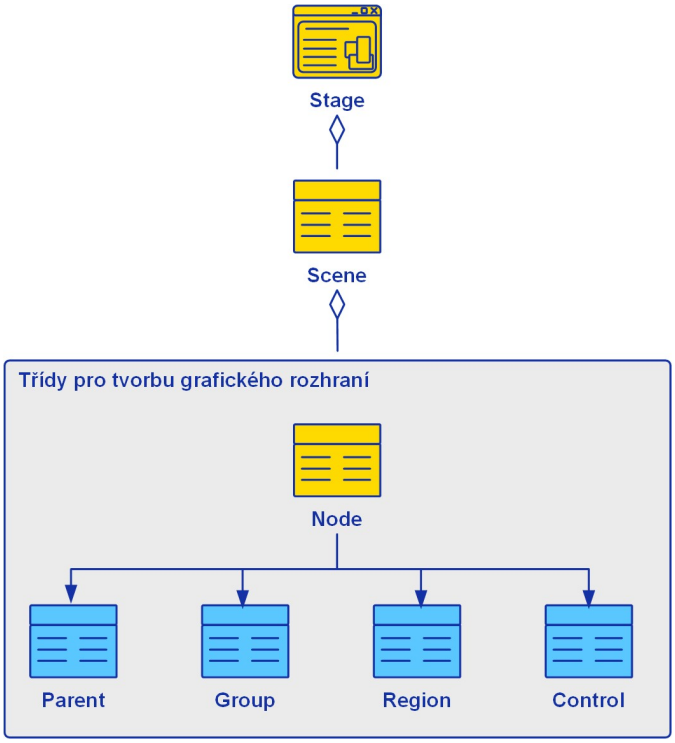
3.3.2 Základní prvky JavaFX

Jedním ze základních prvků JavaFX je tzv. stage, který definuje prostor, na kterém se může nacházet scene. Scene je kontejnerem pro položky grafického rozhraní. JavaFX prvky stage a scene zapouzdřuje třídami `Scene` a `Stage`. Při vytváření JavaFX aplikace je zapotřebí do stage přidat alespoň jeden objekt typu `Scene.`

Stage je kontejner nejvyšší úrovně. Všechny aplikace JavaFX mají automaticky přístup k jednomu objektu stage, který se označuje jako primary stage. Primary stage poskytuje systém runtime při spuštění aplikace JavaFX. (4 str. 580)

[1] https://gstreamer.freedesktop.org/

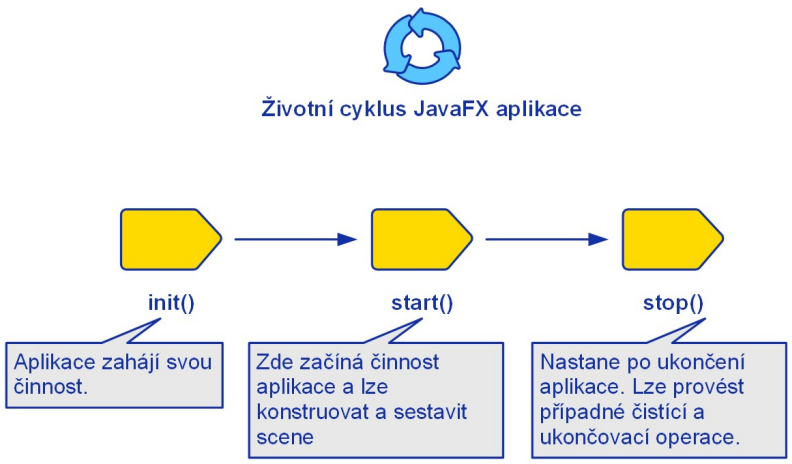
[2] https://webkit.org/



Obrázek 2: Základní prvky JavaFX. Zdroj: autorem vytvořeno v programu MS Visio.

Jednotlivé prvky scény se označují jako uzly. Uzlem je třeba příkazové tlačítko. Uzly se však mohou skládat i ze skupiny uzlů. Uzel kromě toho může mít svůj podřízený uzel. V tomto případě se uzel s podřízenou položkou označuje jako nadřazený uzel nebo větvený uzel. Uzly bez podřízených uzlů jsou koncové uzly a nazývají se listy. Kolekce všech uzlů na scéně se označuje jako scene graph, který tvoří strom. Graf scény obsahuje jeden speciální uzel, kterému se říká kořenový uzel. Jedná se o uzel nejvyšší úrovně a zároveň jediný uzel v grafu scény které nemá svůj nadřazený uzel. S výjimkou kořenového uzlu tedy všechny uzly mají své nadřazené uzly. Všechny uzly pak přímo nebo nepřímo pocházejí od kořenového uzlu. Základní třída všech uzlů se nazývá `Node`. Existuje několik dalších tříd, které jsou přímými nebo nepřímými podtřídami třídy `Node`. Patří k nim mimo jiné třídy `Parent`, `Group`, `Region` a `Control`. (4 str. 581)

JavaFX aplikace musí být potomkem třídy `Application` z package `javafx.application`. Třída Application definuje tři metody životního cyklu aplikace, které lze při implementaci aplikace překrýt.



Obrázek 3: Životní cyklus JavaFX aplikace. Zdroj: autorem vytvořeno v programu MS Visio.

Zde je uveden stručný popis těchto tří metod: (4 stránky 582-585)

- **`init()`**

  K jejímu volání dochází, když aplikace zahájí svou činnost. Tato metoda zajišťuje různé inicializace. Nemůže však vytvořit stage ani konstruovat scene. Pokud není potřeba nic inicializovat, tuto metodu není nutné překrývat, protože je k dispozici výchozí verze.

- **`start()`**

  Po metodě `init()` je volána metoda `start()`. Zde začíná činnost aplikace a metoda dovoluje konstruovat a sestavit scene. Přijímá parametr typu `Stage`. Jedná se o primary stage, který poskytuje systém runtime. Tato metoda je abstraktní a je tedy nutné ji v aplikaci implementovat.

- **`stop()`**

  Po ukončení aplikace je volána metoda `stop()`. V této metodě lze provést případné čistící a ukončovací operace. Pokud takové činnosti nejsou nutné, lze využít prázdnou výchozí verzi.

Ačkoliv by bylo možné celé uživatelské rozhraní zkonstruovat přímo ve zdrojovém kódu je vhodné využít možnosti přehledně jej deklarativně definovat v samostatném XML souboru s příponou `.fxml`. K tomuto souboru je možné připojit sadu kaskádových stylů, kterými lze prvky uživatelského rozhraní stylovat obdobně jako v případě tvorby běžných webových stránek.

```
<?xml version="1.0" encoding="UTF-8"?>
<?import java.lang.*?>
<?import javafx.scene.control.Button?>
<?import javafx.scene.layout.BorderPane?>

<BorderPane xmlns:fx=http://javafx.com/fxml
    fx:controller="application.FXPrototypeController">
      <center>
            <Button Text="OK"/>
      </center>
</BorderPane>
```

Zdrojový kód 4: Ukázka .fxml souboru. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Zde je uvedena ukázka implementace jednoduché aplikace, který zobrazí okno, ve kterém bude na primary stage zobrazena scene definovaná v předchozí ukázce zdrojového kódu č. 4.

```
package application;

import javafx.application.Application;
import javafx.fxml.FXMLLoader;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.layout.BorderPane;


public class Main extends Application {
  @Override
  public void start(Stage primaryStage) {
    try {
      BorderPane root = (BorderPane) FXMLLoader.load(
        getClass().getResource("FXPrototype.fxml"));
      Scene scene = new Scene(root,10,10);
      scene.getStylesheets().add(getClass()
        .getResource("application.css").toExternalForm());
      primaryStage.setScene(scene);
      primaryStage.show();
    } catch(Exception e) {
      e.printStackTrace();
    }
  }

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

```
}
```

Zdrojový kód 5: Ukázka zdrojového kódu jednoduché JavaFX aplikace zobrazující jedno okno s tlačítkem definovaným v rámci BorderPane v .fxml souboru z ukázky zdrojového kódu č.4. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Kromě možnosti definovat grafické rozhraní v `.fxml` souboru lze využít i technologii FXGraph, která je Domain-Specific-Language pro definování objektového grafu Java FX aplikace. Pro více informací o FXGraph odkazuji na (6)

Následující ukázka ilustruje jaký `.fxgraph` soubor odpovídá `.fxml` souboru ve výše uvedené ukázce zdrojového kódu č. 4.

```
import javafx.scene.control.Button
import javafx.scene.layout.BorderPane
import application.FXPrototypeController

component FXPrototype controlledby FXPrototypeController {
  BorderPane {
    center : Button
    {
      Text : "OK"
    }
  }
}
```

Zdrojový kód 6: Ukázka .fxgraph souboru, ze kterého je generován .fxml soubor. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

V praktické části bude grafické rozhraní implementováno právě pomocí technologie JavaFX, přičemž bude použita i technologie FXGraph pro tvorbu prvků uživatelského rozhraní a jejich rozložení.

3.4 JRuby

JRuby je dalším interpretem Ruby. Spouští ten samý kód Ruby jako ten spouštěný standardním Ruby interpretem. Zároveň je však lepším interpretem Ruby. Získáváme např. možnost použití vícevláknového prostředí, které využije všechna jádra procesoru a zároveň virtuální stroj, který byl laděn více než jeden a půl dekády. JRuby je zároveň defacto dalším prostým `.jar` souborem. Není tedy zapotřebí instalovat další běhové prostředí, pokud je v operačním systému již nainstalován virtuální stroj pro jazyk Java. (7 str. 14)

JRuby lze získat z oficiálních stránek http://jruby.org/download, kde jsou k dispozici zdroje pro instalaci na nejrozšířenější operační systémy. Pro operační systém Windows je k dispozici standardní instalátor. Po řádně provedené instalaci získáme především JRuby kompilátor, který umožní kompilovat zdrojový kód napsaný v jazyce Ruby do standardních

.jar souborů. Kromě kompilátoru je po instalaci k dispozici i JRuby interpret, kterým lze spouštět programy napsané v jazyce Ruby a např. i interaktivní konzoli jirb. Pro tuto práci je ale nejvíce podstatný kompilátor a interpret. Kompilátor lze spustit z příkazové řádky pomocí příkazu `jruby -S jrubyc`. Za tento se uvádí název souboru obsahujícího kód v Ruby, který má být zkompilován. Pro kompilaci ukázky zdrojového kódu č. 1 z kapitoly 3.1 lze tedy použít příkaz `jruby -S jrubyc ukazka.rb`.

Výstupem kompilátoru je standardní Java `.class` soubor (v touto případě tedy `ukazka.class`), obsahující metodu `main()`, který lze rovnou spustit pomocí příkazu `java` z příkazové řádky. Nicméně je zapotřebí příkazu `java` předat v parametru `-cp` cestu k souboru `jruby.jar`[1] protože zkompilovaný program je závislý na některých částech JRuby. Ukázkový soubor bychom tedy mohli spustit příkazem `java -cp .; C:\jruby-9.1.6.0\lib\jruby.jar ukazka.class`.

3.5 Integrace JRuby a Java

Na první pohled by se mohlo zdát, že integrace mezi Java a Ruby pomocí JRuby spočívá v pouhých voláních mezi programy nebo prostou kompilací Ruby programu do jazyka Java a následným spuštěním, či spuštěním, respektive interpretací programu napsaného v Ruby z programu implementovaného v Java. Toto však není pravda a v rámci této kapitole budou popsány možné způsob vcelku úzké integrace těchto dvou jazyků. Bude popsána, jak možnost používat kód implementovaný v Java z programu implementovaného v Ruby, tak i obrácený postup, tedy jak používat kód implementovaný v Ruby z programu implementovaného v Java, na což bude kladen větší důraz, protože tento postup bude ve zbytku práce podstatně důležitější.

Je možná na místě poznamenat, že JRuby má řadu uplatnění i v běžném vývoji software a nejde pouze o okrajovou záležitost, která bude proprietárně použita při realizaci studentského projektu. Autor této práce s ní např. setkal i během své praxe jakožto s technologií, která umožňovala tvorbu automatizačních skriptů pro robustní informační systém implementovaný v Java. Rovněž z literatury, ze které tato práce vychází je zmíněno

[1] Tento soubor je při standardní instalaci umístěn ve složce lib tam, kam bylo JRuby nainstalováno.

několik možností praktického využití. Níže uvedené příklady možného použití předpokládají, že v daných situacích je aplikace vyvíjena v Ruby či Ruby on Rails[1], přičemž nebudeme rozebírat důvody proč může být někdy vhodné v těchto technologiích implementovat program či např. webové aplikace. Možnosti praktického využití JRuby jsou např. následující: (7 stránky 31-33)

- Pokud je vyvíjen program v jazyce Ruby a je potřeba použít knihovnu nebo knihovny (např. pro práci s PDF dokumenty knihovnu iText[2]), které nejsou přímo pro Ruby k dispozici, ale existují vhodné kvalitní knihovny pro Java.
- Při vývoji webové aplikace s použitím frameworku Ruby on Rails je možné přistoupit např. k databázi pomocí příslušných Java knihoven. Zde se domnívám, že analogicky by se dalo říct, že by mohlo být někdy vhodné mít možnost prostřednictvím Java knihoven být schopen provést volání jiného systému, který je implementován v Java, a přitom neumožňuje komunikaci prostřednictvím standardní technologie, pro kterou by zároveň existovala kvalitní implementace pro Ruby.
- Pokud je vyvíjen program v Ruby a potřeba implementovat např. vizualizaci SVG obrázků nebo vytvořit jednoduché grafické rozhraní, k čemu má Java na rozdíl od Ruby lepší prostředky.
- Ve chvíli, kdy je implementována Java aplikace, ve které je zapotřebí umožnit uživatelům používat nějaký skriptovací jazyk.

3.5.1 **Používání Java z Ruby**

Aby bylo možné z Ruby použít nějakou třídu z Java, pak je potřeba na začátku programu zavolat metodu `require` s parametrem `"java"`, čímž dojde k načtení potřebných rozšíření Ruby, aby bylo možné Java třídy začít vůbec používat. Poté již lze pracovat s Java třídami obdobně jako, kdyby šlo o třídy implementované přímo v Ruby. Zde je ukázka, jak z Ruby programu vytvořit instanci Java třídy `java.util.ArrayList`, do které je vloženo několik čísel a ta jsou poté vypsána na konzoli. Pro účely této ukázky byla záměrně zvolena takto jednoduchá třída, protože mimo jiné je její použití vcelku přímočaré a zároveň je dále při průchodu pomocí cyklu zajímavé vidět funkčnost iterátoru `each` z Ruby. Pokud vytváříme

[1] http://rubyonrails.org/

[2] http://www.itextpdf.com

instanci Java třídy, pak na ni stejně jako na standardní Ruby třídě voláme konstruktor, tedy metodu `new`, přičemž její název uvádíme včetně package, ve které se Java třída nachází. Je potřeba říct, že tato ukázka bude funkční proto, že JRuby má na classpath třídy, které jsou součástí Java SE a můžeme je tedy takto přímo používat.

```ruby
require "java"

kolekce = java.util.ArrayList.new
# Metoda p je inspekcni metoda vypisujici podrobnejsi popis daneho objektu
p kolekce

kolekce.add(1)
kolekce.add(2)
kolekce.add(3)
kolekce.add(4)
kolekce.add(5)

kolekce.each do |item|
  p item
end
```

Zdrojový kód 7: Ukázka použití Java z Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

JRuby rovněž řeší převody mezi typy jazyka Ruby a typy jazyka Java. V tomto směru se lze v běžných situacích spolehnout na to, že JRuby zajistí příslušný převod, aniž by programátor musel v tomto směru provádět další implementaci.

Pokud je zapotřebí z Ruby např. vytvořit instanci Java třídy nebo zavolat statickou metodu, pak je nutné zajistit, že bude přidána na classpath (pokud se nejedná o třídu, která je standardně na classpath přidána samotným JRuby ve výchozím nastavení). V následující ukázce je uvažována triviální Java třída mající jednu metodu, která spočte medián ze setříděného seznamu celých čísel, přičemž, pokud je počet prvků sudý, pak je použito celočíselné dělení při výpočtu mediánu.

```java
package console.prototype;
import java.util.List;

public class Statistics {
  public long median(List<Long> nums){
    if(nums.size() % 2 == 1){
      return nums.get(nums.size() / 2);
    }else{
      long first = nums.get(nums.size() / 2 - 1);
      long second = nums.get(nums.size() / 2);
      return (first + second) / 2;
    }
  }
}
```

Zdrojový kód 8: Ukázka Java třídy, která bude v ukázce zdrojového kódu č. 9 použita z Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Aby bylo možné tuto třídu použít z Ruby, tak je zapotřebí provést standardní Java kompilaci výše uvedené třídy do `.jar` souboru, ať již přímo z příkazové řádky pomocí `javac`, nebo prostřednictvím vývojového prostředí. Následující ukázka předpokládá, že tento soubor bude pojmenován `statistics.jar`. V případě použití třídy z Ruby je zapotřebí uvést její název, který bude ve tvaru `Java::NazevPackage::NazevTridy`. Pomocí dvojice dvojteček v Ruby je značeno, že se třída nachází v Ruby modulu. Následujícím způsobem by mohl vypadat soubor `statistics.rb` s programem v Ruby používající třídu `Statistics`.

```ruby
require "java"
statistics = Java::ConsolePrototype::Statistics.new
puts "Median([1,2,3,9,10,11]): #{statistics.median([1,2,3,9,10,11])}"
puts "Median([1,2,3,5,6]): #{statistics.median([1,2,3,5,6])}"
```

Zdrojový kód 9: Ukázka použití Java třídy z ukázky zdrojového kódu č. 8 v Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

Výše uvedený zdrojový kód spuštěný z příkazové řádky pomocí příkazu
`jruby statistics.rb` by vyvolal výjimku
`NameError: missing class name ('console.prototype.Statistics').`

Tato výjimka by byla způsobena tím, že JRuby nezná, respektive se mu nepodaří nalézt třídu `console.prototype.Statistics`. Obecně je zapotřebí vlastní třídy respektive např. externí knihovny přidat na classpath, čehož lze docílit několik způsoby. Prvním z nich je při spuštění programu předat do JVM pomocí parametru `-J`[1] parametr `-cp` následovaný názvem `.jar` souboru, který má přidat na classpath při spuštění programu `statistics.rb`. Již

[1] Následující parametr je předán do Java Runtime.

korektní spuštění výše uvedeného programu by tedy mohlo vypadat takto: `jruby -J-cp Statistics.jar statistics.rb`.

Další možností je přidat příslušné cesty do systémových proměnných `RUBYOPT` a `CLASSPATH`, přičemž JRuby v `RUBYOPT` hledá kód implementovaný v Ruby a v `CLASSPATH` hledá Java třídy. Přidání do `CLASSPATH` lze provést i ve zdrojovém kódu Ruby programu přidáním prvku do globální proměnné `$CLASSPATH`. Kromě tohoto lze i použít přímo metody `require` nebo `require_relative`. (7 stránky 39-40)

Metoda `require_relative` umožní pohodlně přidat statistics.jar soubor, který se nachází ve stejné složce jako `statistics.rb` souboru, pokud je ji předán název tohoto souboru. Této metodě lze kromě názvu souboru předat i relativní cestu, pokud by se soubor nacházel v další složce, např. tedy zavolat `require_relative "lib/statistics.rb"`, pokud by se soubor `statistics.rb` nacházel ve složce `lib`. Výše uvedenou ukázku bychom tedy mohli upravit a spustit pouhým příkazem `jruby statistics.rb`.

3.5.2 **Používání Ruby z Java**

Nejjednodušším způsobem, jak lze z Java spustit kód v Ruby je pomocí metody `runScriptlet()`, které je implementována v třídě `ScriptingContainer`[1]. Této metodě je možné předat jeden parametr typu `java.lang.String`, který bude obsahovat kód v Ruby, který chceme vykonat. (7 str. 62)

Aby bylo možné třídu `ScriptingContainer` v Java aplikaci použít, je zapotřebí přidat na classpath knihovnu `jruby.jar`, která je součástí nainstalovaného JRuby. Metoda `runScriptlet()` má návratový typ `java.lang.Object`. Návratovou hodnotou metody je výsledek posledního provedeného výrazu, přičemž JRuby tento výsledek konvertuje na Java objekt. Ukázková konzolová Java aplikace, která provede zavolání Ruby kódu prostřednictvím `ScriptingContainer` by mohla vypadat následujícím způsobem.

```
package console.prototype;

import org.jruby.embed.ScriptingContainer;

public class Program {
```

[1] http://jruby.org/apidocs/org/jruby/embed/ScriptingContainer.html

```
  public static void main(String[] args) {
    ScriptingContainer container = new ScriptingContainer();
    container.runScriptlet("puts 'Hello World from JRuby'");
  }
}
```

Zdrojový kód 10: Ukázka použití Ruby z Java pomocí ScriptingContainer. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Z výše popsaného je velmi podstatné to, že pomocí metody `runScriptlet()` je možné spustit v podstatě libovolný kód v Ruby. Pokud je připraven soubor se zdrojovým kódem v Ruby, pak pomocí metody `runScriptlet()` lze zavolat metodu `require`, která příslušný obsah souboru s daným názvem vykoná, což nemusí nutně znamenat, že se provede určitý program a ten dodá rovnou výstup, ale např. se tím může připravit možnost později pomocí instance `ScriptingContainer` spouštět metody a pracovat s třídami, které jsou v onom souboru se zdrojovým kódem v Ruby připraveny. V Java aplikaci bude vhodné pro přehlednost oddělit soubory se zdrojovým kódem v Ruby do samostatné složky či více složek dle rozsahu aplikace. Aby bylo možné poté příkaz `require` použít, je potřeba využívat ještě jednu metody třídy `ScriptingContainer`, a to metodu `setLoadPaths()`, která umožní přidat seznam cest, na kterých se při použití příkazu `require` pokusí JRuby nalézt požadované soubory ke zpracování.

V případě, že bude vhodné volané metodě v Ruby předat parametry, pak je to možné učinit skrze globální proměnné v Ruby, které lze předat pomocí metody `put` na instanci třídy `ScriptManager`. Praktičtější a z hlediska objektově orientovaného programování výhodnější postup je být schopen předat parametr (objekt) přímo volané metodě, který v kontextu Ruby programu nebude globální a zároveň bude jak z pohledu Java, tak Ruby programu disponovat tím samým chováním, na což lze v terminologii Java nahlížet tak, že bude implementovat rozhraní a v terminologii Ruby bude obsahovat modul, protože jazyk Ruby nemá zcela odpovídající ekvivalent tomu, co je v Java rozhraní.

Následující ukázka shrnuje základní teoretické poznatky, jak lze pracovat s Ruby objekty a jejich metodami z Java aplikace za využití objektově orientovaného přístupu s použitím rozhraní.

Ukázka ilustruje následující situaci. V této situaci je uvažována třída `Actor`, která reprezentuje obecně nějakého aktéra, ať již postavu člověka, či např. vozidlo pohybující se po dvourozměrné mřížce. Tento aktér má souřadnice, na kterých se aktuálně vyskytuje a

bude mít možnost vykonat tah, ve kterém zvolí, jakým směrem se posune. Typ zvoleného tahu bude reprezentovat Java `Enum ActionType`.

```
package console.prototype;

public enum ActionType {
  MOVE_LEFT, MOVE_RIGHT, MOVE_UP, MOVE_DOWN
}
```

Zdrojový kód 11: Definice výčtového typu ActionType, který bude později použit v Java rozhraní při integraci Java a Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Dále bude existovat Java rozhraní definující požadované chování, které je od aktéra očekáváno.

```
package console.prototype;

public interface IActor {
	long getPositionX();
	long getPositionY();
	void setPositionX(long position);
	void setPositionY(long position);
	ActionType makeAction();
}
```

Zdrojový kód 12: Definice Java rozhraní IActor, která bude použito při integraci Java a Ruby. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Z hlediska Ruby bude připraven soubor nazvaný `actor.rb`, který bude obsahovat definici třídy `Actor`. V té je možné pro zjednodušení implementace na začátku zavolat metodu `java_import`[1], po jejímž vykonání není potřeba při práci s typy `IActor` a `ActionType` uvádět názvy včetně package, respektive modulu, do kterého patří. Tato třída `Actor` bude definovat metody vyžadované implementací rozhraní `IActor`, které pro následnou funkčnost Java programu budou vloženy pomocí metody `include` do třídy `Actor`. Tento krok je nezbytný, aby bylo později možné provést konverzi mezi datovým typem `Actor` v Ruby na typ `IActor` v Java programu. S prvky výčtového typu z Java pracuje Ruby jako s konstantami, proto když metoda `makeAction()` vrací pro zjednodušení vždy tah typu `MOVE_LEFT`, lze jej zkrátka použít jako Ruby konstantu `MOVE_LEFT` z modulu `ActionType`.

[1] https://github.com/jruby/jruby/wiki/CallingJavaFromJRuby

```
require "java"
java_import "console.prototype.IActor"
java_import "console.prototype.ActionType"

class Actor
  include IActor

  def initialize
    @positionX = 0
    @positionY = 0
  end

  def setPositionX(position)
    @positionX = position
  end

  def setPositionY(position)
    @positionY = position
  end

  def getPositionX()
    return @positionX
  end

  def getPositionY()
    return @positionY
  end

  def makeAction()
    return ActionType::MOVE_LEFT
  end
end
```

Zdrojový kód 13: Implementace Ruby třídy Actor, která implementuje rozhraní IActor a používá v Javě definovaný výčtový typ ActionType. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

V Java programu, pak s touto Ruby třídou lze pracovat tak, že je zavolán konstruktor třídy `Actor` a výsledek volání metody `runScriptlet()` přetypován na typ `IActor`, což je právě možné proto, že v Ruby třídě `Actor` byl provedena ona metoda `include` s parametrem určujícím příslušné rozhraní. Dále lze na proměnné `actor` volat příslušné metody a `ScriptingContainer` respektive JRuby se postará o jejich provedení na Ruby objektu typu `Actor`, který byl vytvořen na začátku ukázky.

```
package console.prototype;

import java.util.Arrays;
import org.jruby.embed.ScriptingContainer;

public class Program {
	public static void main(String[] args) {
		ScriptingContainer container = new ScriptingContainer();
		container.setLoadPaths(Arrays.asList("libs"));
```

```
            container.runScriptlet("require 'actor.rb'");

            IActor actor = (IActor) container.runScriptlet("Actor.new");
            System.out.println(actor.getPositionX());
            System.out.println(actor.getPositionY());

            actor.setPositionX(5);
            actor.setPositionY(8);
            System.out.println(actor.getPositionX());
            System.out.println(actor.getPositionY());

            ActionType actType = actor.makeAction();
            System.out.println(actType);
        }
}
```

Zdrojový kód 14: Použití Ruby třídy implementující rozhraní IActor v Java prostřednictvím ScriptingContainer. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

3.5.3 Strategie propojení Java a Ruby

Existuje řada postupů jak světy Java a Ruby propojit. Můžeme předat Java třídu do Ruby programu, vytvořit Ruby třídu implementující / rozšiřující Java typ, nebo zkrátka použít předávání řetězců. Neexistuje jedno obecně správné řešení pokrývající všechny možné situace. (7 str. 74)

Obecně lze říci, že je vhodné se snažit minimalizovat počet volání, které se mezi Java a Ruby provádí tam, kde je to účelné a zároveň toho lze rozumným objektově orientovaným návrhem efektivně dosáhnout. Např. může být méně efektivní z Ruby programu v cyklu tisíckrát zavolat Java metodu a postupně z ní získávat jednotlivé výstupy, ve srovnání s jedním voláním, které v Java programu připraví ono větší množství dat, a to v jednom objektu předá do Ruby.

Z hlediska předávání dat z Ruby do Java máme tři základní možnosti: (7 stránky 75-76)

- Vrátit Ruby objekt implementující Java rozhraní
- Vrátit Ruby objekt rozšiřující Java třídu
- Vytvořit Java objekt v Ruby a tento vrátit

První dvě možnosti mají společné to, že vrací Ruby objekt spjatý s běhovým prostředím JRuby, ze kterého vzešly. Pokud na těchto objekt zavoláme z Java metody, pak se provedou na ono Ruby objektu v běhovém prostředí JRuby, odkud pocházejí. Zde by mohly nastat komplikace v případě, že bychom implementovali vícevláknovou aplikaci, protože

předávání objektů mezi různými vlákny by mohlo vyústit v chybu běhového prostředí JRuby, pokud bychom nepoužívali THREADSAFE mód, ale touto již vcelku pokročilou problematikou se práce nezabývá, protože zatím není předpoklad, že by bylo zapotřebí v aplikaci disponovat možností pracovat ve více vláknech. Třetí způsob je odolnější vůči chybám spojených s tvorbou vícevláknových aplikací a může být efektivnější kvůli minimalizaci volání mezi jazyky Java a Ruby. Nicméně je méně elegantní a zvyšuje nároky na implementaci programu na straně Ruby. (7 stránky 75-76)

Osobně se budu snažit v této práci využívat především první a druhou možnost, jejichž základní funkčnost byla ověřena v rámci teoretické přípravy v této práci. V případě opodstatněných důvodů, např. výkonnostního charakteru aplikace budu postupovat pomocí třetí možnosti.

3.6 JUnit

Unit testy jsou automatizované testy, které verifikují funkcionalitu komponenty, třídy nebo metody. Unit testy jsou základním stavebním kamenem automatizovaného a regresního testování, které poskytují dlouhodobou stabilitu a spravovatelnost softwarového projektu. (8)

Pro tvorbu unit testů pro klíčové třídy JavaFX aplikace v této práci bude použit pro Java aplikace standardní, stabilní a dobře zdokumentovaný framework JUnit, který lze získat na oficiálních stránkách http://junit.org/junit4/.

Jednotlivé testy se pomocí JUnit implementují jako třídy, u kterých se pomocí anotací metod definují jednotlivé testy či se metody označí jako speciální metody, které mají být provedeny před či po provedení testů. Uvnitř testovacích metod se používají metody JUnit, kterými jsou ověřovány předpoklady, tzn. např. ověříme předpokládané chování metody pro dané vstupy.

V následující ukázce je definována třída reprezentující jednoduchou kalkulačka, za kterou následuje ukázka unit testu implementovaného s použitím frameworku JUnit, ve které je pomocí metody `assertEquals()` ověřováno, že metoda evaluate třídy `Calculator` funguje korektně.

```
public class Calculator {
  public int evaluate(String expression) {
    int sum = 0;
    for (String summand: expression.split("\\+"))
```

```
      sum += Integer.valueOf(summand);
    return sum;
  }
}
```

Zdrojový kód 15: Třída Calculator, která bude testováno pomocí unit testu implementovaného pomocí frameworku JUnit. Zdroj: (9)

```
import static org.junit.Assert.assertEquals;
import org.junit.Test;

public class CalculatorTest {
  @Test
  public void evaluatesExpression() {
    Calculator calculator = new Calculator();
    int sum = calculator.evaluate("1+2+3");
    assertEquals(6, sum);
  }
}
```

Zdrojový kód 16: Ukázka implementace unit testu třídy Calculator z ukázky zdrojového kódu č. 15 s použitím frameworku JUnit. Zdroj: (9)

3.7 Apache Log4j 2

Logování je běžnou součástí aplikací. V této práci bude použito logování především za účelem ladění aplikace. Pro Java aplikace existuje řada možností, jak logování implementovat. V této práci bude použito logování s použitím v kontextu Java aplikací velmi rozšířeného a dobře zdokumentovaného frameworku Apache Log4j 2, který lze získat na oficiálních stránkách https://logging.apache.org/log4j/2.x/download.html. Tento lze do do JavaFX projektu zakomponovat různými způsoby. Pro tuto práci budou použity přímo stažené knihovny Apache Log4j 2, které budou projektem přímo referencovány[1]. Samotné logování lze poté v základní podobě ve zdrojovém kódu implementovat tak, že je zavoláním metody `getLogger()` na třídě `LogManager` získána instance třídy implementující rozhraní `Logger`. Na této instanci implementující rozhraní `Logger`, lze volat metody provádějící logování dle určitých kategorií, pro které lze konfigurovat jaké prostředky mají být pro zápis logovacích informací použity. Mezi tyto metody patří `error()`, `info()`, `debug()` a `trace()`. Zde je uvedena ukázka získání instance implementující rozhraní `Logger` a na ni zavolání logovacích metod `info()` a `debug()`.

```
package application.world;
```

[1] Další možnost je např. s použitím nástrojů Maven, Ivy či Gradle, přičemž artefakty pro tyto nástroje lze získat zde: https://logging.apache.org/log4j/2.x/maven-artifacts.html

```java
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public class World {
  private static final Logger logger = LogManager.getLogger(World.class);

  public World(Canvas canvas, File worldFile) throws IOException {
    logger.info("Creating new World instance");
    worldGrid = new WorldGrid(canvas);
    logger.debug("World Grid created");
    worldSettings = WorldSettings.load(worldFile);
    logger.debug("World Settings loaded from file");
    jrubyContainer = new JRubyContainer();
    logger.debug("JRubyContainer initialized");
  }
...
```

Zdrojový kód 17: Výňatek zdrojového kódu ilustrující použití logování. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Logování lze konfigurovat řadou způsobů, mezi které patří např. soubory ve formátech XML, JSON či YAML. Dále lze logování konfigurovat i přímo ve zdrojovém kódu aplikace. Pro více informací o konfiguraci logování odkazuji na (10).

V této práci bude použitý základní jednoduchý způsob konfigurace logování s použitím XML souboru, ve kterém bude definováno, které úrovně logovacích zpráv, kterých prvků mají být zapisovány do konzole, a které to prostého textového souboru v daném umístění.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="WARN">
  <Properties>
  <Property name="filename">logs/app.log</Property>
  </Properties>
  <Appenders>
    <Console name="Console" target="SYSTEM_OUT">
      <PatternLayout pattern="%d{HH:mm:ss} %-5level %logger{36} - %msg%n"/>
    </Console>
    <File name="FileLog" fileName="${filename}" append="false">
      <PatternLayout pattern="%d{HH:mm:ss} %-5level %logger{36} - %msg%n"/>
    </File>
  </Appenders>
  <Loggers>
    <Logger name="application.Main" level="debug" additivity="false">
      <AppenderRef ref="FileLog"/>
      <AppenderRef ref="Console"/>
    </Logger>
    <Logger name="application.world.World" level="debug" additivity="false">
      <AppenderRef ref="FileLog"/>
      <AppenderRef ref="Console"/>
    </Logger>
    <Root level="trace">
      <AppenderRef ref="FileLog"/>
      <AppenderRef ref="Console"/>
    </Root>
```

```
  </Loggers>
</Configuration>
```

Zdrojový kód 18: Ukázka konfigurace logování. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4 Praktická část

4.1 Analýza požadavků

V této kapitole jsou popsány funkční a nefunkční požadavky na v rámci této práce vyvíjený software. Na zde definované požadavky navazují následující kapitoly, ve kterých je postupně vytvořen návrh architektury aplikace, popsány případy užití, vytvořen návrh grafického rozhraní aplikace a návrh klíčových prvků software.

Specifikace softwarových požadavků se obecně považuje za počátení vstup k objektové analýze a k následnému objektovému návrhu. Požadavky jsou základem všech systémů. Jsou v podstatě vyjádřením toho, co by měl systém dělat. Požadavky by měly být jediným vyjádřením, co by měl systém dělat, nikoli toho, jak by to měl dělat. To je nesmírně důležitý rozdíl. Můžeme určit, co by měl systém dělat a jaké chování by měl poskytovat, aniž bychom cokoli říkali o způsobu, jak bude dané funkce dosaženo. (11 stránky 76-78)

Požadavky se vyplatí rozdělit na funkční a nefunkční. Funkčním požadavkem je formulace toho, co by měl systém dělat – popisuje požadovanou funkci systému. Nefunkční požadavek je omezující podmínka uvalená na daný systém. Nefunkční požadavky specifikují, nebo spíše vymezují způsob, jímž bude systém implementován. (11 str. 80)

4.1.1 Funkční požadavky

1. Aplikace umožní uživateli pomocí programovacího jazyka Ruby programovat aktéra, který se dle zadaných jednoduchých příkazů bude pohybovat a interagovat s prvky rozloženými do čtvercové mřížky. Mřížka může obsahovat 0..n prvků, se kterými může aktér interagovat, nicméně některá pole mřížky mohou být prázdná a aktér na ně zkrátka může vstoupit, ale neposkytnou mu žádný vstup pro interakci.
2. Simulace naprogramované činnost aktéra bude fungovat na bázi jednotlivých tahů, kde v jednom tahu bude proveden jeden z povolených příkazů, který programu sdělí, co aktér v daném tahu udělá.

   2.1. Povolené příkazy, které může aktér provést jsou následující:

   - Posun o pozici doleva

- Posun o pozici doprava
- Posun o pozici nahoru
- Posun o pozici dolu
- Konec programu, čímž bude určeno, že se nebudou další tahy (příkazy) vykonávat

2.2. Kromě příkazů v bodu 2.2. může aktér provést následující operace, které mohou být součástí tahu, v jehož rámci můžou být provedeny i víckrát, ale nejde o operace, kterým by program pro aktéra definoval, co má udělat v následujícím tahu:

- Provedení textového výpisu
- Načtení vstupu z prvku mřížku, pokud to prvek umožňuje
- Zjištění aktuálních souřadnic aktéra
- Zjištění aktuálních souřadnic aktéra
- Zjištění aktuálních souřadnic prvků mřížky, se kterými je možné interagovat
- Základní práce s běžnými programovými konstrukcemi jazyka Ruby

3. Aplikace umožní uživateli pomocí konfiguračního souboru definovat obrázek aktéra a jeho výchozí pozici v mřížce, barvy jednotlivých čtverců mřížky a obrázky pro vybrané čtverce mřížky. Pro konfigurací dané prvky mřížky aplikace umožní definovat řetězcové hodnoty, které budou moci sloužit jako vstup pro program simulující chování aktéra.
4. Konfiguraci dle bodu 3 bude možné měnit za běhu aplikace výběrem příslušného konfiguračního souboru, ke kterému jsou přidruženy potřebné obrázky.
5. Aplikace poskytne uživateli při programování aktéra jednoduchý textový editor se zvýrazňováním syntaxe.
6. Aplikace umožní importovat program pro aktéra ze souboru .rb.
7. Aplikace umožní exportovat program pro aktéra do souboru .rb.
8. Aplikace umožní simulaci chování aktéra spustit, tak že v krátkých časových intervalech budou vykonávány jednotlivé tahy aktéra.
9. Aplikace umožní spuštění dle bodu 8 pozastavit a následně opět pokračovat.
10. Aplikace umožní simulaci chování aktéra krokovat, tak že uživatel pro vykonání následující tahu musí stisknout příslušné tlačítko.
11. Aplikace umožní simulaci resetovat tak, že se vrátí stav mřížky a aktéra vrátí do původního stavu, jak byl načten z konfiguračního souboru dle bodu 3.
12. Aplikace umožní zobrazení textového výstupu aktéra (pokud nějaký provede)

13. Aplikace umožní zobrazení ladících informací, ve kterých budou zachyceny důležité informace o stavu simulace.
14. Aplikace umožní uživateli zobrazení podrobností o prvcích mřížky, se kterými může aktér interagovat.
15. Aplikace bude informovat uživatele o chybových stavech, s tím že v prototypu budou pokryty alespoň implementačně jednodušší situace zachycení chybového stavu programu aktéra, který uživatel naprogramuje.

4.1.2 Nefunkční požadavky

1. Aplikace bude implementována s použitím technologie JRuby 9.1.6.0.
2. Aplikace bude implementována s použitím technologií Java 8 SE a Java FX 8.
3. Aplikace bude provozovatelná na nejrozšířenějších operačních systémech s tím, že prototyp bude vyvinut a otestován pro operační systém Windows 10 Pro.
4. Aplikace bude rozšiřitelná.
5. Aplikace implementována v anglickém jazyce, a to včetně zdrojového kódu.
6. Aplikace bude používat snadno konfigurovatelné logování do standardních prostředků jako je standardní výstup (konzole) a textový soubor.
7. Klíčové prvky implementace budou pokryty unit testy.

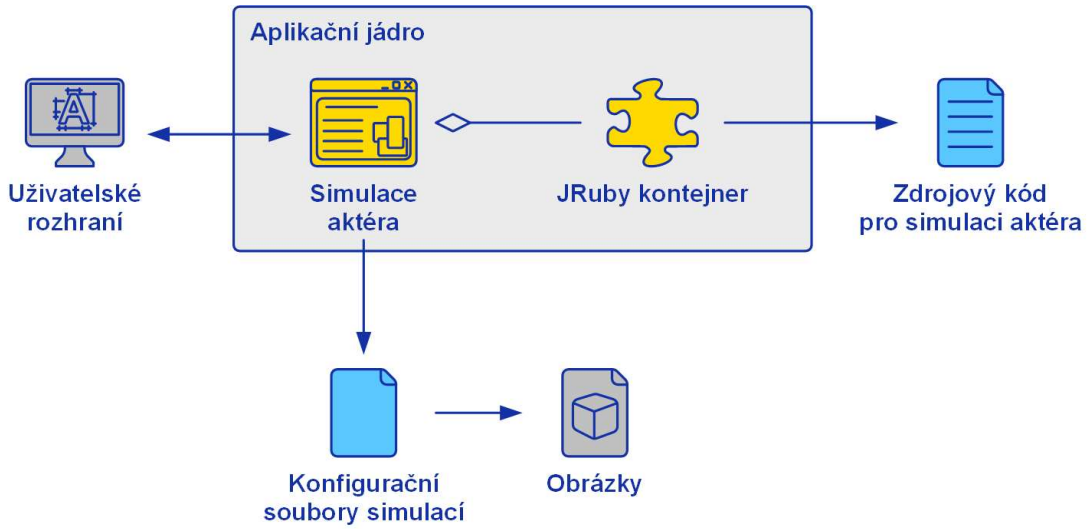
4.2 Architektura a návrh aplikace

V této kapitole je popsána navržená architektura aplikace. Dále je zde uveden seznam případů užití včetně jejich popisu. V neposlední řadě je zde zachycen i návrh uživatelského rozhraní a návrh tříd aplikace, jejichž popis dále rozvedu v kapitole věnující se vývoji aplikace.

Architektura softwaru je vyšší úrovní softwarového návrhu. Je to rámec, do něhož se vsazují detailnější součásti návrhu. Architektura by měla definovat hlavní stavební bloky programu. V závislosti na velikosti programu by měl být každý stavební blok samostatnou třídou nebo dokonce samostatným subsystémem složeným z několika tříd. Každý stavební blok je třídou nebo kolekcí tříd či rutin spolupracujících na vyšších funkcích, jako je třeba interakce s uživatelem, zobrazení webových stránek, interpretace příkazů, zapouzdření provozních pravidel nebo přístup k datům. Odpovědnost jednotlivých stavebních bloků by měla být definována jasně a srozumitelně. Stavební blok by měl mít jednu oblast odpovědnosti. O odpovědnostech ostatních bloků by měl vědět jen to nejnutnější. (12 stránky 68-69)

4.2.1 Obecný pohled na architekturu aplikace

Obecný pohled na architekturu zachycuje následují obrázek. Při návrhu architektury aplikace jsem nejdříve identifikoval klíčové logicky oddělené prvky a definoval vazby mezi nimi.

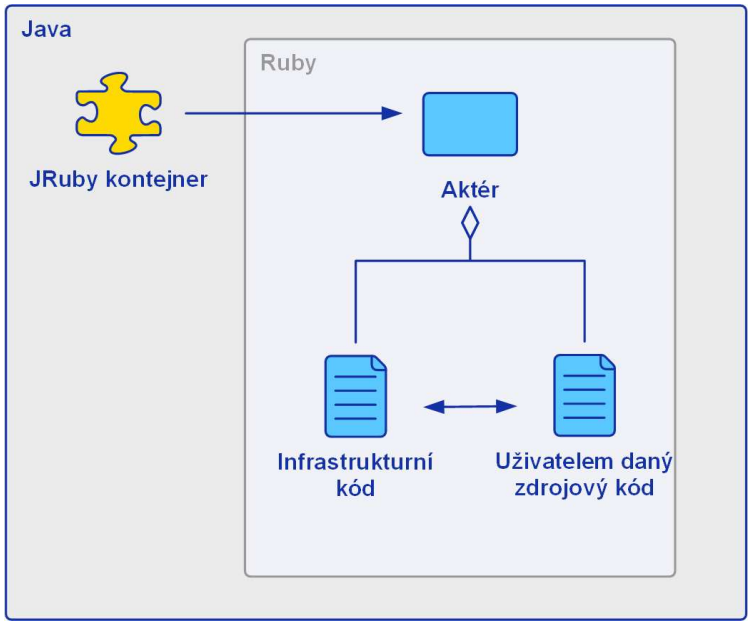


Obrázek 4: Obecný pohled na architekturu aplikace. Zdroj: autorem vytvořeno v programu MS Visio.

Samostatně stojí uživatelské rozhraní, které bude implementováno pomocí technologie Java FX. Toto bude zodpovědné zejména za získávání vstupů od uživatele a zobrazování mřížky

simulace aktéra. Druhou velmi důležitou části je tzv. Simulace, která spolu s JRuby kontejnerem tvoří hlavní jádro aplikace a budou tvořit největší část implementace. Část Simulace bude udržovat stav simulace, informace o zobrazené mřížce, o rozmístění prvků, pozici aktéra atd. Tato část bude v sobě obsahovat JRuby kontejner, ve kterém bude spouštěn uživatelem vytvořený zdrojový kód pro simulaci aktéra. V tomto budou udržovány příslušné Ruby objekty a s jeho pomocí bude s těmito komunikováno, respektive volány jejich metody, předávány jim parametry, a naopak získávány návratového hodnoty z uživatelem definovaného programu pro simulaci aktéra, aby bylo možné provést příslušnou požadovanou operaci jako např. posun aktéra po mřížce. Zároveň bude zapotřebí kromě uživatelem napsaného zdrojového kódu vytvořit i určitou základní infrastrukturu v Ruby, aby do ní bylo možné uživatel napsaný kód zasadit, což plyne z teoretických východisek, kde byly rozebírány možnosti a postupy integrace Java a Ruby. Proto je tomuto věnován samostatný prvek ve výše uvedeném obrázku, protože jde rovněž o podstatnou součást implementace aplikace. Posledními zmíněnými samostatnými prvky jsou konfigurační soubory a k nim přidružené obrázky. Konfigurační soubory budou definovat prvky mřížky, výchozí pozici aktéra atd. Tyto budou ve formátu XML a k nim přidružené obrázky budou v prototypu podporovány v obecně rozšířeném formátu PNG. Podrobnější informace o formátu konfiguračních souborů uvedu v kapitole věnující se vývoji aplikace.

Následující obrázek ilustruje základní principy integrace Java a Ruby. JRuby kontejner bude obsahovat referenci na Ruby objekt reprezentující aktéra. Ruby třída pro aktéra bude obsahovat určitý zdrojový kód, který bude JRuby kontejner volat a jehož prostřednictvím bude např. zjištěno jakou akci v daném tahu má aktér provést a jeho prostřednictvím budou do objektu i předávány informace z Java programu. Součástí této Ruby třídy bude dynamicky doplněný zdrojový kód zadaný uživatelem, kterým uživatel definuje chování aktéra v simulaci.



Obrázek 5: Základní principy integrace Java a Ruby. Zdroj: autorem vytvořeno v programu MS Visio.

Interně bude v tomto JRuby kontejneru klíčovým prvkem instance třídy ScriptingContainer, jejíž princip byl popsána v kapitole 3.5, ve které budou Ruby objekty udržovány a jehož prostřednictvím budou volány jejich metody, vytvářeny instance apod. V rámci volby strategie integrace jsem se rozhodl použít přístup, že implementuji na straně Java aplikace rozhraní, které bude jako modul vloženo do Ruby třídy, protože to značně usnadní implementaci na straně Java aplikace, kde bude možné volat metody přímo na objektu, který bude typu tohoto rozhraní a ScriptingContainer zajistí provolání příslušné Ruby metody.

4.2.2 Seznam a popis případů užití

V této kapitole je uveden seznam případů užití, za kterým následuje popis jednotlivých případů užití. V této aplikaci nerozlišuji mezi různými aktéry[1] z hlediska případů užití, proto jsou případy užití uvedeny v tabulce a není u nich uveden aktér, které je může používat. Do aplikace není žádné přihlášení a libovolný její uživatel může spustit libovolný případu užití. Při popisu případů užití jsem se omezil na méně formální slovní popis, který v kontextu této práce považuji za dostačující, protože případy užití nejsou příliš složité a důraz je kladen

[1] Aktérem je zde myšlen aktér z hlediska UML diagramu případu užití, a nikoliv aktér jakožto programovatelný prvek aplikace, který byl popisován v předchozích kapitolách.

zejména na implementační část aplikace. Nicméně v případě složitějších případů užití by bylo vhodné použít formální prostředky jazyka UML a případy užití popsat podrobně.

ID případu užití	Název případu užití
UC01	Nahraj konfiguraci pro simulaci
UC02	Nahraj uživatelem zadaný zdrojový kód pro simulaci aktéra
UC03	Spusť simulaci kontinuálně
UC04	Pozastav simulaci
UC05	Proveď jeden krok simulace
UC06	Resetuj simulaci
UC07	Exportuj zdrojový kód pro simulaci do souboru
UC08	Importuj zdrojový kód pro simulaci ze souboru
UC09	Zobraz ladící informace aplikace
UC10	Načti výchozí konfiguraci simulace
UC11	Zobraz informace o prvku mřížky simulace

Tabulka 1: Seznam případů užití. Zdroj: autor.

UC01 – Nahraj konfiguraci pro simulaci

1. Aplikace zobrazí standardní okno pro výběr souboru .xml s konfigurací pro simulaci.
2. Uživatel vybere soubor s konfigurací pro simulaci.
3. Aplikace načte konfiguraci a na jejím základě zobrazí mřížku simulace s aktérem na příslušné pozici a příslušnými v konfiguraci definovanými prvky mřížky.

UC02 – Nahraj uživatelem zadaný zdrojový kód pro simulaci aktéra

1. Uživatel zadá do příslušného pole zdrojový kód a stiskne tlačítko pro načtení tohoto zdrojového kódu.
2. Aplikace načte zdrojový kód pro simulaci aktéra a tento bude připraven ke spuštění.

UC03 – Spusť simulaci kontinuálně

1. Aplikace bude provádět jednotlivé kroky simulace v krátkém časovém intervalu, aby bylo možné sledovat jednotlivé kroky aktéra.
2. Uživatel má možnost simulaci pozastavit a pokud je pozastavena, pak může simulaci znovu spustit, s tím že pokračuje od stavu, ve kterém byla pozastavena.

UC04 – Pozastav simulaci

1. Aplikace pozastaví simulaci ve stavu, v jakém se právě nachází a umožní z tohoto stavu později pokračovat.

UC05 – Proveď jeden krok simulace

1. Aplikace provede jeden krok simulace a zůstane ve stavu, v jakém se po provedení jednoho kroku bude nacházet a umožní z tohoto stavu později pokračovat.

UC06 – Resetuj simulaci

1. Pokud byla simulace spuštěna při spuštění tohoto případu užití, je zastavena.
2. Aplikace vrátí simulaci do výchozího stavu, v jakém se nacházela po načtení konfiguračního souboru simulace.

UC07 – Exportuj zdrojový kód pro simulaci do souboru

1. Aplikace zobrazí standardní okno pro výběr souboru .rb, do kterého má být uživatel zadaný zdrojový kód pro aktéra exportován.
2. Uživatele vybere požadované umístění souboru a jeho název.
3. Aplikace provede export do souboru.

UC008 – Importuj zdrojový kód pro simulaci ze souboru

1. Aplikace zobrazí standardní okno pro výběr souboru .rb, ze kterého má být zdrojový kód pro aktéra importován.
2. Uživatel vybere požadovaný soubor.
3. Aplikace načte obsah souboru a zobrazí jej v příslušném poli pro zadávání zdrojového kódu.

UC09 – Zobraz ladící informace aplikace

1. Aplikace zobrazí dialogové okno, ve kterém budou zobrazeny podstatné informace o stavu aplikace a simulace.
2. Uživatel bude moci dále v aplikaci pracovat a v tomto dialogovém okně sledovat stav aplikace a simulace.

UC10 – Načti výchozí konfiguraci simulace

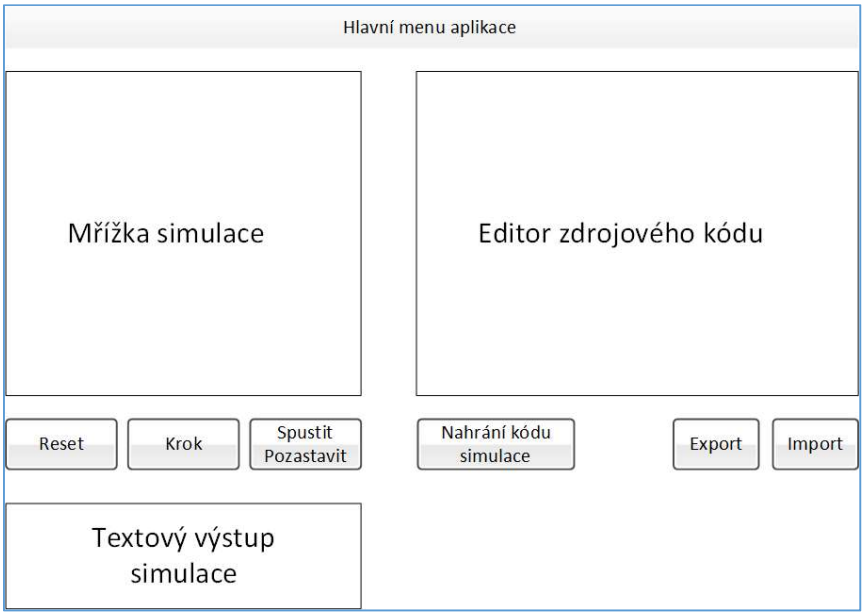
1. Aplikace načte výchozí konfiguraci simulace při spuštění aplikace. Tato výchozí konfigurace bude součástí aplikace.

UC11 – Zobraz informace o prvku mřížky simulace

1. Uživatel označí příslušný prvek na mřížce.
2. Aplikace zobrazí informace o vybraném prvku.

4.2.3 **Návrh uživatelského rozhraní**

V této kapitole uvádím stručný popis návrhu uživatelského rozhraní aplikace. Na následujícím obrázku je uveden Wireframe Diagram pro hlavní okno aplikace.



Obrázek 6: Wireframe Diagram hlavního okna aplikace. Zdroj: autorem vytvořeno v programu MS Visio a dále upraveno v programu MS Paint.

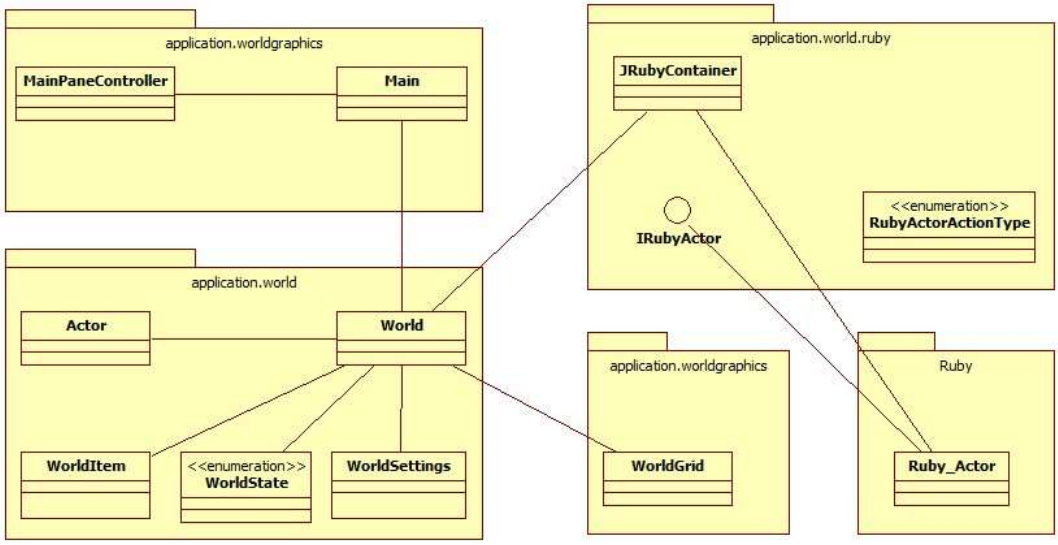
V horní části aplikace bude hlavní menu aplikace, ze kterého bude možné spouštět načtení konfiguračního souboru simulace, ukončit aplikaci a zobrazit dialog s ladícími informacemi o stavu aplikace. Hlavní část okna bude rozdělena na levou a pravou stranu, kde v levé bude zobrazena mřížka simulace, pod kterou bude tlačítka pro obsluhu simulace a pole pro textový výstup aktéra simulace. V pravé části bude editor zdrojového kódu, pod kterým budou tlačítka pro nahrání zadaného zdrojového kódu pro aktéra simulace a zároveň zde budou tlačítko pro import a export zdrojového kódu.

4.2.4 **Návrh tříd aplikace**

Architektura by měla upřesnit, jaké hlavní třídy by měly být v programu použity. Měl by být identifikován rozsah odpovědnosti každé hlavní třídy. Kromě toho by mělo být zřejmé, jak bude taková třída komunikovat s ostatními třídami. Architektura by měla obsahovat popis hierarchie tříd, popis přechodů stavů a persistence objektů. Je-li systém dostatečně velký, měla by architektura popisovat rovněž způsob uspořádání tříd do subsystémů. Architektura by měla popisovat také návrh dalších tříd, o nichž se v projektu uvažuje. (12)

V této kapitole je uveden Class Diagram zobrazující důležité třídy aplikace, k nímž je dále uveden v strukturovaný stručný popis jejich účelu a kompetencí. Klíčové třídy jsou rozebrány podrobně v kapitole věnující se přímo vývoji aplikace. Dále v textu budu používat

slovo „World“, které bude názvem či součástí názvu některých tříd. Proto zde definuji jeho význam. Simulace aktéra se skládá z toho, že vykonává určité chování na základě uživatelem definovaného zdrojového kódu v rámci mřížky, která obsahuje různé prvky. Je na místě definovat pojem, který bude v sobě toto vše zahrnovat. World tedy reprezentuje „Svět, ve kterém probíhá simulace aktéra, obsahující samotného aktéra, mřížku a její prvky, přičemž zachycuje stav v jakém se simulace aktuálně nachází. Tedy např. kde se v mřížce nachází aktér, jak jsou rozmístěné prvky v mřížce, zdali je aktuálně spuštěna kontinuální simulace či byl proveden pouze jeden samostatný krok simulace atd.“.



Obrázek 7: Class Diagram klíčových tříd aplikace. Zdroj: autorem vytvořeno v program StarUML.

Důležité třídy aplikace budou rozděleny do čtyř Java balíčků a jednoho Ruby balíčku[1]:

1. **`application`** obsahuje třídy související přímo s JavaFX aplikací a uživatelským rozhraním.
   1.1. `MainPaneController` slouží pro obsluhu události uživatelského rozhraní aplikace, které dále propaguje do třídy `Main`.
   1.2. `Main` je hlavní třídou JavaFX aplikace, která obsahuje metodu `start()`, která je volána při spuštění aplikace. Tato třída obsahuje klíčovou instanci třídy `World`, do které propaguje akce (události), které uživatel vykoná v aplikaci. Zároveň do

[1] V terminologii Ruby se pojem balíček nepoužívá, ale zde je uvažován jako seskupení logicky souvisejících tříd a v rámci zachování konzistenci názvosloví zůstávám u tohoto pojmu.

instance třídy `World`, předá referenci na instanci třídy `Canvas`, na kterou bude vykreslována mřížka simulace, kterou bude spravovat samostatná třída.

2. **`application.world`** obsahuje klíčové třídy reprezentující jednotlivé prvky „světa simulace“.
   2.1. `Actor` reprezentuje aktéra simulace na straně Java aplikace, jehož stav je synchronizován s Ruby instancí aktéra obsaženou v `JRubyContainer`.
   2.2. `World` reprezentuje „celý svět simulace“ viz definice na začátku kapitoly Návrh tříd aplikace.
   2.3. `WorldItem` reprezentuje jeden prvek mřížky, na které se odehrává simulace. Nese informaci o grafické reprezentaci tohoto prvku a zároveň může nést informaci o vstupu, který může aktér získat z tohoto prvku.
   2.4. `WorldState` je výčtový typ reprezentující stav „světa simulaci“. Možné hodnoty jsou následující:
      - `INITIAL` – výchozí stav, kdy je načtena konfigurace „světa simulace“, ale zatím nebyl načten zdrojový kód pro aktéra simulace. Z tohoto stavu nelze simulaci spustit.
      - `READY` – reprezentuje stav, kdy je aplikace připravena na spuštění simulace. V tomto stavu je kód pro aktéra simulace načten.
      - `PAUSED` – reprezentuje stav kdy již byla simulace spuštěna a je pozastavena. V tomto stavu se nachází simulace, pokud je prováděna po jednotlivých krocích, nebo pokud je pozastavena z kontinuálního spuštění.
      - `RUNNING` – reprezentuje stav, kdy je simulace spuštěna kontinuální a v daných časových intervalech jsou vykonávány automaticky jednotlivé kroky simulace.
      - `TERMINATED` – reprezentje stav, kdy je simulace dokončena.
   2.5. `WorldSettings` reprezentuje načtenou konfiguraci „světa simulace“. Obsahuje informaci o výchozí pozici aktéra, rozmístění prvků mřížky atd.
3. **`application.worldgraphics`** obsahuje zatím pouze jednu třídu určenou pro práci s grafickou části simulace.
   3.1. `WorldGrid` obaluje JavaFX Canvas, na kterém umožňuje vykreslovat jednotlivé prvky simulace.
4. **`application.world.ruby`** obsahuje třídy, které slouží k intergraci s Ruby.

4.1. `JRubyContainer` reprezentuje kontejner JRuby, který v sobě obsahuje zejména ScriptingContainer, ve kterém se nachází Ruby instance aktéra. Dále zprostředkovává komunikaci mezi Java a Ruby částmi aplikace.

4.2. `IRubyActor` je rozhraní použité v rámci integrace Java a Ruby. Toto rozhraní definuje metody třídy reprezentující aktéra v Ruby, které budou volány z Java aplikace. Mezi tyto metody patří zejména metoda vracející informaci o typu provedené akce, synchronizace souřadnic a předávání dalších informací o stavu „světa simulace“.

4.3. `RubyActorActionType` je výčtový typ definující podporované akce, které může aktér v rámci simulace vykonávat.

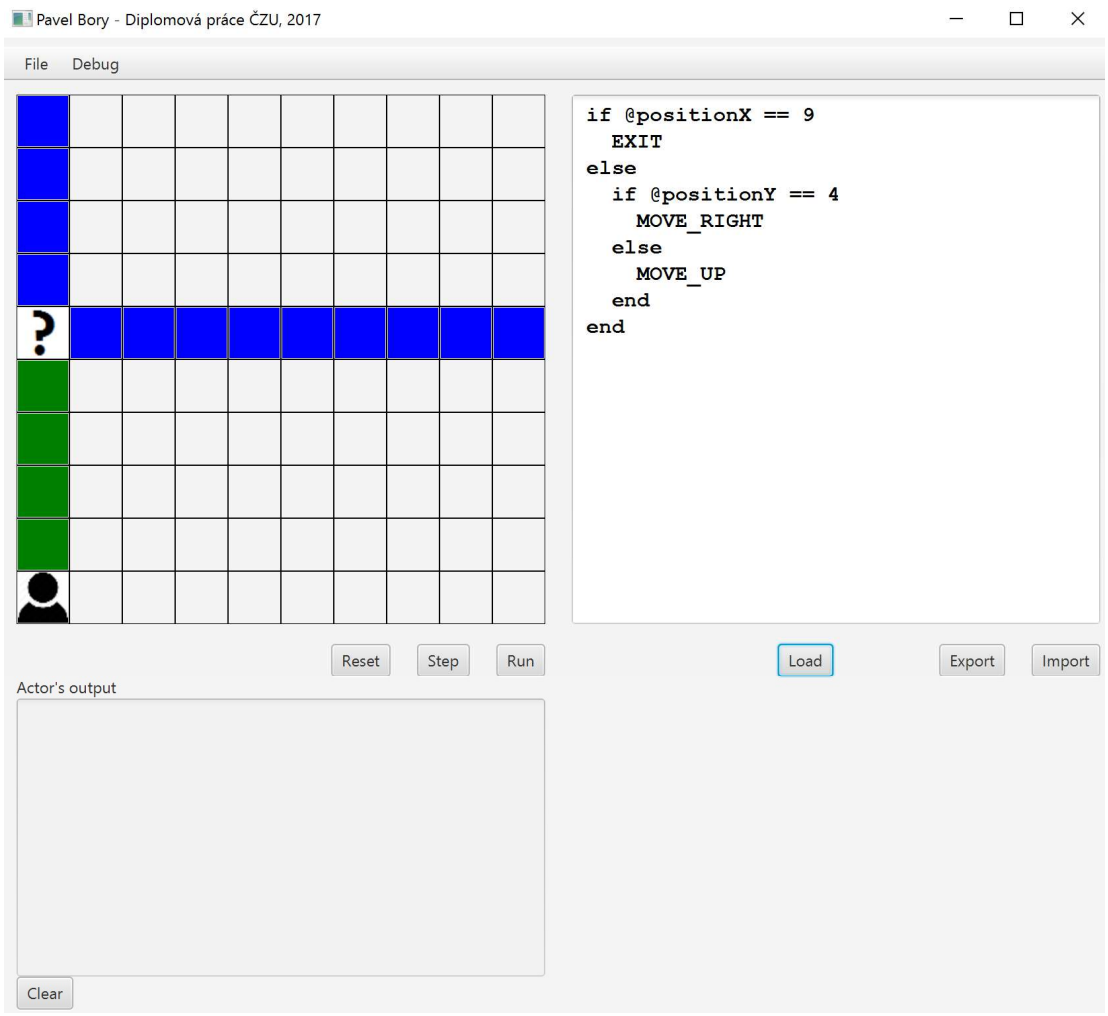
5. **`Ruby`** – nejde přímo o Java balíček, ale o seskupení tříd na straně Ruby, přičemž aktuálně obsahuje pouze jednu třídu aktuálně jednu Ruby třídu – `Actor`, která reprezentuje aktéra simulace, jemuž uživatel může dodat zdrojový kód vykonávaný při simulaci.

4.3 Vývoj aplikace

V této kapitole je popsán vývoj aplikace, přičemž je kladen důraz na popis klíčových prvků v jednotlivých částech aplikace. Nejdříve je popsáno uživatelské rozhraní a jsou zdokumentovány prvky JavaFX, které patří do části aplikace implementující uživatelské rozhraní. Dále následující popis formátu xml souborů, s nimiž bude aplikace pracovat. Poté následuje podrobný popis klíčových tříd aplikace, které implementují hlavní funkčnosti aplikace na straně Java. V závěru kapitoly jsou zdokumentovány prvky, které řeší integraci Java a Ruby a je popsána kostra Ruby kódu, do které bude doplňován uživatelem definovaný kód.

4.3.1 Uživatelské rozhraní

Aplikace se sestává z jednoho hlavní okna, ze kterého je možné otevřít dialogové okno s ladícími informace. V tomto hlavním okně jsou prvky uživatelského rozhraní rozmístěny dle návrhu z kapitoly 4.2.3.



Obrázek 8: Hlavní okno aplikace s krátkým zdrojovým kódem pro aktéra. Zdroj: autorem vytvořeno v aplikaci vyvíjené v rámci této práce a dále upraveno v MS Paint.

Prvky hlavního okna jsou deklarovány v souboru `MainPane.fxgraph`, na jehož základě je generován soubor `MainPane.fxml`, přičemž pro rozmístění prvků jsou použity zejména instance tříd `GripPane` a `FlowPane`. Toto hlavní okno je řízeno pomocí třídy `MainPaneController`.

```
component MainPane controlledby MainPaneController {
  GridPane {
    ^id : "MasterGrid",
    rowConstraints : [
      // Top Menu row
      RowConstraints {
        percentHeight : 5
      },
      // Main area row
      RowConstraints {
        percentHeight : 95
```

```
      }
    ],
...
```

Zdrojový kód 19: Výňatek zdrojového kódu MainPane.fxgraph obsahujícího deklarace ovládacích prvků hlavního okna aplikace. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

`MainPanelController` definuje zejména obslužné metody pro události jednotlivých ovládacích prvků jako jsou např. tlačítka a prvky menu. Propojení s prvky deklarovanými v `.fxgraph` souboru je zajištěno JavaFX s použitím anotace `@FXML`.

```
package application;

import javafx.fxml.FXML;
import javafx.scene.control.TextArea;
import javafx.scene.input.MouseEvent;

public class MainPaneController {

  @FXML
  private TextArea textAreaCode;

  @FXML
  public void BtnLoadCodeClicked(MouseEvent event) {
    Main.loadCodeForActor(textAreaCode.getText());
  }

...
```

Zdrojový kód 20: Výňatek zdrojového kódu třídy MainPaneController. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Kromě hlavního okna aplikace je implementováno i okno, která bude zobrazovat informace užitečné při ladění programu. Toto okno lze otevřít z nabídky Debug z hlavního menu aplikace a bude zobrazeno jako nemodální dialog, aby bylo možné pokračovat v práci s hlavním oknem aplikace. Z hlediska použitých prvků je toto okno s ladícími informacemi triviální, protože bude obsahovat pouze jeden prvek typu `TextArea`, ve kterém budou ladící informace zobrazeny.

```
package application

import javafx.scene.layout.FlowPane
import javafx.scene.control.TableView
import javafx.scene.control.TextArea
import javafx.geometry.Insets

component DebugInfoDialog {
  FlowPane {
    alignment : "TOP_LEFT",
    prefHeight : 700,
    prefWidth : 270,
```

```
    children : [
      TextArea {
        editable : false,
        prefHeight : 650,
        prefWidth : 250,
        ^id : "debugDialogMainTextArea"
      }
    ]
  }
}
```

Zdrojový kód 21: Zdrojový kód DebugInfoDialog.fxgraph reprezentující okno pro zobrazování ladících informací. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Vzhled prvků uživatelského rozhraní je částečně definován v souboru `application.css`, ve kterém jsou uvedeny styly podporované JavaFX pro vybrané prvky uživatelského rozhraní.

```
#bottomPanelProgrammingGrid {
  -fx-padding: 15 0 0 10;
}

#debugDialogMainTextArea {
  -fx-font-size: 16;
  -fx-opacity: 1.0;
  -fx-text-fill: black;
  -fx-font-family: "monospace";
}
```

Zdrojový kód 22: Ukázka části stylů aplikace ze souboru application.css Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4.3.2 **Konfigurační soubory**

Konfiguraci aplikace lze načítat z konfiguračních souborů ve formátu XML. Kořenovým elementem souboru je element `world`. V tomto se musí nacházet elementy `actor`, `gridColors` a `gridItems`. V elementu `actor` se pomocí atributů `posX`, `posY` a `img` definují výchozí souřadnice aktéra a název souboru s obrázkem, který bude graficky reprezentovat aktéra. Veškeré v tomto souboru odkazované soubory s obrázky musí být v rámci implementace prototypu umístěny ve stejné složce jako konfigurační XML souboru. V elementu `gridColor` se může nacházet 0..n elementů `cell` pomocí nichž lze definovat barvy jednotlivých buněk mřížky. V elementu `gridItems`, lze definovat prvky mřížky, které budou graficky reprezentovány příslušnými obrázky definovanými pomocí atributu img a zároveň se v atributu `val` definuje hodnota, kterou bude moci aktér z prvku mřížky načíst.

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

```
<world>
  <actor posX="2" posY="7" img="actor.png">
  </actor>
  <gridColors>
    <cell x="5" y="9" color="GREEN"></cell>
    <cell x="5" y="8" color="RED"></cell>
    <cell x="5" y="7" color="BLUE"></cell>
    <cell x="5" y="6" color="ORANGE"></cell>
    <cell x="6" y="9" color="VIOLET"></cell>
    <cell x="6" y="8" color="DARKGREEN"></cell>
    <cell x="6" y="7" color="DARKBLUE"></cell>
    <cell x="6" y="6" color="PURPLE"></cell>
  </gridColors>
  <gridItems>
    <item x="0" y="8" img="input_num.png" val="6"></item>
    <item x="0" y="7" img="input_num.png" val="28"></item>
    <item x="0" y="6" img="input_num.png" val="496"></item>
    <item x="1" y="3" img="input_text.png" val="Paul"></item>
    <item x="7" y="2" img="input_text.png" val="Peter"></item>
  </gridItems>
</world>
```

Zdrojový kód 23: Ukázka konfiguračního souboru aplikace. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Aplikace při svém startu načte výchozí konfigurační soubor. Z hlavního menu aplikace lze spustit funkčnost, která umožní nahrát konfiguraci z vlastního konfiguračního souboru ve výše uvedeném formátu. Výběr vlastního konfiguračního souboru je realizován pomocí standardního dialogu pro výběr souboru, přičemž v aplikace je toto zajištěno pomocí třídy `FileChooser`.

4.3.3 Třída Main

Třída `Main` je hlavní třídou JavaFX aplikace. Obsahuje metodu `start()`, která je volána při spuštění aplikace. Táto třída v sobě nese mimo jiné především reference na prvky uživatelského rozhraní, se kterými je během práce s aplikací manipulováno a nese v sobě referenci na instanci klíčové třídy `World`. Při událostech uživatelského rozhraní vyvolaných uživatelem budou ze třídy `MainPaneController` volány metody třídy `Main`, která je buď sama přímo zpracuje jako např. při zobrazení okna s ladícími informacemi, nebo zajistí zavolání příslušné metody na instanci třídy `World`. Zde je uvedena ukázka části kódu třídy `Main` obsahující metodu `start()`, ve které probíhá úvodní inicializace potřebných prvků aplikace.

```
public class Main extends Application {
  private static final Logger logger = LogManager.getLogger(Main.class);
  private static World world;
  private static Canvas mainCanvas;
  private static Stage primaryStage;
```

```
  private static Stage debugInfoStage;
  private static TextArea debugInfoTextArea;
  private static TextArea actorsOutputTextArea;
  private static Button btnSimulationRun;

  @Override
  public void start(Stage primaryStage) {
    try {
      logger.info("Application starting");
      GridPane root =
        FXMLLoader.load(getClass().getResource("MainPane.fxml"));
      Scene scene = new Scene(root,1280,768);
      scene.getStylesheets().add(getClass().getResource("application.css")
        .toExternalForm());
      primaryStage.setScene(scene);
      primaryStage.show();
      primaryStage.setTitle("Pavel Bory - Diplomová práce ČZU, 2017");
      Main.mainCanvas = (Canvas)root.lookup(MainCanvasIdSelector);
      Main.primaryStage = primaryStage;
      Main.actorsOutputTextArea =
        (TextArea)root.lookup(ActorsOutputTextAreaIdSelector);
      Main.btnSimulationRun =
        (Button)root.lookup(BtnSimulationRunIdSelector);
      Main.world = initializeWorld(mainCanvas,null);
      initializeDebugDialog();
      logger.info("First World initialization on application startup
successfully completed. Displaying Primary Stage.");
     } catch(Exception e) {
       logger.error("Application failed to start: ", e);
     }
}
...
```

Zdrojový kód 24: Výňatek zdrojového kódu třídy Main. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4.3.4 **Třída World**

Tato třída reprezentuje celý „Svět simulace“. Na začátku třídy je obdobně jako v jiných důležitých třídách vytvořena instance implementující rozhraní `Logger` pro logování. Dále jsou v této třídě definovány konstanty určující rozměry mřížky a parametry časovače, kterým je řízena simulace, pokud je tato spuštěna v kontinuálním režimu. V konstruktoru třídy `World`, který je vidět v následující ukázce, jsou do atributů uloženy instance důležitých objektů reprezentující jednotlivé prvky simulace. Do konstruktoru této třídy je předáván soubor obsahující příslušnou konfiguraci simulace a objekt typu `Canvas`, na který bude vykreslována grafická reprezentace simulace.

```
public class World {
  private static final Logger logger = LogManager.getLogger(World.class);
  public static final int WORLD_COLUMNS_COUNT = 10;
  public static final int WORLD_ROWS_COUNT = 10;
  public static final int TIMER_DELAY = 0;
  public static final int TIMER_PERIOD = 1500;
```

```
  private WorldGrid worldGrid;
  private WorldSettings worldSettings;
  private Actor actor;
  private JRubyContainer jrubyContainer;
  private WorldState state;
  private Timer timer;

  public World(Canvas canvas, File worldFile) throws SAXException,
IOException, ParserConfigurationException{
    logger.info("Creating new World instance");
    worldGrid = new WorldGrid(canvas);
    logger.debug("World Grid created");
    worldSettings = WorldSettings.load(worldFile);
    logger.debug("World Settings loaded from file");
    jrubyContainer = new JRubyContainer();
    logger.debug("JRubyContainer initialized");
    setInitialWorldState();
    logger.debug("World set to it's INITIAL state");
    drawCurrentState();
    logger.debug("Worlds graphical representation has been drawn on the
canvas");
    setState(WorldState.INITIAL);
    updateDebugInfo();
    logger.info("Creating new World instance - Successfuly completed");
  }
...
```

Zdrojový kód 25: Výňatek zdrojového kódu třídy World obsahující atributy a konstruktor. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Třída `World` dále implementuje metodu `step()`, která vykoná jeden krok simulace. Před provedením kroku simulace se ověří, že jej lze provést, a že např. není simulace již v ukončeném stavu nebo např. není v úvodním stavu, kdy ještě není uživatelem daný zdrojový kód pro aktéra načten. Tato metoda přijímá jeden parametr typu `boolean`, který určuje, zdali je volána v situace, kdy je simulace krokována uživatel nebo v režimu kontinuálního spuštění simulace. Pokud lze krok simulace vykonat, pak je zavolána metoda `makeActorAction()` na instanci `JRubyContainer`, který zajistí zavolání příslušné uživatelem v Ruby implementované metody aktéra. Na základě návratového hodnoty této metody je příslušným způsobem změněna pozice aktéra v rámci mřížky nebo je simulace ukončena. Zároveň je případně předán třídě `Main` textový výstup aktéra k zobrazení, pokud nějaký aktér vytvořil. V rámci metody `step()` jsou rovněž v příslušných situacích vyvolány výjimky, které reprezentují určuté chybové situace simulace:

- `WorldActorOutOfGridException` reprezentuje situace, kdy se aktér pokusí posunout mimo mřížku simulace.

- `WorldActorUnssuportedOperationException` reprezentuje situaci, kdy metoda vracející požadovanou akci aktéra vrátí neočekávanou hodnotu, která není simulací podporována.
- `WorldNotInStateToPerformSimulationStep` reprezentuje situaci, kdy je simulace ve stavu, kdy není možné provést krok simulace. Např. ještě není načten zdrojový kód pro aktéra a uživatel by se pokusil sputit simulaci, nebo např. situaci, kdy je simulace již ve finálním stavu a uživatel by se pokusil provést další krok simulace.

```
public void step(boolean singleStep) throws WorldActorOutOfGridException,
  WorldActorUnsupportedOperationException,
  WorldNotInStateToPerformSimulationStep
  {
    if(singleStep) {
      if(getState() == WorldState.TERMINATED || getState() ==
        WorldState.INITIAL)
      {
        throw new WorldNotInStateToPerformSimulationStep(...);
      }else{
        setState(WorldState.PAUSED);
      }
    }else{
      if(getState() != WorldState.RUNNING){
        return;
    }
  }

  RubyActorActionType actionType = jrubyContainer.makeActorAction(actor,
    getActualItemInputForActor(actor));
  String actorsOutputText = jrubyContainer.gatherOutputText();
  if(actorsOutputText != null && !actorsOutputText.isEmpty()){
    Main.appendActorsOutput(actorsOutputText);
  }
  switch (actionType) {
    case MOVE_RIGHT:
      if(actor.getPositionX() >= WORLD_COLUMNS_COUNT - 1){
        setState(WorldState.TERMINATED);
        throw new WorldActorOutOfGridException(...);
      }else{
        actor.setPositionX(actor.getPositionX() + 1);
      }
      break;
    case MOVE_LEFT:
      ...
      ...
      ...
  }
  drawCurrentState();
  updateDebugInfo();
}
```

Zdrojový kód 26: Výňatek zdrojového kódu metody step() ze třídy World. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Aplikace rovněž umožňuje definovat pole mřížky, které obsahují vstup pro aktéra, který může načíst, pokud na ně vstoupí. Za účelem získání vstupu pro aktéra na dané pozici (pokud je na dané pozici vstup pro aktéra definován) je implementována metoda `getActualItemInputForActor()`.

```
private String getActualItemInputForActor(Actor actor){
  if(worldSettings.getGridWorldItems()
      [(int) actor.getPositionX()]
      [(int) actor.getPositionY()] != null){
    return worldSettings.getGridWorldItems()
      [(int) actor.getPositionX()]
      [(int) actor.getPositionY()].getValue();
  }
  return null;
}
```

Zdrojový kód 27: Metoda getActualItemInputForActor() třídy World. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Při testování a v případě dalšího rozvoje prototypu se domnívám, že bude vhodné mít k dispozici v aplikaci přehledný výpis aktuálního stavu klíčových prvků aplikace. Pro tento účel je implementována metoda `updateDebugInfo()`, která sestaví pomocí instance `StringBuilder`[1] (ten byl použit z důvodu vyšší efektivity při sestavování textu v porovnání se skládáním instancí typu `String`) textovou reprezentaci aktuálního stavu aplikace, která je předána metodě `setDebugDialogText()` třídy `Main`.

```
private void updateDebugInfo(){
  StringBuilder sBuilder = new StringBuilder();
  sBuilder.append("ACTOR:\n");
  sBuilder.append(String.format("Pos[x,y]: [%s,%s]\n",
    actor.getPositionX(),actor.getPositionY()));
  HashMap<String, Object> actorData = jrubyContainer.getActorData();
  for(String key : actorData.keySet()){
    sBuilder.append(String.format("data[%s] = %s\n",key,actorData.get(key)));
  }
  sBuilder.append("\n\nGRID ITEMS:\n");

  for(WorldItem worldItem : worldSettings.getWorldItems()){
    ...
  Main.setDebugDialogText(sBuilder.toString());
}
```

Zdrojový kód 28: Výňatek zdrojového kódu metody updateDebugInfo() třídy World. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

[1] https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html

Třída World rovněž implementuje metodu `drawCurrentState()`, která řídí volání příslušných metod na instanci třídy `WorldGrid`, která zajistí vykreslení grafické reprezentace simulace na instanci typu `Canvas`, která byla předána do konstruktoru třídy `World`.

```
public void drawCurrentState(){
  worldGrid.drawGrid(worldSettings.getGridCellColors());
  worldGrid.drawItems(worldSettings.getGridWorldItems());
  worldGrid.drawActor(actor.getPositionX(),actor.getPositionY(),
    worldSettings.getActorImg());
}
```

Zdrojový kód 29: Výňatek zdrojového kódu metody drawCurrentState() třídy World. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4.3.5 Třída WorldSettings

Třída `WorldSettings` představuje nastavení simulace. Ve svých atributech nese informace o výchozích souřadnicích aktéra, obrázek graficky reprezentující aktéra, prvky mřížky a další informace k simulaci.

```
public class WorldSettings {
  private static final Logger logger =
    LogManager.getLogger(WorldSettings.class);
  private long actorInitialPositionX;
  private long actorInitialPositionY;
  private Image actorImg;
  private Color[][] gridCellColors;
  private WorldItem[][] gridWorldItems;
```

Zdrojový kód 30: Výňatek zdrojového kódu ze třídy WorldSettings. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Zároveň obsahuje metodu `load()`, která z daného souboru načte nastavení simulace, které uloží do výše uvedených atributů. V následující ukázce je uvedena část zdrojového kódu metody `load()`. Třída `WorldSettings` obsahuje kromě metody `load()` ještě metodu `getWorldItems()`, která vrací kolekci všech prvků simulace, které jsou rozmístěny do mřížky, a ze kterých může aktér načítat vstupní informace.

```
public static WorldSettings load(File file) throws SAXException, IOException,
  ParserConfigurationException
{
  Document doc = DocumentBuilderFactory.newInstance()
    .newDocumentBuilder().parse(file);
  Node elActor = doc.getElementsByTagName("actor").item(0);
  long actorPosX = Long.parseLong(elActor.getAttributes()
    .getNamedItem("posX").getNodeValue());
  long actorPosY = Long.parseLong(elActor.getAttributes()
    .getNamedItem("posY").getNodeValue());
```

```
  String actorImgFileName = elActor.getAttributes()
    .getNamedItem("img").getNodeValue();
  Image actorImg = new Image(new File(file.getParent() + "\\" +
    actorImgFileName).toURI().toString());

  Color[][] gridCellColors = null;
  if(doc.getElementsByTagName("gridColors").getLength() > 0){
    NodeList cellColors = doc.getElementsByTagName("gridColors")
      .item(0).getChildNodes();
    gridCellColors = new Color
      [World.WORLD_COLUMNS_COUNT]
      [World.WORLD_ROWS_COUNT];
    for(int i = 0;i<cellColors.getLength();i++){
      if(cellColors.item(i).getNodeName().equals("cell")){
        int colorPosX = Integer.parseInt(cellColors.item(i)
          .getAttributes().getNamedItem("x").getNodeValue());
        int colorPosY = Integer.parseInt(cellColors.item(i)
          .getAttributes().getNamedItem("y").getNodeValue());
        Color color = Color.valueOf(cellColors.item(i)
          .getAttributes().getNamedItem("color").getNodeValue());
        gridCellColors[colorPosX][colorPosY] = color;
      }
    }
  }

  ...

  return new WorldSettings(actorPosX,actorPosY,actorImg,
    gridCellColors,gridWorldItems);
```

Zdrojový kód 31: Výňatek zdrojového kódu metody load() třídy WorldSettings. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4.3.6 Třída WorldGrid

Třída `WorldGrid` slouží k vykreslování grafické reprezentace simulace na instanci typu `Canvas`, kterou si uchovává ve svém atributu. V následující ukázce jsou vidět metody této třídy, které slouží k vykreslení mřížky simulace, aktéra a prvků rozmístěných do mřížky simulace.

```
public class WorldGrid {
  private final int cellWidth;
  private final int cellHeight;
  private Canvas canvas;

  ...

  public void drawGrid(Color[][] gridCellColors){
    GraphicsContext gc = canvas.getGraphicsContext2D();
    gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());

    gc.setStroke(Color.BLACK);
    for(int i = 0 ; i <= World.WORLD_COLUMNS_COUNT ; i++){
      int x = i * cellWidth;
      gc.strokeLine(x, 0, x, canvas.getHeight());
    }
```

```
    for(int i = 0 ; i <= World.WORLD_ROWS_COUNT ; i++){
      int y = i * cellHeight;
      gc.strokeLine(0, y, canvas.getWidth(), y);
    }

    ...
  }

  public void drawActor(long x, long y,Image actorImg){
    GraphicsContext gc = canvas.getGraphicsContext2D();
    gc.drawImage(actorImg, x * cellWidth + 1, y * cellHeight + 1);
  }

  public void drawItems(WorldItem[][] items){
    ...
  }
}
```

Zdrojový kód 32: Výňatek zdrojového kódu třídy WorldGrid, který ukazuje metody pro vykreslování grafické reprezentace simulace. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4.3.7 JRubyContainer, IRubyActor a RubyActorActionType

V této kapitole jsou popsány klíčové prvky aplikace, které slouží pro integraci s Ruby. Rozhraní `IRubyActor` definuje seznam metod, které bude možné z Java aplikace zavolat na Ruby instanci aktéra. Tyto metody umožní nastavit pozici aktéra v mřížce, načíst do instance aktéra zdrojový kód definovaný uživatelem, provést uživatelem naimplementované chování, předat aktérovy vstupní informace (pokud je na pozici aktéra v mřížce přítomen prvek, který obsahuje vstupní informace pro aktéra), získat aktuální vnitřní stav proměnných aktéra a získat textový výstup aktéra.

```
public interface IRubyActor {
  void setPositionX(long position);
  void setPositionY(long position);
  void loadActorActionMethod(String code);
  void setInput(String input);
  RubyActorActionType makeAction();
  HashMap<String, Object> getData();
  String gatherOutputText();
}
```

Zdrojový kód 33: Rozhraní IRubyActor definující seznam metod, které lze z Java aplikace zavolat na Ruby instanci aktéra. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Výčtový typ `RubyActorActionType` definuje typy akcí, které může uživatel použít při implementaci chování aktéra.

```
public enum RubyActorActionType {
  MOVE_LEFT, MOVE_RIGHT, MOVE_UP, MOVE_DOWN, TERMINATE
}
```

Zdrojový kód 34: Výčtový typ RubyActorActionType. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Třída `JRubyContainer` zapouzdřuje přístup k instanci Ruby aktéra. V sobě obsahuje instanci `ScriptingContainer`, ve kterém je ve vykonáván kód Ruby a je v něm uchovávána instance Ruby aktéra.

```
public class JRubyContainer {
  private static final String JRubyStdLibPath = "C:\\jruby-
9.1.6.0\\lib\\ruby\\stdlib";

  private IRubyActor actor;
  private ScriptingContainer container;

  public JRubyContainer(){
    container = new ScriptingContainer();
    container.setLoadPaths(Arrays.asList("libs",JRubyStdLibPath));
    container.runScriptlet("require 'actor.rb'");
  }
...
```

Zdrojový kód 35: Výňatek zdrojového kódu třídy JRubyContainer. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Dále třída `JRubyContainer` obsahuje metodu `loadCodeForActor()`, která je v aplikaci zavolána ve chvíli, kdy uživatel definuje zdrojový kód pro aktéra a nechá jej nahrát do Ruby instance aktéra. Kromě této metody ještě obsahuje důležitou metodu `makeActorAction()`, která zajistí vykonání zdrojového kódu definovaného uživatelem a vrátí výsledek akce provedené aktérem.

```
public void loadCodeForActor(Actor currentJavaActorState,String code){
  actor = (IRubyActor) container.runScriptlet("Actor.instance");
  actor.setPositionX(currentJavaActorState.getPositionX());
  actor.setPositionY(currentJavaActorState.getPositionY());
  actor.loadActorActionMethod(code);
}

public RubyActorActionType makeActorAction(Actor currentJavaActorState,
String input) throws WorldActorUnsupportedOperationException {
  actor.setPositionX(currentJavaActorState.getPositionX());
  actor.setPositionY(currentJavaActorState.getPositionY());
  actor.setInput(input);
  try{
    return actor.makeAction();
  }catch(Exception e){
    throw new WorldActorUnsupportedOperationException("The Actor action
caused exception. Check if the return value is one of the allowed actions.");
  }
}

public HashMap<String, Object> getActorData(){
  if(actor != null){
    return actor.getData();
  }
  return new HashMap<String,Object>();
```

```
}

public String gatherOutputText(){
  if(actor != null){
    return actor.gatherOutputText();
  }
  return null;
}
```

Zdrojový kód 36: Výňatek zdrojového kódu třídy JRubyContainer. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Nedílnou součástí aplikace je předpřipravený Ruby zdrojový kód, který obsahuje kostru třídy `Actor` a zejména importy příslušných Java typů použitých při integraci Java a Ruby. Ten zdrojový kód v Ruby je přilozen jako samostatný .rb soubor k Java aplikaci. V tomto je zajímavá zejména metoda `loadActorActionMethod()`, která zajistí vložení zdrojového kódu jakožto těla metody `makeAction()` do instance třídy `Actor`.

```
require "java"
require "Singleton"
java_import "application.world.ruby.IRubyActor"
java_import "application.world.ruby.RubyActorActionType"
java_import "java.util.HashMap"

MOVE_LEFT = RubyActorActionType::MOVE_LEFT
MOVE_UP = RubyActorActionType::MOVE_UP
MOVE_RIGHT = RubyActorActionType::MOVE_RIGHT
MOVE_DOWN = RubyActorActionType::MOVE_DOWN
EXIT = RubyActorActionType::TERMINATE

class Actor
  include IRubyActor
  include Singleton

  def self.actor_code
    @@actor_code
  end

  def loadActorActionMethod(code)
    @@actor_code = code

    def self.makeAction
      eval(Actor.actor_code)
    end
  end

  def setPositionX(position)
    @positionX = position
  end

  def set_data(key,value)
    if(@data == nil)
      @data = HashMap.new
    end
    @data.put(key.to_s,value)
  end

  ...

end
```

Zdrojový kód 37: Výňatek z implementace Ruby kostry obsahující kód pro třídu Actor, do jejíž metody makeAction je nahráván uživatelem definovaný zdrojový kód. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

4.4 Testování

Aplikace byla uživatelsky otestována na základě požadovaných funkčností. Zároveň byly implementovány unit testy s použitím frameworku JUnit, které budou v této kapitole popsány. Pomocí unit testů jsou pokryty klíčové třídy aplikace:

- `WorldSettings`
- `World`
- `JRubyContainer` a v něm obsažená implementace kostry třídy `Actor`

4.4.1 Unit testy

Pro třídy `WorldSettings`, `World` a `JRubyContainer` byly implementovány testy rozdělené do analogicky pojmenovaných tříd `WorldSettingsTest`, `WorldTest` a `JRubyContainerTest`. V unit testech, které potřebují pracovat s prvky z JavaFX je zapotřebí platformu JavaFX inicializovat, protože jinak by z kódu unit testu nebylo možné prvky této platformy používat. Za tímto účelem je v testech, které tyto prvky potřebují implementován následující kód, který vynutí inicializaci JavaFX platformy.

Následuje ukázka implementace unit testu, který ověřuje funkčnost třídy `WorldSettings`, kde je ověřeno, že dojde k očekávanému chování po načtení xml souboru s nastavením simulace, a že je možné získávat v aplikaci potřebné informace z tohoto nastavení.

```
public class WorldSettingsTest {

  @Test
  public void testLoadSettings() throws SAXException, IOException,
      ParserConfigurationException, InterruptedException {
    Thread thread = new Thread(new Runnable() {
      @Override
      public void run() {
        new JFXPanel();
        Platform.runLater(new Runnable() {
          @Override
          public void run() {
            new Main().start(new Stage());
          }
        });
      }
    });
        thread.start();

    File worldFile = new File(Main.DefaultWorldFilePath);
    WorldSettings worldSettings;
    worldSettings = WorldSettings.load(worldFile);
    assertEquals(0,worldSettings.getActorInitialPositionX());
    assertEquals(9,worldSettings.getActorInitialPositionY());
    assertEquals(Color.GREEN, worldSettings.getGridCellColors()[0][9]);
    assertEquals(Color.BLUE, worldSettings.getGridCellColors()[0][3]);
    assertEquals("31",worldSettings.getGridWorldItems()[0][4].getValue());
  }
}
```

Zdrojový kód 38: Ukázka implementace unit testu pro třídu WorldSettings pomocí JUnit. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

4.5 Ukázka použití aplikace

V této kapitole jsou uvedeny dvě ukázky použití aplikace. Na začátku ukázek je uvedeno, jaký úkol má uživatel pomocí aplikace řešit a jakým způsobem má aktéra naprogramovat. Poté je uvedeno řešení v podobě Ruby kódu, kterým by bylo možné danou úlohu řešit. V ukázkách je rovněž ilustrativní obrazek mřížky simulace a obsah XML souboru definujícího „svět simulace“.

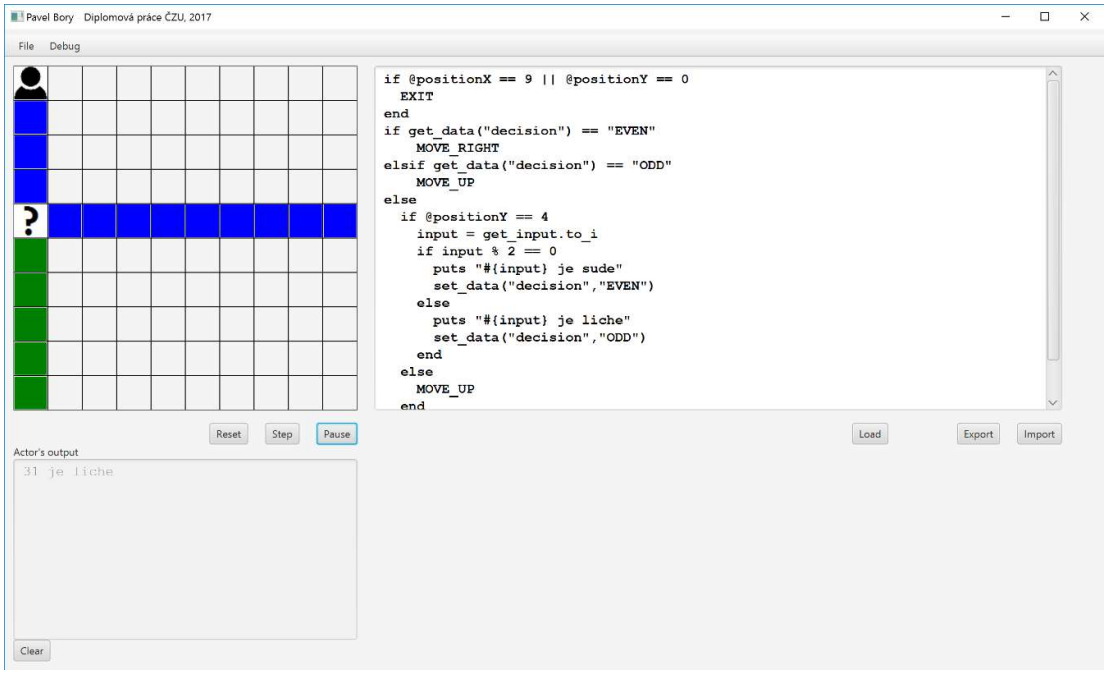
4.5.1 Ukázka 1

V této ukázce má uživatel za úkol napsat kód, ve kterém si bude možné procvičit práci s podmíněným zpracováním kódu. Aktér začíná v levém dolním rohu mřížky a má za úkol se vydat nahoru a pokud na místě vyznačeném otazníkem aktér získá jako vstup číslo, které je sudé, pak se vydá doprava, jinak půjde nahoru. Aktér dojde až na okraj mřížky, kde se zastaví a simulace skončí. Zároveň aktér vypíše, zdali je číslo sudé či liché. Zde je uvedena XML souboru definujícího svět této simulace:

```
<?xml version="1.0" encoding="UTF-8"?>
<world>
  <actor posX="0" posY="9" img="actor.png">
  </actor>
  <gridColors>
    <cell x="0" y="9" color="GREEN"></cell>
    <cell x="0" y="8" color="GREEN"></cell>
    <cell x="0" y="7" color="GREEN"></cell>
    <cell x="0" y="6" color="GREEN"></cell>
    <cell x="0" y="5" color="GREEN"></cell>
    <cell x="0" y="3" color="BLUE"></cell>
    <cell x="0" y="2" color="BLUE"></cell>
    <cell x="0" y="1" color="BLUE"></cell>
    <cell x="0" y="0" color="BLUE"></cell>
    <cell x="1" y="4" color="BLUE"></cell>
    <cell x="2" y="4" color="BLUE"></cell>
    <cell x="3" y="4" color="BLUE"></cell>
    <cell x="4" y="4" color="BLUE"></cell>
    <cell x="5" y="4" color="BLUE"></cell>
    <cell x="6" y="4" color="BLUE"></cell>
    <cell x="7" y="4" color="BLUE"></cell>
    <cell x="8" y="4" color="BLUE"></cell>
    <cell x="9" y="4" color="BLUE"></cell>
  </gridColors>
  <gridItems>
    <item x="0" y="4" img="input.png" val="31"></item>
  </gridItems>
</world>
```

Zdrojový kód 39: Konfigurační soubor simulace pro ukázku použití aplikace č. 1. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Zde je uveden obrázek grafického rozhraní aplikace, kde je vidět simulace v koncovém stavu a uživatelem připravený zdrojový kód, který je jedním z možných řešení výše definované úlohy. Zároveň je vidět textový výstup aktéra simulace.



Obrázek 9: Ukázka použití aplikace č. 1. Zdroj: autorem vytvořeno v aplikaci vyvíjené v rámci této práce a dále upraveno v MS Paint.

4.5.2 **Ukázka 2**

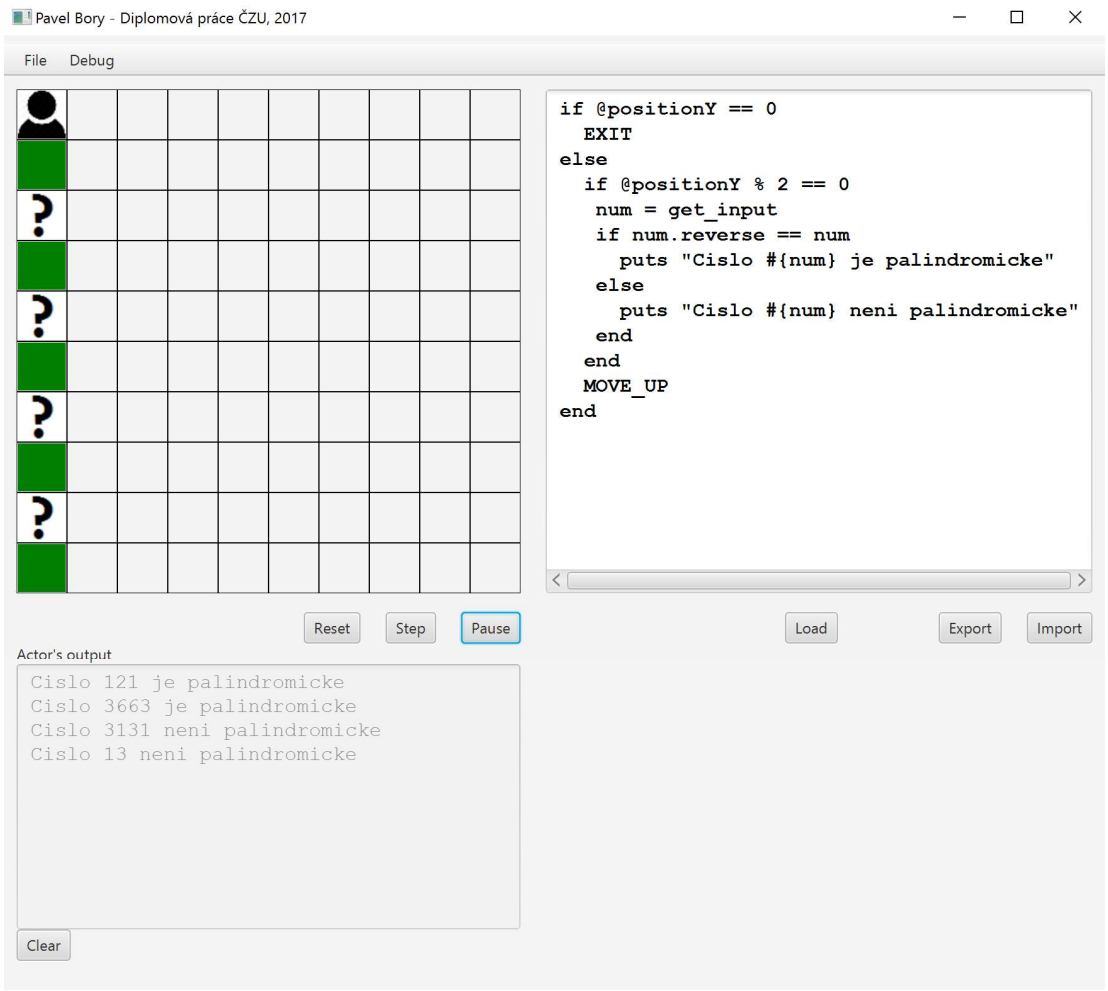
V této ukázce má uživatel za úkol napsat kód, ve kterém si procvičí práci s podmíněným zpracováním kódu a cykly. Aktér začíná v levém dolním rohu (tedy na souřadnicích [0,9]) a bude se pohybovat směrem nahoru dokud nedojde na okraj mřížky. V každém sudém řádku se nachází číslo a aktér má vypsat, zdali je dané číslo palindromatické či nikoliv. Zde je uvedena XML souboru definujícího svět této simulace:

```
<?xml version="1.0" encoding="UTF-8"?>
<world>
	<actor posX="0" posY="9" img="actor.png">
	 </actor>
	<gridColors>
		<cell x="0" y="9" color="GREEN"></cell>
		<cell x="0" y="7" color="GREEN"></cell>
		<cell x="0" y="5" color="GREEN"></cell>
		<cell x="0" y="3" color="GREEN"></cell>
		<cell x="0" y="1" color="GREEN"></cell>
	</gridColors>
```

```
	<gridItems>
		<item x="0" y="8" img="input.png" val="121"></item>
		<item x="0" y="6" img="input.png" val="3663"></item>
		<item x="0" y="4" img="input.png" val="3131"></item>
		<item x="0" y="2" img="input.png" val="13"></item>
		<item x="0" y="0" img="input.png" val="1234"></item>
	</gridItems>
</world>
```

Zdrojový kód 40: Konfigurační soubor simulace pro ukázku použití aplikace č. 1. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

Zde je uveden obrázek grafického rozhraní aplikace, kde je vidět simulace v koncovém stavu a uživatelem připravený zdrojový kód, který je jedním z možných řešení výše definované úlohy. Zároveň je vidět textový výstup aktéra simulace.



Obrázek 10: Ukázka použití aplikace č. 2. Zdroj: autorem vytvořeno v aplikaci vyvíjené v rámci této práce a dále upraveno v MS Paint.

5 Závěr

Tato práce se zabývala návrhem a implementací prototypu software pro podporu výuky objektově orientovaného programování. Software měl být navržen tak, aby umožnil tvořit programy, kterými budou ovládány grafické objekty v předpřipravených prostředích.

V teoretické části práce byly analyzovány pro tuto práci vhodné technologie a možnosti jejich integrace. Analýzována byla technologie JavaFX pro tvorbu uživatelského rozhraní aplikace a programovací jazyky Java a Ruby, přičemž jazyk Ruby byl zvolen jako programovací jazyk, kterým bude moci uživatel programovat chování aktéra v předpřipravených prostředích konfigurovatelných pomocí XML souborů, ve kterých bude moci interagovat s grafickými objekty. Pro integraci Java a Ruby byla zvolena technologie JRuby, která umožní jejich vzájemné propojení, přičemž v teoretické části byly analyzovány možné strategie této integrace. Aplikace bude pro usnadnění ladění a dalšího rozvoje provádět logování pomocí frameworku log4j.

Dále byly definovány funkční a nefunkční požadavky, navržena architektura aplikace, navrženo základní grafické rozhraní, definovány případy užití aplikace a byly navrženy klíčové třídy pro tuto aplikaci. V rámci návrhu tříd byl i na základě teoretických východisek navržen i způsob integrace Java a Ruby pomocí JRuby a byly popsány klíčové prvky pro tuto integraci.

Poté byl implementován prototyp aplikace, jehož klíčové prvky jsou řádně zdokumentovány a popsány v této práci. Zároveň byly připraveny a zdokumentovány dva příklady použití této aplikace. Aplikace byla na závěr uživatelsky otestována a její klíčové části zdrojového kódu jsou pokryty pomocí unit testů implementovaných za použití frameworku JUnit.

6 Seznam použitých zdrojů

1. **TIOBE Software BV.** TIOBE Index. *TIOBE Index.* [Online] 11. 11 2016. http://www.tiobe.com/tiobe-index/.

2. **Dave Thomas with Chad Fowler, Andy Hunt.** *Programming Ruby 1.9 & 2.0 The Pragmatic Programmers' Guide.* United States of America : The Pragmatic Programmers, LLC., 2013. 978-1-93778-549-9.

3. **Microsoft.** Creational Patterns: Prototype, Factory Method, and Singleton. *Microsoft Developer Network.* [Online] [Citace: 4. 2 2017.] https://msdn.microsoft.com/en-us/library/orm-9780596527730-01-05.aspx.

4. **Schildt, Herbert.** *Java 8 Výukový kurz.* Brno : Computer Press, 2016. 978-80-251-4665-1.

5. **Sharan, Kishori.** *Learn Java FX 8 Building User Experience and Interfaces with Java 8.* United States of America : Apress Media, LLC, 2015. 978-1-4842-1143-4.

6. **The Eclipse Foundation.** Efxclipse/Tooling/FXGraph. *The Eclipse Foundation open source community website.* [Online] 2017. [Citace: 5. 2 2017.] https://wiki.eclipse.org/Efxclipse/Tooling/FXGraph.

7. **Charles O Nutter, Thomas Enebo, Nick Sieger, Ola Bini, and Ian Dees.** *Using JRuby Bringing Ruby to Java.* United States of America : The Pragmatic Programmers LLC, 2011. 1-934356-65-4.

8. **Microsoft.** Test Early and Often. *Microsoft Developer Network.* [Online] 2017. [Citace: 5. 1 2017.]

9. ***junit4 Wiki.*** [Online] 2016. [Citace: 3. 1 2017.] https://github.com/junit-team/junit4/wiki.

10. Apache Software Foundation. Log4j - Configuring Log4j 2. [Online] 2017. [Citace: 30. 1 2017.] https://logging.apache.org/log4j/2.x/manual/configuration.html.

11. Jim Arlow, Ila Neustadt. *UML 2 a unifikovaný proces vývoje aplikací.* Brno : Computer Press, a.s., 2007. 978-80-251-1503-9.

12. McConnell, Steve. *Dokonalý kód.* Brno : Computer Press, a.s., 2006. 80-251-0849-X.

13. Ruby-Doc.org. *Dokumentace Ruby.* [Online] 2017. http://ruby-doc.org/.

14. Oracle and/or its affiliates. JavaFX 2.2. *Oficiální dokumentace.* [Online] 2014. https://docs.oracle.com/javafx/2/api/overview-summary.html.

15. ScriptingContainer. *Dokumentace JRuby.* [Online] 2015. http://jruby.org/apidocs/org/jruby/embed/ScriptingContainer.html.

16. The Eclipse Foundation. Efxclipse/Tooling/FXGraph. [Online] 2017. https://wiki.eclipse.org/Efxclipse/Tooling/FXGraph.

17. Apache Software Foundation. Apache Log4j 2. [Online] 2017. https://logging.apache.org/log4j/2.x/.

7 Přílohy

1. Příloha 1: Zdrojový kód metody step() třídy World
2. Příloha 2: Zdrojový kód metody updateDebugInfo třídy World
3. Příloha 3: Implementace Ruby kostry obsahující kód pro třídu Actor, do jejíž metody makeAction je nahráván uživatelem definovaný zdrojový kód
4. Příloha 4: Třída WorldGrid

7.1 Zdrojový kód metody step() třídy World

```
/**
    Performs one step of the Simulation, that consists of the following
    substeps:
      - Check whether the World is in correct state to perform next step (for
        example that it is not TERMINATED)
      - Call Ruby Actor makeAction code through JRubyContainer and save what
        Ruby Actor want's to do in this step of the Simulation
      - Gather Actor's text output (if any)
      - Based on the Actor's makeAction result perform requested action (if
        possible)
      - Re-draw World Grid after the step of the Simulation
      - Update Debug Info Dialog

     @param singleStep Determines whether the method is called within the
            Continuous Simulation Run Mode or not
     @throws WorldActorOutOfGridException
     @throws WorldActorUnsupportedOperationException
     @throws WorldNotInStateToPerformSimulationStep

     @author Pavel Bory
*/
public void step(boolean singleStep) throws WorldActorOutOfGridException,
  WorldActorUnsupportedOperationException,
  WorldNotInStateToPerformSimulationStep
{
  logger.info("Performing step. Single step: {}",singleStep);
  // Check if the WorldState enables to perform Actor's action
  if(singleStep){
    if(getState() == WorldState.TERMINATED || getState() ==
      WorldState.INITIAL){
      logger.info("World in state: {} can't perform next step",getState());
      throw new WorldNotInStateToPerformSimulationStep(String.format("World
        is in state: %s, so it cannot perform step of the
        Simultion",getState()));
    }else{
      setState(WorldState.PAUSED);
    }
  }else{
    if(getState() != WorldState.RUNNING){
      logger.debug("World in state {} can't perform a non single step. To
        make continuos steps it is expected that world is in state
        RUNNING",getState());
```

```
      return;
    }
  }

  // Call Ruby Actor's method that performs user programmed function and
  // store Actor's text output (if any) and append it to the GUI
  RubyActorActionType actionType =
jrubyContainer.makeActorAction(actor,getActualItemInputForActor(actor));
  String actorsOutputText = jrubyContainer.gatherOutputText();
  if(actorsOutputText != null && !actorsOutputText.isEmpty()){
    // Simplified solution of newline symbols conversion between Java and
    // Ruby
    //Strings. __NEWLINE__ is internally considered to represent
    // newline symbol that comes from the Ruby Actor. It should be handled
    // properly in next versions of this application and robust
    // special String symbols conversion mechanism should be implemented in
    // the JRubyContainer.
    actorsOutputText = actorsOutputText.replaceAll("__NEWLINE__", "\n");
    Main.appendActorsOutput(actorsOutputText);
  }

  // Perform requested action (if possible)
  logger.debug("Action type: {}",actionType);
  switch (actionType) {
    case MOVE_RIGHT:
      if(actor.getPositionX() >= WORLD_COLUMNS_COUNT - 1){
        setState(WorldState.TERMINATED);
        throw new WorldActorOutOfGridException("The actor went of of the grid
          on the x-axis");
      }else{
        actor.setPositionX(actor.getPositionX() + 1);
      }
      break;
    case MOVE_LEFT:
      if(actor.getPositionX() <= 0){
        setState(WorldState.TERMINATED);
        throw new WorldActorOutOfGridException("The actor went of of the grid
          on the x-axis");
      }else{
        actor.setPositionX(actor.getPositionX() - 1);
      }
      break;
    case MOVE_UP:
      if(actor.getPositionY() <= 0){
        setState(WorldState.TERMINATED);
        throw new WorldActorOutOfGridException("The actor went of of the grid
          on the y-axis");
      }else{
        actor.setPositionY(actor.getPositionY() - 1);
      }
      break;
    case MOVE_DOWN:
      if(actor.getPositionY() >= WORLD_ROWS_COUNT - 1){
        actor.setPositionY(actor.getPositionY() + 1);
      }else{
        setState(WorldState.TERMINATED);
```

```
        throw new WorldActorOutOfGridException("The actor went of of the grid
          on the y-axis");
      }
      break;
    case TERMINATE:
      setState(WorldState.TERMINATED);
      break;
  }
  drawCurrentState();
  updateDebugInfo();
  logger.info("Performing step. Single step: {} - COMPLETED",singleStep);
}
```

Příloha 1: Zdrojový kód metody step() třídy World. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.

7.2 Zdrojový kód metody updateDebugInfo třídy World

```
/**
    Constructs Debug Info String (StringBuilder used for optimized string
    creation) and passes it to the GUI
*/
private void updateDebugInfo(){
  logger.debug("Constructing debug info string");
  StringBuilder sBuilder = new StringBuilder();
  sBuilder.append("ACTOR:\n");
  sBuilder.append(String.format("Pos[x,y]: [%s,%s]\n",
    actor.getPositionX(),actor.getPositionY()));
  HashMap<String, Object> actorData = jrubyContainer.getActorData();
  for(String key : actorData.keySet()){
    sBuilder.append(String.format("data[%s] = %s\n",key,actorData.get(key)));
  }

  sBuilder.append("\n\nGRID ITEMS:\n");
  for(WorldItem worldItem : worldSettings.getWorldItems()){
    sBuilder.append(String.format("Pos[x,y]: [%s,%s], val: %s\n",
    worldItem.getPositionX(),worldItem.getPositionY(),worldItem.getValue()));
  }

  sBuilder.append("\n\nWORLD:\n");
  sBuilder.append(String.format("State: %s\n",getState()));
  sBuilder.append(String.format("Actor initial position X:
    %s\n",worldSettings.getActorInitialPositionX()));
  sBuilder.append(String.format("Actor initial position Y:
    %s\n",worldSettings.getActorInitialPositionY()));
  sBuilder.append(String.format("Columns count: %s\n",WORLD_COLUMNS_COUNT));
  sBuilder.append(String.format("Rows count: %s\n",WORLD_ROWS_COUNT));
  sBuilder.append(String.format("Timer interval ms: %s\n",TIMER_PERIOD));
  sBuilder.append(String.format("Timer delay ms: %s\n",TIMER_DELAY));

  logger.debug("Constructing debug info string - COMPLETED");
  Main.setDebugDialogText(sBuilder.toString());
}
```

Příloha 2: Zdrojový kód metody updateDebugInfo()třídy World. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse

7.3 Implementace Ruby kostry obsahující kód pro třídu Actor, do jejíž metody makeAction je nahráván uživatelem definovaný zdrojový kód.

```
require "java"
require "Singleton"
java_import "application.world.ruby.IRubyActor"
java_import "application.world.ruby.RubyActorActionType"
java_import "java.util.HashMap"

MOVE_LEFT = RubyActorActionType::MOVE_LEFT
MOVE_UP = RubyActorActionType::MOVE_UP
MOVE_RIGHT = RubyActorActionType::MOVE_RIGHT
MOVE_DOWN = RubyActorActionType::MOVE_DOWN
EXIT = RubyActorActionType::TERMINATE

class Actor
  include IRubyActor
  include Singleton

  def self.actor_code
    @@actor_code
  end


  def loadActorActionMethod(code)
    @@actor_code = code

    def self.makeAction
      eval(Actor.actor_code)
    end
  end

  def setPositionX(position)
    @positionX = position
  end

  def setPositionY(position)
    @positionY = position
  end

  def setInput(input)
    @input = input
  end

  def get_input
    return @input
  end

  def getData
    return @data
```

```
  end

  def print(text)
    if @output == nil
      @output = text
    else
      @output += text
    end
  end

  def puts(text)
    print(text + "__NEWLINE__")
  end

  def gatherOutputText
    result = @output
    @output = nil
    return result
  end

  def set_data(key,value)
    if(@data == nil)
      @data = HashMap.new
    end
    @data.put(key.to_s,value)
  end

  def get_data(key)
    if(@data == nil)
      @data = HashMap.new
    end
    return @data.get(key)
  end
end
```

Příloha 3: Implementace Ruby kostry obsahující kód pro třídu Actor, do jejíž metody makeAction je nahráván uživatelem definovaný zdrojový kód. Zdroj: autorem vytvořeno ve vývojovém prostředí RubyMine.

7.4 Třída WorldGrid

```
public class WorldGrid {
  private final int cellWidth;
  private final int cellHeight;

  private Canvas canvas;

  public WorldGrid(Canvas canvas){
    this.canvas = canvas;
    cellWidth = ((Double)canvas.getWidth()).intValue() /
      World.WORLD_COLUMNS_COUNT;
    cellHeight = ((Double)canvas.getHeight()).intValue() /
      World.WORLD_ROWS_COUNT;
  }

  public void drawGrid(Color[][] gridCellColors){
    GraphicsContext gc = canvas.getGraphicsContext2D();
    gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());
```

```
    gc.setStroke(Color.BLACK);
    // vertical lines
    for(int i = 0 ; i <= World.WORLD_COLUMNS_COUNT ; i++){
      int x = i * cellWidth;
      gc.strokeLine(x, 0, x, canvas.getHeight());
    }

    // horizontal lines
    for(int i = 0 ; i <= World.WORLD_ROWS_COUNT ; i++){
      int y = i * cellHeight;
      gc.strokeLine(0, y, canvas.getWidth(), y);
    }

    if(gridCellColors != null){
      for(int col = 0; col < World.WORLD_COLUMNS_COUNT; col++){
        for(int row = 0; row < World.WORLD_ROWS_COUNT; row++){
          if(gridCellColors[col][row] != null){
            gc.setFill(gridCellColors[col][row]);
            gc.fillRect(col * cellWidth + 1, row * cellHeight + 1, 38, 38);
          }
        }
      }
    }
  }

  public void drawActor(long x, long y,Image actorImg){
    GraphicsContext gc = canvas.getGraphicsContext2D();
    gc.drawImage(actorImg, x * cellWidth + 1, y * cellHeight + 1);
  }

  public void drawItems(WorldItem[][] items){
    GraphicsContext gc = canvas.getGraphicsContext2D();
    if(items != null){
      for(int col = 0; col < World.WORLD_COLUMNS_COUNT; col++){
        for(int row = 0; row < World.WORLD_ROWS_COUNT; row++){
          if(items[col][row] != null){
            gc.drawImage(items[col][row].getImg(), col * cellWidth + 1, row *
              cellHeight + 1);
          }
        }
      }
    }
  }
}
```

Příloha 4: Zdrojový kód třídy WorldGrid. Zdroj: autorem vytvořeno ve vývojovém prostředí Eclipse.