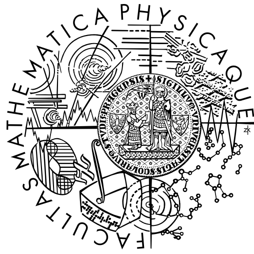


MATEMATICKO-FYZIKÁLNÍ FAKULTA
Univerzita Karlova

BAKALÁŘSKÁ PRÁCE

Ondřej Tichavský

Program pro vytváření příběhů

Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D.
Studijní program: Informatika
Studijní obor: Programování a vývoj software Bc.

Praha 2024

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů. Tato práce nebyla využita k získání jiného nebo stejného titulu.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona v platném znění, zejména skutečnost, že Univerzita Karlova má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle §60 odst. 1 autorského zákona.

V .... dne ....

Podpis autora

Chtěl bych poděkovat vedoucímu práce, panu RNDr. Martinu Pergelovi, Ph.D., za jeho cenné rady a pomoc při vedení této práce.

Název práce: Program pro vytváření příběhů

Autor: Ondřej Tichavský

Katedra: Katedra softwaru a výuky informatiky

Vedoucí bakalářské práce: RNDr. Martin Pergel, Ph.D., Katedra softwaru a výuky informatiky

Abstrakt: Cílem projektu „WorldsFactory“ je vytvořit software umožňující tvorbu komplexních, logických a rozsáhlých příběhů. Software bude uchovávat uživatelské nápady, myšlenky a hotové části příběhu, vizualizovat je pomocí nástrojů pro vyhledávání, zobrazovat timeline, vizuální mapy a grafy vztahů mezi koncepty. Bude také zahrnovat možnost tvorby rozvětvených příběhů s více možnostmi událostí na základě definovaných podmínek, což je užitečné pro tvorbu gamebooků nebo her. Software umožňuje export dat do knihoven, které sledují průběh příběhu a jsou integrovatelné s herními enginy. Projekt zahrnuje i implementaci jednoduchého herního enginu, demonstrujícího funkčnost jedné takové přídavné knihovny.

Klíčová slova: Přiběh, Gamebook, Herní engine, Graf událostí

Title: Program for stories creation

Author: Ondřej Tichavský

Department: The Department of Software and Computer Science Education

Supervisor: RNDr. Martin Pergel, Ph.D., The Department of Software and Computer Science Education

Abstract: The goal of the "WorldsFactory" project is to create software enabling the creation of complex, logical and extensive stories. The software will store the user’s ideas, thoughts and finished parts of the story, visualize them using search tools, display timelines, visual maps and graphs of relationships between concepts. It also includes the ability to create branching stories with multiple event options based on defined conditions, which is useful for creating gamebooks or games. The software allows data to be exported to libraries that track the progress of the story and can be integrated with game engines. The project also includes the implementation of a simple game engine, demonstrating the functionality of the additional library.

Keywords: Story, Gamebook, Game engine, Event graph

Obsah

Úvod 4

I Definice příběhu 6

1 Nerozvětvený příběh a definice nerozvětveného světa 8
- 1.1 Definice nerozvětveného světa 8
  - 1.1.1 Účel příběhu 9
- 1.2 Alternativní definice světa 9

2 Popis příběhu 10
- 2.1 Popis nerozvětveného příběhu 10
  - 2.1.1 Vnoření událostí 10
  - 2.1.2 Cestování časem 11
- 2.2 Popis rozvětveného příběhu 11
  - 2.2.1 Příběh pro Gamebooky 12
  - 2.2.2 Rozvětvený příběh a rozvětvený svět 14
- 2.3 Praktická vylepšení rozvětveného světa 15
  - 2.3.1 Graf událostí 15
  - 2.3.2 Povolení podmínkových skoků metody události 16
  - 2.3.3 Lineární událost 16

II Worlds Factory 18

3 Hlavní struktura programu 20

4 Tvorba světa 22
- 4.1 Knihovny 22
- 4.2 Vlastnosti události 22
- 4.3 Zobrazování světa 23
  - 4.3.1 Zobrazování stavu světa 23
  - 4.3.2 Zobrazovací mapa 24
  - 4.3.3 Zobrazování grafů 24
  - 4.3.4 Zobrazování grafu událostí 24
- 4.4 Ukládání dat 25
- 4.5 Interpret 25
- 4.6 Identifikátory a Tagy 26

5 Uživatelská dokumentace 27
- 5.1 Instalace 27
- 5.2 Management projektu 27
  - 5.2.1 Otevření projektu 27
  - 5.2.2 Vytvoření nového projektu 29
- 5.3 Vytváření světa 31
  - 5.3.1 Vytváření třídy, objektu, událostí 32

5.3.2 Vytváření property, nebo metody 33
5.4 Úprava událostí 36
5.4.1 Zobrazování vztahů mezi eventy 38
5.4.2 Event container 39
5.5 Export dat pro herní engine 40
5.6 Ukázky jdenoduchých projektů 41

6 Architektura programu 42
6.1 Stakeholders 43
6.2 Uživatelské požadavky 43
6.3 Architektura 44
6.4 Implementace 45
6.4.1 Prostředí uživatelského rozhraní 46
6.4.2 Management Projektů 47
6.4.3 World Editor 47
6.4.4 Ukládání a načítání dat pomocí Loaderů 48
6.4.5 Library Editor, Object Editor, Event Editor 48
6.4.6 Identifikátory a Tagy 49
6.4.7 Story Editor 49
6.4.8 Zobrazování světa 51
6.4.9 Generování dokumentace 52
6.5 AvaloniaGraphs 52
6.5.1 Rozhraní 52
6.5.2 Implemetnace layoutovacího algoritmu 56

7 WorldsFactory Knihovny 57
7.1 Motivace 57
7.2 Moment posunu v příběhu 57
7.2.1 Kontrola při každé změně hodnoty 57
7.2.2 Kontrola při signálu 57
7.3 Implementace 57
7.3.1 Uživatelské rozhraní 58

III Herní engine 61

8 Spuštění příkladu herního enginu 62
8.1 Spuštění 62
8.2 Příklad spuštění prvního příběhu - Derek a vlci 63

9 Základní popis herního enginu 67
9.1 Příběh 67

10 Implementace příběhu do herního enginu 68
10.1 Přidání WorldsFactoryJavaLibrary knihovny do projektu 68
10.2 Inicializace WorldsFactoryJavaLibrary knihovny 68
10.3 Svazování tříd s příběhem 69
10.4 Vylepšení mobilního klienta 69
10.5 Generování dokumentace 70

Závěr **71**

Seznam použité literatury **72**

Seznam obrázků **73**

Úvod

Historie příběhu

Příběhy popisují události v libovolném světě a tudíž i v našem světě. Jsou tu s námi tedy od nepaměti. V minulosti se příběhy přenášely z ústního podání, kdy se přenášely z generace na generaci. Nevýhodou bylo, že při přenášení mohlo dojít k jeho pozměnění. Zároveň paměť lidí je značně omezená, tudíž tvorba soudržitých a konzistentních příběhů byla omezena. Problém ústního předávání byl vyřešen až s rozvojem technologie, kdy se začaly zaznamenávat do knih. Nejenže to umožnilo přenášení příběhů bez jejich změnění, ale zároveň se zjednodušila tvorba. Tvorba příběhů se stala jednodušší díky tomu, že se mohly zaznamenávat informace o světě do poznámek.

Tvorba rozvětvených příběhů, kdy směr příběhu ovlivňuje posluchač, respektive čtenář, byla možná ve formě gamebooku, nicméně mnohem relevantnější začala být při nastupu digitálních technologií, které umožnily tvorbu interaktivních her.

Tvorba příběhů se stala mnohem jednodušší díky efektivnímu zaznamenávání informací o světě. Jak již bylo řečeno, lidská mysl není schopna si zapamatovat veškeré detaily o světě, pro zachování soudržnosti a projevení souvislostí. Nové technologie umožňují tvorbu větších, komplexnějších, detailnějších světů s minimem logických chyb. Bohužel i dnes je běžné, že příběh je děravý jako ementál.

U velmi rozsáhlých světů je výskyt chyb více pochopitelný, obzvláště u dlouhých seriálů, protože platí pravidlo „čím větší dílo, tím složitější svět“. Jako příklad může sloužit seriál Teorie velkého třesku, kde jedna z hlavních postav, jménem Sheldon Cooper, prohlašuje, že v dětství neměl žádného kamaráda, ale v pozdějších sériích se dozvídáme, že alespoň jednoho kamaráda měl.

Je to příklad nelogičnosti, kterého se snažíme vyvarovat, a kdyby autor příběhu zaznamenával informace o světě v průběhu tvorby, nemuselo by k tomu dojít.

Klarifikace zadání

Cílem tohoto projektu je navrhnout software, nástroj, pomocí něhož bude možné vytvářet příběhy. Tím je myšleno, že má být schopen uchovávat uživatelovy nápady, myšlenky, jeho hotové části příběhu, vztahy mezi koncepty příběhu a aby přehledně všechny tyto informace vizualizoval na základě poskytnutých nástrojů pro vyhledávání v těchto datech. Může se jednat o zobrazování timeline, vizuálních map, grafů vztahů mezi koncepty v závislosti na podmínkách konkrétních vlastnostech, vyhledávání v textu, vyhledávání podle podmínek...

Tím program umožní, že tvůrce bude mít možnost vytvářet složité, promyšlené, rozsáhlé a logické příběhy.

Práce hledá obecnou definici příběhu, dle které je následně software strukturován tak, aby pokrýval co nejvíce uživatelských požadavků týkajících se tvorby příběhu. Tato obecnost mimo jiné zahrnuje i možnost vytvářet příběhy, které jsou

rozvětvené. V nějaký čas může tedy nastat jedna z více možností událostí, na základě definované podmínky. Taková vlastnost je užitečná například při tvorbě příběhu pro gamebooky nebo hry.

Software pak také podporuje exportování dat příběhu. Ty jsou dále přebírány přídavnými knihovnami, které umožňují sledovat dění v příběhu a postupné posouvání v něm. Vnitřní data knihovna zpřístupňuje jiným aplikacím, jako například herním enginům.

V další části se práce zabývá počáteční implementací návrhu, tedy konkrétnějším popisem jeho struktury a návrhu softwaru. Rozdělení projektu do kontejnerů a jednotlivých modulů. Pracovní název je *WorldsFactory*.

V následující části se pak projekt zabývá jednoduchým herním enginem, který má využívat přídavnou Java knihovnu a demonstrovat její funkčnost a možný způsob použití.

Podobné programy

Většina podobných programů, se kterými jsem se setkal, se zaměřuje pouze na vybrané časti zadání mého projektu. Software, který by se pokoušel o totéž jsem nenašel (to neznamená že neexistuje). Zde jsou hlavní typy programů, se kterými jsem se setkal:

- Jedním typem jsou programy pro vytváření příběhu, které obvykle ukazují příběh rozdělený do kapitol s několika osami příběhovými linkami. Často oplývají možností přidávat do příběhu postavy, události, místa a podobně. Příkladem takovéto aplikace může být software se jménem Plottr [1].

  Velikou nevýhodou takovýchto programů je nemožnost definovat události světa, které nejsou součástí příběhu. Zároveň program podporuje pouze příběh v nerozvětvené formě.

- Programy pro tvorbu světa - jako například WorldAnvil [2] - Podobný program, jako náš projekt, až na to, že neumožňuje psát rozvětvené příběhy a nelze používat interpreter. Je schopen popisovat svět a na základě tohoto světa dále vytvářet konkrétní dílo.

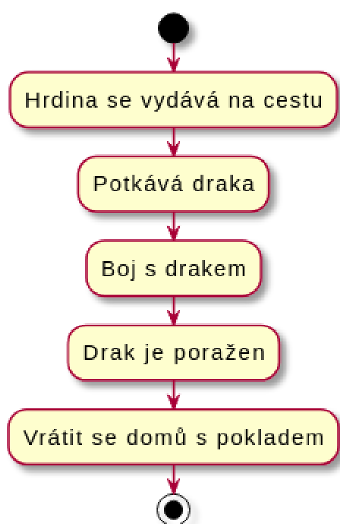
- Programy pro tvorbu rozvětvených příběhů - jako například Twine [3] - Tento program podporuje psát rozvětvené příběhy zároveň s programováním vlastního kódu, nicméně neumožňuje vytvářet vlastní svět, alespoň ne jednoduchým způsobem. Zároveň nedokáže data přehledně zobrazovat. Vytvořený kód pak lze také exportovat a použít v jiné aplikaci.

Část I

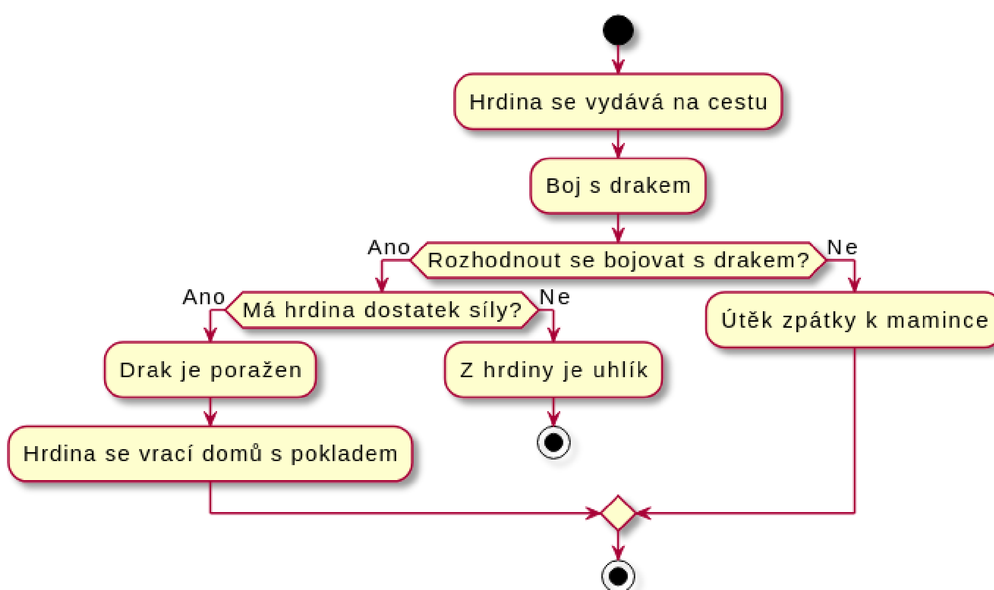
Definice příběhu

Před tím, než se ponoříme do samotného pátrání po vhodné definici příběhu, ujasníme si základní terminologii nad intuitivní definicí příběhu a světa. V případě, mluvíme-li o lineárním příběhu, jedná se o příběh s jedinou dějovou linií. Pokud je nelineární, může příběh přeskakovat mezi více dějovými linkami. Pak také rozlišujeme rozvětvený, respektive nerozvětvený příběh. Rozvětvený příběh je příběh, kdy se vypravěč může na základě nějakých podmínek rozhodnout, jak bude příběh pokračovat. Dvě vypravování stejného příběhu se pak mohou lišit.

Nerozvětvený příběh



Rozvětvený příběh



Obrázek 1: Rozvětvený a nerozvětvený příběh

1. Nerozvětvený příběh a definice nerozvětveného světa

Co je to příběh? Jelikož se jedná o termín, který má na první pohled nejednoznačnou a těžko vyjádřitelnou definici, pokusíme se nejprve podívat na konkrétnější typ příběhu, který by mohl být intuitivnější.

Vezmeme v úvahu příběh, který se odehrává v našem reálném světě. Nejklasičtější situací, kdy mluvíme o příběhu, je případ, kdy se snažíme popsat událost, která se nám v minulosti přihodila. Obvykle se takový příběh skládá z několika navazujících událostí. Každá událost se odehrává v konkrétním čase a zahrnuje nějaké objekty světa, obvykle postavy nebo předměty, které se v závislosti na čase a fyzikálních zákonech nějak mění nebo něco provádějí.

Dalším typem příběhu je například popis událostí ve stylu otázky "Co by kdyby", popisující události, které reálně nenastaly.

Z našich příkladů můžeme vyvodit, že příběh závisí na konkrétním světě. Pojem svět vychází z pozorování pomocí našich smyslů okolního reálného světa. Světem považujeme například náš reálný svět, tj. vše, co pozorujeme. Nicméně svět může být také smyšlený.

1.1 Definice nerozvětveného světa

Obecně by se dal svět zadefinovat jako uspořádaná čtveřice, která zahrnuje kolekci konceptů, kolekci stavu světa, kolekci fyzikalnich zákonů a čas.

Časem je myšlena libovolná množina, jejíž prvky se nazývají *okamžiky*. Často je časem myšleno datum a denní čas, může tím ovšem být cokoli. Často je místo okamžiku použito spojení „konrkétní čas“.

Kolekci konceptů lze de facto chápat jako libovolnou množinu, nicméně představuje soubor všech objektů, které existují ve světě (nebo se v něm někdy vyskytovaly). Tyto objekty mohou mít své vlastnosti, jež mohou nabývat hodnot v konkrétním okamžiku.

Kolekce stavů světa je množina, kde stav světa představuje soubor vlastností všech konceptů a je identifikován nějakým okamžikem.

Nakonec kolekce fyzikálních zákonů představuje množinu pravidel, které určují závislosti stavů světa mezi sebou a způsob, jakým dochází ke změnám vlastností konceptů mezi dvěma stavy světa. Popřípadě mohou fyzikální zákony omezovat hodnoty vlastností konceptů.

Tato definice se na první pohled zdá být velmi obecná. Dokonce tak obecná, že by v něm mohl být popsán i libovolný rozvětvený příběh. Bohužel, opak je pravdou, protože existují případy, které bychom chtěli také označit za světy, ale neodpovídají této definici. Hlavním problémem této definice je konstantnost času. Při rozvětveném příběhu musíme počítat i s větvením nezávislým na čase. Stav světa by pak nemohl být identifikován pomocí času. Z toho důvodu tuto definici budeme nazývat *nedokonalou definicí nerozvětveného světa*.

Mějme například svět tvořený koncepty hlavní hrdina, vedlejší postavy, hlavní záporná postava, nějaká místa, předměty a tak dále. Svět je dále tvořený stavy, kde stav světa je popsán pro konkrétní čas a zahrnuje vlastnosti všech konceptů. Například, jakou náladu má hlavní hrdina a kde se nachází. Fyzikální zákony pak mohou určovat pravidla světa. Například, že pozice nějaké postavy může mít hodnotu pouze jednoho konkrétního místa, nebo že seslání kouzla stojí postavu určité množství její energie.

Problém této definice pak nastane v případě, chceme-li říci, že ve světě dojde k nějaké události, při které si může hlavní hrdina vybrat, jak se bude chovat. Konkrétní stav světa je identifikován okamžikem.

1.1.1 Účel příběhu

Každý příběh vznikl s určitým účelem, buď byl vytvořen pro zábavu, poučení nebo jen jako záznam událostí, který by mohl být později užitečný. Zároveň lidská představivost je závislá na datech získaných pomocí našich smyslů v reálném světě. Příběhy ve světech, které se příliš liší od reálného světa, jsou velmi obtížně popsatelné. Alespoň tak obtížně, že čtení není příliš zábavné pro čtenáře.

Při hledání definice jsme se snažili o co nejobecnější definici světa tak, aby pokryl všechny případy, které bychom chtěli považovat za svět. To se nám nepodařilo, a tak jsme se omezili na konkrétní typy příběhů.

Praktičtější, stále ale nedokonalou definicí pro některé připady by byla definice zahrnující časovou osu, která uspořádává události. Zároveň bychom mohli zavést prostor, jakožto množinu míst, kde událost by byla změnou vlastností množiny konceptů v konkrétním čase a na konkrétním místě v prostoru.

1.2 Alternativní definice světa

Místo stavu světa bude svět tvořen uspořádaným seznamem událostí, kde událost je množina změn vlastností konceptů. Jinak bude svět strukturován podobně jako u předchozí definice, tedy množinou konceptů, časem a množinou fyzikálních zákonů. Z množiny konceptů půjde vybrat podmnožinu, kterou lze označit za prostor a jejím prvkům se bude říkat místa. Častokrát, místo pro svět intuitivnější pojem stav světa, se používá alternativní pojem *událost*. S touto definicí se dále lépe pracuje při uvažování nad definicí příběhu, proto v následujících kapitolách budeme upřednostňovat tuto definici.

2. Popis příběhu

Před samotnou implementací programu pro tvorbu příběhu je dobré si nejprve ujasnit, co vlastně považujeme za příběh, jaké jsou různé typy a jakým způsobem jej lze popsat. To povede k obecné definici, která bude základem jádra programu.

Bez důkladnějšího pozorování, intuitivně, by příběh měl být něco jako seřazený seznam informací, obvykle podle časové linie, pojednávající o vývoji vlastností relativně malé množiny konceptů (pojem *koncept* vychází z definice světa, viz předchozí kapitola).

2.1 Popis nerozvětveného příběhu

Výhradně nerozvětvený příběh se vyskytuje v knihách (kromě herních knih) nebo ve scénářích. Jeho průběh není závislý na nějaké podmínce a při každém vypravování příběhu se odehrává stejně. Svět můžeme popsat pomocí konceptů a jejich vlastností v konkrétním čase. Změny vlastností se následně dějí při událostech (eventech), které nastávají v určité době. Jelikož naše představivost neumožňuje pracovat s něčím tak abstraktním jako svět bez prostoru, můžeme o každém příběhu říkat, že se odehrává v nějakém prostoru a čase.

Díky jednoduchosti nerozvětveného příběhu můžeme popsat přesný stav světa v určitém čase. To je vlastnost, které bude chtít každý spisovatel využít. Náš budoucí program by tedy měl mít možnost definovat čas a následně dle konkrétní hodnoty času si zobrazit aktuální stav světa.

Díky tomu, že příběh v knize nebo ve scénáři nikdy nevypráví o více věcech naráz, můžeme příběh popsat jako seřazený seznam událostí, nebo konceptů v konkrétním čase. Příběh nemusí být tvořen pouze událostmi, protože spisovatel může chtít ve svém díle popsat pouze stav světa v konkrétním čase, nikoli změnu stavu.

Takto popsaný příběh může být dále rozšířen a pozměněn podle potřeb konkrétního typu díla. Například při tvorbě knihy může být příběh rozdělen do kapitol, které se dále dělí na podkapitoly s případnými poznámkami. V případě scénáře může být příběh rozdělen do jednotlivých scén, které se dále dělí na jednotlivé záběry.

2.1.1 Vnoření událostí

Jelikož hledání vhodného popisu příběhu bude využito při tvorbě softwaru, který má být užitečným nástrojem pro práci spisovatele, předestřu již nyní jedno praktické vylepšení naší definice, částečně z důvodu, že při následujících úvahách se toto vylepšení využije nepřímo, tedy ne ze stejného důvodu.

Jakožto spisovatel při své práci budeme chtít popisovat náš smyšlený svět a události v něm, ovšem častokrát ne do konkrétních detailů. Kupříkladu v našem smyšleném světě se bude odehrávat událost královská svatba. Ta bude probíhat v přibližném časovém úseku a zároveň tento event (událost) bude ve výsledku pouhé shrnutí mnoha jiných eventů, které jsou jeho součástí. Například přípravy

svatby, předsvatební noc a ráno, kdy se může něco důležitého dít, ceremonie, vražda nevěsty, následný chaos. . . V těchto časech mohou nastat jiné eventy, které jsou součástí královské svatby a ty se mohou dále dělit na další konkrétnější a podrobnější eventy. Takovým událostem můžeme říkat *podeventy* a *nadeventy*. V naší definici se jedná o rozšíření definice události, která může mimo změn vlastností konceptů světa obsahovat také množinu podeventů. Každý podevent ovšem nesmí trvat příliš dlouho, tak, aby nepřesahoval délku nadeventu.

2.1.2 Cestování časem

Můžeme pozorovat jisté zavádějící situace, kdy se spisovatel může snažit popsat něco pomocí nerozvětveného světa, protože by se na první pohled mohlo zdát, že je svět v dané situaci nerozvětvený, přičemž daná situace není danou definicí popsatelná. Příkladem budiž svět, ve kterém dochází k cestování v čase. Cestování samo o sobě je nelogické (časový paradox). Spisovatel tedy vždy musí definovat zákony, jakými se jeho svět řídí a jakými paradox řeší. Častým způsobem je deklarace fyzikálního zákonu, který říká, že při cestování do minulosti se cestuje do paralelního vesmíru, který je do doby vracení stejný jako předchozí.

Takový svět by se dal popsat vytvořením paralelního vesmíru, který je v jiném prostoru. Software by mohl umožňovat vytvoření kopie světa (všech konceptů a událostí s označením do jakého paralelního vesmíru spadá). Zde narážíme na hranice definice nerozvětveného světa. Chtěli bychom každé kopii přiřadit svůj speciální čas, ale zároveň čas byl definován pomocí pouze jedné časové osy.

Problém tedy není řešitelný pomocí nerozvětveného příběhu a je tedy potřeba sáhnout po obecnějším typu světa, tedy světu rozvětveném. V takovém případě se jednotlivé paralelní vesmíry začnou lišit až v nějaké chvíli. V té chvíli dochází ke větvení světa. Všimněme si, že jedna větev závisí na následujícím vývoji příběhu z druhé větve. K vybrání správné větve v příběhu dochází na základě nějaké podmínky. To implikuje, že podmínka může být závislá dokonce i na vývoji světa z jiné větve.

2.2 Popis rozvětveného příběhu

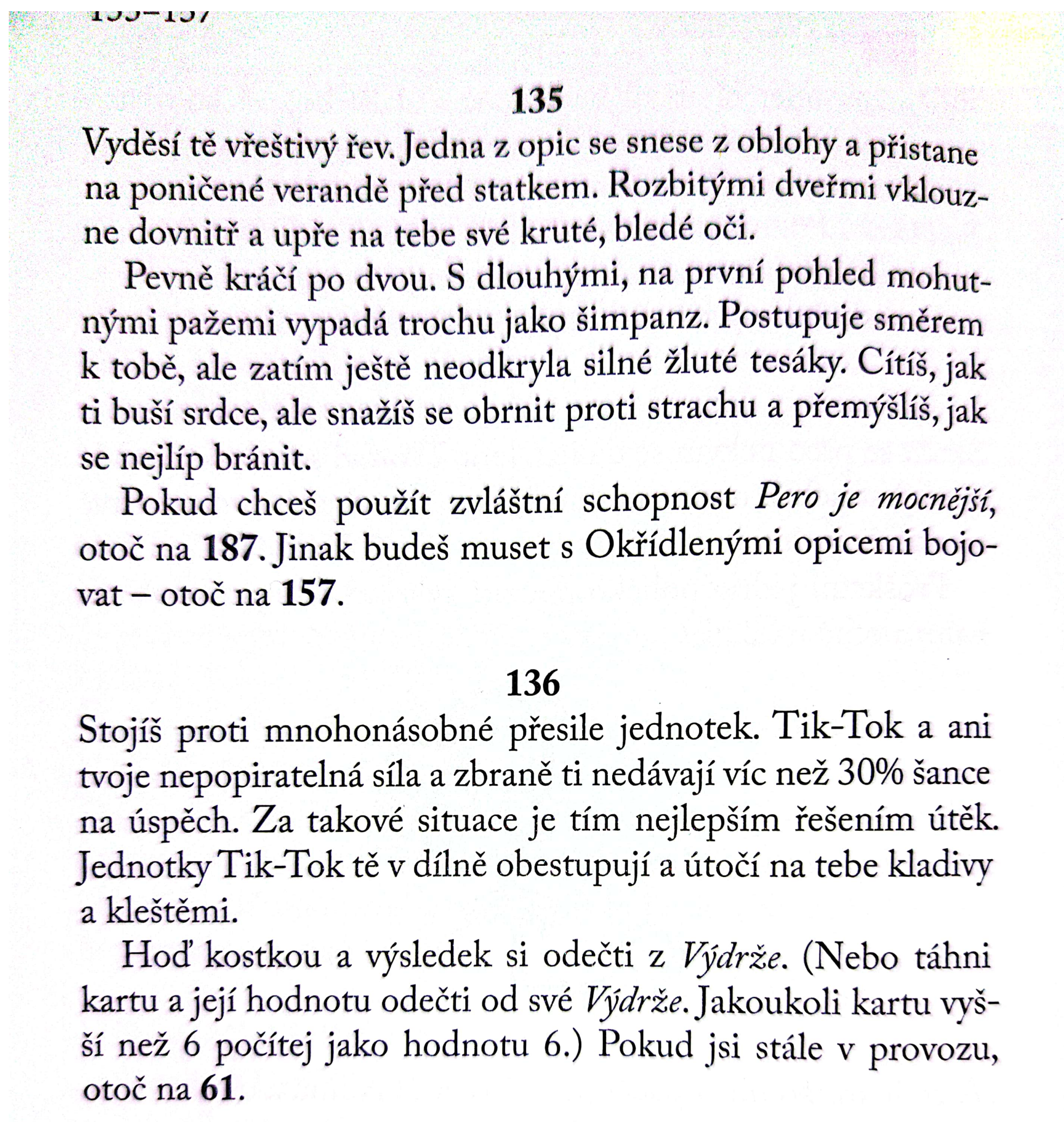
Doposavad bylo vždy jednoznačné, jakou hodnotu má nějaká vlastnost určitého konceptu v konkrétním čase. Nyní, ovšem, o tuto vlastnost přicházíme. Příběh se může rozvětvovat, tedy v nějaký čas může nastat jedna z více možností událostí.

Veškeré vlastnosti budou tedy proměnné, a pokud budeme sledovat stav světa, hodnoty proměnných budou udávat naposledy nastavenou hodnotu (software by ideálně mohl držet odkaz na místo, kde byla hodnota naposledy změněna).

Pro popis rozvětveného příběhu budeme postupovat po menších krocích, protože je tento problém o něco složitější. Podíváme se tedy nejprve na to, jakým způsobem lze popsat příběh pro gamebooky a až poté se podíváme na definici rozvětveného světa, ze které bude dále vycházet definice rozvětveného příběhu.

2.2.1 Příběh pro Gamebooky

Gamebooky jsou knihy, které umožňují čtenáři rozhodovat o tom, jak se bude příběh dále vyvíjet. Obvykle uživatel čte určitou část textu a na konci má možnost vybrat si z několika různých událostí, z kterých nastane pouze jedna.



135

Vyděsí tě vřeštivý řev. Jedna z opic se snese z oblohy a přistane na poničené verandě před statkem. Rozbitými dveřmi vklouzne dovnitř a upře na tebe své kruté, bledé oči.

Pevně kráčí po dvou. S dlouhými, na první pohled mohutnými pažemi vypadá trochu jako šimpanz. Postupuje směrem k tobě, ale zatím ještě neodkryla silné žluté tesáky. Cítíš, jak ti buší srdce, ale snažíš se obrnit proti strachu a přemýšlíš, jak se nejlíp bránit.

Pokud chceš použít zvláštní schopnost *Pero je mocnější*, otoč na **187**. Jinak budeš muset s Okřídlenými opicemi bojovat – otoč na **157**.

136

Stojíš proti mnohonásobné přesile jednotek. Tik-Tok a ani tvoje nepopiratelná síla a zbraně ti nedávají víc než 30% šance na úspěch. Za takové situace je tím nejlepším řešením útěk. Jednotky Tik-Tok tě v dílně obestupují a útočí na tebe kladivy a kleštěmi.

Hoď kostkou a výsledek si odečti z *Výdrže*. (Nebo táhni kartu a její hodnotu odečti od své *Výdrže*. Jakoukoli kartu vyšší než 6 počítej jako hodnotu 6.) Pokud jsi stále v provozu, otoč na **61**.

Obrázek 2.1: Úryvek z gamebooku *Oz - gamebook* od Jonathana Greena[4]

Velikou změnou oproti lineárnímu příběhu je, že čas může být relativní. Nemusí plynout rovnoměrně, často se ve hrách stává, že hráč hraje mimo čas. Respektive v rámci příběhu se nachází v nějakém přibližném čase, ale jinak si může dělat co chce a jak dlouho chce bez ohledu na čas v příběhu. To platí i pro gamebooky.

Příkladem může být otevřený svět, kde se může hráč volně pohybovat a přecházet mezi jednotlivými místy. Text pak obvykle představuje popis místa a dané možnosti představují typy akcí, které může postava provádět. Přesun z místa na místo pak neposune čas ve světě.

Příběh bychom si mohli představit jako strom, kde každý uzel představuje text a hrany představují jednotlivé možnosti, mezi kterými uživatel může vybírat. Při výběru možnosti se pak změní stav světa pro přehlednost spisovatele.

Takovýto popis příběhu má několik výhod:

1. Při posunu v příběhu a po konečně mnoha krocích dorazíme na konec příběhu.
2. Konkrétní uzel grafu jednoznačně určuje předešlý vývoj světa, a tedy při výběru uzlu může software zobrazit aktuální stav světa.
3. Strom lze spisovateli přehledně zobrazit a spisovatel má jasnou kontrolu nad tím, jakými průchody čtenář může projít.

Tato jednoduchost je ovšem vykoupena svoji neobecností, protože:

1. Spisovatel se často snaží o slučování dvou větví příběhu, aby nedošlo ke přílišnému rozvětvení. Tím vzniknou slabě souvislé cykly.
2. Teoreticky mohou dva bloky textu mít možnosti vedoucí k sobě navzájem, čímž by vzniknul silně spojitý cyklus.

   O takovéto cykly navíc mívá spisovatel zájem. Představte si příběh, kdy se hrdina dostane do labyrintu, kde každé místo je reprezentováno jedním blokem textu. V případě že by měl uživatel poskytnutou hádanku, jež by mu pomohla nalézt cestu z labyrintu ven, stává se příběh uvěřitelný do takové míry, že by se podobný příklad mohl v reálném gamebooku vyskytovat.

   Právě díky cyklům se může hráč dostat mimo čas příběhu.
3. Chybí možnost popisu světa pomocí eventů, jako je u nerozvětveného příběhu. Veškeré změny vlastností se provádí pouze při pohybu v grafu, jenže my bychom chtěli popisovat změny ve světě v libovolném čase neovlivněný příběhem.

Příběh tedy nepopisujeme stromem, ale zkusíme to pomocí obecnějšího grafu.

V případě slučování dvou větví příběhu asi nevznikne moc veliký problém. Jedinou nevýhodou je, že nelze zobrazit přesný stav světa jen dle konkrétního uzlu grafu, ale je potřeba specifikovat celou cestu od kořene grafu.

U silně souvislých cyklů, pokud bychom chtěli zobrazit přesný stav světa, je potřeba specifikovat opět konkrétní cestu od kořene grafu. Hůř se graf zobrazuje a je složitější dávat si pozor na všechny možné průchody grafem. Zrovna v případě gamebooku můžeme předpokládat, že graf obvykle nebude jedna veliká silně souvislá komponenta, tudíž by software mohl umožňovat zobrazení grafu rozděleného podle silně souvislých komponent.

Náš graf žádným způsobem nesouvisí s časem, přesto bychom chtěli popisovat události ve světě, které se v konkrétním ději. Obvykle se jedná o časový úsek, ve kterém se příběh zrovna nijak nevětví, takže daný úsek příběhu proběhne vždy stejným způsobem. De facto se v tomto úseku příběh chová jako nerozvětvený příběh. Takovýto úsek pojmenujeme jako *lineání event*.

Lineární event v rámci své definice obsahuje délku trvání, seznam vlastností, které mění, blok textu, který se zobrazí čtenáři a seznam možností kam dál se může příběh ubírat.

Doposavad se jednalo v podstatě o uzel grafu rozšířený o změny vlastností konceptů světa. Dále ovšem bude obsahovat takzvané podeventy, kde každý podevent je v podstatě lineární event, který má definovaný *offset*, respektive dobu, po jak dlouhé době začíná od začátku lineárního eventu. Pochopitelně doba trvání + offset podeventu nesmí přesáhnout dobu trvání + počátek nadeventu.

Tento obecnější popis příběhu nám nejen že umožňuje popsat děje ve světě mezi větveními grafu, ale také umožňuje popsat nerozvětvený příběh samotný. Takový příběh by obsahoval jediný lineární event a všechny eventy z definice výše pro nerozvětvený příběh by byly podeventy. Díky tomu budeme schopni popisovat co se děje ve světě.

Podeventy zároveň odpovídají podeventům u nerozvětveného příběhu, kde jsme již zmínili jejich praktické výhody při vymýšlení vlastního světa.

Díky tomu, že lineární event je v podstatě rozšířený uzel grafu, můžeme graf sestrojit z uzlů, které budou pouze lineárními eventy.

2.2.2 Rozvětvený příběh a rozvětvený svět

Nejprve rozlišme rozvětvený příběh a rozvětvený svět z pohledu vývoje her. Rozvětvený příběh je to, co požaduje herní engine (objekty, jejich vlastnosti a události, které nastávají za určitých podmínek). Naproti tomu, rozvětvený svět je soubor všech informací o světě, ve kterém se hra odehrává. Podobně jako u nerozvětveného příběhu, spisovatel tvoří svět a následně na základě vybraných částí tohoto světa vytváří příběh.

Pro jednoduchost si budeme představovat vylepšený gamebook, který není tištěný na papíře.

Změnou oproti běžnému gamebooku je, že podmínkou pro přechod do dalšího uzlu grafu je splnění nějaké podmínky, kterou nemusí být pouze vybrání možnosti čtenářem.

Jak již bylo naznačeno v příkladu problému s cestováním v čase, tato podmínka může být opravdu libovolná a může záviset na informací o libovolném konceptu, nebo přímo času. Představme si že aktuálně čteme nějaký blok textu. K přechodu do dalšího bloku textu může dojít různými způsoby, například uplynutím určité doby.

Nyní se můžeme odprostit od vylepšeného gamebooku a zaměřit se na libovolnou hru. Jakožto hráč se nacházíme v nějakém stavu hry. Koncepty stavu světa jsou nastaveny na nějaké hodnoty, které se mění podle herního enginu.

Máme množinu událostí, kde každá má podmínku za jaké má proběhnout. Každá událost má tedy také nějakou metodu, která je volána, pokud je pod-

mínka splněna. Metoda může měnit vlastnosti konceptů světa. Na rozdíl od ní, je podmínkou metoda vracící pravdivostní hodnotu a zároveň nesmí měnit stav konceptů světa.

Jak nyní může ovlivňit příběh stav hry?

Jednoduše při každé změně stavu nějaké vlastnosti, která je součástí světa, se vyhodnotí podmínky všech událostí. Pokud nějaká podmínka vrátí pravdivostní hodnotu, zavolá se metoda události.

Definice rozvětveného světa

Rozvětvený svět se skládá z množiny konceptů a množiny událostí. Koncept představuje libovolný objekt ve světě, který se považuje za jeho součást. Obsahuje množinu proměnných, které určují vlastnosti daného konceptu. Událost je dvojice metod, a to podmínková metoda a metoda události. Podmínková metoda vrací pravdivostní hodnotu a nesmí měnit stav konceptů světa. Metoda události žádnou hodnotu nevrací, může měnit stav konceptů, ale nesmí obsahovat podmínkové skoky, jako jsou *if*, *for* nebo *while* statementy.

Dále můžeme také definovat stav světa jako seznam uskutečněných událostí.

Definice rozvětveného příběhu

Rozvětvený příběh je složen z podmnožiny událostí rozvětveného světa, dále konkrétních hodnot vlastností všech konceptů a na konec množiny koncových událostí.

Nevýhoda definice rozvětveného světa

S vyšší obecností přichází problémy. Jeden docela podstatný vychází z faktu, že veškeré vlastnosti konceptů jsou pouze proměnné bez konkrétní hodnoty v daném stavu světa. To znamená, že nelze zobrazit konkrétní stav světa spisovateli s přesnými hodnotami. Místo nich ovšem můžeme alespoň zobrazit hodnoty naposledy změněné příběhem. Konkrétní stav světa pak nebude ukazovat hodnoty, které by vlastnosti měly během hry. Důvod je ten, že vlastnosti mohou být měněny herním enginem.

Zajímavým postřehem je, že tvůrce by mohl mít možnost označit vlastnost konceptů za neměněnitelnou enginem. Následně, pokud by se o to engin pokusil, mohla by být vyhozena vyjímka.

Dalším možným nástrojem pro spisovatele by mohlo být propojení programu na vytváření příběhu s herním enginem tak, že by užiatel procházel hrou v enginu a veškerá data by byla posílána do programu pro tvorbu příběhu. Zde by pak měl tvůrce možnost sledovat aktuální stav světa s přesnými hodnotami.

2.3 Praktická vylepšení rozvětveného světa

2.3.1 Graf událostí

Podle aktuální definice by opravdu mohl spisovatel vytvářet svůj rozvětvený svět. Byla by to ovšem značně nepříjemná práce. Uživatel by musel pro každý

jeden event zadefinovat přesnou podmínku tak, aby byl proveden pouze ve chtěné chvíli. Při větším množství událostí by se v tom začal tvůrce ztrácet.

Jedno jednouché a praktické vylepšení můžeme pochopitelně přímo odkoukat již z popisu příběhu pro gamebooky. Zde se tvořil graf podle jednotlivých bloků textů. Stejně tak můžeme uspořádat události do grafu, kde každá událost definuje, po kterých událostech může jedině nastat. Popřípadě bychom mohli rozdělovat události do skupin, kde být součástí skupiny znamená, že vede hrana do každého dalšího prvku.

Také bychom mohli vytvářet složitější struktury, které by obsahovaly množinu událostí spolu s definovanými jedinými vstupními a výstupními událostmi, kde jinudy by opustit skupinu nešlo. To by se vyplatilo například pro oddělování podle časových období světa.

Nicméně ani jedno vylepšení nám nepomůže v případě přílišné provázanosti všech událostí. To je omezení, se kterým žádný program nemůže pomoci a je to na spisovateli vytvořit svět přehledně a ne příliš zamotaně.

Ve výsledku vytváří ze rozvětveného světa jakýsi stavový automat.

2.3.2 Povolení podmínkových skoků metody události

Dalším praktickým vylepšením je povolení podmínkových skoků v metodě události. Zákaz zde byl z dobrého důvodu. Pokud tato metoda bude mít podmínkový skok, například *if* a *else* statement, pak to znamená, že se defakto jedná o dva různé eventy s dvěma různými podmínkami. Pro výsledný program je jednoznačně jednodušší pracovat s metodami událostí bez podmínkových skoků, protože pokud si chce uživatel zobrazit nějaký konkrétní stav světa, musí vybrat, které události měly proběhnout a v jakém pořadí. Pokud by metoda události obsahovala podmínku, pak by musel uživatel určit výsledek podmínky. Výsledek nemůže být vyvozen z hodnot vlastností konceptů, protože, jak již bylo řečeno, ony hodnoty jsou pouze poslední změny provedené příběhem, respektive metodami událostí.

Na druhou stranu, při tvorbě příběhu by mohlo být pro uživatele pohodlnější nevytvářet mnoho událostí, kde každá provádí mnoho stejných nastavení vlastností konceptů, nýbrž vytvořit jednu událost, která by obsahovala několik podmínek.

2.3.3 Lineární událost

Následující vylepšení řeší problematiku kombinování nerozvětveného příběhu v rozvětveném světě. Například ve hrách, se obvykle, před tím, než samotný hráč začne rozhodovat o tom, jakým směrem se příběh vydá, může odehrát nějaký příběh, který je vždy odvyprávěn stejným průběhem. V takových situacích bychom chtěli, aby mohl spisovatel přistupovat k oné části příběhu jako k nerozvětvenému příběhu, tedy aby mohl popsat přesný stav světa v konkrétním čase a co především, aby mohl jednotlivé události v této části zobrazovat na časové ose.

Nicméně náš požadavek může zajít ještě dále. Spisovatel by samozřejmě mohl chtít začít popisovat příběh jako nerozvětvený v libovolném místě příběhu, ne pouze na začátku před prvním větvením.

Dále by tvůrce mohl chtít využívat vlastností událostí v nerozvětveném příběhu. Mohl by chtít vytvářet nadeventy, které by následně mohl rozdělovat na další podeventy. Proto, aby mohl pokračovat rozvětvený příběh, musí být všechny podeventy uskutečněny.

Toto všechno se pokusíme vyřešit pomocí takzvaného *lineárního eventu*, respektive *lineární události*. Bude se jednat o běžnou událost rozvětveného světa, která bude rozšířena o další vlastnosti. Zaprvé bude muset definovat čas. Musí říci jakou strukturu zde čas má, ale zároveň musí být jednotlivé hodnoty času lineárně uspořádané, abychom mohli zobrazovat časovou osu. Nejen že musí definovat strukturu času, ale zároveň musí definovat, která proměnná představuje aktuální čas, podle kterého mají být uskutečňovány podeventy.

Lineární event dále potřebuje definovat dobu trvání. Konkrétní čas, kdy událost začíná není třeba určovat, protože konkrétní čas započnutí se může lišit. Jiné eventy rozvětveného světa mohou nastat až po uplynutí doby trvání lineárního eventu.

Následně bude pochopitelně event obsahovat metodu události, která bude moci měnit vlastnosti konceptů světa. Ta je spuštěna na konci lineárního eventu, ale nemusí obsahovat nic.

Dále lineární event bude obsahovat podeventy, které mají definovaný offset, tedy dobu, po které mají nastat od začátku nadeventu. Také bude mít vlastní dobu trvání a svoji množinu podeventů.

Tímto způsobem lze popsat nerozvětvený příběh. Pokud bychom chtěli vytvořit příběh, který je pouze nerozvětvený, stačí nám vytvořit jeden veliký lineární event. V případě že by bylo ho potřeba prodloužit, stačilo by zvětšit dobu trvání.

Nyní uvažme situaci, kdy vytvoříme linearní event, načež nás napadne vytvořit uprostřed něj větvení. V takovém případě bychom chtěli mít nástroj pro rozdělení jednoho lineárního eventu na dva v konkrétním časovém offestu.

Podobně tak by bylo šikovné mít nástroj pro spojení dvou lineárních eventů nad stejnou proměnnou času.

Část II

Worlds Factory

V této částí bakalářské práce se budeme zabývat teoretickými i praktickými záležitostmi softwarového díla.

3. Hlavní struktura programu

Hlavní částí softwaru je program, který je hlavním nástrojem pro tvorbu příběhů, a to program, v němž bude spisovatel navrhovat svůj svět, podle něj následně vytvoří konkrétní příběh. V následující fázi, podle navrženého příběhu, bude vytvářet konkrétní dílo.

Součástí programu by měly být také nástroje pro tvorbu konkrétního díla, jako knihy, scénáře a gamebooky. Všechny tyto nástroje mohou být přímo součástí hlavního programu. Software by ovšem měl také podporovat tvorbu příběhu pro hry. Tvorba samotného příběhu může být uskutečněna v hlavním programu, ale výsledkem by měla být data, která následně převezme herní engine. Díky tomu jsou součástí celého softwaru také knihovny v různých jazycích. Tyto knihovny nejenže umožňují čtení exportovaných dat, ale také poskytují rozhraní pro herní engine, které klarifikuje způsob použití exportovaných dat.

Základní představu ohledně definice příběhu máme již za sebou, nicméně zatím se jednalo pouze o teoretickou úvahu. Mnoho konkrétnějších programátorských detailů je potřeba teprve vyřešit.

Podívejme se nyní na základní strukturu hlavního programu:

- **Management projektů**

  Základním požadavkem je jistě tvorba nových projektů a jejich management. To zahrnuje vytváření nových projektů, jejich ukládání, načítání a mazání.

- **Interpretace světa**

  Druhým požadavkem je interpretace světa podle definice v programu a jeho vytváření a editování.

- **Zobrazování světa**

  Následně třetí častí jsou nástroje pro zobrazování částí světa ze všech různých úhlů, ideálně za pomoci grafů a tabulek. To by zároveň mělo umožňovat filtrování a vyhledávání dat světa. Tento úsek programu by mohl být také schopen propojovat program s herním enginem a streamovat data ze hry přímo do programu.

- **Tvorba příběhu**

  Čtvrtou částí je tvorba příběhu. Jde o výběr podmnožiny událostí světa. Tato část programu by měla být také schopna zobrazovat příběh pomocí grafu událostí. Lineární události by měly být zobrazeny na časové ose.

- **Export dat**

  Další část zahrnuje export dat pro případ použití v herním enginu. Tato část zahrnuje validaci zdrojových kódů a podporu debugingu.

- **Tvorba díla**

  V případě knih, scénářů a gamebooků přímo zahrnuje nástroje pro tvorbu díla. Podporuje psaní textu s vkládáním komentářů a odkazů na konkrétní části světa. Jednotlivé části díla jsou rozděleny podle příběhu. V případě tvorby díla u hry se jedná o export dat, který byl vyčleněn do samostatné části.

- **Podpora ovládání**

  Poslední částí je podpora ovládání programu. Ideálně vše by mělo být ovládatelné pomocí klávesnice a všechny zkratky by měly být nastavitelné uživatelem pro zajištění pohodlného ovládání.

4. Tvorba světa

Program musí umožňovat vytvářet svět podle definice rozvětveného světa. To znamená, že uživateli bude umožněno vytvářet koncepty, události a definovat vztahy mezi nimi. Dále podle praktických vylepšení bude možné vytvářet lineární události a definovat graf událostí.

4.1 Knihovny

Pro popis konceptů světa využijeme objektově orientovaný přístup. Jednotlivé koncepty budou reprezentovány jako instance tříd. Třida pak bude obsahovat atributy, které budou odpovídat vlastnostem konceptu. To umožní popisovat celé množiny konceptů, které mají stejné vlastnosti, pomocí jediné třídy. Podobně jako u objektově orientovaného programování, využijeme také dědičnosti tříd. Klient by pak vytvářel tyto třídy a jeijich instance. Tím by vytvářel popis svého světa.

Jelikož eventy jsou tvořeny metodami, pro uživatele by bylo užitečné vytvářet jiné metody v kontextu konkrétní třídy.

Pro zjednodušení zakážeme vícenásobnou dědičnost, abychom se vyvarovali diamantovému problému. Tedy problému s nejednosznačností volání metody s implementací ve více předcích.

Myšlenkou je, že by se třídy shlukovaly do skupin, tak aby žádní příbuzní nebyli oddělení od sebe. Tato skupina by pak byla nazvána jako *knihovna*. Výhodou by bylo možné sdílení pouze knihoven mezi různými projekty. To by bylo užitečné, aby nemusel uživatel při vytváření nového světa popisovat každý typ objektu, který by chtěl použít.

Rozšířením nějaké knihovny by vznikla jiná. Pak při importu dvou knihoven by mohly nastat konflikty v názvech tříd. Konfliktní třídy by mohly být identifikovány pomocí názvu knihovny, které jsou součástí.

Vyhledávání správného typu objektu uřivatelem by mohlo být uskutečněno pomocí hledání od kořenové třídy, tedy třídy bez předka. Následně by šlo v hledání pokračovat hledáním v jeho potomcích, čímž by uživatel přesněji a přesněji specifikoval hledaný typ objektu, dokud by nebyl nalezen. Popřípadě by mohl automaticky vytvořit nový, pokud by požadovaný dopsud neexistoval.

Možným vylepšením by pak bylo vyhledávání návrhů tříd, podle názvu třídy o jakou by se asi mělo jednat. Takové vyhledávání by se dalo uskutečnít pomocí vyhledání synonym.

4.2 Vlastnosti události

Událost by měla mít možnost vlastnit přidané vlastnosti, podobně jako koncept. Událostí by pak byla defakto třída, která by mohla rozšířovat jiné třídy o základní vlastnosti eventu. Toto by mohlo být užitečné například ve chvíli, kdybychom chtěli zobrazovat událost na mapě. Události bychom přidali vlastnost pozice a pracovali bychom s ní jako s obyčejným objektem, respektive konceptem.

Dalším využitím může být například obohacení události o počítadlo, kolikrát byla uskutečněna.

4.3 Zobrazování světa

Různé nahlížení světa by měl mít možnost si uživatel uložit, a pokud by později chtěl zobrazení opakovat, mohl by tak učinit jedním kliknutím, místo opětovného nastavování všech parametrů.

4.3.1 Zobrazování stavu světa

Jak již bylo dříve uvedeno, přesný stav světa v rozvětveném světě nemůže být zobrazen bez herního enginu. Můžeme zobrazovat pouze poslední změny, které byly provedeny příběhem. Nicméně i toto může být pro tvůrce užitečné. Uživatel vybere konkrétní události, popřípadě konkrétní průchody metodami událostí a program zobrazí které koncepty a které jejich vlastnosti byly změněny.

Ve skutečnosti je zde celá situace složitější a jedná se o těžší úkol, než na první pohled vypadá. Problémem je, že pro vyhodnocení změn vlastností konceptů musíme provést metodu události. To znamená, že potřebujeme použít nějaký interpret, který bude schopen vyhodnotit kód a který si bude pamatovat všechny hodnoty.

Vyvstává zde otázka, co má nastat ve chvíli, kdy v rámci metody události dojde k nastavení vlastnosti nekonstantní hodnotou, obvykle jinou vlastností, respektive proměnnou. Problémem je, že neznáme přesnou hodnotu této proměnné, ale pouze naposledy změněnou hodnotu příběhem, pokud k nějakému nastavení vůbec došlo.

Máme zde dvě možnosti. Buď použít onu naposledy použitou hodnotu, nebo nenastavovat vlastnost vůbec. Místo toho si lze zapamatovat, že se měla vlastnost změnit a zapamatovat si třeba odkaz do kódu, kde měla být změněna.

První možnost je vhodná pro nerozvětvený příběh, protože předpokládáme, že v takovém případě žádný herní engine, který by vstupoval do dat světa není. Zároveň může být tato možnost nepříjemná v případě rozvětveného příběhu, protože může dojít vyhodnocení nějaké vlastnosti a uživatel by pak počítal s tím, že takovou hodnotu vlastnost v daném stavu světa má kdykoli.

Při druhé možnosti se uživateli zeslabuje možnost prohlédnout jak přibližně vypadá stav světa. Jednou z možností je, že by si mohl hráč vybrat, jakou možnost chce použít. Ale jako ještě lepší se jeví, kdyby program rozlišil, které hodnoty jsou konstantní a které ne. Program by mohl zobrazovat seznam změněných vlastností a pozici změny ve všech případech.

S tím souvisí, že potřebujeme nastavit počáteční hodnoty všech vlastností konceptů, a nebo počítat s tím, že všechny potřebné vlastnosti se nastaví na primitivní hodnotu, než se začnou používat v dalších výpočtech.

Z toho důvodu by dávalo smysl nastavovat defaultní hodnoty vlastnostem konceptů. To pak defakto znamená, že před první zmímnkou o daném objektu ve světě by byla jeho vlastnost nastavena na tuto hodnotu.

Další možnou funkcí by mohlo být zobrazení průběžných změn konkrétní vlastnosti konkrétního konceptu (při dané cestě událostí). Uživatel by tak mohl například sledovat průběžný vývoj míry hladu nějaké postavy a viděl by, kolikrát postava hladověla při daném průchodu příběhem (a bez informací podané herním enginem).

4.3.2 Zobrazovací mapa

Motivací je snaha zobrazovat objekty na mapě při konkrétním zobrazení stavu světa. Musíme označit konkrétní vlastnost nějaké třídy za pozici. Teoreticky můžeme označit i více vlastností tříd, ale musí být alespoň stejného typu. Ještě praktičtější by bylo, kdyby uživatel zadefinoval funkci, která přebírá za parameter nějaký typ objektu a vrací konkrétní pozici. Těchto funkcí by pak mohl být libovolný počet, ale musely by všechny vracet stejný typ pozice.

Následně by mohl být nastaven požadavek, že struktura pozice musí být vektor čísel.

To, že bychom určovali pozici pomocí funkce a nikoli pomocí výčtem vlastností objektů, nám umožňuje vytvořit strukturu pozice libovolné složitosti a z libovolných vlastností různých objektů.

Program by následně získal zobrazení stavu světa a prošel by veškeré objekty, které mohou být parametrem nějaké z funkcí. Pokud by funkce vrátila validní pozici, zobrazí objekt na mapě. To mimo jiné znamená, že by jeden stejný objekt mohl být na více místech zároveň, protože funkcí pro získání pozice může být více.

Samotné zobrazení může být samozřejmě provedeno pouze do dimenze 3. Pro vyšší dimenze by bylo rozdělení s jezdci, kde by uživatel nastavoval konkrétní hodnoty pro zbylé souřadnice zvlášť.

4.3.3 Zobrazování grafů

Další možností zobrazení nějaké části světa je pomocí grafů. Opět předpokládáme nějaký stav světa, ve kterém chceme tentokrát zobrazit vztahy mezi objekty. Ty mohou být orientované, nebo neorientované. Příkladem může být například rodokmen postav. Vrcholy grafu jsou zde očividně objekty. Jejich typ je jeden z množiny typů, které uživatel vybral. Vlastnost patřit do rodiny je pak dána vlastností objektu být potomkem někoho. To je reprezentováno referencí na jiný objekt.

Hranou mezi dvěma vrcholy je pak tedy reference v objektu na druhý objekt.

4.3.4 Zobrazování grafu událostí

Zobrazení samotného grafu je zde relativně přímočaré. Vrcholy jsou události, které jsou spojeny hranami, podle toho, jaké události mohou nastat po kterých. Ideální by bylo seskupovat eventy podle skupin jak bylo již zmíňeno v kapitole *Popis příběhu*, sekce *Praktická vylepšení rozvětveného světa*, subsekce *Graf událostí*. Pokud by se jednalo o množinu, kde každá událost je spojena s každou, nemusely by být hrany zobrazeny. Jen je potřeba vizuálně rozlišit mezi typy množin.

Předestřeme jedno teoreticky praktické vylepšení pro zobrazování grafu událostí pro rozvětvené příběhy. Ve většině případů, kdy nedochází k cestování časem,

by bylo užitečné zavedení přibližné časové osy. Jednotlivým událostem bychom mohli přiřadit konkrétní čas, ale mohli bychom také přiřadit časový interval, ve kterém může událost nastat. Otázkou je, jestli by měl být časový interval součástí podmínky pro vykonání události. Nepříjemností zde je, že by byl uživatel nucen definovat nějaký obecný čas pro celý svět. Zároveň si všimněme, že mírně podobného výsledku se dá docílit za pomoci množin událostí. Rozdíl je mimo jiné, že časová osa by ukazovala přesné hodnoty, zatímco množiny by pouze zobrazovaly pořadí skupin událostí.

Ve výsledku se jedná o praktickou věc, měla by být ale dobrovolná.

4.4 Ukládání dat

Pro snadné sdílení světa mezi uživateli by měly být soubory světa oddělené od souborů zbytku programu. Zároveň by software měl podporovat práci více uživatelů na stejném projektu. Jedním z nejjednodušších způsobů jak toho dosáhnout je použití předpokládat, že uživatelé budou používat nástroje podobné nástroji *GIT* [5]. Podle toho můžeme ukládání dat našeho programu uzpůsobit. Snahou bude rozdělovat jednotlivé části programu, které jsou editovatelné uživatelem do jednotlivých co nejvíce logicky separovaných souborů. Tyto soubory musí být také pojmenovány dobře pochopitelným způsobem. Navíc bychom se měli vyvarovat ukládání nějakých prvků programu do společných seznamů, protože při změně dvou prvků různými uživateli může dojít ke konfliktu. Pokud bychom potřebovali vytvořit nějakou logickou množinu, ideální je interpretace množiny jako množiny souborů ve společné složce v adresářové struktuře.

Složitější řešení by mohlo vypadat tak, že by byla centrální databáze projektu, kterou by ostatní uživatelé mohli upravovat. Změny provedené jedním uživatelm by pak mohli být propagovány ihned ostatním uživatelům. Centrální databáze by řadila příchozí požadavky jednoltivých uživatelů. Pokud by nemohl být požadavek splněn z důvodu konfliktu, byl by požadavek zamítnut a podavatel by byl s komfliktem obeznámen.

4.5 Interpret

Součástí programu musí být komponenta, respektive nějaký interpret, která je schopna spouštět metody událostí pro zobrazování stavu světa. Zároveň musí být program schopen exportovat data pro herní engine. Obě činnosti jsou pro nás důležité. Interpret může zpracovávát pouze kód napsaný v konkrétním jazyce. Na druhou stranu uživatel by byl rád, kdyby jednotlivé metody mohly být napsány v jeho oblíbeném jazyce, repektive podle potřeby herního enginu. Zde máme možnost vyřešit tuto situaci tím, že metody by byly napsány v nějakém speciálním jazyce, který program dokáže interpretovat a při exportu by byl tento kód přeložen do vybraného jazyka uživatelem. Toto řešení je velmi složité pro implementaci, ale při použití dat v herním enginu by kód běžel bez ztráty výkonu.

Druhou možností je, že by export neprováděl překlad kódu do vybraného jazyka, ale zůstal by ve formě jazyka interpretu. Za běhu by se pak kód příběhu interpretoval. Toto řešení je mnohem jednodušší na implementaci, ale může způsobit výraznou ztrátu výkonu.

První možnost by mohla být jednodušeji realizovatelná za použití velkých jazykových modelů. Schopnost takovýchto modelů překládat kód z jednoho jazyka do druhého je v dnešní době na relativně dobré úrovni. Zatím nevýhodou je špatná testovatelnost překladu, spojená s nevyzpytatelností výsledků.

4.6 Identifikátory a Tagy

Skoro vše, co uživatel vytvoří, musí mít v programu svůj identifikátor. Díky tomu je vhodné identifikátory nějak standardizovat pro celý program pro jednotlivé typy konceptů.

Samotný identifikátor bude dělen na několik částí. První částí je takzvaný prefix. Ten je rozdělen na dvě části. První část začíná zavináčem a končí dvojtečkou. Určuje typ konceptu. Druhou částí může být doupřesnění. Například prefix identifikátoru property nějaké třídy se skládá z *@property:* a názvu třídy *názevTřídy_*. Podtržítko určuje konec názvu třídy, tudíž název třídy nesmí končit podtržítkem. Další částí je samotný název konceptu. Zde jsou pro příklad tři různé identifikátory různých typů:

@class:ExampleClass
@property:ExampleClass_x
@method:ExampleClass_myMethodsName

5. Uživatelská dokumentace

5.1 Instalace

Repozitář celého projektu je dostupný v příloze této práce.

Obsahuje mimo veškerý zdrojový kód také sestavený program pro linux a windows. Verze pro linux byla testována na distribuci *Ubuntu 20.04.3*. Verze pro windows z časových důvodů testována zatím nebyla.

Zdrojový kód bez sestaveného programu je součástí příloh této práce.

Uživatel má dvě možnosti spuštění aplikace. Zde jsou jednotlivé kroky:

Přímé spuštění

Nejprve v případě linuxu:

```
cd bc_thesis_tichavsky/WorldsFactory/publish/net7.0/linux-x64/
chmod +x WorldsFactory
./WorldsFactory
```

Pro windows:

```
cd bc_thesis_tichavsky/WorldsFactory/publish/net7.0/win-x64/
.\WorldsFactory.exe
```

Sestavení aplikace

Pro sestavení aplikace je potřeba mít nainstalovaný .NET 7.0 SDK.

```
cd bc_thesis_tichavsky/WorldsFactory/
dotnet restore
dotnet build
dotnet run
```

5.2 Management projektu

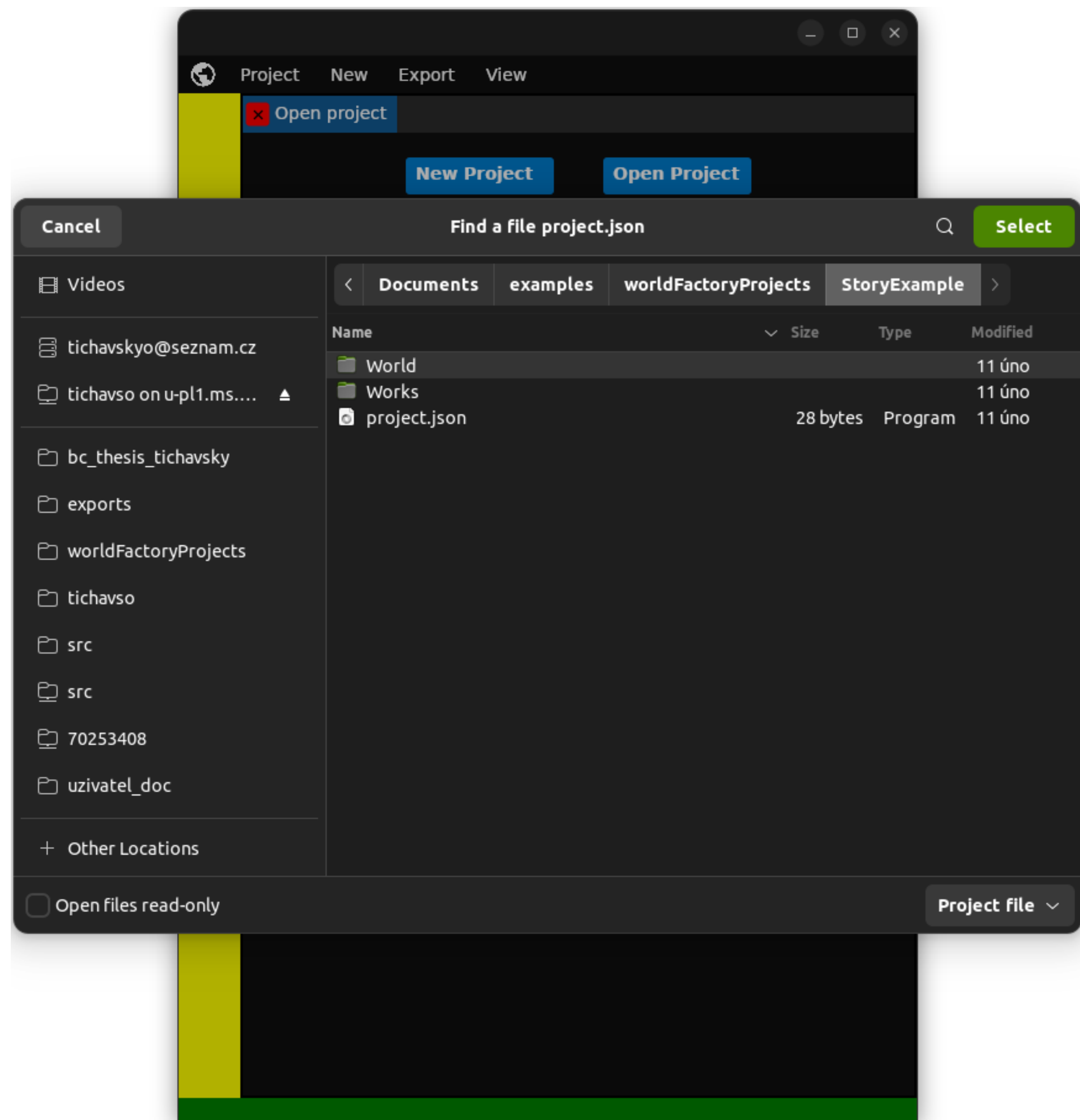
5.2.1 Otevření projektu

Po spuštění aplikace se uživateli zobrazí okno se seznamem všech známých projektů a seznamem nedávno otevřených projektů. Na jednotlivé projekty lze kliknout a tím započíst akci otevření daného projektu, viz obrázek 5.1.

Obrázek 5.1: Okno pro výběr projektu

Chtěl-li by uživatel otevřít projekt, který není v seznamu, může tak učinit pomocí tlačítka *Open Project*, které se nachází v pravém horním rohu okna s nápisem

Open Project. Klinutím na toto tlačítko se zobrazí dialogové okno, viz obrázek 5.2, které umožní uživateli vybrat projekt ze souborového systému. Uživatel musí vybrat soubor s názvem *project.json*. Tento soubor se nachází v kořenovém adresáři projektu.

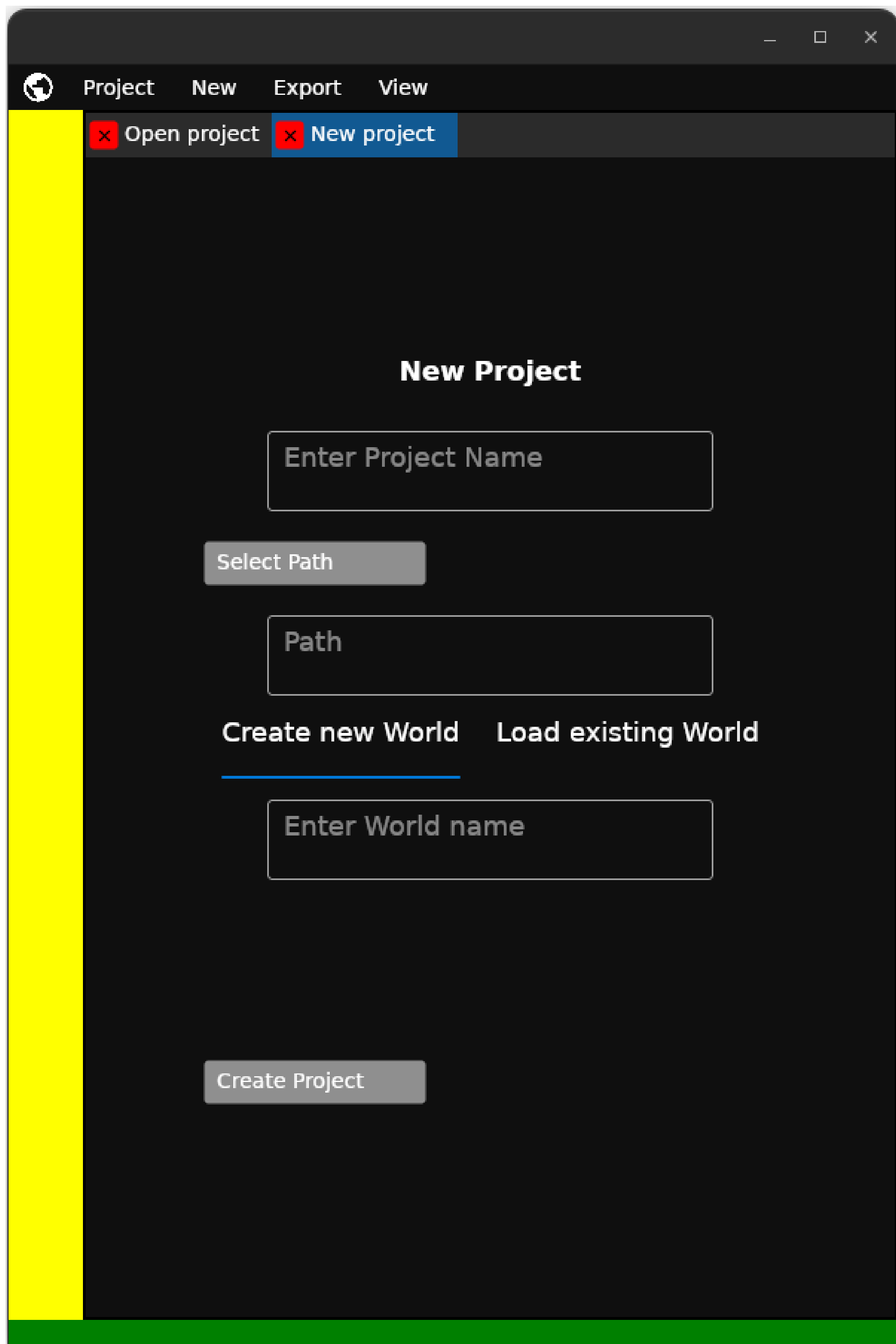


Obrázek 5.2: Dialogové okno pro výběr souboru projektu

Po vybrání souboru dojde k otevření daného projektu. Pokud není struktura projektu validní, dojde k zobrazení notifikace.

5.2.2 Vytvoření nového projektu

Po spuštění aplikace se uživateli zobrazí okno se seznamem všech známých projektů a seznamem nedávno otevřených projektů. V pravém horním rohu okna se nachází tlačítko pro vytvoření nového projektu s nápisem *New Project*. Kliknutím na toto tlačítko se zobrazí nová karta s formulářem pro vytvoření nového projektu, viz obrázek 5.3.



Obrázek 5.3: Okno pro vytvoření nového projektu

Uživatel musí vyplnit název projektu, cestu, kde se má projekt uložit a také musí specifikovat jaký svět použije. Buď vytvoří nový prázdný svět, pak musí

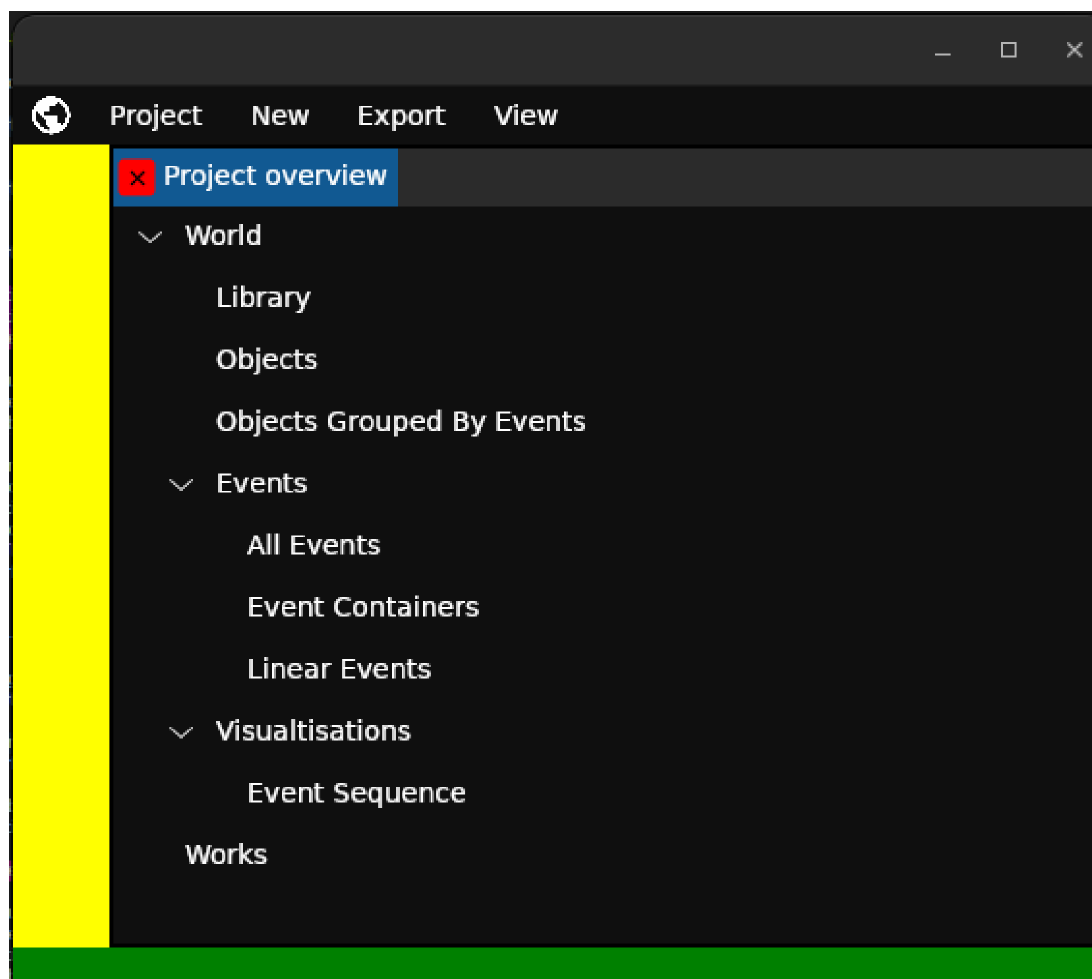
vyplnit jeho název.

Pokud by chtěl uživatel použít existující svět, klikne na tlačítko *Load existing World*. Tím se zobrazí pole pro vyplnění cesty k souboru se světem. Uživatel může vyhledat soubor pomocí tlačítka *Select Path*, které zobrazí dialogové okno pro výběr souboru ze souborového systému. Zde musí uživatel vybrat soubor s názvem *world.json*. Tento soubor se nachází v kořenovém adresáři světa uvnitř nějakého projektu.

Pokud uživatel vyplní všechny potřebné informace, může stisknout tlačítko *Create Project*. Tím dojde k vytvoření nového projektu a jeho automatického otevření.

5.3 Vytváření světa

Po otevření projektu se uživateli zobrazí karta s názvem *Project Overview*. Tato karta obsahuje základní informace o projektu v podobě stromové struktury všech částí. Viz obrázek 5.4.



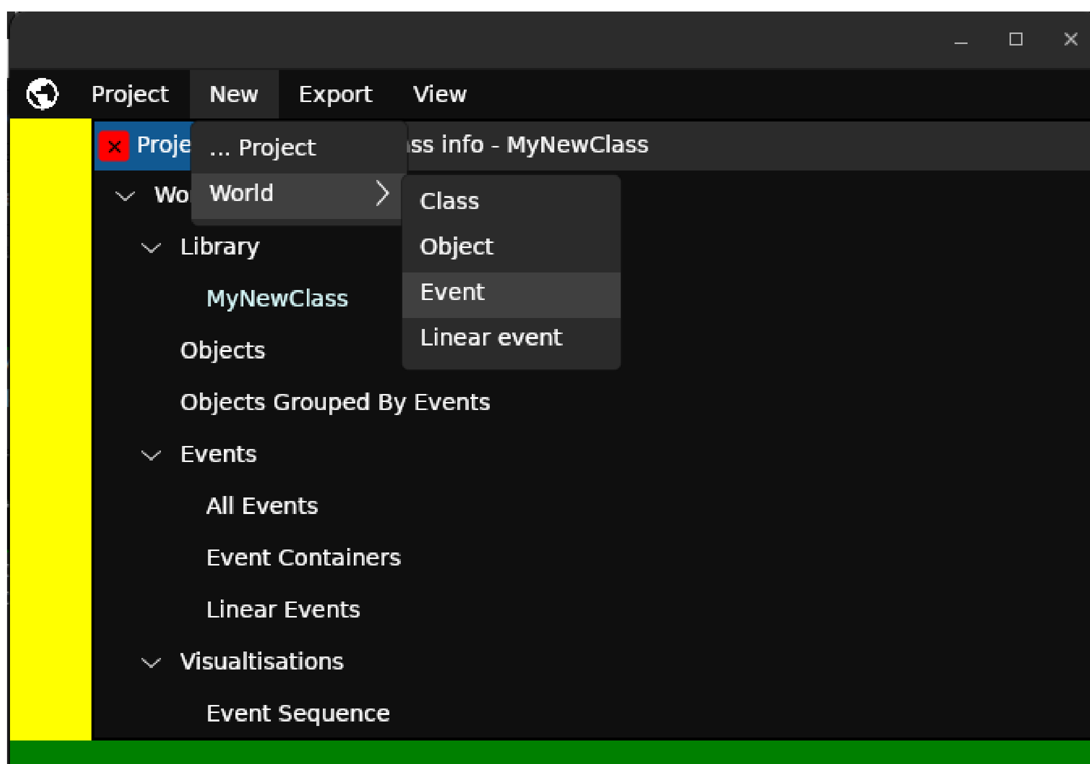
Obrázek 5.4: Okno po otevření nového projektu

Hlavní rozdělení projektu je na *World* a *Works*. World se dále dělí na *Library*, *Objects* a *Events*. Works v této verzi zatím nic neobsahuje.

Stromová struktura zároveň slouží jako navigační panel. Kliknutím na jednotlivé části, které již byly vytvořeny uživatelem, se zobrazí nová karta s podrobnostmi.

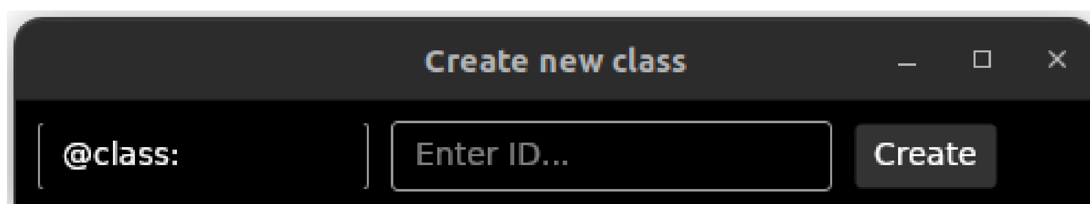
5.3.1 Vytváření třídy, objektu, událostí

Po otevření projektu, uživatel může vytvářet nové třídy, objekty a události pomocí horní lišty okna. Zde se nachází navigační panel, který obsahuje také *New*.T o zobrazí různé možnosti, zde uživatel může vybrat *World* a následně jednu z možností *Class*, *Object* nebo *Event*. Každé z tlačítek zobrazí nové okénko s formulářem pro vytvoření nového konceptu. Obvykle stačí zadat název, popřípadě typ. Vše je ukázáno na obrázku 5.5.



Obrázek 5.5: Navigační panel pro vytvoření nového konceptu

Po kliknutí na některé z tlačítek se zobrazí nové okno s formulářem pro vytvoření nového konceptu. Příklad pro vytvoření nové třídy je na obrázku 5.6.



Obrázek 5.6: Formulář pro vytvoření nové třídy

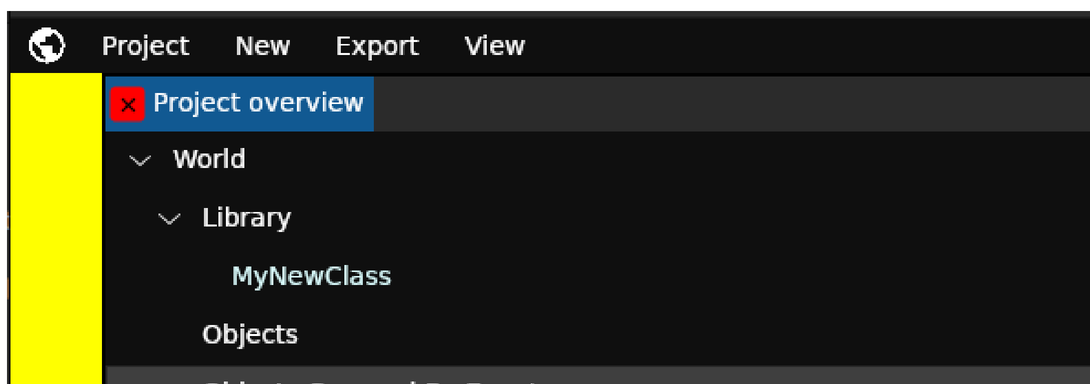
Uživatel musí zadat název třídy. Typem třídy je reference na jinou třídu. Je tedy potřeba zadat identifikátor dané třídy. Ten se skládá z prefixu a názvu třídy. Prefix je zde *@class:*, tedy například *@class:ExampleClass*.

Pro vytvoření nové třídy je nyní nutné potvrdit formulář pomocí tlačítka *Create*.

Ostatní koncepty se vytváří podobným způsobem.

5.3.2 Vytváření property, nebo metody

Předpokládáme, že uživatel již vytvořil nějakou třídu. *Project overview* karta vypadá, jako na obrázku 5.7.



Obrázek 5.7: Project overview s vytvořenou třídou

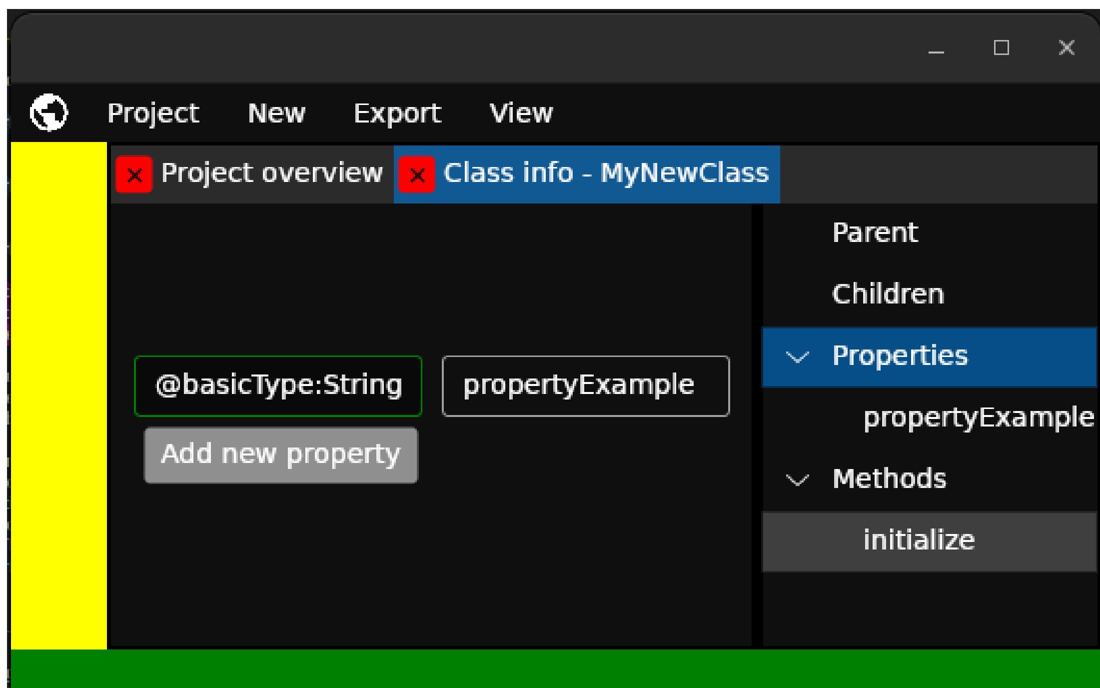
Na kartě *Project Overview* musí uživatel nalézt a kliknout na danou třídu. Tím se zobrazí nová karta s názvem *Class Info - Název*.

Zde uživatel vidí seznam všech properties a metod dané třídy. Také vidí, jestli třída dědí od jiné.

Vytváření property

Pro vytvoření nové property, uživatel klikne na tlačítko *Properties*. Tím se zobrazí seznam všech properties spolu s tlačítkem pro vytvoření nové. Kliknutím na tlačítko *Add new property* se do seznamu přidá nová nevyplněná property. Uživatel musí vyplnit název a typ. Zároveň zde uživatel může změnit existující property.

Situace znázorněna na obrázku 5.8.



Obrázek 5.8: Vytvoření nové property

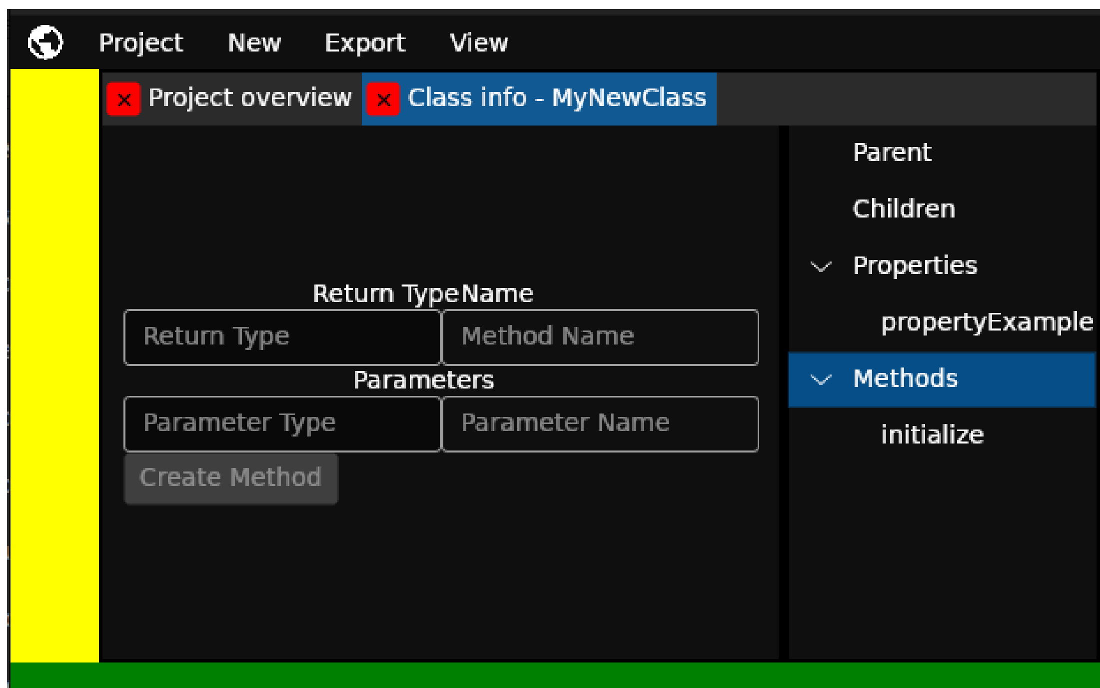
V případě úspěchu by se měla property objevit v seznamu properties.

Pro editaci property, uživatel klikne na danou property v seznamu. Tím se zobrazí formulář pro editaci.

Pro vytvoření property typu pole, uživatel musí ke jménu property napsat hranaté závorky s velikostí pole. Pro vícedimenzionální pole se hranaté závorky opakují.

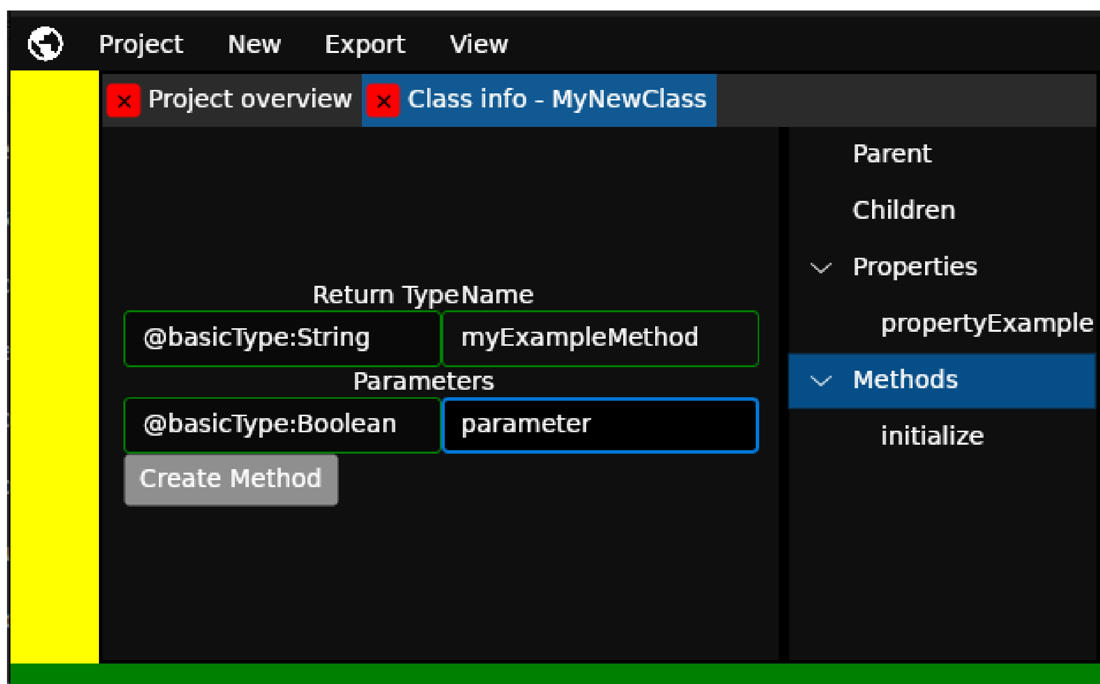
Vytváření metody

Pro vytvoření nové metody, uživatel klikne na tlačítko *Methods*. Tím se zobrazí formulář pro vytvoření nové metody, stejně, jako je na obrázku 5.9.



Obrázek 5.9: Formulář pro vytvoření nové metody

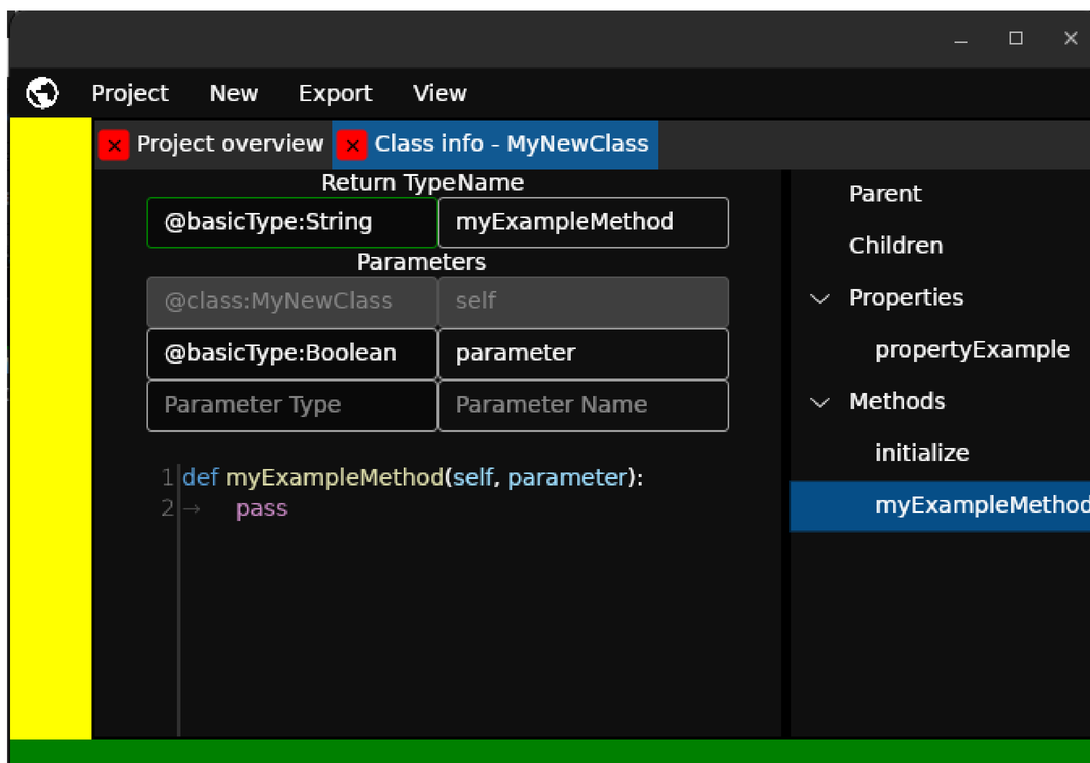
Uživatel musí vyplnit název, typ vrácené hodnoty a parametry. Příklad je na obrázku 5.10.



Obrázek 5.10: Příklad vyplněného formuláře pro vytvoření nové metody

Po Kliknutí na tlačítko *Create Method* se ověří validita dat a v případě úspěchu by se měla metoda objevit v seznamu metod.

Pro editaci metody, uživatel klikne na danou metodu v seznamu. Tím se zobrazí formulář pro editaci. Uživatel může změnit název, typ vrácené hodnoty, parametry a tělo metody. Příklad můžete vidět na obrázku 5.11.



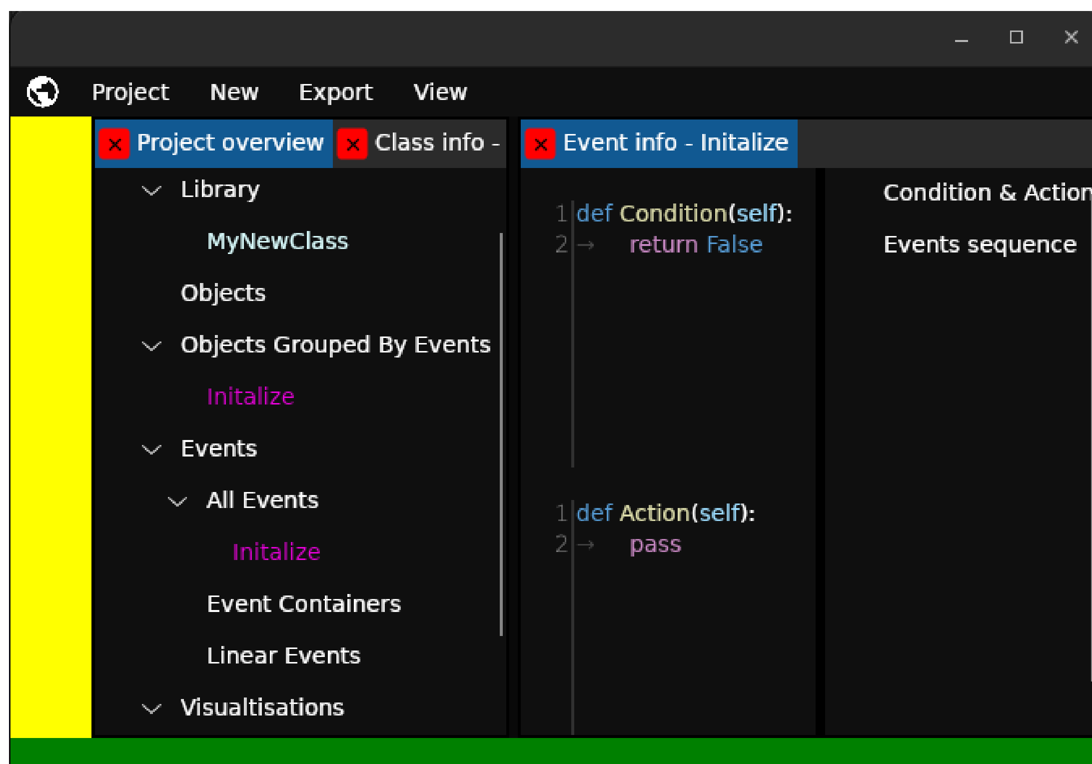
Obrázek 5.11: Podrobosti o metodě

Metoda je napsána v jazyce Python. K objektům a eventům se dá přistupovat pomocí globálních proměnných *objects* a *events*, nebo lze použít id objektu nebo eventu.

```
def Example(self, x, y, description):
    self.x = x
    self.y = y
    self.description = description
    objects.Map.places[x][y] = self
    @object:Map.places[x][y] = self
    pass
```

5.4 Úprava událostí

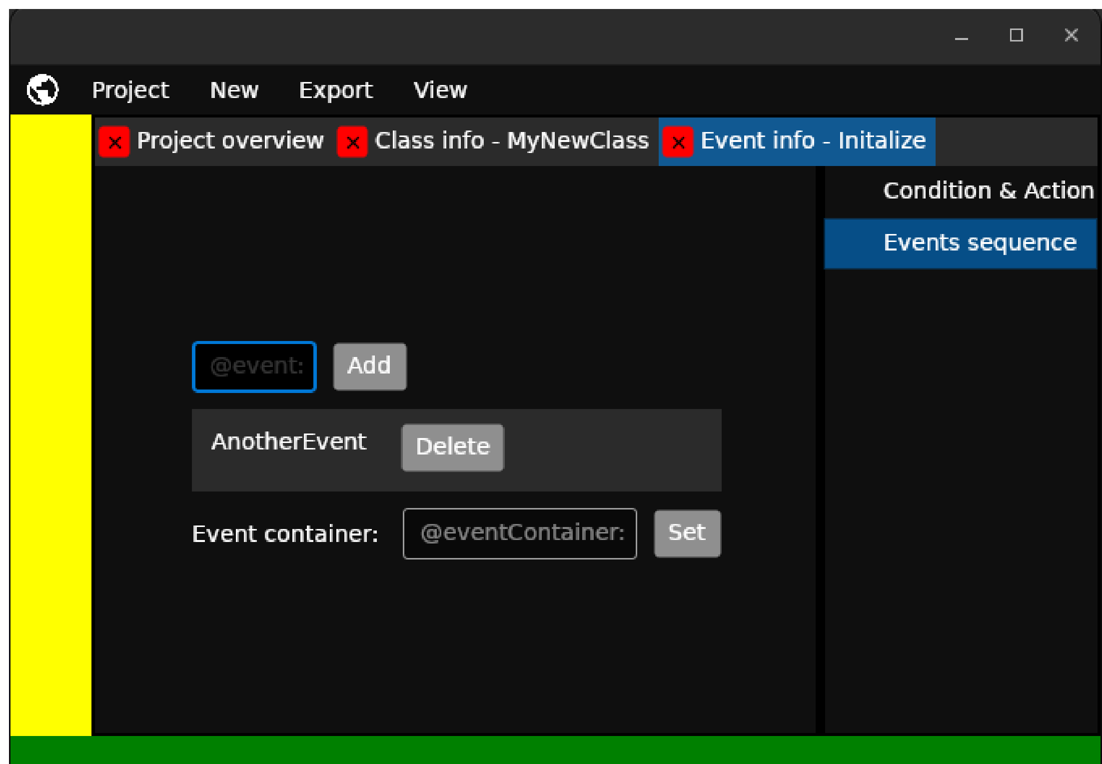
V případě, že uživatel již vytvořil nějakou událost, může ji upravit. Na kartě *Project Overview* musí nalézt danou událost a kliknout na ni. Tím se zobrazí nová karta s názvem *Event Info - Název*, na které uživatel vidí obě funkce události. Obě zde může upravovat. Vše se automaticky ukládá, jakmile uživatel ztratí zaměření (focus) na danou kartu. Příklad je na obrázku 5.12.



Obrázek 5.12: Podrobnosti o události

Uživatel může nastavovat vztahy mezi událostmi. Na kartě *Event Info - Název* lze v postranním navigačním panelu na pravé straně karty překliknout tlačítkem *Events sequence*. Uživateli se zobrazí na stejné kartě panel, kde lze přidávat jiné eventy do sekvence. Tento seznam říká, pouze po kterých událostí může tato událost nastat.

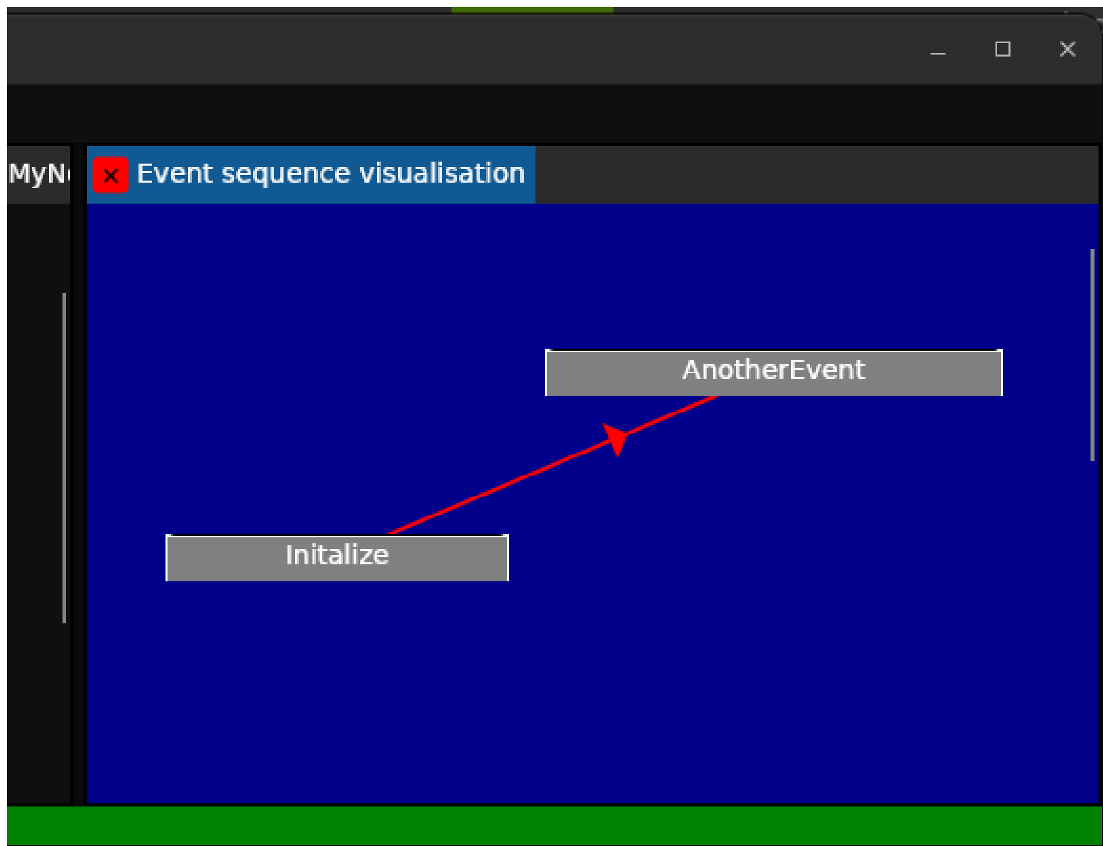
Příklad je na obrázku 5.13.



Obrázek 5.13: Nastavení sekvence událostí

5.4.1 Zobrazování vztahů mezi eventy

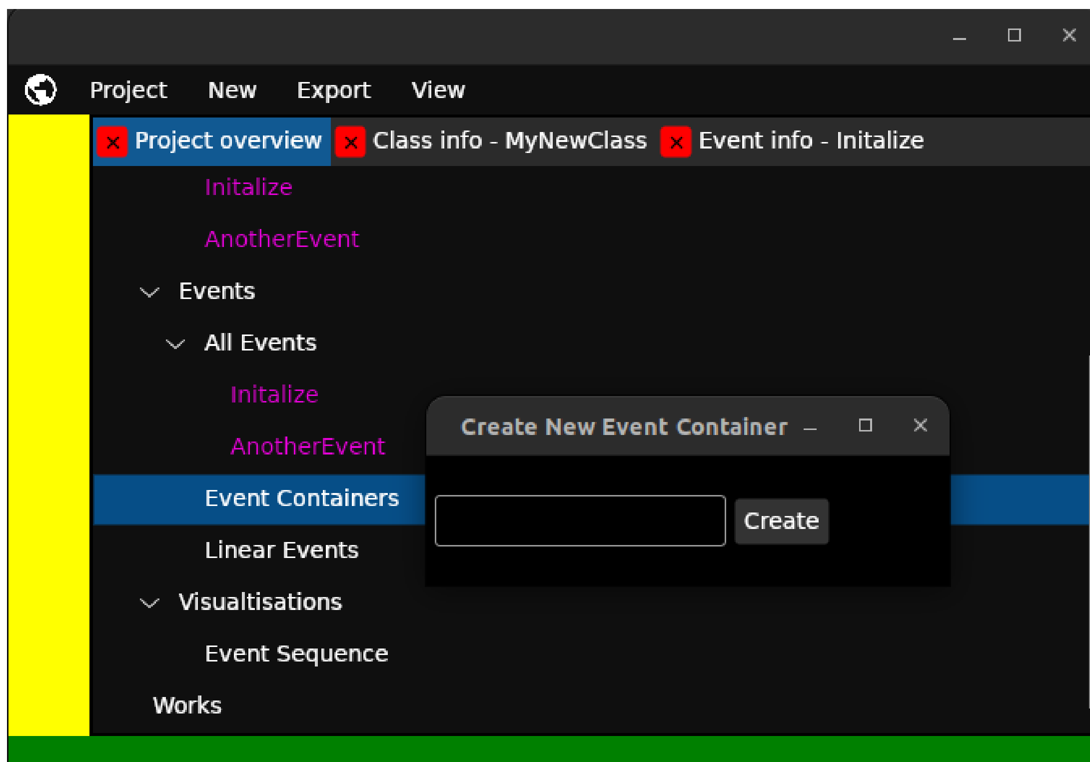
Vztahy eventů je možné zkoumat pomocí zobrazení ve formě grafu. Na kartě *Project Overview* lze v menu pod prvkem *Visualisation* vybrat *Event Sequence*. Tím se zobrazí nová karta, která zobrazuje všechny eventy a kreslí šipky mezi nimi, viz obrázek 5.14.



Obrázek 5.14: Zobrazení vztahů mezi eventy

5.4.2 Event container

Event container je nástroj, který umožňuje uživateli eventy třídit do skupin, a ty dále do dalších skupin. *Event container* lze vytvořit kliknutím pravým tlačítkem myši na prvek v seznamu všech event containerů, který se nachází na kartě *Project Overview*. Příklad vytvořeného event containeru je na obrázku 5.15.



Obrázek 5.15: Vytvoření event containeru

Přidání eventu do event containeru

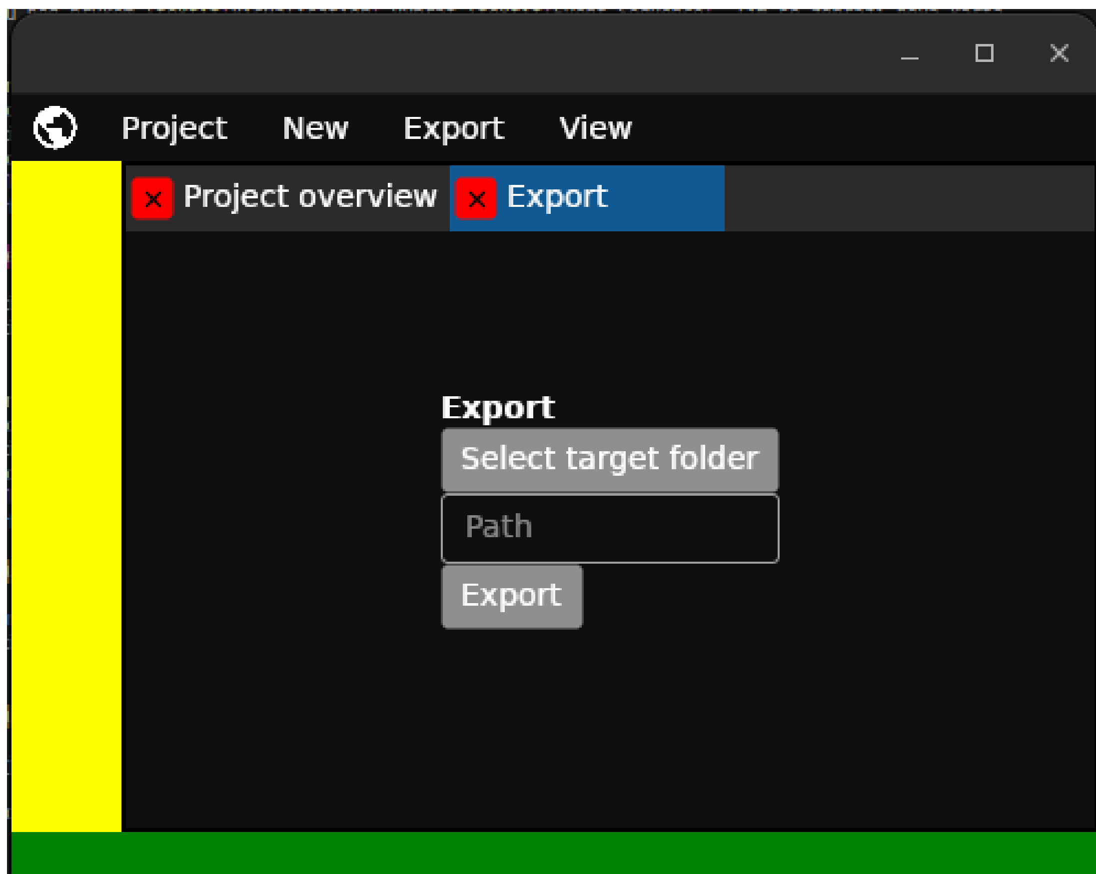
Na kartě *Event Info - Název* pro libovolný event lze v postranním navigačním panelu na pravé straně karty překliknout tlačítkem *Event container*. Zde lze zadat identifikátor konkrétního event containeru. Každý event může být pouze v jednom event containeru.

Zobrazení kontejnerů v Event Sequence vizualizaci

Kontejnery eventů slouží také pro lepší přehlednost při prohlížení eventů a vztahů mezi nimi. Při zobrazení grafu vztahů mezi eventy dojde k vytvoření speciálních uzlů, které reprezentují event containery. Při pohybu myší nad zobrazením grafu si může uživatel přibližovat a oddalovat graf. Pokud si uživatel přiblíží uzel reprezentující event container, zobrazí se mu eventy a další event containery, které obsahuje.

5.5 Export dat pro herní engine

V horním navigačním panelu uživatel klikne na tlačítko *Export* a následně *All data*. Tím se zobrazí nová karta s názvem *Export*. Viz obrázek 5.16.



Obrázek 5.16: Export dat

Zde uživatel musí vybrat cestu v souborovém systému, kam se exportovaná data mají uložit. Po kliknutí na tlačítko *Export* dojde k validaci dat světa. To proběhne za pomoci Python interpretu, který se pokusí exportovaná data interpretovat. V případě úspěchu se data uloží do souboru. V jiném případě se zobrazí notifikace s chybou a v kartě *Export* se zobrazí chybová hláška s popisem chyby.

5.6 Ukázky jdenoduchých projektů

Pro ukázku, jak aktuální stav programu může být využit v praxi, jsou součástí repozitáře dva jednoduché projekty. Nacházejí se v adresáři

```
./bc_thesis_tichavsky/WorldsFactoryProjectExamples
```

Postup pro otevření takového projektu již byl zmíněn výše v sekci 5.2.

Oba projekty jsou ukázky rozvětvených příběhů.

Protože se jedná zatím o minimální produkt celkového projektu, jsou tyto projekty velmi jednoduché a ukazují pouze velmi základní koncepty.

6. Architektura programu

V této kapitole budeme pojednávat o konkrétní implementaci programu. Rozebereme uživatelské požadavky, architekturu softwaru a její jednotlivé části.

Hlavní program, ve kterém bude uživatel vytvářet svět a příběh má pracovní název *WorldsFactory*. Při výběru jazyka pro implementaci byla snaha vybrat jazyk, který by využíval garbage collector, protože spolehlivost toho, že nedojde ke zpomalení programu kvůli nutnosti zavolání garbage collectoru zde není kritická. Pozitivum garbage collectoru pak převyšuje negativa.

Jelikož nám docela záleží na efektivitě programu, jasná volba je použití kompilovaného jazyka.

Výběr se omezil na jazyky Java a C#. Je to z důvodu, že oba jazyky jsou velmi používané a mají velkou podporu. Pro oba jazyky existuje veliké množství knihoven a vývojářských nástrojů.

Oba dva jazyky mají co nabídnout a jsou si i docela podobné, přesto vidím u jazyka C# výhodu u některých syntaktických vychytávkách, které u jazyka Java nenajdeme. Konečným vítězem byl tedy jazyk C#.

S ohledem zpět bylo možné uvažovat i o jiných jazycích, se kterými jsem se v době výběru jazyka nesetkal. Například jazyk Groovy, jakožto nadstavba nad jazykem Java, se z dnešního pohledu tváří jako velmi mocný jazyk. Koneckonců, kdybych se rozhodl pro jeho použití u Java knihovny, která je součástí této práce viz dále, umožnil by mi vytvoření příjemnějšího uživatelského rozhraní.

Co se týče jazyka interpretu, pomocí kterého budou psány jednotlivé metody uživatelem, musí jím být jazyk, který je jednoduchý na použití a porozumění pro uživatele. Jazyk musí počítat s tím, že uživatelé mají pouze velmi základní znalosti programování, nebo žádné. Jedno z možností je si vytvořit vlastní jazyk. Toto řešení přináší problémy s tím, že neexistují žádné naučné materiály. Zároveň bychom museli vytvořit vlastní interpret.

Z těchto důvodů jsem se rozhodl použít jazyk Python. Python je jednoduchý a velmi rozšířený jazyk. Zároveň existují knihovny v C#, které umožňují spouštět kód v Pythonu přímo v C# programu.

Další věcí, kterou je potřeba vyřešit, je výběr správné knihovny pro uživatelské rozhraní. Požadavek, který jsme si sami stanovili je podpora operačních systémů Windows, Linux a MacOS. Zároveň bychom chtěli, aby knihovna obsahovala, nebo aby existovaly jiné knihovny pro zobrazování grafů a map. Ideální by byla také knihovna pro textový editor pro psaní kódu.

Chtěli bychom, aby knihovna pro uživatelské rozhraní neměla chyby, neměla zastaralý interface a ideálně aby byla dobře dokumentovaná.

Jednou z možností je vytvořit nezávislou aplikaci pro frontend, která by byla napsána jako webová aplikace. Využívat by mohla například populární knihovnu React.

S backendem by pak komunikovala pomocí jasně definovaného API.

Nevýhodou je, že by se jednalo o webovou aplikaci, nikoli o desktopovou.

Další možností je využít knihovnu Electron.NET, což je zaobalení nativní aplikace Electron s vestavěnou aplikací ASP.NET Core.

Následující možností je využít knihovnu Avalonia. Tato knihovna je multiplatformní a podporuje .NET Core. Jedná se o knihovnu vycházející z podobné C# knihovny pro uživatelské rozhraní WPF. Rozdílem je, že WPF není multiplatformní.

Avalonia je relativně nová knihovna, ikdyž k založení společnosti došlo již v roce 2013. Do použitelného stavu se ovšem dostala až v minulých několika letech. Na knihovně se stále aktivně pracuje.

Právě tato knihovna byla vybrána pro implementaci uživatelského rozhraní.

6.1 Stakeholders

Určitý seznam lze nalézt v dokumentaci softwaru:

./bc_thesis_tichavsky/WorldsFactory/documentation/requirements/stakeholders.md

Přesto zde shrneme, pro které skupiny lidí je náš program určen, protože z toho mohou plynout zajímavé poznatky a nápady směřující k dalším user requirementům. První hlavní skupinou jsou pochopitelně spisovatelé, scénáristé a herní spisovatelé. Program by měl tedy uchovávat texty konkrétních děl podle typů. Dále by ovšem program mohl být užitečný například pro překladatele, nebo dabéry, kteří nad vytvořenými dialogy a texty příběhu pracují. Tyto texty by tedy neměly být pouze ve formátu vhodný pro spisovatele, ale také pro tyto skupiny lidí.

Další skupinou mohou být například režiséři, kteří by mohli chtít po programu nějaké specifické vlastnosti, jako informace pro managování natáčení jednotlivých scén. Například vytváření rozvrhu natáčení.

Skladatelé a zvukoví designéři by mohli chtít nahlížet do příběhu a podle zanesených dat vytvářet hudbu a zvukové efekty.

Další skupinou mohou být například návrháři postav nebo úrovní her.

V neposlední řadě Dungeon Master, který by mohl využívat program pro předpřipravení různých scénářů příběhu a ukládání historie příběhu.

Jistě by se nalezlo i mnoho dalších skupin lidí. Tyto jsou ovšem ty nejvýznamnější.

6.2 Uživatelské požadavky

Seznam základních uživatelských požadavků lze nalézt v dokumentaci softwaru:

./bc_thesis_tichavsky/WorldsFactory/documentation/requirements/user_requirements/project_managing.md

Pro větší přehlednost jsou rozděleny do kategorií podle míry logické souvislosti. Například uživatelské požadavky *Po spuštění*, které se týkají pouze toho, co uživatel vidí po spuštění programu. Například uživatel vidí informace o nových aktualizacích softwaru, nebo vidí seznam všech projektů.

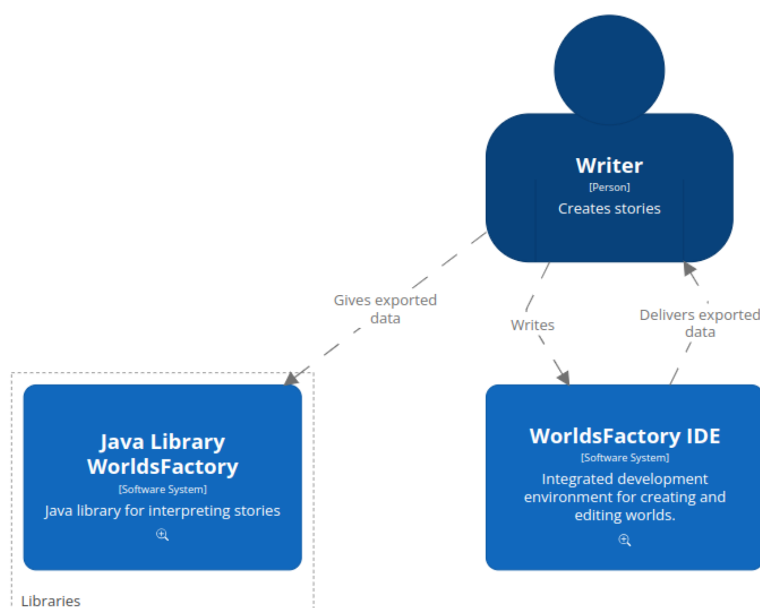
Další skupinou je například *Management projektů*. Jak již bylo zmíněno, seznam lze nalézt v dokumentaci softwaru.

6.3 Architektura

Z architektonického hlediska bude moci být náš software označen za software s monolitickou architekturou. To znamená, že všechny části programu jsou v jednom celku. Veškeré instance softwaru jsou propojené, tak, že má software jediné architektonické kvantum. Díky tomu, že celý software bude složen z jediného procesu, se na naše dílo budou vztahovat všechny pozitivní vlastnosti typické pro monolitické architektury. Netýkají se nás problémy se vzdálenou komunikací mezi jednotlivými částmi programu. Nemusíme se zabývat problémy s distribucí a nasazením, jako v případě nemonolitických architektur.

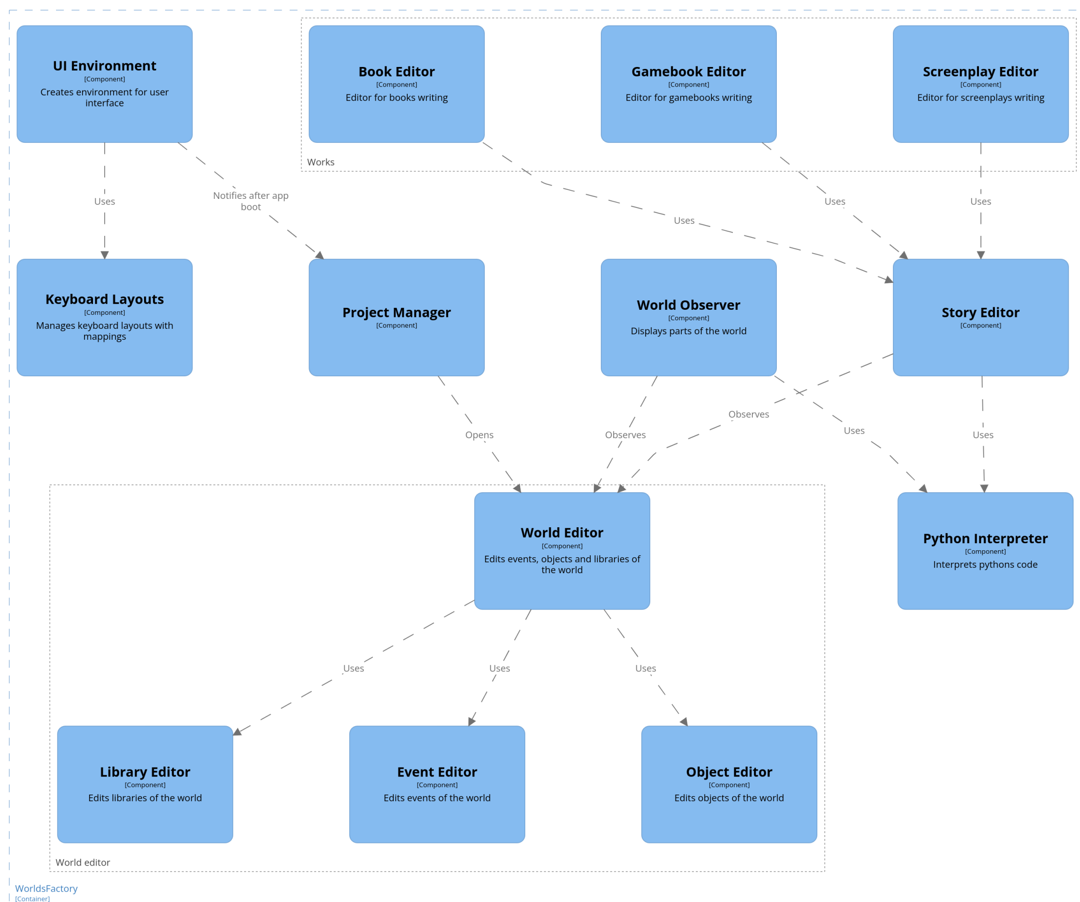
K architektuře softwaru budeme přistupovat podle modulárního monolitického architektonického stylu. To znamená, že bude software rozdělen podle business logiky na jednotlivé moduly. Každý modul bude mít svou vlastní vrstvu pro uživatelské rozhraní, persistenci dat a business logiku.

Dokumentace softwaru využívá pro popis architektury softwaru C4 model. Na obrázku 6.1 je zobrazena architektura softwaru jako celek. Vyobrazuje hlavní komponenty softwarového systému.



Obrázek 6.1: C4 model - Softwarový Systém

Podstatně zajímavější je pohled na strukturu jediného kontejneru WorldsFactory IDE na obrázku 6.2. Pohled zobrazuje jednotlivé moduly hlavního programu a jejich vztahy.



Obrázek 6.2: C4 model - WorldsFactory Container

Při spuštění programu dojde k inicializaci prostředí uživatelského rozhraní v modulu *UI Environment*. Dojde k vytvoření okna programu se základními vizuálními komponentami.

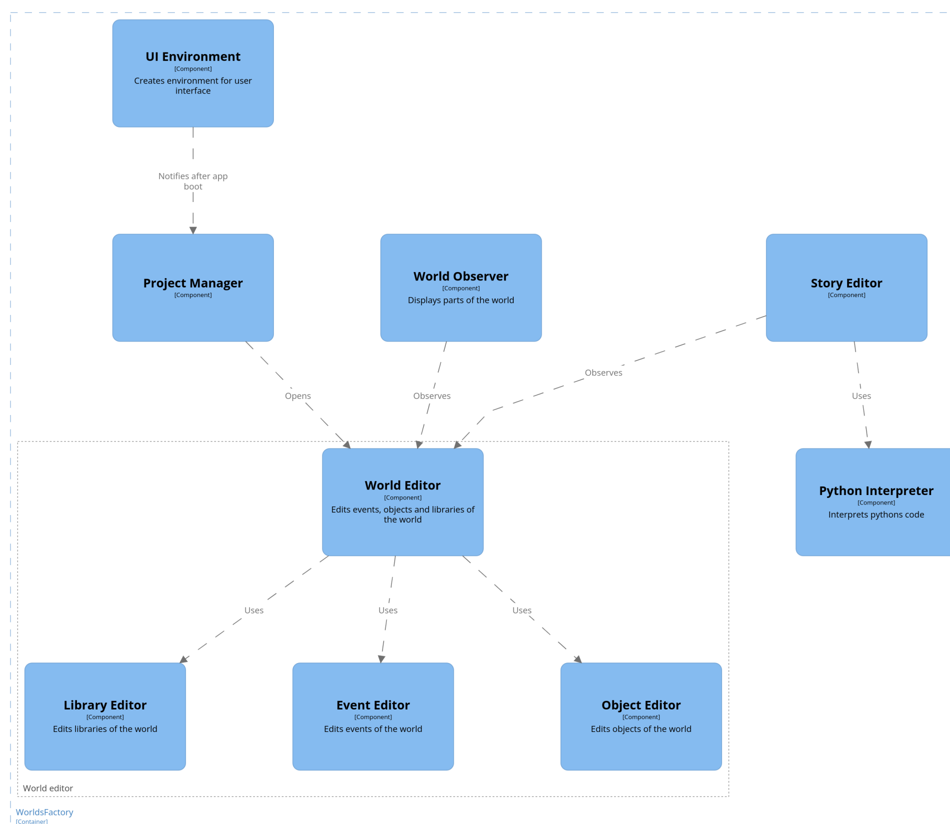
Následně je předána iniciace modulu *Project Management*, který zobrazí své uživatelské rozhraní.

Mírně matoucí by se mohlo zdát, proč *Story Editor* používá modul *Story Interpreter*. Story Editor má splňovat požadavek export dat pro herní engine. Během exportu dat je provedena kontrola kódu právě za pomoci Story Interpreteru. Ten se pokusí kód interpretovat, nemusí žádné metody spouštět.

6.4 Implementace

Jelikož se celkově jedná o relativně rozsáhlé dílo, na jehož vypracování se podílí pouze jeden vývojář, pokusili jsme se vybrat z množiny uživatelských požadavků minimální produkt. Takový produkt, na kterém by bylo možné demonstrovat základní prinicpy tvorby rozvětveného světa.

Přibližné zjednodušení požadavků můžete vidět pomocí změny v architektuře na obrázku 6.3.



Obrázek 6.3: C4 model - Simplified WorldsFactory Container

Zde je vidět, že jsme se rozhodli vynechat moduly pro vytváření konkrétního díla, tedy moduly *Screenplay Editor*, *Gamebook Editor* a *Book Editor*. Story Editor byl ponechán, ale omezen pouze na export dat pro herní engine, a to všech eventů světa.

Podobně tak jsme zjednodušili vytváření světa, kde jsme se rozhodli lineární události ponechat až na následující verzi a knihovny jsme omeznili pouze na jedinou.

V poslední řadě jsme také vynechali modul zajišťující vytváření klávesových layoutů.

6.4.1 Prostředí uživatelského rozhraní

Tento modul se zasluhuje o zobrazení okna programu a základních vizuálních komponent. Zároveň poskytuje statické metody pro zobrazování notifikačních okének. V programu je možné modul nalézt pod složkou screen. Odpovídá namespacu *WorldsFactory.screen.*

Jednotlivé komponenty uživatelského rozhraní se dělí do oken a každé okno je dále rozděleno pomocí *Controls*, které dále obsahují další komponenty. To je základní dělení podle knihovny Avalonia UI. Některé třídy mají název ukončený

slovem *View*, což znamená že se jedná o třídu zobrazitelnou uživatelským rozhraním. Jejich název obvykle končí koncovkou *.axaml.cs*. Každá view má vlastní *XAML* soubor, který obsahuje popis vzhledu. Obvykle je název souboru stejný, akorát s koncovkou *.axaml*. Některé views mají také vlastní ViewModel třídu, která obsahuje většinu logiky pro zobrazení. Jedná se o takzvaný *MVVM* pattern, který je v knihovně Avalonia UI podporován.

Obsah hlavního okna aplikace je rozdělen do takzvaných *Cards* (každá karta obsahuje vlastní View). Jednotlivé karty lze přesouvat, skládat na sebe a měnit velikost, podobně jako to bývá u programů pro programování kódu. Jednotlivé karty se dají také zavírat.

Tata funkcionalita byla vytvořena jako jedna z prvních. Během implementace bylo zjištěno, že dokumentace knihovny Avalonia nebyla v té době úplně dokonalá. Důsledkem mimo jiné je, že kód této části programu není přehledný a zasluhoval by refaktorizaci.

6.4.2 Management Projektů

Tuto komponentu lze nalézt pod složkou *project*. Odpovídá modulu namespacu *WorldsFactory.project*. Zasluhuje se o uchovávání odkazů na všechny projekty, které byly vytvořeny nebo otevřeny. Tato data jsou pak uchovávána ve složce aplikačních dat. Záleží na operačním systému, kde jsou aplikační data reálně uložena.

Podle těchto dat pak modul zobrazuje seznam všech projektů, které byly vytvořeny nebo otevřeny. Zároveň si pamatuje naposledy otevřené projekty.

Modul umí zobrazit formulář pro vytvoření nového projektu. Zde uživatel může zadat základní informace o projektu a o světě. Buď může vytvořit nový svět, nebo vybrat existující svět ze souborového systému.

V případě, kdy si uživatel vybere nějaký projekt, dojde k takzvanému *otevření projektu*. V té chvíli jsou všechny karty zavřeny a dochází k načítání dat projektu. Jedna instance programu předpokládá práci na jediném projektu.

Schopností tohoto modulu je přistupovat k datům o projektu. Toho následně dokáže využívat ve formě karty s uživatelským rozhraním pro navigaci uživatele v projektu. Toto je zajištěno pomocí třídy *OverviewProjectView*.

6.4.3 World Editor

Načtení projektu zahrnuje také načtení světa. K tomu dochází pomocí třídy *World Loaderu*. Načítá postupně všechny knihovny, objekty a události. Před samotným načtením provede kontrolu, jestli má svět správnou strukturu.

World Editor odpovídá kódu v adresáři *world*. A jeho úkolem je zastřešovat data světa. Dokáže poskytovat seznamy všech částí světa.

Velmi hojně užívaný je zde návrhový vzor *Observer* [6], který umožňuje snadnou aktualizaci všech otevřených karet, pokud dojde ke změně dat světa.

6.4.4 Ukládání a načítání dat pomocí Loaderů

Podobný princip, jako je u načtení světa pomocí třídy World Loaderu, je použit podobný princip také pro ostatní moduly a komponenty. Loader je třída zajišťující persistentní logiku. Dokáže ukládat a načítat data ze souborového systému podle lokace dat ve složce projektu. Máme tedy také *Library Loader*, *Object Loader*, *Event Loader*, *Class Loader* a *MethodsBody Loader*. S každým loaderem pak program komunikuje pomocí jasně definovaných interfaců, jako je například *ILibraryLoader*. Tím je zajištěna vyšší úroveň koheze programu.

Loadery ukládají data o projektu pomocí knihovny pro serializaci dat podle *json* formátu. konkrétně se jedná o knihovnu Newtonsoft.json [7].

6.4.5 Library Editor, Object Editor, Event Editor

Všechny tyto moduly lze nalézt pod složkou *world*. Jedná se tedy defakto o podmoduly World Editoru.

Library Editor

Library Editor byl v rámci minimálního produktu zjednodušen na jedinou knihovnu. To znamená že se musí zabývat pouze vytvářením nových tříd, jejich funkcí a proměnných. Tyto proměnné jsou v programu označovány jako *Properties*.

Při vytváření nových properties třídy jsme se rozhodli, pro jednodušší kontrolu uživatelova kódu, zavést typované proměnné. To znamená, že uživatel musí při vytváření nové proměnné vybrat typ, který bude proměnná představovat. Další výhodou vynucení typování je při překladu uživatelova kódu do kódu herního enginu. To sice také není v této verzi implementováno, ale je to věc, se kterou můžeme počítat do budoucna.

Object Editor

Object Editor je zodpovědný za vytváření nových objektů. Objekty jsou vytvářeny z tříd, které jsou definovány v Library Editoru.

Měl by nastavovat defaultní hodnoty jednotlivým objektům. Jelikož se ale zatím nepočítá se zobrazováním stavu světa, tak tato funkcionalita zatím není implementována.

Event Editor

Event Editor je zodpovědný za vytváření nových událostí. Každá událost obsahuje podmínkovou funkci a metodu události. Třída *Event* také vlastní takzvaný *SequenceManager*, který specifikuje, po kterých událostech může tato nastat. Zároveň je zde informace o tom, zda li je událost součástí nějaké *EventContainer*, respektive, jestli spadá do nějaké pojmenované množiny událostí.

6.4.6 Identifikátory a Tagy

Skoro vše, co uživatel vytvořil, má v programu svůj identifikátor. Jedná-li se o třídu, dědí ze třídy *ConceptWithID* (nebo nějaký jeho předek). Při jeho vytvoření je pak automaticky přiřazen do množiny všech identifikátorů, které jsou managované třídou *IDConceptManager*. Takovýchto kolekcí identifikátorů je v programu několik. Máme kolekci všech identifikátorů, která je dostupná z modulu *World Editor*, respektive přímo třídy World. A dále máme *IDClassManager* a *IDObjectManager*, které obsahují všechny identifikátory tříd, respektive objektů.

ConceptWithID implementuje *TaggedConcept* interface, který zajišťuje otagování daného konceptu. Pojem koncept zde ale není myšleno ve smyslu jako při definici příběhu. Jedná se o něco, co uživatel vytváří v programu a je součástí světa. Tedy třídy, objekty, události, atd. Tagování dále bude sloužit při vyhledávání konceptů uživatelem.

Podobně jako je IDConceptManager, máme také TagManager, který ke každému tagu mapuje všechny koncepty s tímto tagem.

Samotný identifikátor je dělen na několik částí. První částí je takzvaný prefix. Ten je rozdělen na dvě části. První část začíná zavináčem a končí dvojtečkou. Určuje typ konceptu. Druhou částí může být doupřesnění. Například prefix identifikátoru property nějaké třídy se skládá z *@property:* a názvu třídy *názevTřídy_*. Podtržítko určuje konec názvu třídy, tudíž název třídy nesmí končit podtržítkem. Další částí je samotný název konceptu.

```
@class:ExampleClass
@property:ExampleClass_x
@method:ExampleClass_myMethodsName
```

6.4.7 Story Editor

Zatím podporuje pouze export dat pro herní engine a to ve formě Python kódu. Externí knihovna, která by byla použita v herním enginu pak používá interpret pro jeho spuštění. Modul lze nalézt pod složkou *works*. Přesněji pod složkou *src/works/storyInterpreterWork*.

Export bere v potaz všechna data světa a vytváří kód, který je pak možné spustit v herním enginu. Kontroluje jeho spustitelnost a popřípadě oznamuje uživateli chyby. Dokáže lokalizovat chybu v kódu. Lepší by bylo, kdyby uživatel mohl vybrat, které části světa mají být příběhem, tedy která data je potřeba exportovat.

Zatím se počítá s exportem výhradně pro herní engine naprogramovaný v jazyce Java. Zde jsem bohužel nenašel dostatečně dobrou knihovnu pro spuštění Python kódu, která by dokázala volat metody v Pythonu a sledovat jeho data.

Z toho důvodu jsme si mírně ulehčili práci za cenu ztráty efektivity. Knihovna bude interpretovat data v separátním procesu, se kterým následně bude komunikovat pomocí standardního vstupu a výstupu. Viz kapitolu *WorldsFactory Knihovny*, sekce *Implementace*7.3.

Export objektu a tříd

Třída se exportuje jako třída v Pythonu. Jméno je odvozeno od názvu původní třídy s prefixem *class_*.

Pokud má třída property, její název s typem se zapíše do seznamu všech properties a vygeneruje se setter a getter.

Zde je takový příklad kódu, který by byl vygenerován pro třídu *ExampleClass* s property x:

```
class class_ExampleClass:
  properties = {
    "x": "@basicType:Integer"
  }
  @property
  def x(self):
    return self._x
    pass
  @x.setter
  def x(self, value):
    self._x = value
    set_property("x", value, self.properties["x"])
```

Metoda *set_property* na standardní výstup vypisuje informace o změně hodnoty property.

V případě exportu metody se vytvoří dvě metody. Jedna původní, psaná uživatelem, a druhá metoda, která volá první a chytá výjimky.

```
class class_Example:
  properties = {
    "x": "@basicType:Integer",
    "y": "@basicType:Integer",
    "description": "@basicType:String"
  }

  def Initialize(self, x, y, description):
    try:
      return self._Initialize( x, y, description)
    except Exception as e:
      message = e.message if hasattr(e, "message") else
          str(e)
      class_ = "@class:Place"
      method = "@method:Place_Initialize"
      raise MethodRuntimeException("An Exception in
         @class:Place during method ...\n" + message)

  def _Initialize(self, x, y, description):
    self.x = x
    self.y = y
    self.description = description
    objects.Map.places[x][y] = self
```

```
22      pass
23      pass
24    pass
```

Export událostí

Událost se exportuje velmi podobně jako třída. Vytvoří se třída v Pythonu, do které se exportují obě funkce. Rozdílem je, že třída je označena za singleton, aby uživatel nemohl dělat nesmyslné věci.

Dále třída obsahuje seznam eventů, které mohou následovat po této události. Všimněme si rozdílu. Ve World editoru uživatel nastavoval po kterých událostech může tato událost nastat. To je při tvorbě světa logičtější. V této chvíli se ale programu více hodí opačný přístup.

Kromě exportu jednotlivých událostí se také přidá kód pro posun v příběhu. Metoda, která drží naposledy provedený event a která při její zavolání spustí všechny podmínkové funkce událostí, které mohou následovat. Pokud nějaká událost nastane, zavolá se její metoda události.

Export polí

Property může mít také typ pole nějakého typu, popřípadě vícedimenzionální pole. Všimněme si, že při exportu běžné property je pro nás velmi důležité použití setterů a getterů, tedy metod, které jsou volány, když uživatelský kód s property pracuje. V případě pole bohužel nemáme tuto výhodu.

Pro export pole vytvoříme speciální typ a property bude pak jeho instancí.

```
1 class class_Map:
2   properties = {
3     "places": "@class:Place"
4   }
5   places = ArrayObject((2,2), "places", "@class:Place")
6   pass
```

ArrayObject je defakto vlastní implementace vícedimenzionálního pole, které se skládá z tzv. *CustomListů*. U nich pak lze sledovat změny hodnot a notifikovat knihovnu o změně stavu.

6.4.8 Zobrazování světa

Komponenta *World Observer* byla v rámci výběru minimálního produktu osekána na pouhé zobrazování grafu eventů. Graf zobrazuje uzly jako události. Hrany jsou orientované, a pokud vede hrana z události A do události B, znamená to, že po té co nastane událost A, může následovat událost B. Naopak pokud hrana nevede z A do B, znamená to, že událost B nemůže následovat po události A. Navíc dokáže zobrazit Eventy setříděné podle *Evnet containerů*.

Složku pro komponentu *World Observer* lze nalézt pod jmenným prostorem *WorldsFactory.world.visualisations*. Popřípadě pod adresářem:

`./bc_thesis_tichavsky/WorldsFactory/src/world/visualisations`

Pro zobrazení takového grafu, tento modul obsahuje nové *UserControl* s názevem *RelationGraphWithSubGraphs*.

To využívá knihovnu *AvaloniaGraphs*, pro zobrazení grafu.

6.4.9 Generování dokumentace

Projekt podporuje generování dokumentace pomocí nástroje *Doxygen*.

```
cd ./bc_thesis_tichavsky/WorldsFactory/documentation
doxygen Doxyfile
```

Dokumentace by měla být vygenerována ve stejné složce jako se nachází soubor *Doxyfile*.

6.5 AvaloniaGraphs

Během práce na programu jsme se dostali také k problému zobrazování částí světa. První věcí, kterou jsme se rozhodli implementovat, bylo zobrazování grafu událostí, který zobrazuje vztahy mezi nimi. Osvětluje uživateli jaké události mohou následovat po jiných událostech.

Zde jsme ovšem narazili na naše nedostatečné zjišťování, kolik knihoven tento problém řeší. Během rozhodování se, jakou knihovnu použít, jsme samozřejmě toto hledisko brali v potaz a pro vykreslování grafů v knihovně Avalonia UI jsme našli celou řadu knihoven. Při konkrétnějším studování jsme ovšem zjistili, že většina z nalezených podporovala pouze vykreslování grafů s anglickým překladem *chart*. Na vykreslování grafů s uzly a hranami jsme nalezli knihovnu jedinou *GoDiagram*[8], jejíž licenční podmínky byly pro nás nepřijatelné. Dle licence je možné používat knihovnu pouze prvních 30 dnů od zkompilování programu. Následně by knihovna sama zařídila ukončení celého programu.

Rozhodli jsme se tedy vytvořit knihovnu vlastní. Výhodou je, že můžeme přidat funkcionality, které jsou na míru šité našemu programu. Přesto se jedná o zcela oddělenou část softwarového díla.

Z aktuálního pohledu jsme zcela přehlédli knihovnu *AvaloniaGraphControl* [9], která vypadá, že by mohla splňovat naše požadavky.

Kód knihovny je dostupný v repozitáři projektu pod adresářem *AvaloniaGraphs*.

6.5.1 Rozhraní

Použití knihovny je docela snadné. Uživatel vytvoří instance objektů *Node* a následně *Edge*, pro dva uzly.

Ty pak jsou předány do instance třídy *Graph*.

```
1  var node0 = new GraphNode() { ContentControl = new TextBlock() { Text = "Node 0" } };
2  var node1 = new GraphNode() { ContentControl = new TextBlock() { Text = "Node 1" } };
3  var node2 = new GraphNode() { ContentControl = new TextBlock() { Text = "Node 2" } };
4  var node3 = new GraphNode() { ContentControl = new TextBlock() { Text = "Node 3" } };
5
6  graph = new Graph()
7  {
8      Nodes = {
9          node0,
10         node1,
11         node2,
12         node3
13     },
14     Edges = {
```

```
        new GraphEdge(node0, node1) { IsDirected = true },
        new GraphEdge(node1, node2) { IsDirected = true },
        new GraphEdge(node2, node0) { IsDirected = true },
        new GraphEdge(node3, node1) { IsDirected = true }
    },
    Layout = new SpringGraphLayout()
    {
        Iterations = 100,
        Width = 800,
        Height = 400,
        WithAnimation = true
    },
};
```

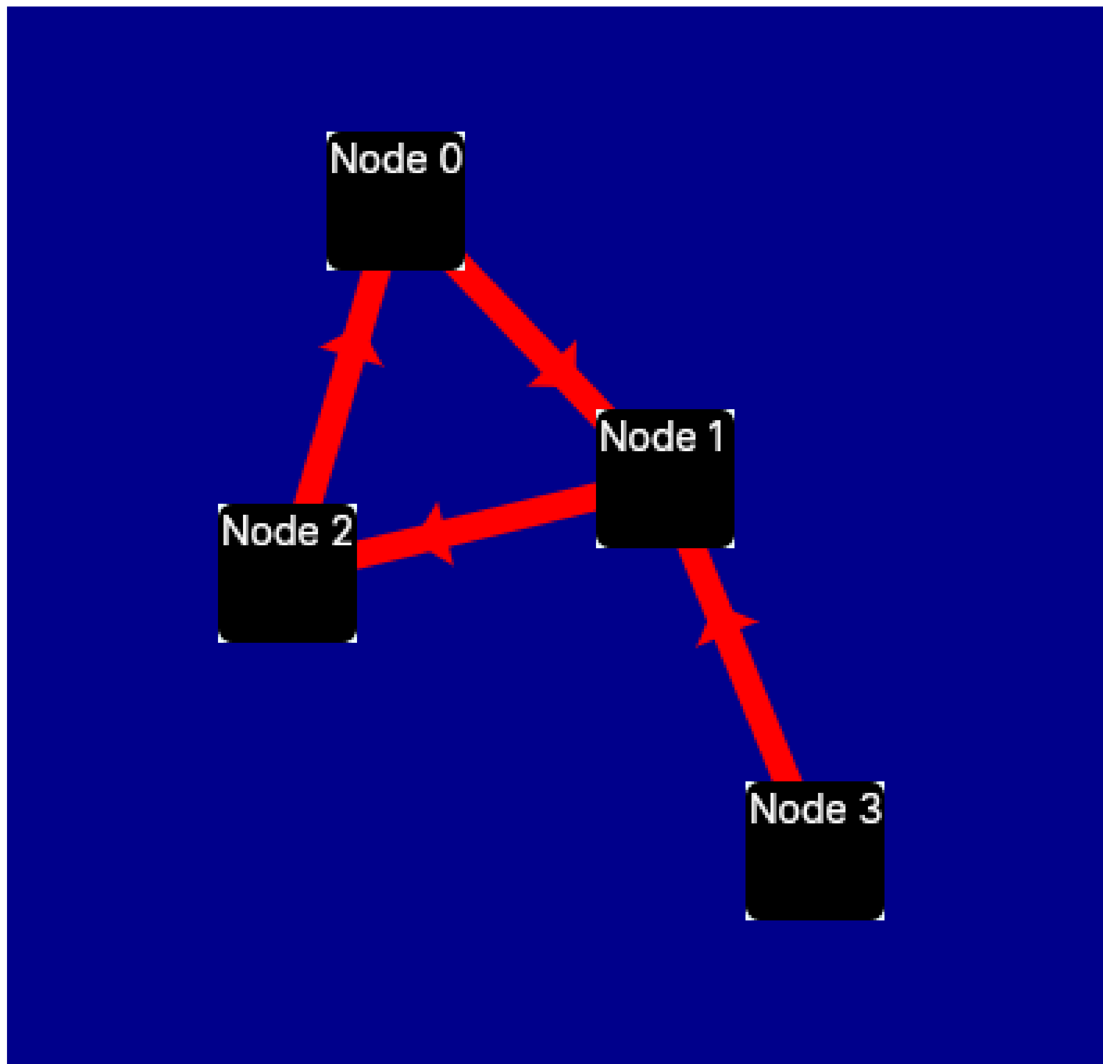
Všímeme si, že každý uzel má nastavený *ContentControl*, který specifikuje *Control*, který se má zobrazit na pozici uzlu.

Následně je potřeba vytvořit speciální *UserControl*, který bude umět graf vykreslit. Konkrétně se jedná o třídu *GraphView*.

```
var graphView = new GraphView(graph);
graph.Layout?.ApplyLayout(graph);

mainPanel.Children.Add(graphView);
```

Vizte obrázek 6.4, kde je vizualizace grafu popsaný výše.



Obrázek 6.4: Vizualizace jednoduchého grafu v AvaloniaGraphs

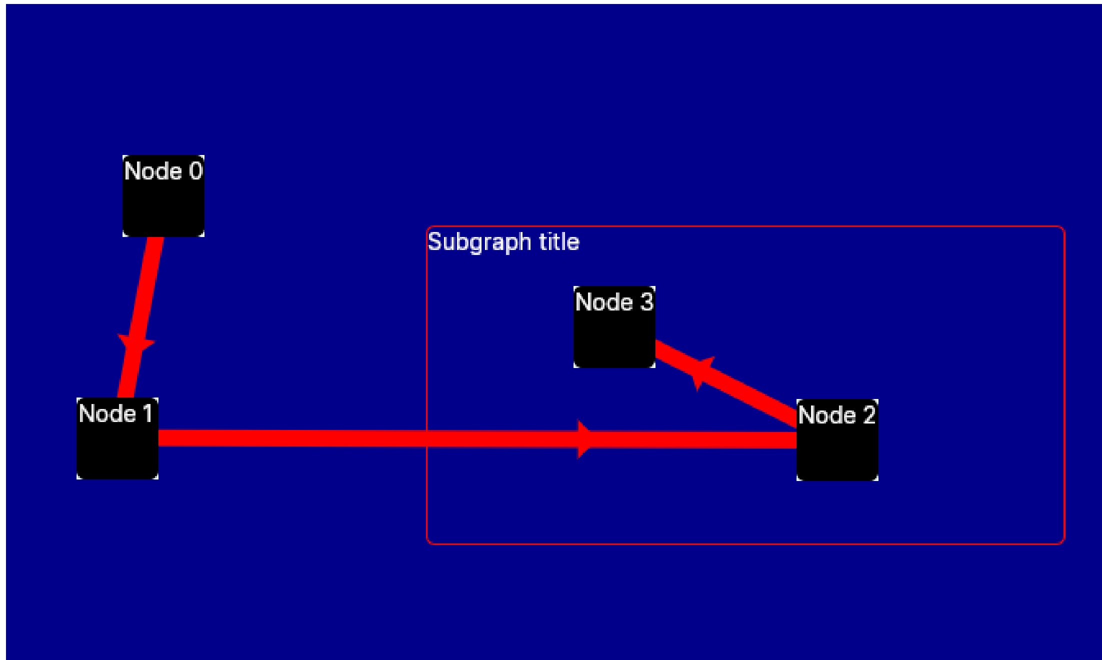
Knihovna dále umožňuje zobrazovat graf jako uzel v jiném grafu. Pro dosažení takehoto efektu je potřeba vytvořit speciální třídu *SubGraph*. Tato třída rozšiřuje třídu *GraphNode* a umožňuje vytvářet vnořené grafy.

```
var subGraph = new SubGraph()
{
    ContentControl = new TextBlock() { Text = "SubGraph
       " },
    Graph = new Graph()
    {
        Nodes = {
            node2,
            node3
        },
        Edges = {
            new GraphEdge(node2, node3) { IsDirected =
               true }
        },
        Layout = new SpringGraphLayout()
        {
            Iterations = 100,
            Width = 800,
            Height = 400
        }
    }
};
```

Pro správné zobrazení musí ovšem graf, který by měl obsahovat nějaký subgraf, využít správný Layoutovací algoritmus.

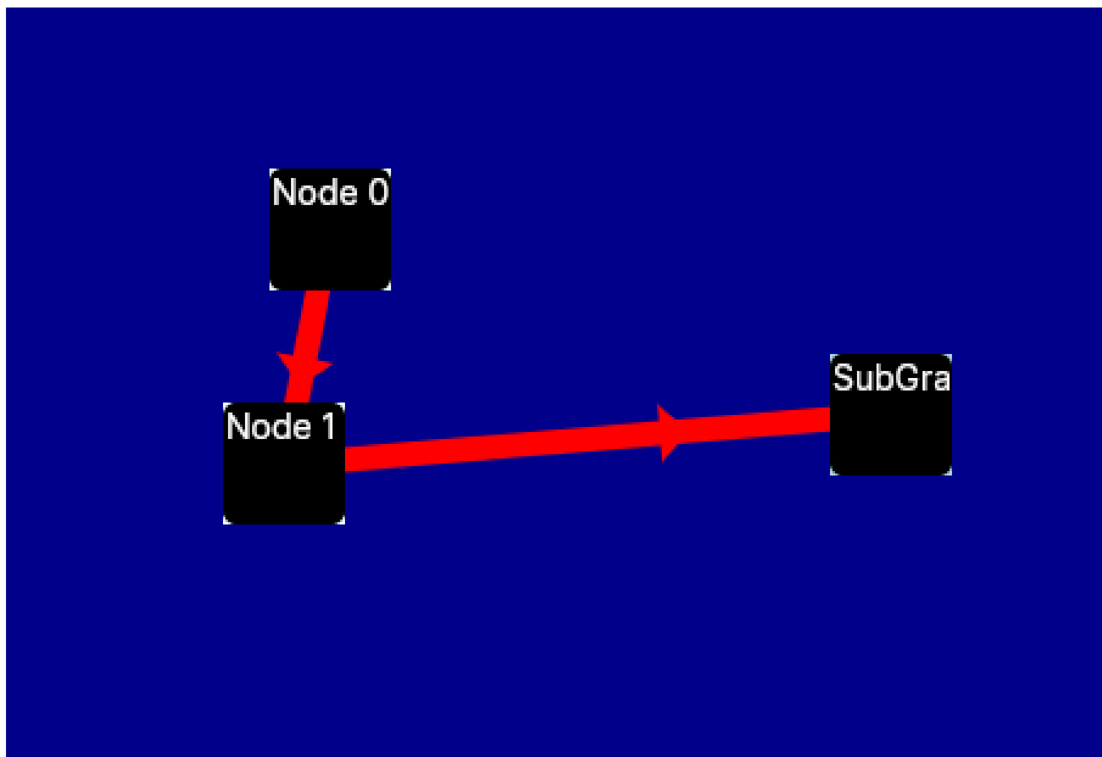
```
Layout = new SpringGraphLayoutWithSubGraphs()
{
    Iterations = 1000,
    Width = 800,
    Height = 400,
}
```

Vizte obrázek 6.5, kde je vizualizace grafu s vnořeným grafem.



Obrázek 6.5: Vizualizace grafu s vnořeným grafem v AvaloniaGraphs

Při dostatečném oddálení pomocí kolečka myši se místo vnořeného grafu zobrazí pouze uzel, který reprezentuje vnořený graf (protože subgraph je zase jen uzel). Následující obrázek 6.6 ukazuje tento efekt.



Obrázek 6.6: Vizualizace grafu s vnořeným grafem v AvaloniaGraphs, po oddálení

Knihovnu lze nalézt v podadresáři

`./bc_thesis_tichavsky/AvaloniaGraphs`

6.5.2 Implemetnace layoutovacího algoritmu

SpringGraphLayout podporuje zatím pouze jeden základní layoutovací algoritmus pro nalezení pozic uzlů. Nejprve algoritmus hledá komponenty slabé souvislosti. Prostor pro zobrazení se rozdělí na tabulku a každé z komponent se přiřadí jedna pozice. Každá z nich má pak jasně daný prostor, do kterého se může vykreslit.

Na každé se následně aplikuje layoutovací algoritmus. Konkrétně trochu pozměněný *Fruchterman Reingold* [10] algoritmus.

Frucherman Reingold algoritmus je založen na představě, že hrany mezi uzly jsou pružiny a uzly jsou navzájem odpuzovány, nebo přitahovány. Iterativně se hledá pozice s nejnižší energií.

Změna v algoritmu spočívá v připočtení síly mezi dvěma vrcholy spojené cestou délky dvou hran. Pro takové dva vrcholy se spočítá úhel mezi oběma hranami cesty a pokud je úhel příliš malý, přidá se mezi vrcholy síla odpuzení. Navíc k třetímu uzlu ležícímu na dané cestě se přičte síla přiblížení směrem k oběma vrcholům.

SpringGraphLayoutWithSubGraphs hledá pozice uzlů také pro vnořené grafy. Výpočet se provádí rekurzivně, protože každý vnořený graf má definován vlastní layoutovací algoritmus. Layoutovacímu algoritmu se tedy vždy předá jak veliký prostor může využít a v layoutovacím algoritmu rodičovského grafu se pak všechny pozice uzlů posunou na správnou pozici.

Během hledání pozic uzlů se také vypočítává, jaká je ideální velikost subgrafu, při které se má zobrazit vnořený graf.

7. WorldsFactory Knihovny

7.1 Motivace

Jedním z hlavních cílů minimálního produktu projektu je vytvoření knihovny, která bude schopna zpracovávat exportovaná data a interpretovat je. Zjednodušeně řečeno vyprávět příběh, který je ovlivněn herním enginem.

Knihovna i herní engine jsou dvě oddělené entity, kde obě chtějí pracovat se stejnými daty, daty světa. Obě tedy potřebují mít způsob, jak k těmto datům přistupovat. Jednou z možností je, že by si všechna data světa pamatovala knihovna. To by bylo možné v případě, kdyby uživatel nejprve pracoval na světě, a až poté by vytvářel herní engine. Ideálnější přístup by mohl být, kdyby data měl v držení herní engine a knihovně by byl poskytnut interface, přes který by k datům mohla přistupovat.

7.2 Moment posunu v příběhu

Další otázkou je, v jaké chvíli by mělo docházet k posunu v příběhu, respektive ověření, jestli k posunu nemá dojít.

7.2.1 Kontrola při každé změně hodnoty

Jednou z možností je, že by ke kontrole docházelo při každé změně nějaké hodnoty světa (respektive příběhu). Toto by bylo proveditelné pouze v případě, že by veškerá data byla ukládána v knihovně. Pokud by data držel herní engine, musel by se také postarat o to, že při libovolné změně hodnoty by byla notifikována knihovna. To by znamenalo v každém setteru navíc zavolat metodu. To nezní příliš pozitivně, ale lepší řešení zde asi nenajdeme.

Ještě jedním řešením by bylo neustále dokola ověřovat všechny aktuální podmínkové funkce. To by bylo ale vykoupeno ztrátu výkonu.

Pokud jde tedy uživateli o tento případ, ideální je držet všechna data v knihovně.

7.2.2 Kontrola při signálu

Další možností je, že by k ověření nedocházelo automaticky, ale po konkrétním signálu od herního enginu. V takové chvíli není žádný problém nechat data v držení herního enginu a asi by to bylo i praktičtější řešení.

7.3 Implementace

Jelikož knihovna bude vytvářena již pro hotový herní engine, který je naprogramovaný v jazyce Java, je nutné, aby knihovna byla také napsána v jazyce Java.

Jelikož jsme v době práce na programu nenalezli žádnou vhodnou Java knihovnu, která by dokázala používat pythonovský interpret a vyhovovala by našim

potřebám, rozhodli jsme se spustit pythonovský program jako externí proces, se kterým bude knihovna komunikovat.

Minimálně knihovna *Jython* [11] není vhodná, protože nepodporuje Python verze 3.x. Zde jsme jistě udělali chybu v rozhodnutí. Nemohli jsme nalézt lepší řešení, a tak jsme se rozhodli o komunikaci pomocí standardního vstupu a výstupu.

Lepším řešením by bylo už jen ponechat proces jako samostatnou mikroslužbu, která by komunikovala přes nějaký specializovaný middleware.

Oddělený proces implikuje, že veškerá data světa budou u obou procesů duplikována. To znamená, že máme problém s tím, že každý setter u herního enginu musí notifikovat knihovnu. Musíme data obou světů provázat, aby se automaticky synchronizovala.

7.3.1 Uživatelské rozhraní

Pokusíme se nalézt uživatelské rozhraní, které bude založeno na anotacích. Ideální api by mohlo vypada takto:

```
@StoryObject("myClassName")
public class EngineObject {
    @StoryObjectName
    private String name;

    public EngineObject(String name) {
        this.name = name;
    }

    @LinkedField("myProperty")
    private String description;

    public String getDescription() {
        return description;
    }

    public void setDescription(String description) {
        this.description = description;
    }
}
```

Třída v herním enginu má anotaci *@StoryObject* a musí obsahovat anotaci *@StoryObjectName* nad jednou z proměnných. Tato proměnná bude sloužit jako identifikátor objektu.

Dále, má-li třída nějakou property, měla by být označena anotací *@LinkedField*. Tato anotace by měla obsahovat název. Následně v době kompilace by příslušný anotační procesor měl nalézt příslušnou setter metodu a vložit do ní kód, který bude notifikovat knihovnu o změně hodnoty.

Takovéto řešení by bylo opravdu velmi elegantní, bohužel reálná implementace není tak jednoduchá. Java kompilátor nepodporuje přidávání kódu během kompi-

lace. Teoreticky to možné je. Existuje jeden projekt, nazvaný Project lombok[12], kterému se přesně toto podařilo. Vychází ovšem z nezdokumentovaných vlastností Java kompilátoru a jevilo se nám to jako složité řešení, které by vyžadovalo zdlouhavé zkoumání. Můžeme to uvažovat, ovšem, jako možné rozšíření.

Nicméně, tento problém je specifický pro jazyk Java. V mnoha jiných jazycích podobného elegantního API dosáhnout lze. Stačí mít správnou podporu od kompilátoru, takovou, aby umožňoval generovat kód před překladem.

Pro naše API se hodí použití anotací. Především proto, že často uživatel vytváří nejprve herní engine a až poté příběh. Chtěli bychom tedy existující kód změnit pouze přidáním anotací. Výsledné API vypadá takto:

```
@WorldsFactoryClass(className = "MyClass")
public static class MyClass {
    public final String objectName;

    public MyClass(String objectName) {
        this.objectName = objectName;
        WorldsFactoryStoryManager.bindObject(objectName,
           this);
    }

    @WorldsFactoryPropertySetter(name = "firstProperty")
    public void firstSetterExample(String value) {
        WorldsFactoryStoryManager.setProperty("
           firstProperty", value, objectName);
        System.out.println("Setter example called");
    }
}
```

Je to defakto stejné, jako v předchozím případě, jen s tím rozdílem, že anotace negenerují žádný kód.

Samotná inicializace příběhu se provádí pomocí třídy *StoryInitializer*, což je pouze factory třída pro vytvoření instance *WorldsFactoryStory*.

```
CompletableFuture<WorldsFactoryStory> story = new
    StoryInitializer()
      .withStoryName("MyStory")
      .withCode(exampleCode)
      .withDebugMode(true)
      .withEventGraphCondition(EventGraphCondition.
         MOVE_MAX_BY_ONE)
      .withCheckingConditionsAfterEachSet(true)
      .build();

WorldsFactoryStoriesManager.endStory(story.join().getName()
    );
```

Metoda *withEventGraphCondition(EventGraphCondition.MOVE_MAX_BY_ONE)* říká, jestli při posunu v příběhu má dojít k opětovnému ověření nových podmínkových funkcí, takže by se posouval příběh dokud by bylo možné.

Metoda *withCheckingConditionsAfterEachSet(true)* říká, jestli má dojít k posunu v příběhu po každé změně libovolné hodnoty.

Statická metoda *endStory* ukončí příběh a ukončí separátní proces.

V případě většího počtu příběhů je možné u anotace třídy specifikovat, v kontextu jakého příběhu se má třída použít. Defaultně se přiřadí k prvnímu vytvořenému příběhu.

Autoload

Při použití knihovny v herním enginu jsme zjistili, že bychom od knihovny potřebovali jakési automatické načítání objektů. Kupříkladu, dojde-li k napojení objektu na příběhový objekt, chtěli bychom, aby vlastnosti objektů byly automaticky nastaveny. Prakticky by to mohlo být provedeno zavoláním metod příslušných vlastností v herním enginu, označené pomocí anotace. Takové automatické načtení vlastností by ovšem mohlo vést k tomu, že bychom se snažili načíst referenci na objekt, který zatím v herním enginu není. V takové chvíli by mohlo být praktické mít možnost vytvářet takové objekty automaticky, ideálně zavoláním konstruktoru bez parametrů.

Zde je potřeba vyřešit problém s hledáním tříd se správnou anotací. Problém je v tom, že prohledávat třídy celého classloaderu může být velmi zdlouhavý proces. Knihovna by tedy měla umožňovat určovat konkrétní balíčky, ve kterých pouze má k hledání dojít.

Testování

Jelikož u této část práce nelze jednoduše program spustit a ověřit správnost, byla vytvořena sada testů imitující běžné použití knihovny. Testy lze spustit příkazem *mvn test* v adresáři projektu *worldsfactory_java_library*.

Část III

Herní engine

8. Spuštění příkladu herního enginu

Součástí této práce je také program, který má představovat herní engine. Jeho úkolem je demonstrovat použití softwaru *WorldsFactory* a přidružené knihovny pro interpretaci exportovaných dat. Pro demonstraci nám vystačí jednoduchý engine, který použije již zmíněnou knihovnu.

Využil jsem svůj vlastní engine, na kterém jsem pracoval částečně i v rámci jiných projektů. Z toho důvodu je složitější než by bylo potřeba.

Idea za celým programem byla vytvořit něco jako vylepšený hratelný gamebook. Uživatel hraje za určitou postavu, která se nachází v prostoru reprezentovaném tabulkou míst. Hráč se může pohybovat z místa na místo. Každé místo je reprezentováno pomocí obrázku na pozadí a textového popisu.

8.1 Spuštění

Herní engin vyžaduje pro správný běh exportovaná data projektu z programu *WorldsFactory*. Postup pro export dat je popsán v kapitole 5, nicméně pro připravené příklady není export nutný. Příklady projektů můžete nalézt v tomto adresáři:

`./bc_thesis_tichavsky/WorldsFactoryProjectExamples`

Jejich již exportované soubory můžete nalézt v adresáři:

`./bc_thesis_tichavsky/GameEngine/Server/world/src/main/resources/story}`

Nalézají se zde dva soubory značící dva různé příběhy.

Herní engine se dělí na dva oddělené programy - server a klient.

Obě aplikace používají pro komunikaci síťové připojení. Bohužel samotné navázání spojení může být problematické. Obě zařízení, na kterých aplikce běží, musí být na stejné lokální síti. To ale častokrát k navázání spojení nestačí. Nejjistějším způsobem je vytvoření hotspotu na straně serveru.

Spuštění serveru

Server je aplikace napsána v jazyce Java. Pro spuštění je vyžadována alespoň verze 17.

Server lze poté spustit pomocí následujícího příkazu:

```
java -jar ./bc_thesis_tichavsky/GameEngine/Server/world/target/world-0.1-jar-with-dependencies.jar
```

Program lze případně zkompilovat pomocí příkazu:

```
cd ./bc_thesis_tichavsky/GameEngine/Server/world
mvn clean package -Dmaven.test.skip assembly:single
```

Testy se zde přeskakují, protože se zastaví na čekání vstupu od uživatele, které bylo narychlo přidáno, aby si mohl hráč vybrat, který z předpřipravených příběhů spustit.

Po spuštění programu by se měly vypsat možnosti příběhu, ze kterých si musí uživatel vybrat. Následně se spustí server a vypíše důležité informace, jako IP adresa a port socketu.

Spuštění klientské aplikace

Klientská aplikace je psána v jazyce Java pro Android, pro mobilní zařízení. Potřebné je mít verzi minimálně SDK 29, což odpovídá Androidu verze 10.

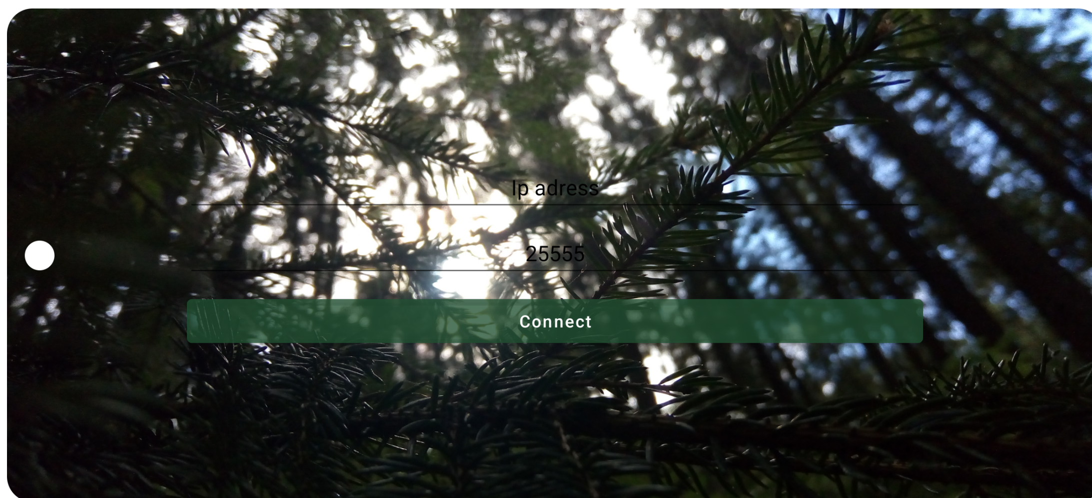
Pro její spuštění je potřeba překopírovat soubor

```
./bc_thesis_tichavsky/GameEngine/MobileClient/apk/release/app-release-unsigned.apk
```

na mobilní zařízení a spustit jej. Systém se tomu bude bránit, nicméně měl by nakonec spuštění umožnit.

Pokud nechceme aplikaci kopírovat na mobilní zařízení, je možné ji spustit z emulátoru, například přímo z *Android Studia*. V takové situaci je asi nejsnazší otevřít projekt z repozitáře a zkompilovat jej. Výhodou takového případu je, že pokud server i klient běží na stejném zařízení, neměl by být problém s navázáním spojení.

Po spuštění mobilní aplikace klienta, se zobrazí uživateli formulář pro navázání spojení, kde je potřeba vyplnit lokální IP adresu serveru a port, který server vypsal. Viz obrázek 8.1.

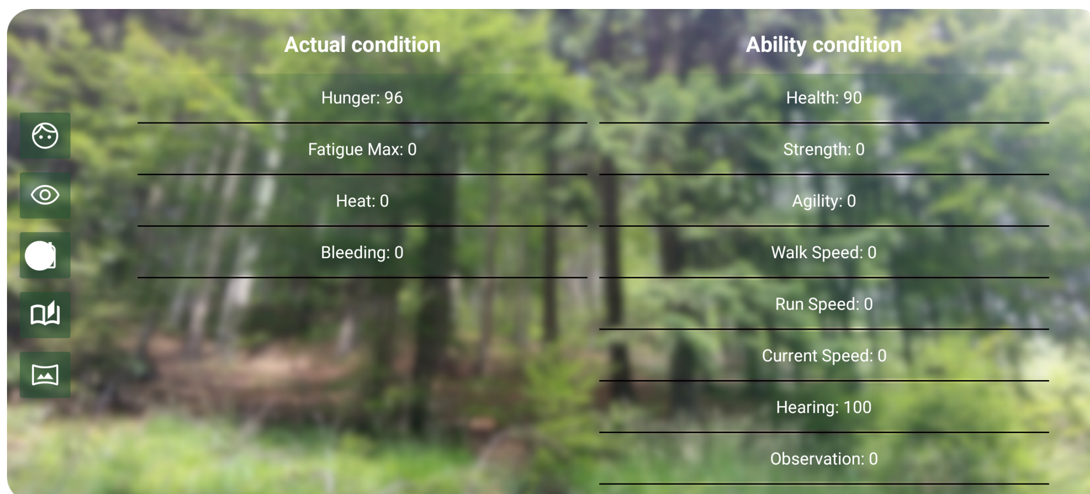


Obrázek 8.1: Formulář pro navázání spojení

Při opětovném zapnutí se aplikace automaticky pokusí navázat spojení podle posledního nastavení. Pokud by se ale server zapnul až po klientovi, je potřeba restartovat klienta, nebo zadat IP adresu a port znovu.

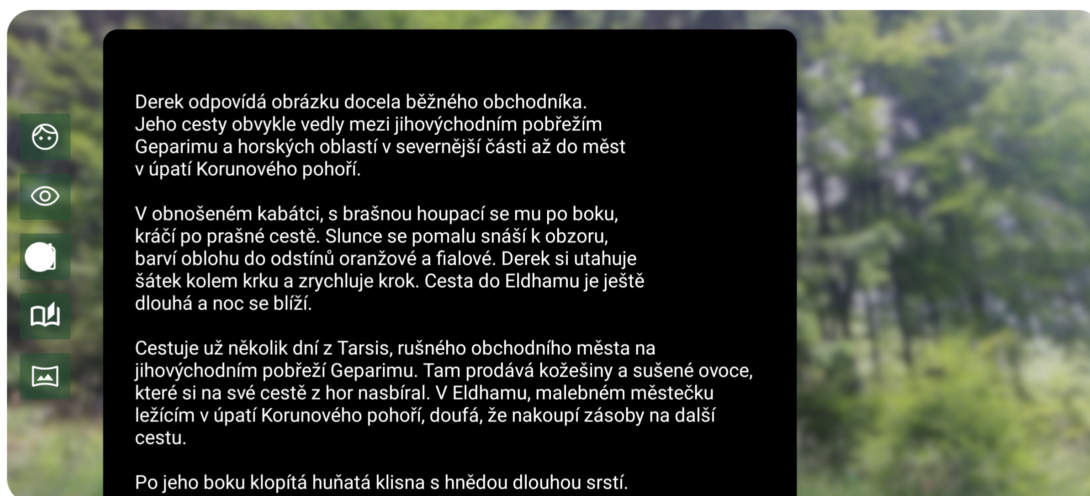
8.2 Příklad spuštění prvního příběhu - Derek a vlci

Po navázání spojení server začne posílat klientovi data o hře a příběhu. Klientovi se ukáže panel s obrázkem aktuální pozice hráče a výpisem informací o jeho postavě. Ty v tuto chvíli nejsou důležité, protože příběh je nijak nebere v potaz. Jak stav aplikace vypadá, lze vidět na obrázku 8.2.



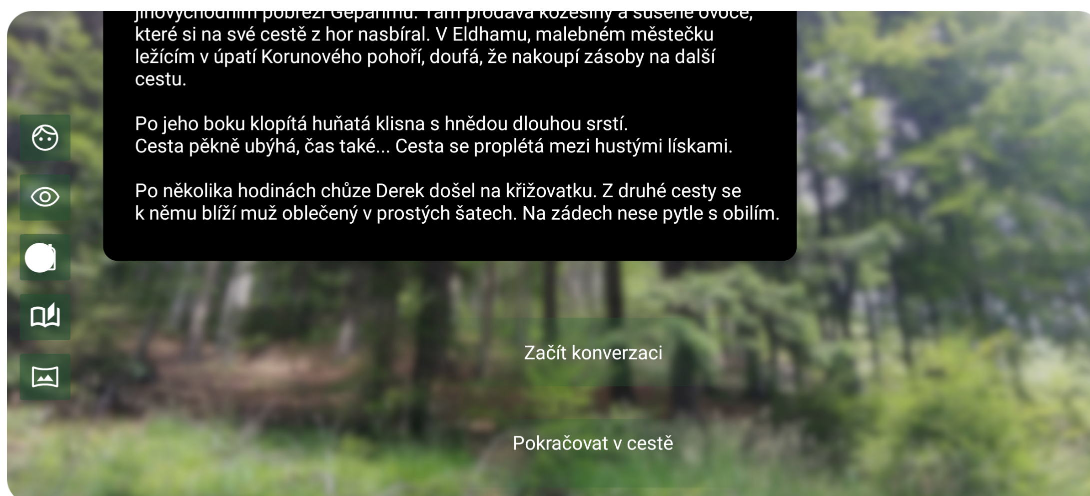
Obrázek 8.2: Počáteční obrazovka hry

Na levé straně okna se nachází sloupec s tlačítky, mezi kterými lze překlikávat. Horní tlačítko značí přechod na okno s informacemi hráče, což je počáteční okno aplikace. Druhé shora zobrazuje seznam všech objektů aktuálně viditelných. Třetí tlačítko zobrazuje seznam předmětů v inventáři hráče. Následující tlačítko je pro nás v tuto chvíli nejdůežitější. Viz obrázek 8.3.



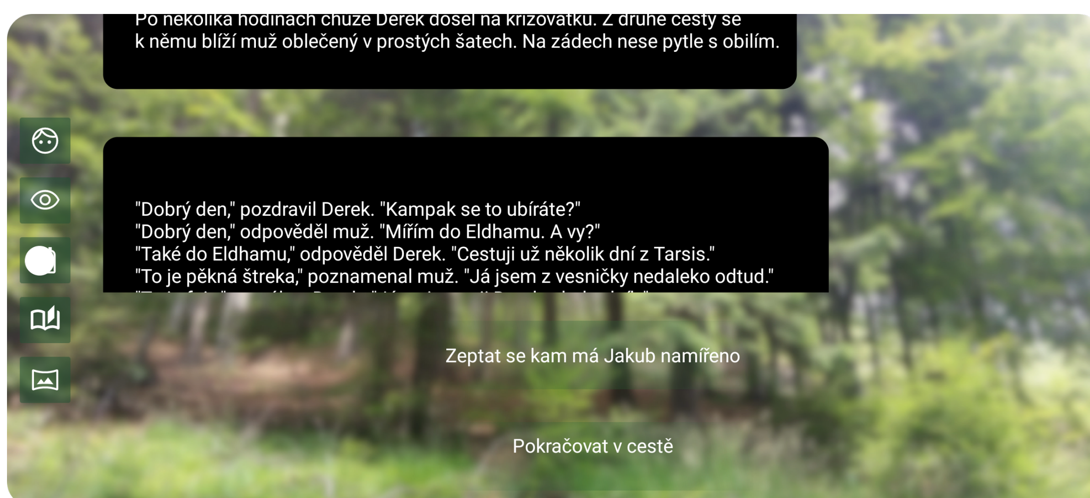
Obrázek 8.3: Okno s dosavadním vývojem příběhu

Zde lze nalézt dosavadní vývoj příběhu hráče ve formě seznamu textových zpráv. Navíc po scrollování dospod, lze nalézt tlačítka pro posun v příběhu, viz obrázek 8.4. Nebo mohou být viditelná na levé straně vedle tlačítek pro přechod na jiné okno.



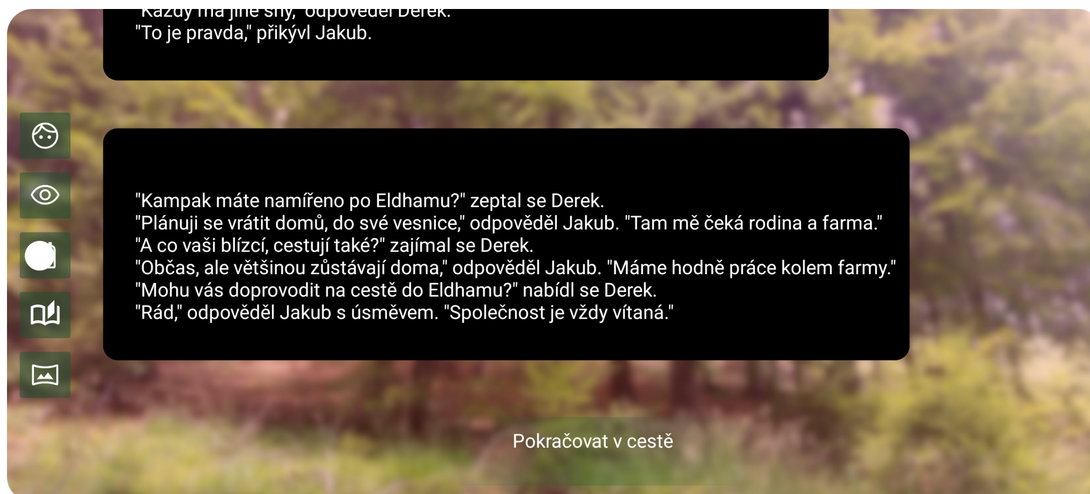
Obrázek 8.4: Okno s možnostmi posunu v příběhu

Pokud uživatel jedno z těchoto tlačítek zvolí, například *Začít konverzaci*, Příběh by se měl posunout a zobrazit nové možnosti. Viz obrázek 8.5.



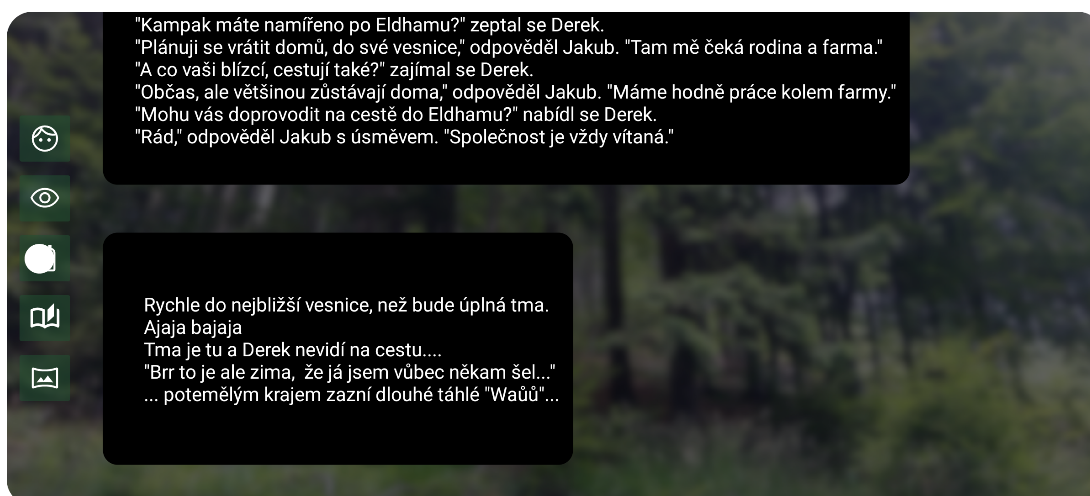
Obrázek 8.5: Zahájení konverzace

Po přejití do nového stavu zmáčknutím tlačítka *Zeptat se kam má Jakub namířeno*, dojde opět k posunu v příběhu, ale také, protože rozhovor trvá už nějakou dobu, dochází k západu slunce. Viz obrázek 8.6.



Obrázek 8.6: Zeptat se kam má Jakub namířeno

Následně po pokračování v cestě příběh končí. Viz obrázek 8.7.



Obrázek 8.7: Konec hry

Poslední zpráva indikuje, že hlavní postavu možná sežrali vlci.

Pokud by ale hráč nekonverzoval tak dlouho, mohla jeho postava stihnout dojít do vesnice před setměním.

9. Základní popis herního enginu

Samotný herní engine je schopen věcí, které pro naše účely nejsou důležité. Shrneme zde tedy pouze ty základní informace, které jsou následně využity. Jak již bylo zmíněno, uživatel hraje za určitou postavu, která se nachází v prostoru reprezentovaném tabulkou míst. Hráč se může pohybovat z místa na místo. Každé je reprezentováno obrázkem na pozadí a textovým popisem.

Na každém místě se může nacházet množina předmětů, množina postav. Každá postava má svůj inventář předmětů a lokaci na mapě, na které se nachází.

Chování postavy

Uživatel by měl mít možnost ovládat postavu pomocí udávání příkazů, co má postava udělat. Takovéto rozhodnutí se nazývá *Behaviour*. Každé behaviour má definovanou množinu *Behaviour requirement* a pokud chce postava provést nějakou akci, musí splnit všechny požadavky. Všechny typy požadavků, které aktuálně postava splňuje, má uložené, a nemusí se tedy znovu zjišťovat. Například, chce-li postava provést akci *najíst se*, musí splňovat požadavek vlastnění alespoň jednoho předmětu typu jídlo. Takového požadavku postava nabývá, když se předmět typu jídlo dostane do jejího inventáře.

Vytváření her

Program umožňuje vytvářet hry pro více hráčů. Pro jednoduchost byl Server omezen na jednu hru v jeden moment pro jednoho hráče. Po odpojení se klienta zavřením aplikace by se server měl automaticky připravit na nového hráče ale se stejným příběhem. Pro výběr jiného příběhu je třeba server restartovat.

9.1 Příběh

Data příběhu vytvořená v našem programu *WorldsFactory* se propojují s herním engimem pomocí Java knihovny.

K hernímu enginu byly přidány třídy *Story*, které zastřešují příběh popisovaný pomocí takzvaných *StoryMessages*. Každá odpovídá defakto jednomu uzlu v grafu příběhu jako gamebooku. Je tedy složena ze zprávy a množiny *StoryOptions*, mezi kterými může hráč vybírat. Story objekt si pak také pamatuje aktuální *StoryMessage*.

Obdobně musely být vytvořeny třídy v softwaru *WorldsFactory*. Zde, pomocí eventů, může uživatel vytvářet příběh pomocí eventů, při kterých může nastavit aktuální *StoryMessage*.

10. Implementace příběhu do herního enginu

Bylo zapotřebí provést hned několik věcí. Nejprve bylo nutné přidat WorldsFactory knihovnu do projektu serverové aplikace, dále také inicializovat knihovnu a následně propojit třídy s objekty příběhu. Dále bylo zapotřebí vylepšit mobilní aplikaci, aby bylo možné sledovat vývoj příběhu a posouvat se v něm. A nakonec jsme museli rozšířit komunikace mezi serverem a klientem o nové typy zpráv. Konkrétně ze serveru na klienta jsme přidali zprávy pro získání nového textu příběhu a možnosti, které má hráč k dispozici. Naopak z klienta na server byly přidány zprávy pro vybrání možnosti posunu v příběhu.

10.1 Přidání WorldsFactoryJavaLibrary knihovny do projektu

Nejprve na straně Serveru jsme inicializovali samotnou knihovnu *WorldsFactory*. To lze, v případě použití nástroje Maven, přidáním následujícího kódu do souboru *pom.xml*.

```
<dependency>
    <groupId>com.belafon.worldsfactory</groupId>
    <artifactId>worldsfactory_java_library</artifactId>
    <version>1.0-SNAPSHOT</version>
</dependency>
```

Následně lze jar soubor knihovny nainstalovat pomocí příkazu, kde je potřeba nahradit *pathToJarFile.jar* cestou k jar souboru.

```
mvn install:install-file \
    -Dfile="./pathToJarFile.jar" \
    -DgroupId="com.belafon.worldsfactory" \
    -DartifactId="worldsfactory_java_library" \
    -Dversion="1.0-SNAPSHOT" \
    -Dpackaging=jar
```

10.2 Inicializace WorldsFactoryJavaLibrary knihovny

Ve třídě *World* se během incializace objektu vytvoří nový příběh následujícím způsobem:

```
var story = new StoryInitializer()
    .withStoryName("MainStory")
    .withPathToStoryData(path)
    .withDebugMode(true)
    .withEventGraphCondition(EventGraphCondition.
       MOVE_MAX_BY_ONE)
    .withCheckingConditionsAfterEachSet(true)
    .build();

```

```
story.join().tryToMoveInEventGraph();
this.worldsFactoryStory = story.join();
```

Naopak na konci hry se provede *worldsFactoryStory.end();*.

Všimněme si, že pro jednoduchost byl story ve *WorldsFactoryStoriesManager* nastaven jako defaultní příběh. V dalším kódu pak není třeba identifikovat příběh podle jména. To má ale za následek, že nemohou běžet dva příběhy paralelně v jeden moment. Pokud by se tedy uživatel pokusil připojit k serveru ve chvíli, kdy už nějaký příběh běží, server ho nepřipojí.

10.3 Svazování tříd s příběhem

Opět se toto týká pouze serverové části. Zde jsme vytvořili zcela nový balíček *world/story* s objekty objekty *Story* a *StoryMessage*, podobně, jako je to v příběhu.

Dále jsme upravili třídu *Time* o anotace a změnili setter pro nastavení aktuaální části dne.

Poté jsme vytvořili třídu *StoryMapGenerator*, která získá pro danou pozici její jméno v příběhu a vytvoří nový objekt *Place*.

```
public Place[][] generateMap(int sizeX, int sizeY, Map map) {
    Place[][] mapPlace = new Place[sizeX][sizeY];
    for (int i = 0; i < sizeX; i++)
        for (int j = 0; j < sizeY; j++){
            var placePath = new StringBuilder("places[" + i + "][" + j + "]");

            // ...

            // get name of the object in the story by its position
            // in the story map
            var storyPlaceObjectNamePath = placePath + ".__object_name__";
            String storyPlaceObjectName = (String)WorldsFactoryStoryManager.getProperty(
                storyPlaceObjectNamePath.toString(), "map");

            mapPlace[i][j] = new Place(map, i, j, 0, typePlace, storyPlaceObjectName);
        }
}
```

Třídu *Place* jsme pak obohatili o příslušné anotace a svázali s odpovídajícím objektem v příběhu.

Podobně byly oanotovány třídy *Creature* a *Item*. Načítání objektů z příběhu se provádí ve *World* třídě při jeho spuštění. Využívá se při tom speciální metoda *WorldsFactoryStory.getAllObjectNamesOfType*, která vrátí seznam jmen všech objektů příběhu daného typu.

10.4 Vylepšení mobilního klienta

Pro posouvání v příběhu a sledování vývoje příběhu byl na straně klienta vytvořen nový fragment *StoryFragment*. Ten obsahuje tlačítka a kontejner pro zobrazování textu ve formě zpráv. Rozvržení komponent fragmentu lze nalézt v souboru *fragment_story.xml*.

Byl vytvořen nový balíček *game/story* obsahující třídy *Story*, *StoryMessage* a *StoryOption*.

Následně bylo třeba přidat zpracování zpráv ze strany serveru a přidání možnosti odesílat zprávy o vybrání konkrétní možnosti posunu v příběhu. Jednalo se o drobé úpravy ve třídách *ChatListener* a *SendMessage*.

10.5 Generování dokumentace

Jak k serveru tak ke klientovi lze vygenerovat dokumentaci zdrojového kódu pomocí *Doxygen*.

```
cd ./bc_thesis_tichavsky/GameEngine/Server/world/doc
doxygen Doxyfile
```

```
cd ./bc_thesis_tichavsky/GameEngine/MobileClient/doc
doxygen Doxyfile
```

Vygenerovaná data se umístí do stejné složky jako se nachází soubor *Doxyfile*.

Závěr

Práce prezentuje způsob, jak elegantně popisovat nerozvětvené příběhy pomocí časové osy a událostí. Události je možné přesněji specifikovat pomocí podudálostí, což umožňuje spisovateli popisovat části světa na různých úrovních detailu. Tak také může tvůrce postupovat při jejich vymýšlení.

Práce také dále nalezla způsob, jak popisovat rozvětvené příběhy. Nejprve hledala velmi obecný způsob popisu, který nebyl příliš praktický z pohledu tvůrce příběhu. Následně byl tento způsob upravován a vylepšován, až do sofistikované podoby.

Základním způsobem popisu byla množina konceptů s počátečním stavem jejich vlastností a množinu událostí, kde se každá skládala z podmínkové funkce a metody události. Následně byl tento způsob rozšířen o graf událostí, který popisoval, která událost mohla nastat po které.

Byl zaveden také takzvaný „lineární event“, pro popis nerozvětveného příběhu v rozvětveném.

Dále se práce zabývala architekturou konkrétního softwarového díla. Následně jsme se zaměřili na konkrétní implementaci s pracovním názvem „WorldsFactory“, společně s implementací externí knihovny pro herní enginy, které umožňují využití exportovaných dat z programu.

Tímto vším jsme si v této práci prošli. Od teoretických základů, přes návrh, až po konkrétní implementaci základní verze programu.

Oddělování příběhu a herního enginu

Program pro vytváření příběhu, jako je WorldsFactory umožňuje popisovat svět pomocí programovacího jazyka. To v mnoha věcech spisovateli pomáhá ale zároveň dává velikou moc. Uživatel softwaru by si měl uvědomit, že ne vše je nutné programovat v tomto programu.

Vezměme si příklad, kdy uživatel vytváří příběh pro hru, ve které se uživatel proplétá labyrintem. V tomto případě dává smysl vytvořit mapu a políčka v programu WorldsFactory. Čemu by se ale tvůrce měl vyvarovat je tvorba jednotlivých přechodů mezi políčky pomocí eventů. V tomto případě je to ještě relativně na hraně a asi by to v některých situacích bylo obhajitelné. Co ale nechceme je, aby uživatel vytvářel něco jako šachy hned vedle příběhu. Program je toho schopný, ale nemá k tomu sloužit. V případě, že by obsahoval také opravdový příběh vedle mechaniky nějaké hry, bylo by to matoucí.

Jednoduše řečeno, uživatel by se neměl snažit vytvářet herní engine v programu WorldsFactory.

Kam dál

Práce obsahuje pouze velmi základní verzi programu, která zatím bohužel není pro uživatele příliš praktická. Práce si stanovila minimální produkt z velikého návrhu. Směr pokračování je tedy očividný. Hlavní je přidat podporu lineárních eventů a zobrazování stavu světa. To jsou následující primární cíle.

Seznam použité literatury

[1] Kolektiv. Plottr. Software, 2024. [online]. Dostupné z: https://plottr.com/ [Citováno: 2024-02-04].

[2] Kolektiv. World Anvil. Software, 2024. [online]. Dostupné z: https://www.worldanvil.com/ [Citováno: 2024-02-04].

[3] Ch. Klimas. Twine. Software, 2024. [online]. Dostupné z: https://twinery.org/ [Citováno: 2024-02-04].

[4] J. Green. *Oz - gamebook*. Mytago, UK, 2020. ISBN: 978-80-87761-50-2.

[5] L. Torvalds. Git. Software, 2024. [online]. Dostupné z: https://git-scm.com/ [Citováno: 2024-02-04].

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, United States, 1994. ISBN: 0-201-63361-2.

[7] Kolektiv. Newtonsoft.Json. Software, 2024. [online]. Dostupné z: https://www.newtonsoft.com/json [Citováno: 2024-02-04].

[8] Northwoods Software Corporation. Godiagram. Software, 2024. [online]. Dostupné z: https://godiagram.com/avalonia/latest/index.html [Citováno: 2024-02-04].

[9] Kolektiv. AvaloniaGraphControl. Software, 2024. [online]. Dostupné z: https://github.com/Oaz/AvaloniaGraphControl [Citováno: 2024-03-27].

[10] T. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991. S2CID: 31468174.

[11] Kolektiv. Jython. Software, 2024. [online]. Dostupné z: https://www.jython.org/ [Citováno: 2024-02-04].

[12] Kolektiv. Project Lombok. Software, 2024. [online]. Dostupné z: https://projectlombok.org/ [Citováno: 2024-02-04].

[13] J. Novak. *Game Development Essentials: An Introduction*. Delmar Cengage Learning, USA, 2011. ISBN: 1111307652.

[14] A. Thorn. *Game Development Principles*. Cengage Learning PTR, USA, 2013. ISBN: 978-1285427058.

[15] E. Adams. *Fundamentals of Game Design*. Pearson Education, USA, 2013. ISBN: 9780321929679.

[16] M. Menard. *Game Development with Unity*. Cengage Learning PTR, USA, 2011. ISBN: 9781435456587.

[17] J. S. Cho. *The Beginner's Guide to Android Game Development*. Glasnevin Publishing, Irsko, 2014. ISBN: 978-1908689269.

Seznam obrázků

1 Rozvětvený a nerozvětvený příběh 7

2.1 Úryvek z gamebooku *Oz - gamebook* od Jonathana Greena[4] 12

5.1 Okno pro výběr projektu 28
5.2 Dialogové okno pro výběr souboru projektu 29
5.3 Okno pro vytvoření nového projektu 30
5.4 Okno po otevření nového projektu 31
5.5 Navigační panel pro vytvoření nového konceptu 32
5.6 Formulář pro vytvoření nové třídy 32
5.7 Project overview s vytvořenou třídou 33
5.8 Vytvoření nové property 34
5.9 Formulář pro vytvoření nové metody 35
5.10 Příklad vyplněného formuláře pro vytvoření nové metody 35
5.11 Podrobosti o metodě 36
5.12 Podrobnosti o události 37
5.13 Nastavení sekvence událostí 38
5.14 Zobrazení vztahů mezi eventy 39
5.15 Vytvoření event containeru 40
5.16 Export dat 41

6.1 C4 model - Softwarový Systém 44
6.2 C4 model - WorldsFactory Container 45
6.3 C4 model - Simplified WorldsFactory Container 46
6.4 Vizualizace jednoduchého grafu v AvaloniaGraphs 53
6.5 Vizualizace grafu s vnořeným grafem v AvaloniaGraphs 55
6.6 Vizualizace grafu s vnořeným grafem v AvaloniaGraphs, po oddálení . . 55

8.1 Formulář pro navázání spojení 63
8.2 Počáteční obrazovka hry 64
8.3 Okno s dosavadním vývojem příběhu 64
8.4 Okno s možnostmi posunu v příběhu 65
8.5 Zahájení konverzace 65
8.6 Zeptat se kam má Jakub namířeno 66
8.7 Konec hry 66