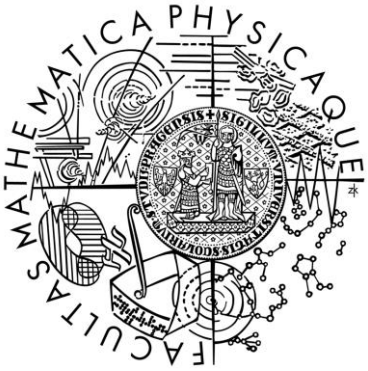
Univerzita Karlova v Praze

Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Martin Mašek

Systém pro zjednodušení tvorby umělé inteligence ve hrách

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek Ph.D.

Studijní program: Informatika

Studijní obor: IOI

Praha 2013

Děkuju panu Mgr. Pavlu Ježkovi Ph.D. za vedení této práce a čas, který mi věnoval.

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů, literatury a dalších odborných zdrojů.

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 v Praze má právo na uzavření licenční smlouvy o užití této práce jako školního díla podle § 60 odst. 1 autorského zákona.

V Praze dne 12. 5. 2013 Martin Mašek

Název práce: Systém pro zjednodušení tvorby umělé inteligence ve hrách

Autor: Martin Mašek

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek Ph.D., Katedra distribuovaných a spolehlivých systémů

Abstrakt:

Na tvorbě umělé inteligence (artificial intelligence, dále AI) do počítačových her se v dnešní době běžně podílí lidé, kteří jsou sice specialisty ve svém oboru, ale nejsou programátory. Proto bylo nutné vyvinout nástroj pro tvorbu AI snadno použitelný i bez znalosti programování. Pro tvorbu umělé inteligence se ukázal velice vhodný koncept Behavior trees. Na základě konceptu byl jako součást bakalářské práce vytvořen framework v jazyce C# pro platformu .NET nazvaný Revive, který umožňuje tvorbu umělé inteligence užitím grafického editoru, je snadno implementovatelný ve hrách a poskytuje možnost sledovat průběh rozhodování zapnutím ladícího režimu. Důraz byl kladen především na jednoduchost implementace do hry. Revive lze reálně použít pro tvorbu AI do her. Demonstrace jednoduché simulace, užitím Revive frameworku, je přiložena k této práci.

Klíčová slova: Umělá inteligence ve hrách, behavior trees, editor

Title: Software System Simplifying AI Development in Computer Games

Author: Martin Mašek

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Pavel Ježek Ph.D.

Abstract: Artificial intelligence (AI) in games is developed by programmers and AI designers. Designers are experts on AI but they are not programmers. Because of that there is a need to create a tool which non-programmers can use in an easy way. The Behavior trees have proven as a very good concept for creating an AI. The concept of Behavior trees enabled the development of Revive framework which is part of this work. The Framework is developed in C# for .NET platform. Revive framework contains a graphical editor which enables creating of the Behavior trees. It has also been designed with a strong focus on ease of implementation and therefore it can be easily implemented in the games. The framework also provides an option to oversee the decision process of Behavior tree. Revive can be used by small games and demonstration of Revive usage is part of this work.

Keywords: Artificial intelligence in games, behavior trees, editor

Obsah

Obsah 1
1. Úvod / Předmluva 5
1.1. Cíle 6
1.2. Struktura 6
2. Použitá technologie 7
2.1. Platforma a operační systém 7
2.2. Použité .NET technologie 7
2.2.1. Komunikace mezi Runtime částí a editorem 7
3. Analýza 8
3.1. Proč potřebujeme nástroj 8
3.2. Možné přístupy tvorby umělé inteligence ve hrách 9
3.2.1. Skripty 9
3.2.2. Konečné automaty 10
3.2.3. Splňování cílů 12
3.2.4. Ostatní 13
3.3. Behavior trees 13
3.3.1. Z čeho se skládají 13
3.3.2. Typy vnitřních vrcholů 14
3.4. Co by měl editor mít 16
3.5. Základní části frameworku 16
3.6. Workflow 18
3.7. Parsování souboru s entitami a akcemi 19
3.8. Oddělení závislosti na definujícím souboru 19
3.9. Výstup dat z editoru 20
3.10. Dodatečné informace entit a akcí 21
3.10.1. Jak definovat dodatečné informace 21
3.10.2. Předání parametrů vrcholům 21
3.10.3. Přiřazení akcí entitám 22
3.11. Komunikace mezi vrcholy 22
4. Vývojová dokumentace A –knihovna a funkcionalita 24

4.1. Knihovna Revive.dll ... 25
4.1.1. Atributy ... 26
4.1.2. Vrcholy stromu ... 26
4.1.3. Runtime ... 30
4.2. Parsování definující assembly ... 31
4.3. Znovu použití existujících stromů ... 32
4.3.1. InheritenceTree ... 33
4.3.2. LookUp vrchol ... 35
4.4. Podpora ladění ... 35
4.4.1. Nutnost dvou knihoven ... 36
4.4.2. Definice kontraktu komunikace ... 37
4.4.3. Jak identifikovat vrcholy ... 38
4.4.4. RepresentantRegister ... 39
4.4.5. Centrální tvorba vrcholů ... 40
4.4.6. Runtime při čekání na potvrzení z editoru ... 41
5. Vývojová dokumentace B – editor a synchronizace ... 43
5.1. MVVM ... 43
5.2. Model ... 44
5.3. ViewModel ... 47
5.3.1. Zobrazení parametrů v editoru ... 48
5.3.2. RepresentantViewModel ... 49
5.3.3. Akce ve ViewModelu ... 51
5.3.4. ViewModel pro editační plochu a menu s akcemi ... 52
5.3.5. Dědičnost a Named tree ... 55
5.3.6. ApplicationViewModel ... 56
5.3.7. Tvorba dialogů ... 57
5.3.8. Jak funguje server. ... 58
5.4. View ... 59
5.4.1. Ostatní okna ... 60
5.4.2. Dialogy ... 60
5.4.3. Hlavní okno ... 61
5.4.4. Editační plocha ... 61
5.5. Synchronizace ... 64

5.5.1. Původní návrh ... 64
5.5.2. Synchronizace dvou modelů ... 64
5.5.3. Synchronizační wizard ... 65
5.5.4. Synchronizace jednotlivých částí ... 65
5.5.5. ViewModel pro synchronizaci ... 68
5.6. Ukládání a načítání aplikace ... 68
6. Práce s editorem ... 69
6.1. Start ... 69
6.1.1. Vytvoření nového projektu ... 69
6.1.2. Načtení existujícího projektu ... 69
6.2. Možnosti menu a horních tlačítek ... 69
6.2.1. Okno s entitami ... 69
6.2.2. Strom dědičnosti ... 69
6.2.3. Zarovnání zobrazeného stromu ... 70
6.2.4. Synchronizace ... 70
6.2.5. Publikování práce ... 70
6.2.6. Ukládání ... 70
6.2.7. Ladění Behavior trees ... 71
6.3. Tvorba Behavior tree ... 71
6.3.1. Editační plocha ... 72
6.3.2. Vytvoření vrcholu ... 72
6.3.3. Typy vrcholů ... 72
6.3.4. Práce se stromem ... 73
7. Práce s knihovnou Revive.dll ... 74
7.1. Co musí splňovat entity ... 74
7.2. Jak vytvořit akce entit ... 75
7.2.1. Tvorba akce ... 75
7.2.2. Použití atributů ... 77
7.2.3. Parametry pro akce ... 78
7.2.4. Komunikace akcí mezi sebou ... 80
7.2.5. Předání vlastních objektů akcím ... 80
7.3. Použití ve hře v Update smyčce ... 81
7.4. Ladění ... 81

8. Obohacujeme Revive framework 82
8.1. Tvorba nového parametru pro akce 82
8.1.1. Nutné body pro implementaci 82
8.1.2. Příklad implementace 82
8.2. Tvorba nového vrcholu pro framework 83
8.2.1. Nutné body pro implementaci 84
9. Srovnání s existujícími nástroji 85
10. Závěr 88
10.1. Zhodnocení práce 88
10.2. Budoucí práce 88
11. Zdroje 90
12. Seznam použitých termínů 91
13. Přílohy 92

1. Úvod / Předmluva

Počítačové hry jsou relativně nový druh zábavy, který v dnešní době není okrajová záležitost skupiny náctiletých. V dobách Pacmana a Tetrisu vypadala herní scéna zcela jinak. Hry se tvořily v malých týmech, či dokonce v jednotlivcích. Vývojář často zastupoval grafika, programátora a designéra v jednom. Se zvyšujícím se výpočetním výkonem vzrůstaly možnosti počítačových her. Tvorba her začala vyžadovat více lidí specializovaných na konkrétní části hry (design, grafika, tvorba umělé inteligence, tvorba uživatelského rozhraní apod.). Dnešní hry jsou technologicky velmi pokročilé, náročné a rozsáhlé projekty, na kterých se podílejí desítky a v některých případech až stovky specialistů. Herní průmysl se rapidně rozrostl na trh o ročním obratu v miliardách dolarů (pro představu obrat jedné z největších herních společností Activision Blizzard, Inc. v roce 2012 činil $4,8 mld.[1]) a tvoří tak podstatnou část zábavního průmyslu.

Důležitou částí hry je umělá inteligence. Hlavním účelem umělé inteligence[2] je oživení hry. Hra samotná může být po vizuální stránce velice pěkná s velmi realistickou fyzikou. Herní objekty (různá monstra, postavy, zvířata apod.) poskytují svou specifickou funkcionalitu. Typicky jde přehrávat nějaké animace či přemístit objekt na jiné místo. Pokud se ale herní objekty nebudou hýbat a reagovat na podněty, nedostaneme zábavnou hru. Toto rozhýbání má na starost právě AI. Dále obstarává spuštění určitých událostí ve vhodnou chvíli (například video ukázka po dokončení důležité části hry) a realistické chování herních objektů. Tyto reagující objekty můžeme nazvat *entitami* (tedy například pes, robot, past jsou entity a zeď či voda nejsou entity). Entity jsou schopny vykonávat určité činnosti, které nazýváme *akce*. Například pes je schopen kousnout, pohnout se na jinou pozici, štěkat apod. AI používá akce entit pro navození dobrého hráčského dojmu. Tento dojem se snaží být často obrazem skutečnosti nebo splnit očekávání hráče. Nikdo neví jak by se skutečný drak měl chovat, ale hráč očekává, že bude chrlit oheň a půjde těžko zabít (toto je jen ilustrační příklad lišící se v závislosti na prostředí hry).

Očekávání hráče může být různé a vytvořit reálně vypadající entity je problém. Hlavním účelem je dát iluzi realističnosti a poskytnout zábavu a přiměřeně těžkého soupeře. Vytvoření takovéto iluze je designový problém. Například počítačem řízený protivník, který si stoupne před hráče a nedělá nic, není dobrý design, stejně tak jako protivník, který zneškodní hráče při prvním pohledu. Proto se tým zabývající tvorbou AI skládá z programátorů a designérů. Cílem programátorů je implementovat design navržený odborníky na design. Aby toho mohlo být dosaženo, je třeba spolupráce,

---

[1] *zdroj:* oficiální finanční výsledky na http://investor.activision.com/results.cfm

[2] *artificial intelligence*, zkráceně *AI*, budeme v textu používat tuto zkratku

kdy se tým odborníků zabývá, jak se mají entity chovat a programátoři jak mají toto chování implementovat. Pro efektivní práci je nutné vytvoření nástroje, který umožní zefektivnit práci obou skupin a minimalizuje nutnost komunikace mezi nimi. Tím pádem se každá skupina zabývá čistě svým zaměřením.

Proto je cílem této práce vytvoření frameworku, nazvaného Revive, pro tvorbu AI, který bude odpovídat požadavkům lidí zabývajících se tvorbou AI do her. Tento framework nazveme Revive. Teď, když máme hrubou představu, proč je framework nutný a co bude jeho účelem, si můžeme vytyčit cíle.

1.1. Cíle

Toto jsou cíle, které by měl framework splňovat.

C1. Lehce implementovatelný do her tak, aby programátor implementující framework nemusel trávit čas složitým nastavováním a programováním.

C2. Dávat velkou volnost na úrovní kódu. Programátoři tvořící akce by neměli být nijak omezeni z hlediska použití vlastních proměnných, datových struktur a užití jiných konstruktů programovacího jazyka hry.

C3. Být založen na technice tvorby AI, která je populárně užita v praxi a tím pádem tvorbu AI odborník rychle pochopí.

C4. Tvorba AI by měla být pro designéra pohodlná a lehce naučitelná. Neměla by bránit lehce používat koncept z bodu 3 a měla by být jasně zobrazitelná.

C5. Framework musí mít možnost sledování průběhu výpočtu. Toto je důležitý ladící nástroj v případech, kdy se entity nechovají podle plánu.

C6. Je nutná možnost synchronizace. Pokud jsou změněné akce či entity, musí být možné tyto změny promítnout do již vytvořené práce bez nutnosti smazání veškeré práce.

C7. Možnost uložení a nahrání rozpracované práce.

1.2. Struktura

Následující **druhá kapitola** obsahuje informace o použité technologii a její zdůvodnění. Ve **třetí kapitole Analýza** probereme důkladně problémy tvorby frameworku a jejich řešení. **Čtvrtá kapitola** poskytuje první část na detailní pohled na technické detaily frameworku a následuje ji **pátá** kapitola s druhou částí o technické implementaci. **Kapitola šest** podává podrobný návod o ovládání editoru, zatímco **kapitola sedm** je návodem na implementaci frameworku v uživatelském kódu. **Osmá kapitola** popisuje jak rozšířit stávající framework (a) o nové vrcholy pro Behavior tree (b) o nové parametry pro vrcholy. K**apitola devátá** se zabývá srovnáním s existujícími editory. Předposlední, **desátá kapitola**, slouží jako zhodnocení výsledku práce. Pak následují kapitoly se zdroji, seznamem použitých termínu a přílohou popisující praktické použití Revive frameworku.

2. Použitá technologie

Aplikace je postavena na technologii .NET a jazyku C#, jelikož má autor s touto platformou zkušenosti. Testována je na operačním systému Windows 8. V době vzniku aplikace byl na trhu jen .NET Framework 4.0, proto je vývoj postaven na této verzi.

2.1. Platforma a operační systém

Platforma: Microsoft .NET Framework 4.0

Jazyk: C#

Operační systém: operační systém podporující .NET Framework 4.0 oficiálně:

- Windows XP
- Windows Vista
- Windows 7

2.2. Použité .NET technologie

Vizuální část frameworku (editor umělé inteligence) bude využívat technologii WPF (Windows Presentation Foundation). Tato technologie umožňuje značnou volnost v návrhu uživatelského rozhraní a dobře odděluje data a vizuální vzhled aplikace. Pro detailní popis WPF je vhodné přečíst knížku od Adama Nathana (1), kterou berou jako referenční knihu i zaměstnanci Microsoftu.

2.2.1. *Komunikace mezi Runtime částí a editorem*

Pro komunikace mezi dvěma procesy nabízí .NET různé druhy možností. Všechny jsou rozebrány na oficiálních stránkách Microsoftu MSDN (2). Jelikož jsou oba naše procesy na stejném stroji, připadají v úvahu tyto dvě možnosti:

- .NET Remoting
- WCF (Windows Communication Foundation)

Podle oficiálních zdrojů uvedených Microsoftem (3) by se .NET Remoting neměl nadále používat, přičemž jako alternativa je zde WCF. Pro případ komunikace mezi dvěma .NET procesy běžícími na stejném stroji má WCF speciální typ bindingu – NetNamedPipeBinding. Tento typ bindingu je rychlá alternativa k .NET Remotingu. Více informací o technologii WCF uvádí Microsoft na svých stránkách (4).

3. Analýza

Tato kapitola se zabývá analýzou tvorby frameworku s cíli definovanými v Úvodu. Nejdříve probereme možné přístupy tvorby AI a zjistíme, že nejvhodnější z nich splňující vytyčené cíle jsou Behavior trees. Dále pak probereme, jaké části by měl mít framework používající Behavior trees a jak vypadá schéma výpočtu takového frameworku. Na závěr probereme detaily ohledně použití Behavior trees.

3.1. Proč potřebujeme nástroj

Jak již bylo zmíněno v úvodu, na tvorbě her se zdaleka nepodílejí jen programátoři. Jako příklad vezměme tvorbu 3D grafiky. Pro tvorbu 3D modelů (pro příklad 3D modelu viz Obrázek 2) se užívají nástroje jako Blender (5) (Obrázek 1). Tyto nástroje jsou vytvořeny programátory a používány designéry. Designéři se naučí používat takovýto nástroj a samostatně tvořit modely. Tyto modely jsou pak užity ve hře.

Obrázek 2: 3D model supermana

(http://thefree3dmodels.com/stuff/characters/superman/14-1-0-5408)

Stejně tak jsou lidé specializovaní na tvorbu umělé inteligence a třeba vytvořit nástroj pro ně. Je nutné vytvoření rychle osvojitelného programu, ve kterém bude možno vytvářet AI stejně tak, jako je možno vytvářet 3D modely v Blenderu. Tento program by měl splňovat cíle z úvodu C3 a C4. Získáme několik výhod. (a) Jelikož už rozhodování entity není pevně dáno v kódu, lépe se provádí změny, neboť nemusíme upravovat zdrojové kódy, což by mohlo vyústit v chyby. (b) Zdrojové kódy netřeba znovu kompilovat v případě změn v AI.

Obrázek 1: Příklad modelování v programu Blender

(http://blog.nuclex-games.com/wp-content/uploads/pages/blender-rigging/imported-turret.png)

Samozřejmě, že je pořád nutné implementovat základní akce pro entity ve zdrojovém kódu, nicméně samotná logika rozhodování je už extrahována mimo. Přehledný a intuitivní editor zároveň umožní účast více lidí a tím pádem podpoří kreativitu. Díky cíli C3 z úvodu pomůže vizuální representace lépe prezentovat vytvořené chování a je daleko přehlednější pro případnou konzultaci a změnu. Velkým přínosem je také zvýšená produktivita, neboť tvorba nástrojem je rychlejší a je třeba méně konzultací mezi lidmi tvořící AI a programátory.

3.2. Možné přístupy tvorby umělé inteligence ve hrách

V této podkapitole se podíváme na způsoby tvorby umělé inteligence ve hrách. Analyzovat je budeme z pohledu designéra. Probrány jsou tři populární postupy – skripty, konečné automaty a koncept splňování cílů. Blíže jsou postupy popsány v knize Mata Bucklanda (6), programátora zabývajícím se AI přes 30 let, člena AI Interface Standards Committee (tvorba jednotných rozhraní pro AI a podpora jejich implementace).

3.2.1. *Skripty*

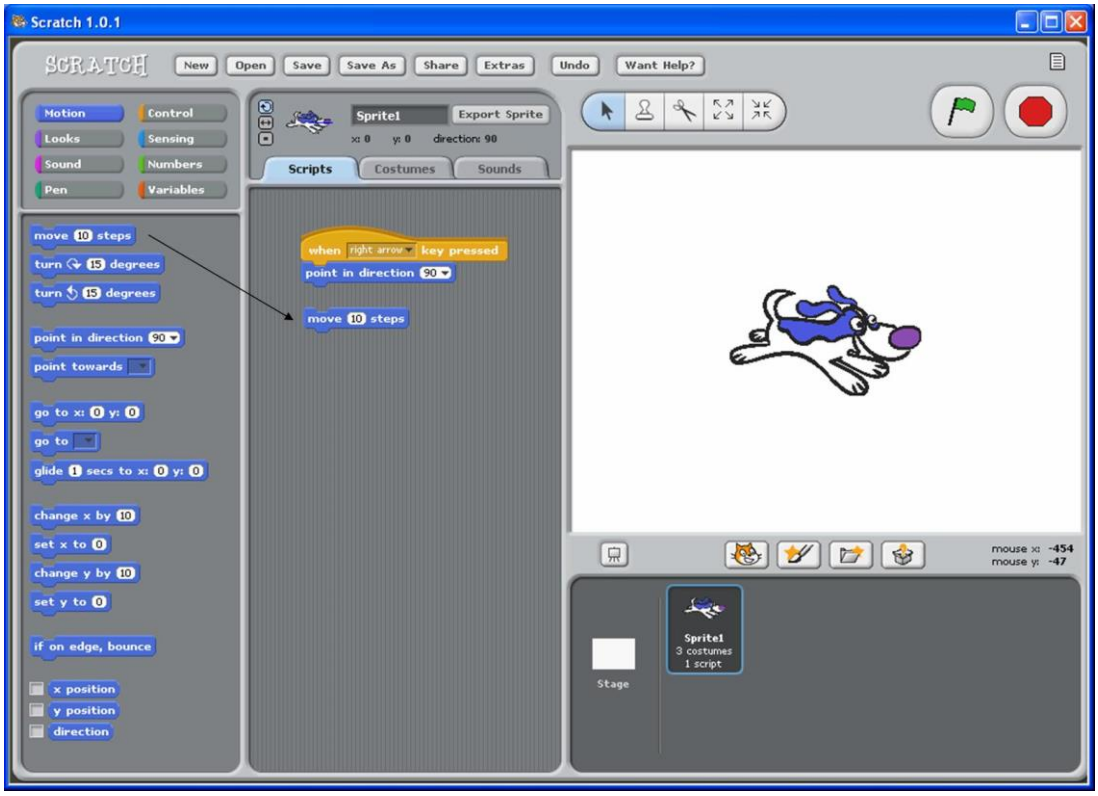
Hlavním rysem skriptovacího jazyka je, že se jedná o programovací jazyk. Skriptovací jazyky mají relativně jednoduchou syntaxi, sémantiku a jsou interpretovány za běhu. Hry používající skripty se většinou drží následujícího konceptu. V jazyku, ve kterém je napsána hra (např. C++) jsou zveřejněny metody, které je možno volat ze skriptu, který je vykonáván virtuálním strojem. Zároveň si lze předávat obousměrně data. Jelikož skriptovací jazyk často poskytuje vyšší abstrakci, je srozumitelnější a lehký na použití. V praxi používaný a velmi oblíbený jazyk je Lua. Lua je použita v mnoha AAA titulech např. Heroes 5, Crysis či World of Warcraft. Příklad skriptu v jazyku Lua je přiložen pod tímto odstavcem (převzato z knihy (6)). Účelem tohoto skriptu je vytvoření strážce, který zavede hráče do královské místnosti v případě, že má hráč k tomu povolení (komentáře jsou zvýrazněny šedě).

```
FUNCTION script_casttle_guard (player)
** create a guard situated at the castle's drawbridge
guard = Guard(GetPos(Drawbridge))

** focus and lock the camera on the guard
LockCamera(guard)

** move the guard toward the player
guard.Move(GetPos(player))
IF Player.Has(GetFlag(AUTHORIZATION_FROM_KING)) THEN
   guard.Speak("Good evening "+player.Name()+"His Majesty is expecting you. Come this way"
   guard.Move(GetPos(Throne_Room))
```

```
    player.Follow(guard)
END FUNCTION
```

Jak vidíme, ve skriptu jsou použity názvy funkcí (`LockCamera`, `Speak`, `Move`), ze kterých je patrný účel funkce. Příkazy jsou odděleny novou řádkou a netřeba nějaké složité syntaktické „omáčky“. Proto je skript relativně srozumitelný.

Jelikož skriptovací jazyk je programovací jazyk, velikou výhodou tohoto přístupu jsou vysoké možnosti tvorby AI. Další výhodou použití skriptů je oddělení herní inteligence mimo hlavní kód hry. Nicméně stále se jedná o programování, které není příliš intuitivní pro designéry a je zdlouhavější z hlediska učení. Dále je problém ve vizualizaci. Kód nelze přímočaře vizualizovat z principu toho, že se jedná o kód který píšeme. Šla by vytvořit například nadstavba, kdy bychom využívaly bloky kódu a skládali je do sebe. Příkladem takového přístupu je například Scratch (Obrázek 3) vyvinutý na MIT k výukovým účelům. Složitost k naučení se skriptování a nepřímočará možnost vizualizace jsou v rozporu s cílem C4.



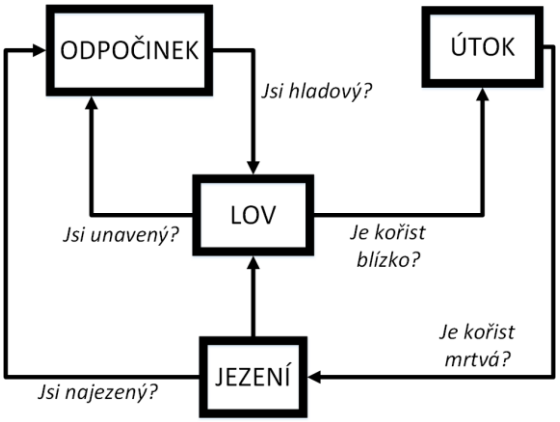
Obrázek 3: Program Scratch umožňuje skládání kusů kódů do bloků. *(http://scratch.mit.edu/)*

3.2.2. *Konečné automaty*

Konečné automaty jsou dnes stále používány, nicméně se ale více a více prosazují techniky založené na splňování cílů (následující podkapitola 3.2.3). Konečný automat se skládá z konečné množiny vrcholů zvaných stavy a hran mezi nimi. Hrany jsou ohodnoceny podmínkami. Automat se vždy nachází v nějakém stavu, který obsahuje logiku toho, co má daná entita, řídící se konečným automatem, dělat.

Po splnění podmínek na hranách se současný stav může změnit na příslušný následující stav definován hranou – automat přejde do nového stavu.

Vysvětleme si to na zjednodušeném příkladu. Mějme lva. Lvi většinou celý den odpočívají a když dostanou hlad, vydají se na lov. Pokud se jim úspěšně podaří ulovit kořist, sní ji a v případě nasycení jdou odpočívat. Lev se tedy může vyskytovat ve stavech odpočinek, LOV, ÚTOK, JEZENÍ. Mezi těmito stavy pak přechází na základě splnění podmínek. To co lev v danou chvíli dělá, závisí na aktivním stavu. Například pokud je ve stavu LOV musí zkoumat okolí a hledat kořist, a posléze se snažit připlížit a vyběhnout (viz Obrázek 4).

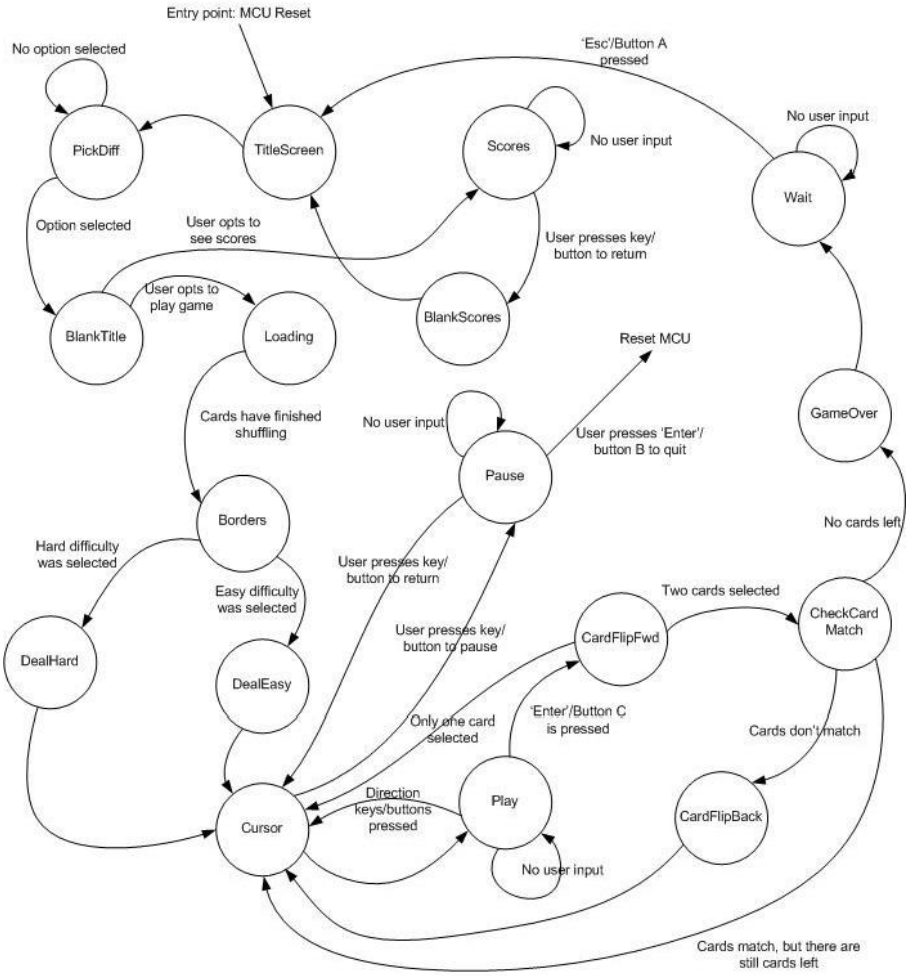


Obrázek 4: Příklad jednoduchého konečného automatu pro lva.

Tvorba AI pomocí konečných automatů má velikou výhodu, neboť po krátké demonstraci konečného automatu je téměř každý schopen pochopit hlavní princip. Proto může rychle začít s designem AI. Konečné automaty můžou být lehce implementovány v kódu, kdy vytvoříme základní třídu pro stav s jednotným rozhraním. Tato reprezentace jde pak lehce vizuálně vykreslit z principu toho, jak konečné automaty fungují. Koncept konečných automatů splňuje všechny naše podmínky stanovené v úvodu. Má ale pár nevýhod.

Komplexnější chování vyžaduje mnoho stavů a přechodů (příklad většího konečného automatu viz Obrázek 5). Tvorba takového automatu je náročná práce a je těžší otestovat, zda všechny průchody automatem dávají smysl a zda neuvízneme v nějakém stavu či cyklu.

Větším problémem je ovšem přidání nového stavu v průběhu vývoje hry a potřeba modifikovat automat. Podívejme se na výše zmíněný automat a představme si přidání dvou stavů a pár nových funkcí pro přechody. Začlenění těchto nových komponent vyžaduje důkladnou kontrolu, zda nám nevznikla smyčka, zda jsou všechny stavy dosažitelné, zda jsou průchody automatem stále smysluplné. Dále se automaty při větším množstvím stavů stávají velice složitými a nepřehlednými. Přehlednost jde z části vyřešit použitím hierarchických automatů, které jako stavy můžou obsahovat jiné konečné automaty. Dalším problémem je znovu použitelnost stavů pro jiné typy automatů. V takovém případě je třeba znovu upravit stav, který chceme znovu využít kvůli zasazení do jiného kontextu. Stav tak víceméně přepíšeme znovu. Jak uvidíme v podkapitole 3.3, implementace konceptu splňování cílů v podobě Behavior trees je výhodnější.



Obrázek 5: Příklad konečného automatu pro hraní karet

(http://people.ece.cornell.edu/land/courses/ece4760/FinalProjects/s2003/jc282/memory_website/images/msd.jpg)

3.2.3. *Splňování cílů*

Splňování cílů je metoda založena na principu uvažování lidí. Lidé uvažují tak, že vykonávají jeden konkrétní cíl, který je buď triviální (vyslovení věty, kopnutí do míče apod.), nebo se skládá z jiných cílů, které je třeba případně dál rozkládat (například postavit most). Představme si modelovou situaci, kdy člověk ráno vstává do práce. V hlavě se nám vytvoří seznam čtyř abstraktních cílů – obléci se, ranní hygiena, snídaně, cesta do práce. Nejdříve rozvíjíme první cíl – obléci se. Tento cíl se skládá z podcílů jít do skříně, vybrat oblečení, svléknout se, obléci oblečení. Cesta do skříně je pak sada atomických cílů v podobě jednotlivých kroků. Po splnění prvního cíle, oblečení se, si jdeme vyčistit zuby a tak dále.

Splňování cílů je obecný princip. Jeho konkrétní implementace může být například kupení cílů na zásobník, jak je v popsáno v již zmíněné knize (6). My použijeme implementaci s názvem Behavior trees, která bude rozebrána v následující podkapitole 3.3. Důvod proč vybrat Behavior trees dobře objasní tento komentář Alexe J. Champandarda, zakladatele blogu zabývajícího se AI ve hrách, člověka z praxe, který zná vývojáře zabývající se AI do AAA her. Jelikož je Alex J. Champandard zkušený v oblasti AI do počítačových her, dokonce se vyjadřuje k AI

na konferencích, můžeme brát jeho komentář k Behavior trees jako pravdivý. Na svém blogu se v článku (7), zabývajícího se retrospektivně jeho článkem o predikci trendů v AI do her, k Behavior trees se vyjádřil takto:

> *„Here at `AiGameDev.com`, I've been evangelizing behavior trees for over a year and a half now, and over the past few months I've been consulting with a handful of AAA studios about behavior trees and their implementations. But in a sense, I completely underestimated the influence of the blog itself as well as the strength of behavior trees in general. It's now safe to say that BTs have become the de facto standard in the games industry for anyone writing a new system from scratch."*

Vidíme tedy, že koncept Behavior trees splňuje přesně náš cíl C3.

3.2.4. *Ostatní*

Samozřejmě je zde více způsobů tvorby AI, ale jejich probrání je nad rámec této bakalářské práce. Výše zmíněné techniky sloužily jako inspirace při hledání optimální techniky použité editorem.

3.3. Behavior trees

Většina následujícího textu čerpá z článků o Behavior trees zveřejněných na stránkách komunity zabývající se tvorbou AI do her (8). Jako dobrý start poslouží úvodní článek (9) ze zmíněné komunitní stránky.

Behavior trees jsou implementací obecného principu splňování cílů (podkapitola 3.2.3). Jak je patrné z anglického názvu jedná se o stromy, které reprezentují chování určitého objektu. Kombinují výhody přehlednosti konečných automatů, přičemž jsou značně modulární a znovu použitelné. Zároveň poskytují stejnou moc jako programovací jazyk při implementaci vhodných vrcholů a práce s nimi je přirozená díky tomu, že se jedná o implementaci konceptu techniky splňování cílů.

3.3.1. *Z čeho se skládají*

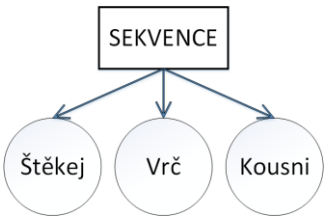
Strom je složený z vrcholů, kterým lze také říkat akce, či chování. My se budeme držet následující terminologie. *Akce* je vrchol stromu, který nemůže mít děti. Každá akce je listem ve stromě. Definice akce odpovídá naší definici v úvodu. *Vnitřní vrchol* bude vrchol, který může mít děti.

Akce

Tvoří konkrétní činnosti, které objekt (entita) může vykonávat a jsou typicky implementované v kódu pro každou herní entitu. Vezměme si jako příklad entity psa. Jeho akce jsou – štěkej, jdi, jez, pij, vrč, spi apod.

Vnitřní vrcholy

Tyto vrcholy jsou společné všem entitám a představují logiku provádění činností. Například můžeme mít vrchol, který představuje vykonání akcí v sekvenčním pořadí. Za příklad vezměme výše zmíněného psa a

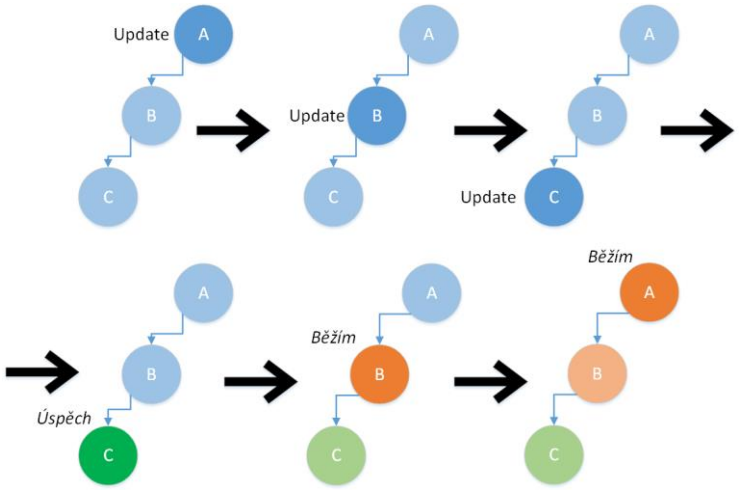


Obrázek 6: Příklad jednoduchého stromu.

vytvořme vrchol s dětmi – štěkej, vrč, kousej. Tento jednoduchý strom pak představuje chování psa, který jediné co dělá je zaštěkání následované zavrčením a nakonec kousne (znázorňuje Obrázek 6).

Každý vrchol je sebe popisující a poskytuje funkci (nazývejme ji `Update`), která po zavolání vykoná chování, které vrchol představuje a vrátí stav o tom, jak chování dopadlo. Použitím výše uvedeného příkladu zavolání `Update` na akci štěkej způsobí, že pes zaštěká a vrátí výsledek „úspěšně dokončeno“. Toto umožňuje modularitu a znovu použitelnost již vytvořených vrcholů a nezávislost vrcholů na sobě, neboť vrcholy pracují se stavy dětí a neví nic o funkcionalitě, která se tak může lehce změnit.

Dále uvažujme, že vrchol může vrátit tyto stavy „běžím“, „úspěch“ a „neúspěch“ (není to podmínkou a reálná implementace může mít stavů více). Demonstraci fungování Behavior tree popisuje Obrázek 7.



Obrázek 7: Příklad propagace Update volání a následné vrácení stavu z posledního vrcholu (každý vrchol vrací svůj stav na základě stavu obdrženého od dítěte). Stav "úspěch" je znázorněn zeleně, stav "běžím" je oranžový. Šipky znázorňují přechody mezi jednotlivými stavy.

3.3.2. *Typy vnitřních vrcholů*

Popíšeme si dva základní typy vnitřních vrcholů.

Dekoratér[1].

Tento typ vrcholů má zpravidla pouze jedno dítě a nějakým způsobem mění či obohacuje funkcionalitu tohoto dítěte. Použití uveďme na příkladu.

Máme akci štěkej. Chtěli bychom omezit počet možných štěkání (akce štěkej tuto funkcionalitu nenabízí), proto můžeme použít dekoratér, který si bude pamatovat počet úspěchů akce štěkání a v případě překročení limitu akci nespustí. Velmi užitečný dekoratér je negace, kdy po úspěchu našeho dítěte vrátíme neúspěch a naopak. Jako poslední příklad uveďme dekoratér, který zopakuje chování svého dítěte několikrát. Při každém úspěšném `Update` kroku svého dítěte vrátí stav „běžím“ a úspěch vrátí až při dosažení potřebného počtu opakování.

Druhým typem jsou **vrcholy s větším počtem dětí**. Sem se řadí dva důležité vrcholy nezbytné pro každý strom – sekvence a selektor.

[1] Anglicky *decorators.*

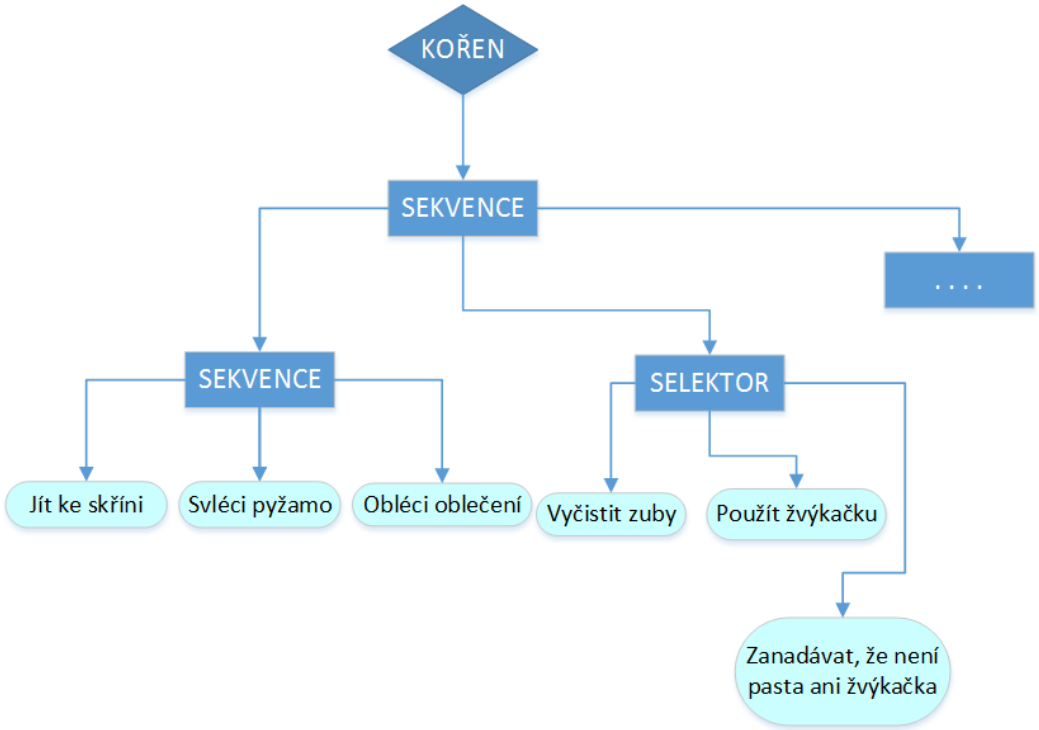
Sekvence

Propaguje `Update` volání svým potomkům v sekvenčním pořadí a v případě prvního neúspěchu selže taktéž. V každém `Update` volání zavolá sekvence `Update` na aktuálním dítěti. V případě úspěchu dítěte se aktuální dítě posune o jednu pozici a pokud se jednalo o poslední dítě sekvencér uspěje. Pokud dítě vrátí stav „běžím“ pozice se nemění a sekvence propaguje stav „běžím“ dále.

Selektor

Je velmi podobný sekvenci. Taktéž volá `Update` na potomcích v sekvenčním pořadí, nicméně selektor uspěje v případě úspěchu prvního dítěte. Pokud dítě vrátí stav „běžím“, selektor taktéž vrátí stav běžím a v případě neúspěchu dítěte jen posune pozici aktuálního dítěte dále (v případě, že se jednalo o poslední dítě selektor selže).

Pojďme si předvést, jak by mohla vypadat část stromu pro ranní vstávání a cestu do práce z kapitoly 3.2.3. Obrázek 8 znázorňuje takový strom.



Obrázek 8: Ukázka části Behavior tree pro cestu do práce

Selektor a sekvence jsou základem pro každý Behavior tree. Vnitřních vrcholů může být velké množství. Uveďme stručně dva vrcholy jako příklad. Paralelní vrchol volá `Update` na všech svých dětech v jednom `Update` volání. Takovýto vrchol se hodí na kontrolování podmínek pro běh stromu, kdy jeden potomek paralelního nodu je regulérní strom obstarávající chování a další potomci jsou akce, které slouží jako podmínky. V závislosti na politice paralelní vrchol například selže v případě nesplnění jedné podmínky. Druhým příkladem buď náhodný vrchol, který může své děti vykonávat v nějakém náhodném pořadí (výpočet stromu tak není deterministicky určen).

3.4. Co by měl editor mít

Pojďme si blíže rozebrat, co by měl editor využívající Behavior trees obsahovat.

Nejprve rozebereme tvorbu stromu v editoru. Editor by měl obsahovat lehce přístupné menu se všemi vrcholy. Jelikož těchto vrcholů může být potenciálně mnoho, je nutné vytvoření kategorií pro vrcholy. Samotná práce s vrcholy by měla probíhat na editační ploše, která umožňuje libovolné přemisťování vrcholu pomocí uchopení myší a táhnutí na požadované místo. Pro přehlednost by měla být možnost oddalování a přibližování editační plochy, která by neměla mít pevně dané rozměry (stromy můžou být různě velké). To, že vrcholy lze spojovat, rozpojovat a mazat je samozřejmost.

Jelikož znovu použitelnost stromů je jedna z důležitých vlastností Behavior trees je nutné tuto funkcionalitu zahrnout v editoru. Proto je nutné vytvoření vrcholu, který ponese odkaz na jiný strom.

Akce, které budou použity v editoru, jsou definovány externě programátory. Tyto akce se v průběhu vývoje hry můžou měnit (např. může být změněno jméno, parametry akce nebo akce může být odstraněna). Pokud chceme používat editor v průběhu vývoje hry, kritickou vlastností je možnost synchronizace již vytvořených stromů s pozměněnými akcemi, tento požadavek máme již z úvodní kapitoly jako cíl C6. I kdybychom uvažovali o použití editoru až po definici všech akcí, nepřítomnost synchronizace by mohla zkomplikovat tvorbu chování. Uvažme například situaci, kdy designér AI tvoří stromy pro entity v průběhu práce zjistí, že by potřeboval určitou akci. Tato akce je typicky lehce implementovatelná v kódu, nicméně nemožnost synchronizace znemožní její použití v editoru. Musel by být vytvořený nový projekt a vytvořené stromy by byly ztraceny.

Důležitým nástrojem je možnost sledovat průběh výpočtu stromu během hry. Tato možnost pomáhá při ladění stromu. Proto minimální požadavek je možnost zapnutí ladícího režimu a sledování výpočtu pro vybrané entity během hry. Tento bod je již definován jako cíl C5.

Jako závěrečný požadavek uvedeme možnost uložení nebo nahrání rozpracované práce (již definováno jako cíl C7 v úvodu). Pojďme si nové požadavky na editor shrnout v cílech.

E1. Lehce přístupné menu s vrcholy.
E2. Editační plocha by neměla mít pevné rozměry.
E3. Možnost posouvat vrcholy libovolně v editační ploše.
E4. Oddalování a přibližování editační plochy.
E5. Vytvoření vrcholu, s jehož pomocí je možné se odkazovat na již vytvořené stromy.

3.5. Základní části frameworku

V této podkapitole prozkoumáme jaké hlavní části musí framework obsahovat.

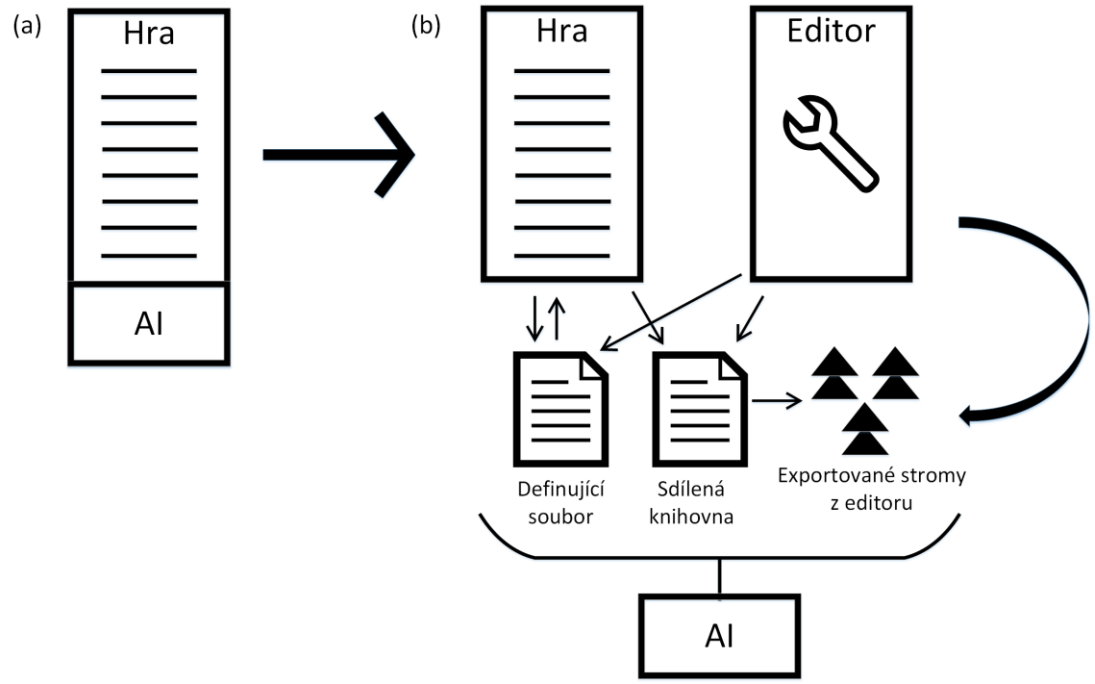
Počítačová hra obsahuje mnoho různých tříd. Pro tvorbu stromu potřebujeme jen entity a jejich akce. Proto je nutná nějaká definice entit a akcí. Budeme požadovat, aby byly entity a akce definovány zvlášť v souboru, který nazveme *definující soubor*. Tato definice může být různá, blíže bude tento problém rozebrán ve zvláštní podkapitole 3.7, kde dospějeme k tomu, že nejvhodnější je použít assembly, kde budou nadefinovány všechny entity a akce. Toto osamostatnění přinese přehlednost a vše bude na jednom místě.

Dále musíme programátorovi, implementující náš framework, předat knihovnu s našimi třídami, které umožňují implementování akcí a entit. Tato knihovna také musí obstarávat veškerou režii spojenou s AI, tj. aktualizovat chování entit a umožnovat ladění. Z uvedených důvodů je jasné, že knihovna bude distribuována uživatelskému programátorovi. Knihovna bude ze dvou částí:

- definice tříd pro entity a akce, které použije uživatelský programátor
- Runtime, který se postará o režii spojenou s AI během hry

Jelikož je primárním účelem této práce tvorba editoru, je jasné, že i editor tvoří část frameworku. Editor bude závislý na definujícím souboru, neboť obsahuje definici entit a akcí. Kromě toho je editor závislý i na knihovně, která je distribuována, neboť účelem editoru je vytvořit data, reprezentující designérovo vytvořenou práci, pro Runtime. Těmto datům musí rozumět jak Runtime, tak editor, proto jsou umístěna v knihovně. Knihovna je tedy sdílená.

Pro lepší orientaci je přiložen Obrázek 9. Pod obrázkem jsou shrnuty informace o jednotlivých částech.



Obrázek 9: Základní přehled částí frameworku a závislostí mezi nimi.

(a) Původní hra bez použití Revive. AI je integrální část hry.

(b) Hra při použití Revive. Došlo k oddělení AI. Definující soubor obsahuje entity a akce hry, proto je na něm závislý editor a hra. Zároveň akce můžou používat třídy hry.

Runtime ve sdílené knihovně používá exportované stromy z editoru. Sdílená knihovna poskytuje kontrakt pro hru i editor.

Sdílená knihovna

Tato knihovna obsahuje kontrakt definující to, jak vypadá vrchol Behavior tree z pohledu kódu, co musí splňovat entita a obsahuje runtime část zajišťující korektní průběh výpočtu stromu pro každou entitu tak, jak byl definován v editoru. Tuto knihovnu využívá programátor pro vytvoření akcí entit. Knihovna se vyskytuje ve dvou verzích – vývojářské a finální. Vývojářská verze obsahuje kód navíc, který zajišťuje podporu ladění tj. sledování výpočtu stromů. Zároveň knihovna obsahuje třídy pro vnitřní vrcholy stromů. Tato knihovna je distribuována koncovému uživateli – programátorovi a umožňuje vytvoření uživatelských entit a akcí.

Assembly s entitami a jejich akcemi

Assembly obsahující všechny třídy reprezentující akce a entity. Je závislá na sdílené knihovně, neboť v ní jsou obsaženy základní třídy pro akce a entity.

Editor

Editor umožňuje tvorbu Behavior tree pro konkrétní entity. Entity a jejich akce získá zpracováním definujícího souboru. Po dokončení práce umožňuje vyexportování stromů do souboru. Tyto data pak použije Runtime část pro vykonání výpočtu stromů. Editor je taktéž závislý na sdílené knihovně, neboť obsahuje definici vrcholů, což je nutné při extrakci dat z definujícího souboru.

Runtime

Runtime obstarává zpracování exportovaných stromů z editoru a jejich následné vykonání. Runtime obsahuje datovou strukturu pamatující si všechny nahrané stromy. Pokud dostane dotaz na vrácení nějakého stromu, jen se podívá do této struktury. Toto je užito při znovu použití některých stromů, kdy máme jeden strom v paměti a existují vrcholy, které se na tento strom odkazují. Tyto vrcholy strom hledají podle názvu. Runtime je obsažen ve sdílené knihovně a tudíž je na knihovně závislá i hra užívající Runtime.

3.6. Workflow

V této podkapitole se podíváme na schéma provádění činností při práci s editorem. Nejprve je nutné vytvořit soubor s entitami a jejich akcemi. K tomu programátor potřebuje sdílenou knihovnu. Pomocí knihovny naimplementuje své vlastní třídy reprezentující akce a entity. V assembly s kódem hry je třeba vytvořit instanci třídy obstarávají veškerou režii okolo a předávat ji entity pro vykonání jejich chování v každé iteraci `Update` smyčky hry. V této části veškerá práce programátora končí.

Následuje práce s editorem. Data pro editor se nahrají buď z uložené práce, nebo se vytvoří nová z definujícího souboru. Následuje tvorba AI. Po dokončení práce se vytvořené stromy exportují. Každý strom se exportuje do zvláštního souboru, jehož jméno je totožné s názvem entity, které strom patřil. Při puštění hry se exportované stromy nahrají a v případě použití vývojářské knihovny se předává informace o aktuálním spuštěném vrcholu do editoru.

3.7. Parsování souboru s entitami a akcemi

Popíšeme si dva přístupy pro definici entit a akcí.

1. Pouhý seznam s názvy entit a jejich akcemi. Pro potřeby zobrazení v editoru by to stačilo. Toto řešení je zcela špatně neboť by vyžadovalo dvojí definici všech akcí a entit. Jednou v definujícím souboru a podruhé v samotném souboru(y) s kódem. Toto představuje značné riziko vzniku typografických chyb a bylo by třeba dvojího editování při každé změně. Tento způsob využívá program JBT, který je srovnáván s Revive v kapitole 0. JBT takto definuje akce a entity ve zvláštním XML souboru.
2. Jako zdroj dat použít přímo assembly kde jsou implementovány entity a jejich akce. Problém parsování dobře řeší mechanizmus .NET Reflection (10). Reflection umožňuje zkoumat metadata programu samotným programem. Velmi užitečná věc je prozkoumání třídy s možností vytvoření instance. Toto se velmi hodí, neboť máme nástroj jak zkoumat třídy definované programátorem spolu s možností vytvoření instancí.

Jak vidíme, způsob 2 je lepší. Proto také upravíme naše názvosloví a budeme používat *definující assembly* místo definující soubor.

Pro potřeby editoru není vhodné data načíst a rovnou s nimi pracovat, ale je vhodnější data zanalyzovat a na jejich základě vytvořit datovou strukturu. Tuto strukturu pak můžeme použít jak v různých částech editoru zpracovávajících data z definujícího souboru, tak při synchronizaci. Struktura by měla

- držet informace (jaké informace bude řečeno dále) o akcích a entitách
- nabízet metody na vyhledání této informace na základě jména akce či entity
- držet informace i o každé třídě, jež má entitu jako potomka, důvodem je možnost definovat Behavior trees i jinde než entitě a pak jej využít ve více entitách (pro bližší informace viz podkapitola 4.2)
- poskytovat seznam chyb v případě, že nějaké budou v definujícím souboru

3.8. Oddělení závislosti na definujícím souboru

Reflection nám poskytuje možnost pracovat přímo s konkrétními třídami definovaných programátorem v definující assembly. Toto má své výhody. Můžeme například pro to, jaké má mít vrchol parametry či vizuální zobrazení, vytvořit zvláštní informační třídy. Dále bychom definovali společný kontrakt pro všechny vrcholy ve formě metod, které by vraceli zmíněné informační třídy. Takové řešení by zajistilo, že by třídy reprezentující vrcholy byly sebe popisující. Toto skýtá velké možnosti pro programátora vytvářející entity a akce. Můžeme mu tímto způsobem poskytnout velké možnosti pro editaci vrcholů a entit (vytvořil by si své vlastní informační třídy).

Nevýhodou tohoto přístupu je, že by bylo třeba nahrávat definující assembly při každém spuštění editoru, neboť by v ní byly třídy držící informace k vrcholům a zároveň samotné třídy reprezentující entity a akce. Toto by nebyl až takový problém, ale při změně definujícího souboru bychom dostali výjimku při načítání dříve

vytvořené práce, neboť bychom neměli stejné třídy. Zároveň ne vždy bychom chtěli propagovat změny v definující assembly (například změny provádíme postupně a nechceme zatím měnit data v editoru). Proto je třeba oddělit zvlášť data použitá editorem a zvlášť data v definující assembly. Vrchol v editoru budeme nazývat *Representant* a vrchol použitý Runtime částí, který je definován ve sdílené knihovně pak *Node*. `Representant` pak bude obsahovat textový klíč shodný s názvem `Nodu`, kterého representuje. Tím pádem snadno zjistíme, jaký vrchol v editoru odpovídá jakému vrcholu v Runtimu a naopak. Stejným způsobem vytvoříme dva druhy tříd pamatujících si entity. Toto jednoduše oddělí data a umožní snadné dohledání správné akce či entity z definující assembly. Zároveň lehce provedeme úpravy při změně v definující assembly jednoduchým přepsáním textového klíče.

Toto řešení není bez problému, neboť opět máme dvojí definici dat – konkrétní třídu v definující assembly a vlastní třídu v editoru. Problém nastává, když chceme vytvořit Behavior trees z dat editoru pro Runtime. Je nutné vytvořit třídu obstarávající překlad mezi třídami používané editorem a třídami v definující assembly. Princip je jednoduchý, překladová třída načte třídy z definující assembly a vytvoří datovou strukturu, která uloží třídy pod jejich názvy. Tato struktura je předaná třídám editoru v průběhu tvorby Behavior trees, které na základě svého textového klíče dohledají správnou třídu a vytvoří její instanci.

3.9. Výstup dat z editoru

Jelikož editor nepracuje přímo s daty z definující assembly, ale Runtime část tyto data používá, je nutné nějakým způsobem převést data z editoru. Jak bylo zmíněno v podkapitole 3.8, překlad mezi třídami v editoru a třídami v definující assembly zajišťuje jiná třída pro toto určená. Výstupem jsou konkrétní stromy složené z tříd z definující assembly (v případě akcí) či z tříd ze sdílené knihovny (v případě vnitřních vrcholů). Vytvořené stromy jsou serializovány a uloženy do souborů. Jména souborů jsou totožná s entitou, která soubor vlastní. V případě, že se jedná o Named tree (Behavior tree na který se jde odkázat a tím pádem ho využít na více místech, blíže podkapitola 4.2), je jméno souboru shodné s textovým klíčem (klíče Representantů jsou řešeny v kapitole o podpoře ladění v 4.4) použitého `Representanty` tohoto stromu. Binární data z těchto souborů jsou pak nahrány Runtimem a použity pro AI.

Nabízí se možnost exportovat vytvořené stromy v pro lidi přehlednějším než binárním formátu – XML. Výhoda tohoto řešení je přehlednost, která nabízí i možnost textové editace a univerzálnost umožňující zobrazení v jiném nástroji. Jelikož při Runtimu chceme mít možnost nahrávat a uvolňovat stromy z paměti, stejně bychom XML data museli zpracovat do binárního formátu před spuštěním hry. Možnost textové editace XML by také mohla způsobit nekonzistenci mezi informací, kterou si o stromech udržuje editor a informací, kterou má Runtime díky nahraným exportovaným datům. Toto by způsobilo problémy při ladění, kdy by dva komunikující objekty měly různá data. Binární serializace tak v tomto případě slouží i jako ochrana před změnou dat. Přístup publikování výstupu editoru jako XML soubor používá JBT editor, který je srovnán s Revive v kapitole 0.

Důležité je i uvědomění si use case. Jelikož editor sám dobře vizualizuje data, není až tak nutné data zveřejňovat pro jiné programy. Zároveň typické použití je export dat a následné použití Runtimem. Převod do XML by tak přinesl zbytečné zkomplikování a jelikož binární serializovaná data obsahují kompletní strukturu stromu, nebylo by složité vytvořit jednoduchý nástroj pro tvorbu XML z těchto dat. Nicméně tato funkce by se dočkala spíše atypického použití. Proto framework používá binární serializaci.

3.10. Dodatečné informace entit a akcí

Analýzou dat v definující assembly získáme jen názvy entit a akcí. Je důležité poskytnout další možnosti uzpůsobení. Vyžadujeme tyto vlastnosti:

- definice jaká akce patří jaké entitě
- definice jména pro zobrazení dané entity či akce v editoru
- definice parametrů pro `Node` spolu s typem hodnot, které může nabývat
- definice vizuální závislost jednoho parametru na hodnotě druhého (Například mějme parametr „typ kouzla“ s hodnotami „ohnivá koule“ a „ledový blesk“. Dále mějme parametr „rozšířit oheň po dopadu“ s hodnotami „Ano“ a „Ne“. Požadujeme mít možnost nadefinovat aby se parametr „rozšiř oheň po dopadu“ zobrazil v editoru jen pokud je vybrána možnost „ohnivá koule“ u parametru „typ kouzla“.)
- definice kategorie pro akce (určité typy akcí bychom chtěli seskupit do jedné logické kategorie)
- definice kategorie pro entity (entity chceme skupit do kategorií a podkategorií)

3.10.1. *Jak definovat dodatečné informace*

Jelikož definující assembly editor jen analyzuje a nevytváří žádné třídy, nemůžeme definovat například metody, které by vracely objekty s dodatečnými informacemi. Velmi vhodným řešením je vytvořit sadu konfigurovatelných atributů. Těmito atributy je pak možno označit příslušné třídy a my pomocí Reflection získáme ke každé třídě atributy i jejich parametry, které jsou pak použity editorem.

3.10.2. *Předání parametrů vrcholům*

Předem nevíme počet a typ parametrů, které budou obsahovat uživatelem definované akce. Pro zachování jednotného rozhraní u vrcholů je třeba vytvoření jedné univerzální třídy ve které můžeme informaci o typu a možných hodnotách parametru držet. Tuto třídu nazveme `NodeParameters` abychom se na ni mohli odkazovat a nedošlo k nějakému nedorozumění. `NodeParameters` je pak předána `Nodu` při konstrukci. Jelikož nás ve výsledku zajímá jen jaký parametr byl nastaven na jakou hodnotu, zmíněná třída, držící parametry, je velice jednoduchá. Nabízí pouze možnosti nastavení parametru na nějakou hodnotu. Vše v textové reprezentaci, neboť

atributy nedovolují užitý tříd ve svých parametrech. Nemůžeme tak pro parametry `Nodů` použít speciální třídu, neboť parametry se nastavují pomocí použití atributů.

3.10.3. *Přiřazení akcí entitám*

Jelikož máme zvlášť třídy pro entity a zvlášť třídy pro akce, musíme mít možnost přiřadit akci entitě. Díky výsledku analýzy shrnutého v podkapitole 3.10.1 přiřazení budeme dělat pomocí atributů. Vytvoříme atribut, který jako parametr dostane typy tříd, které reprezentují akce (je to vhodnější než textová reprezentace náchylná k chybám). Tímto parametrem označíme třídu reprezentující entitu a získáme přiřazení sady akcí k entitě.

Tento jednoduchý způsob není vhodný, neboť vyžaduje značnou manuální práci a typicky bude spousta akcí společná různým entitám. Je třeba aby byly akce nějakým způsobem děděny. K tomu využijeme dědičnosti tříd. Pokud nadefinujeme sadu akcí nějaké třídě, všechny odvozené třídy tyto akce také zdědí. Takový způsob je přirozený, neboť odpovídá tomu, jak funguje dědičnost metod (pomineme-li privátní metody). Tento princip tedy kopíruje přirozený programátorský model, který by byl použit, kdyby byly akce definovány třídou jako metody.

3.11. Komunikace mezi vrcholy

Vrcholy Behavior tree neví o sobě navzájem. Vrcholy jsou sebe popisující a to umožňuje velkou modularitu a znovu použitelnost vrcholů (neexistují žádné externí závislosti). Jsou případy, kdy bychom ale potřebovali porušit tuto vlastnost a vytvořit vrchol, jehož funkcionalita závisí na jiném vrcholu.

Například mějme akci, která zjistí jaké nepřátelské entity jsou poblíž. Tato akce má jako parametr v editoru možnost filtrace pro výběr entity s nejmenším počtem zdraví, výběr nejbližší, výběr entity s nejvyšším útokem apod. Dále máme sadu různých akcí, které odpovídají různým typům útokům. Chtěli bychom mít možnost nadefinovat „Použij ohnivou kouli na nejbližší nepřátelskou entitu s nejmenším počtem zdraví“. Toto je jednoduše proveditelné pomocí sekvence, která bude mít dvě děti – akci zajišťující výběr entity a akci odpovídající vyčarování ohnivé koule. Definice tohoto podstromu je v editoru snadná, nicméně musíme navrhnout prostředek, aby byly `Nody` definované v kódu schopny komunikace.

Jednotné rozhraní vrcholů nám neumožňuje vytvořit výjimky. Proto definujeme objekt, do kterého každý vrchol bude moci uložit data pod nějakým klíčem. Pojmenujme ho `Blackboard`. `Blackboard` budeme předávat skrz `Update` metodu jako parametr. Tímto způsobem se bude propagovat všem `Nodům`. Tvorbu `Blackboard` obstará kořen stromu a tím pádem slouží jako globální prostor pro proměnné. Pokud bychom chtěli vytvořit `Blackboard` lokální, stačí přidat vrchol, který obstará tvorbu `Blackboard` a předá tuto novou instanci dětem. Zde vyvstane problém jak vyhledat patřičný objekt, pokud dostanu lokální `Blackboard` a můj objekt je uložen v `Blackboard` o úroveň výše (výše ve stromu). Řešením je předat výše postavený `Blackboard` objektu níže postavené při konstrukci a jeho uložení. Při žádosti o objekt se koukneme, zda máme uložen objekt pod daným klíčem, pokud

ano, vrátíme ho, pokud ne, požadavek propagujeme na výše postavený `Blackboard`. Takto se rekurzivně dohledá správný objekt.

Tímto byla probrána kapitola zabývající se analýzou na nejvyšší úrovní. Následuje kapitola o vývojové dokumentaci řešící konkrétní technické problémy.

4. Vývojová dokumentace A –knihovna a funkcionalita

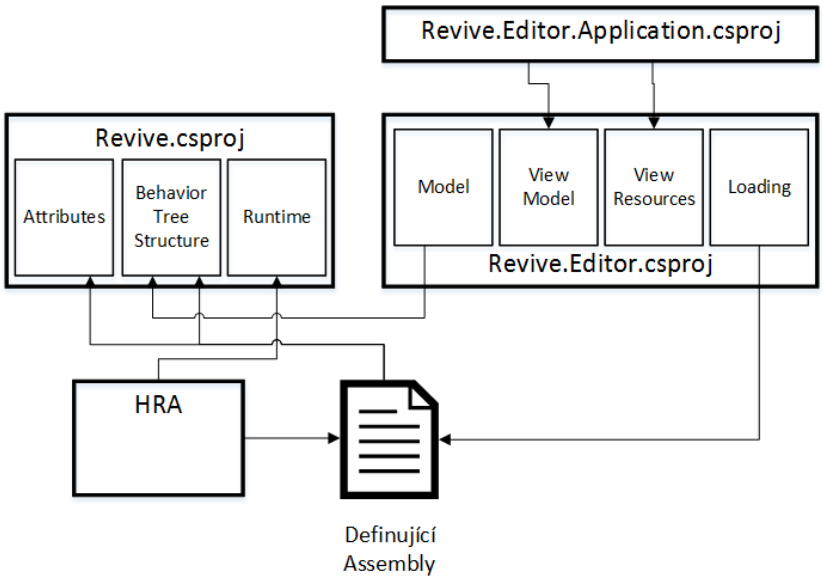
Vývojová dokumentace je rozdělena do dvou kapitol. Tato první část se zabývá funkcionalitou a knihovnou Revive.dll. Druhá část pak vývojovou dokumentací editoru a synchronizace.

Následující kapitola předpokládá pochopení konceptu Revive frameworku jak bylo popsáni v kapitole 3. Budeme se zabývat implementací všech částí frameworku. Jelikož se vrchol Behavior tree vyskytuje jak na straně Runtimu, tak v editoru, je nutné si ujasnit který vrchol myslíme. Runtime pracuje s instancemi třídy `Node` (případně `Task` viz dále) a editor s instancemi třídy `Representant`, resp. `RepresentantViewModel`, resp. `RepresentantView` (záleží na vrstvě v MVVM viz 5).

Program byl tvořen ve Visual Studiu 2010 a skládá se ze tří projektů:

- `Revive.csproj` kompiluje se do *Revive.dll*.
- `Revive.Editor.csproj` kompiluje se do *Revive.Editor.dll*.
- `Revive.Editor.Application.csproj` kompiluje se do *Revive.Editor.Application.exe*.

Závislosti mezi projekty ilustruje Obrázek 10. Připomeneme, že definující assembly je assembly, kde jsou umístěny třídy reprezentující entity a akce.



Obrázek 10: Projekty a jejich závislosti. Revive.Editor.Application reprezentuje vizuální část aplikace. U projektů Revive a Revive.Editor jsou zobrazeny dále jejich základní jmenné prostory:

- **Revive.csproj**
  - **Attributes – Obsahuje atributy. Atributy slouží k definici dalších informací k třídám reprezentující entity nebo vrcholy (například jméno, parametry akce apod.), které vytvoří uživatelský programátor v definující assembly.**
  - **BehaviorTreeStructure – Obsahuje definici kontraktu. Tento kontrakt udává jak mají vypadat třídy reprezentující vrcholy a entity na straně hry (tj. ne v editoru).**
  - **Runtime – Obsahuje logiku pro aktualizaci chování entit.**
- **Revive.Editor.csproj**

- **Model – Obsahuje třídy reprezentující Model vrstvu z MVVM (viz podkapitola 5).**
- **ViewModel – Obsahuje třídy reprezentující ViewModel vrstvu z MVVM.**
- **ViewResources – Obsahuje pár tříd, které jsou použity View vrstvou. Jsou umístěny zde, neboť je potřebuje i ViewModel.**
- **Loading – Obsahuje třídy schopné parsovat definující assembly a vytvořit z ní Model.**

Stručně popíšeme důvody závislostí (konkrétní třídy a jmenné prostory budou rozebrány dále v dokumentaci).

Závislosti Revive.Editor.Application.csproj

Tento projekt obsahuje View vrstvu z MVVM a je tedy závislý na ViewModel vrstvě. Ta je obsažena v `Revive.Editor.csproj` spolu s pár třídami, které také View vrstva potřebuje.

Závislosti Revive.Editor.csproj

Model obstarává tvorbu Behavior tree z práce v editoru pro Runtime. Jelikož je kontrakt toho, jak mají vrcholy vypadat, v `Revive.csproj`, je na `Revive.csproj` tento projekt závislý. Dále je tento projekt závislý na definující assembly, neboť ta je užita k vytvoření datové struktury, reprezentující akce a entity programátora implementující Revive framework. Dále je tato struktura použita Modelem a ViewModelem.

Závislosti hry

Jelikož je hra ke svému fungování potřebuje entity a jejich akce, je závislá na definující assembly. Jelikož aktualizaci AI obstarává část frameworku umístěná ve jmenném prostoru `Revive.Runtime`, je závislá hra na *Revive.dll*.

Závislosti definující assembly

Definující assembly je závislá na *Revive.dll*, neboť je zde kontrakt pro implementování entit a akcí ve jmenném prostoru `Revive.BehaviorTreeStructure`. Kromě toho je závislá na *Revive.dll,* neboť ve jmenném prostoru `Revive.BehaviorTreeStructure` se nachází atributy, které jsou použity například k definici parametrů pro akce či definici jaké akce patří jaké entitě.

Tímto jsme popsali jaké projekty tvoří framework a jaký je jejich účel. Před dalším pokračováním poznamenejme, že konvence projektu je taková, že pro nový jmenný prostor je použita nová složka a pro každou třídu je použit zvlášť soubor s názvem shodným s názvem třídy. Vnoření složek reprezentující jmenné prostory pak odpovídá vnoření jmenných prostorů.

Nejdříve se budeme zaobírat částí implementovanou uživatelským programátorem definovanou ve sdílené knihovně a dále pak hlavní částí aplikace – editorem.

4.1. Knihovna Revive.dll

Tato knihovna je distribuována programátorovi používající framework a obsahuje definici tříd reprezentující vrcholy stromu, rozhraní pro entity, Runtime obstarávající

AI pro entity, atributy pro definování dodatečných informací k entitám a akcím, kontrakt pro ladění a klienta pro spojení s editorem během ladění. Tato knihovna se vyskytuje ve dvou verzích:

- Vývojářská. Tato verze obsahuje režii okolo ladění stromů. Použití této verze způsobí, že Behavior trees použité entitami dělají callbacky do editoru, a tak můžeme sledovat výpočet stromu.
- Finální. Toto je knihovna bez režie spojené s laděním. Tuto knihovnu používá finální verze hry.

Více o principu ladění viz podkapitola 4.4. Pojďme rozebrat jednotlivé části knihovny.

4.1.1. *Atributy*

Atributy obsahuje jmenný prostor `Revive.Attributes` a jsou umístěny v souboru *AttributesCollection.cs*. Hlavním účelem atributů je definování dodatečných informací k třídám reprezentující entity či akce. Jaké informace je možno definovat bylo popsáno v podkapitole 3.10. Pro více informací o konkrétních atributech viz dokumentační komentáře.

Také je zde obsažen enum `ParameterType` sloužící k definici typu parametru. Parametr slouží k získání dat od uživatele a k tomu je třeba, aby byl zobrazen jako WPF komponenta (`ComboBox`, `CheckBox` apod.). Proto je nutné mít typ komponenty jako `enum`. Jak pracovat s parametry je popsáno v uživatelské dokumentaci v podkapitole 7.2.2. Dále jsou ve jmenném prostoru `Revive.Attributes` dvě třídy `DependencyWrapper` a `ParameterWrapper`, které byly umístěny v assembly *Revive.Editor.dll*. Kvůli tomu, že je ale potřebuje jak editor, tak *Revive.dll*, je třeba je nechat v *Revive.dll* kvůli zabránění cyklických závislostí.

ParameterWrapper (třída)

Tato třída si pamatuje informace o parametru, tj. jak se jmenuje, jaký je typ, jaké může mít hodnoty a jaká hodnota má být použita jako defaultní.

DependencyWrapper (třída)

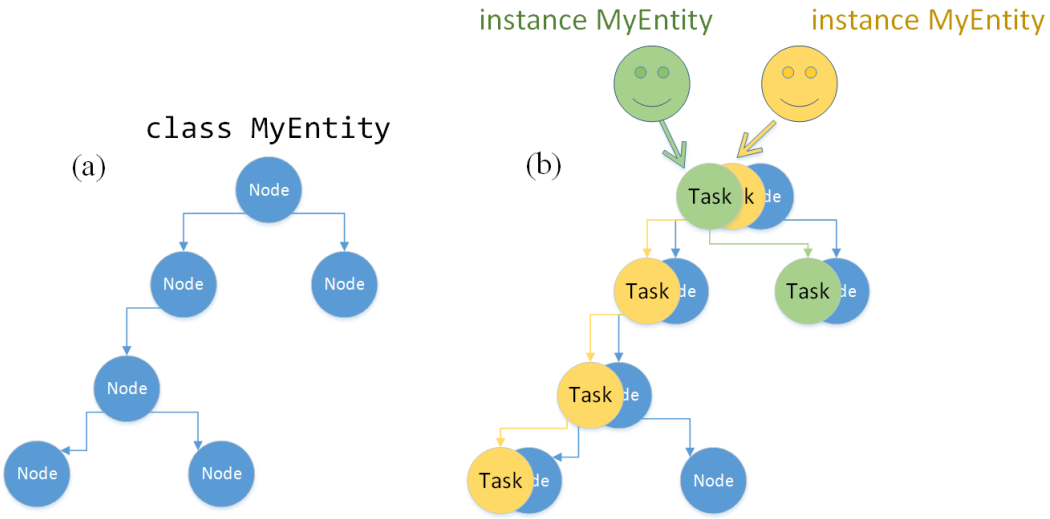
Účelem této třídy je pamatovat si jaký parametr závisí na jakém a na jakých hodnotách.

4.1.2. *Vrcholy stromu*

Kvůli šetření pamětí nemá každá entita[1] uložený Celý behavior tree, ale jen část, kterou potřebuje. Stejné entity (tzn. instance stejné třídy) mají stejné stromy, netřeba tak držet kopie, ale stačí jeden strom pro všechny instance této třídy. Tento strom je tedy definován pro třídu a ne pro instanci. Odpovídá to konceptu v editoru, kde tvoříme strom pro typ entity a všechny instance konkrétního typu mají stejný strom

[1] Tímto termínem myslíme instance třídy, která reprezentuje entitu a implementuje rozhraní IEntity.

jen v jiným stavu výpočtu. Každá instance entity si pak drží podstrom, který používá k výpočtům. Jelikož jeden vrchol může být v několika stavech vzhledem k několika instancím, je třeba oddělení struktury stromu a logiky stromu. Proto máme dvě třídy `Node` a `Task`. `Node` reprezentuje strukturu stromu a každý `Node` vytváří `Task`, který je už specifický pro danou instanci. Entity tedy používají instance třídy `Task` k výpočtu a ne `Node`. Behavior tree složený z instancí třídy `Node` je držen v paměti jen jednou pro třídu, reprezentující entitu. Situaci ilustruje Obrázek 11.



Obrázek 11: Oddělení závislosti výpočtu stromu od struktury stromu.

a) Definice Behavior tree skládajících se z instancí třídy Node. Tyto instance obsahují logiku struktury stromu. Tento strom je společný všem instancím třídy MyEntity a je v paměti jen jeden.

b) Každá instance třídy MyEntity má svůj vlastní podstrom složený z instancí třídy Task. Tyto instance obsahují logiku výpočtu stromu.

Všechny třídy spojené s definicí vrcholů stromu a entit obsahuje jmenný prostor `Revive.BehaviorTreeStructure`. Pojďme si probrat jednotlivé třídy pro vrcholy stromu.

Node (abstraktní třída)

Byl popsán v úvodů této kapitoly a slouží jako základní třída schopna vytvářet `Task`. Při konstrukci dostává `NodeParameters`, kde jsou uloženy případné parametry. Také obsahuje textový a číselný klíč pro identifikaci během ladění. `Node` má definovanou metodu `OnNodeLoad`, která slouží k předání objektů uživatelského programátora během načítání. Více viz podkapitola 4.1.3.

UserNode (abstraktní třída)

Toto je třída od které dědí každý vrchol v definujícím assembly a je implementována uživatelským programátorem. Vytváří `UserTask`. Je důležité, aby každá poděděná třída měla konstruktor s `NodeParameters`! To je kvůli tomu, neboť jsou instance třídy `UserNode` vytvářeny pomocí třídy `Actiovator` (jmenný prostor `Systém.Reflection`) a `Activator` vkládá instanci třídy `NodeParameters` jako parametr konstrukce.

Task (abstraktní třída)

Byl popsán v úvodu. Obsahuje logiku výpočtu vrcholu. Hlavní důležitou metodou je abstraktní metoda `Update` obsahující logiku výpočtu. `Update` metoda vrací jeden ze stavů, který definuje enum `Status` (definovaný ve stejném souboru *Task.cs*). Kromě jiného obsahuje `Task` property `IsCondition`. Nastavením této property na `true` říkáme frameworku, že tento `Task` je podmínka. Toto slouží například paralelnímu `Tasku` (`Parallel`), aby věděl, že takový `Task` má aktualizovat každý `Update` krok i když je `Task` úspěšně dokončen.

DebugableTask (abstraktní třída)

Tento `Task` obsahuje logiku potřebnou pro ladění. Viz podkapitola zabývající se laděním 4.4.

UserTask (abstraktní třída)

Tento `Task` vytváří `UserNode`. Má už implementovanou logiku pro ladění, a tak uživatelský programátor nemusí nic implementovat kromě normální logiky, kterou by implementoval, kdyby nevěděl o ladění.

Kromě zmíněných abstraktních tříd pro vrcholy, které udávají kontrakt, jsou zde i vrcholy naimplementované. Jedná se o vrcholy, které jsou společné všem entitám. Jsou rozděleny do dvou jmenných prostorů:

Revive.BehaviorTreeStructure.InnerNodes (jmenný prostor)

Obsahuje vnitřní vrcholy stromu. Každý vnitřní vrchol musí dědit od třídy abstraktní třídy `CompositeNode`. Každý implementovaný vrchol pak dědí od jedné ze dvou abstraktních tříd, implementujících metody pro práci s dětmi:

- `ArrayCompositeNode` - pro ukládání dětí používá pole. Od této třídy dědí všechny vrcholy, které mají více než jedno dítě (`SequenceNode`, `ParallelNode`, `SelectorNode`).
- `SingleChildNode` – může uložit jen jedno dítě a používá k tomu proměnou. Od této třídy dědí všechny třídy, které mají jedno dítě (`RootNode`, `NegationNode`)

Pro bližší informace o jednotlivých vrcholech viz dokumentační komentáře či uživatelská dokumentace zabývající se vrcholy v podkapitole 6.3.3.

Revive.BehaviorTreeStructure.LeafNode (jmenný prostor)

Obsahuje vrcholy, které jsou listy. Jedná se o `FailNode` (vždy selže), `SuccessNode` (vždy uspěje) a `LookUpNode` (odkazuje se na jiný strom). O principu odkazování se na jiné stromy viz podkapitola 4.3.

Zbývá poslední třída týkající se struktury stromu.

Behavior (třída)

Tato třída „obalí“ `Node` a `Task`, který vytváří a automaticky volá metody instancí třídy `Task` `OnInitialize` a `OnTerminate` když je třeba. `Behavior` nabízí metodu `Tick`, která dělá zmíněné „obalení“ a vrací `Status` instance třídy `Task`. Tím pádem nám stačí volat metodu `Tick` a kód je tak přehlednější.

Tím jsme dokončili přehled tříd týkajících se Behavior tree. Dále následuje definice kontraktu toho, co je entita. Ta je dělána přes jednoduché rozhraní `IEntity`.

IEntity (rozhraní)

Toto jednoduché rozhraní nám umožní uložit instanci třídy `EntityData` a definuje metodu `GetDebugData`, která může vracet data pro editor během ladění (viz podkapitola o ladění 4.4).

EntityData (třída)

Tato třída slouží k uložení instance třídy `Behavior` (reprezentující kořen) a dále zveřejňuje property `Name`, které vrátí jméno třídy vlastnící `EntityData` (resp. vrací textový klíč kořenu, který má být stejný, jako jméno třídy, které reprezentuje entitu).

Poslední dvě třídy v tomto jmenném prostoru jsou `NodeParameters` (zajišťuje zapisování parametrů) a `Blackboard` (zajišťuje předávání parametrů mezi instancemi třídy `Task`).

NodeParameters (třída)

Důvod této třídy a princip byl vysvětlen v podkapitole 3.10.2. Pro ukládání parametrů tato třída používá instanci `Dictionary<string,object>`. Ačkoliv bylo v analýze zmíněno, že je vše v textové reprezentaci, vyplatí se nám ukládat instance třídy `Object` namísto textu. To nám umožní předat instanci třídy `Node` i instance jiných tříd. Toto je prováděno při načítání instance `Node` (viz pokapitola 4.1.3 třída `RuntimeManager`). Takto může uživatelský programátor předat své objekty instancím tříd odvozených od `UserNode`.

Blackboard (třída)

Důvod a účel této třídy byl vysvětlen v analýze v podkapitole 3.11. `Blackboard` je předávaný instancemi třídy `Task` mezi sebou skrz jejich `Update` metodu. `Blackboard` nemůže k ukládání dat použít instanci `Dictionary<string,object>`, neboť by to znamenalo hashování pokaždé, kdy bude `Task` chtít data. Proto Blackboard interně požívá pole instancí třídy `Object`. Bylo zvoleno pole o délce padesát. O tvorbu pole se stará instance třídy `Root` (soubor *RootNode.cs*) a tak lze délku změnit, když by bylo za potřebí. Nám přišla délka padesát jako dostačující.

Typický scénář použití `Blackboard` uživatelským programátorem je definice `enumu` k indexaci do `Blackboard`. Takto nemusíme používat čísla, nebudeme mít problémy s typografickými chybami a vše bude dostatečně rychlé a přehledné.

Oproti konceptu popsaném v podkapitole 3.11 neposkytuje `Blackboard` rekurzivní dohledání a všechny instance třídy `Task` pro instanci dané entity používají jednu instanci třídy `Blackboard`. Toto je zapříčiněno dvěma důvody:

- Není jasné, zda máme dělat rekurzivní dohledání, pokud na daném indexu máme `null`.
- Z principu, kterým se náš `Blackboard` používá, je to zbytečné. Jelikož se k indexaci používají `enumy`, nedojde ke konfliktům.

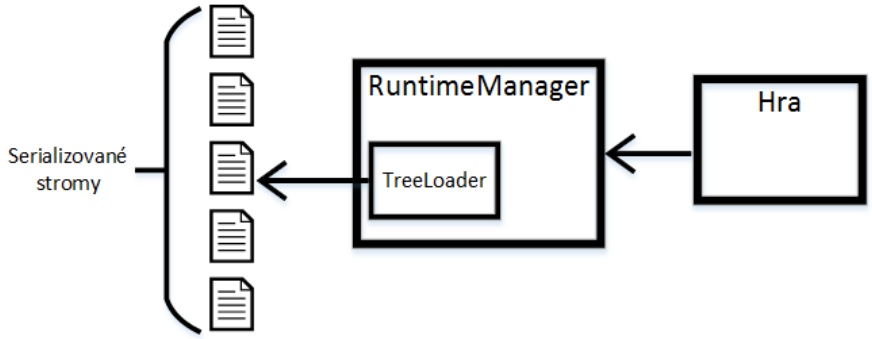
Tímto byl jmenný prostor `Revive.BehaviorTreeStructure` dokončen a můžeme se zabývat další důležitou částí a tou je Runtime.

4.1.3. *Runtime*

Jmenný prostor `Revive.Runtime` obsahuje tři třídy zajišťující vykonání chování entit. Jsou to `RuntimeManager`, `TreeLoader` a `NodeRegister`. Princip pracování Runtimu je následující:

Stromy, které vzniknou z editoru během publikování jsou serializovány do složky, jejíž jméno je stejné jako textový klíč kořenu stromu. Pokud se tedy jedná o strom entity, je název binárního souboru se stromem totožný s jménem entity. Pokud se jedná o Named tree, je název ve tvaru TŘÍDA_STROM kde TŘÍDA je název třídy a STROM je jméno stromu. Tyto soubory mají koncovku `.tree`. Proč je název dělán takto, je zdůvodněno v podkapitole 4.2, zabývající se znovu použitím stromů.

`RuntimeManager` si vytvoří instanci třídy `TreeLoader` při konstrukci. Když je `RuntimeManager` požádán o `Update` entity, z jejího jména zjistí jaký strom má nahrát a požádá instanci třídy TreeLoader o nahrání. Princip fungování ukazuje Obrázek 12.



Obrázek 12: Princip fungování Runtime.

V případě, že je použita ladící verze knihovny, je načtený strom zaregistrován v instanci třídy `NodeRegister`, kterou drží `RuntimeManager`.

Teď, když je princip fungování jasný, pojďme si rozebrat třídy.

TreeLoader (třída)

Dostane cestu k adresáři, kterou si uloží a poskytuje metodu `LoadTree`, která jako parametr dostane název stromu. Tato metoda prozkoumá, zda daný soubor v adresáři existuje a případně ho deserializuje. Pokud soubor neexistuje vrací `null`.

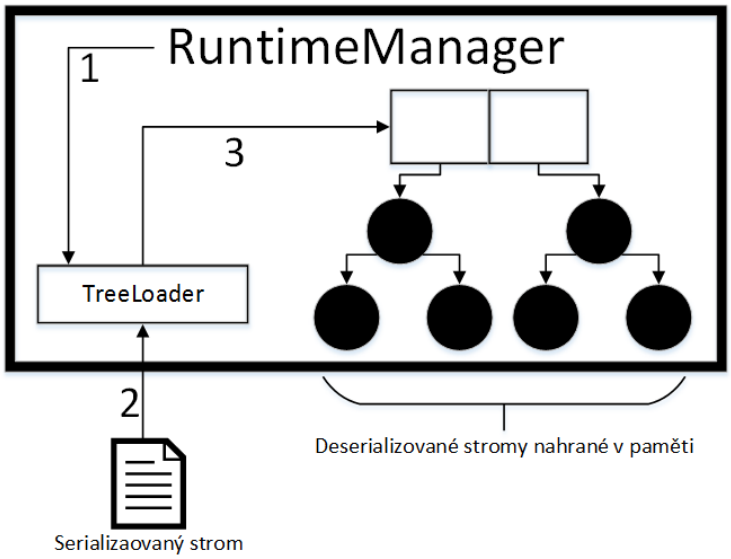
NodeRegister (třída)

Tato třída poskytuje metodu pro registraci stromu `RegisterTree`. Rekurzivně projde strom a zaregistruje všechny neregistrované instance třídy `Node`. Jako klíč použije klíče instancí `Node`. K registraci používá `Dictionary<string,Node[]>`. Textový klíč odpovídá klíči v `Dictionary` a číselný klíč je index v poli. `NodeRegister` není potřeba kvůli principu jak probíhá ladění (viz podkapitola 4.4), je zde ale přítomen pro budoucí rozšíření, kdy bude možno měnit parametry

za běhu, a tím pádem bude nutné identifikovat jaká instance třídy `Node` byla změněna.

RuntimeManager (třída)

Toto je třída, jejíž instanci si vytvoří programátorský uživatel. Pomocí `TreeLoader` nahrává serializované stromy a drží je v paměti (ilustruje to Obrázek 13).



Obrázek 13: Nahrávání stromu třídou RuntimeManager

1. **RuntimeManager je požádán o nahrání stromu či o Update entity, jejíž strom není v paměti. Požádá TreeLoader o nahrání.**
2. **TreeLoade vyhledá strom, deserializuje ho a vrátí ho RuntimeManager**
3. **RuntimeManager si uloží strom k ostatním.**

Poskytuje metody pro před nahrání stromů určitých entit pro zrychlení prvotního `Update` entity. Dále lze nepotřebné stromy od nahrávat pro uvolnění paměti. Nejdůležitější metodou je `Update` zajišťující aktualizaci chování entit. V případě použití ladící verze knihovny tato třída inicializuje `Debug` třídu pro umožnění callbacků od instancí třídy `Task a Update` deleguje na třídu `Debug` (více podkapitola o ladění 4.4). V ladící verzi je také vytvořena instance `NodeRegister`, která zaregistruje každý nahraný strom. Dále `RuntimeManager` obsahuje instanci třídy `NodeParameters`, kterou zveřejňuje. Tuto třídu předá načtenému stromu pomocí `OnNodeLoad` metody. `NodeParameters` může využít uživatelský programátor k uložení svých objektů a předání je instancím svých tříd poděděných od `UserNode`.

4.2. Parsování definující assembly

Parsování bylo probráno v podkapitole 3.7. Zde si probereme konkrétní třídy. Parsování obstarávají třídy z jmenného prostoru `Revive.Editor.Loading`. Tento jmenný prostor je umístěn v *Revive.Editor.dll* assembly. Výsledná třída, držící informace o assembly se jmenuje `AssemblyData`. Tuto třídu vytvoří statická metoda

`LoadData` třídy `DataLoader`. Pojďme si rozebrat třídy, které slouží k reprezentaci dat.

Wrapper (třída)

Toto je základní třída pro reprezentaci o třídě o které poskytuje informace, obsahuje jen jméno k zobrazení pro editor a typ třídy.

EntityWrapper (třída)

Obsahuje informace o třídách reprezentující entity nebo jejich rodiče. Obsahuje příslušná data definovaná atributy a dědí od třídy Wrapper.

ActionWrapper (třída)

Obsahuje informace o třídách reprezentující akce. Obsahuje příslušná data definovaná atributy a dědí od třídy Wrapper.

Tyto třídy jsou použity instancí třídy `AssemblyData`.

AssemblyData (třída)

Obsahuje seznamy `ActionWrapper` a `EntityWrapper` a zároveň ukládá každou instanci `EntityWrapper` a `ActionWrapper` do `Dictionary`, aby bylo možné hledat objekt podle jména.

DataLoader (statická třída)

Poskytuje metodu pro vytvoření `AssemblyData` z definující assembly. Pokud nelze data vytvořit vrátí seznam chyb zabraňující vytvoření.

4.3. Znovu použití existujících stromů

Účelem této podkapitoly vyřešit cíl E5 z našich kritérií na editor z podkapitoly 3.4. Nejdříve je nutné vyřešit jak bude vrchol, odkazující se na jiný strom, implementován. Toto má dvě roviny, vrchol chápaný editorem, tedy `Representant` a vrchol chápaný Runtimem, tedy instance třídy `Node`. Vrchol, který se bude odkazovat na jiný vrchol v Runtimu implementuje třída `LookUpNode`. Pojďme si rozebrat modelové příklady týkající se znovu použitelnosti stromů.

Mějme entity A a B a uvažujme, že entita B chce znovu použít strom definovaný entitou A. Pomineme-li, že je tento scénář nepravděpodobný, tak zde narazíme na problém. Entita A může používat akce, které nejsou definované na entitě B. Pojďme se soustředit na reálnější scénář.

Máme entity A a B a chtěli bychom vytvořit mimo strom, který by použily obě entity. Problémem je, jak zabezpečit, aby strom používal jen akce z průniku množin akcí entity A a entity B. V podkapitole 3.10.3 jsme řešili problém přiřazení akcí entitě. Z principu našeho řešení tedy stačí strom, který má být použit více entitami, nadefinovat na třídě, reprezentující společného předka entit A a B. Tím zaručíme, že strom bude obsahovat jen akce společné oběma entitám.

Problémem by mohlo být, kdyby entity A a B neměli stejného předka (pomineme-li třídu `Object`) a přesto bychom chtěli využít nějaký strom. Tento případ ale znamená, že je nejspíše chybu v designu. Toto by znamenalo, že jsou někde nějaké akce definovány zbytečně nadvakrát a jdeme proti principu dědičnosti. Výjimkou může

být třeba definice rozhraní a implementování tohoto rozhraní dvěma nepříbuznýma třídami. My se nicméně zabýváme definicí akcí pro entity. Jelikož všechny třídy mají společného předka (`Object`), lze vždy vytvořit společnou třídu pro libovolné dvě podtřídy (neboť dědičnost je strom). Proto tento problém nejspíše znamená chybu v designu – chybí zde nějaká třída. Pokud by se ve vzácných případech nejednalo o chybu, jde vždy tento problém vyřešit znovu postavením stromu v patřičné entitě a mít tak stejný strom vícekrát. Tyto speciální případy Behavior trees, na které se jde odkázat pojmenujeme *Named trees*.

Abychom umožnili definici stromů nejen na entitách, ale i na třídách, reprezentující předky entit, je třeba nové datové struktury, která je pojmenována `InheritenceTree`.

4.3.1. *InheritenceTree*

Účelem této struktury je umožnění definice Behavior trees na každé třídě, která je předkem nějaké entity včetně entity samotné. Tuto strukturu používá editor a je vytvořena na základě datové struktury `AssemblyData` vytvořené z definujícího souboru (viz 4.2). `InheritenceTree` reprezentuje strom dědičnosti a je složen z tříd `InheritenceNode`. Jak takový strom vypadá ilustruje Obrázek 14 na konci podkapitoly (strana 34), který ilustruje přidávání Named tree.

InheritenceNode (třída)

Umožňuje definici Behavior tree a přidání referencí na další `InheritenceNode` či `EntityModel` (reprezentuje entitu). Udržuje si seznam `ActionModel` (reprezentuje akce), aby byla uchována informace jaké akce můžou Named tree, definované konkrétním `InheritenceNode`, obsahovat.

Tímto způsobem získáme stromovou strukturu odpovídající dědičnosti tříd v definujícím souboru. Jako listy jsou instance třídy `EntityModel`, reprezentující entity. Schopnost vracet Named tree reprezentuje jednoduché rozhraní `INamedTreeContainer`. Toto rozhraní implementují všechny vrcholy `InheritenceTree`.

INamedTreeContainer (rozhraní)

Je definováno ve jmenném prostoru `Revive.Editor.Model` v assembly *Revive.Editor.dll* a definuje jednu metou `GetNamedTreeRoots` vracející seznam Named trees (tedy seznam instancí třídy `Representant`, které reprezentují kořeny Named trees).

Každý `InheritenceNode` může obsahovat nejeden Behavior tree, který je reprezentovaný instancí třídy `Representant`, která reprezentuje kořen. Aby se na stromy dalo odkazovat, musejí být pojmenovány. Dále je třeba vyřešit klíče `Representantů` obsažených v těchto stromech. Číselný klíč řešit nemusíme (každý strom má unikátní číselné klíče), je třeba vyřešit ten textový. Pokud bychom použili jako klíč jméno stromu, které si vybere uživatel tvořící strom, hrozilo by značné riziko na konflikt klíčů, proto jako klíč použijeme název třídy, oddělovací značku „_“ a pak název stromu. Pokud bychom tedy definovaly Named tree na vrcholu reprezentující třídu „A“ a tento strom bychom pojmenovali „MyTree“, klíč všech

instancí třídy `Representant` ve stromě by byl „A_MyTree“. `InheritenceNode` používá instanci třídy `RootContainer` k držení instancí třídy `Representant` reprezentující kořeny Named tree.

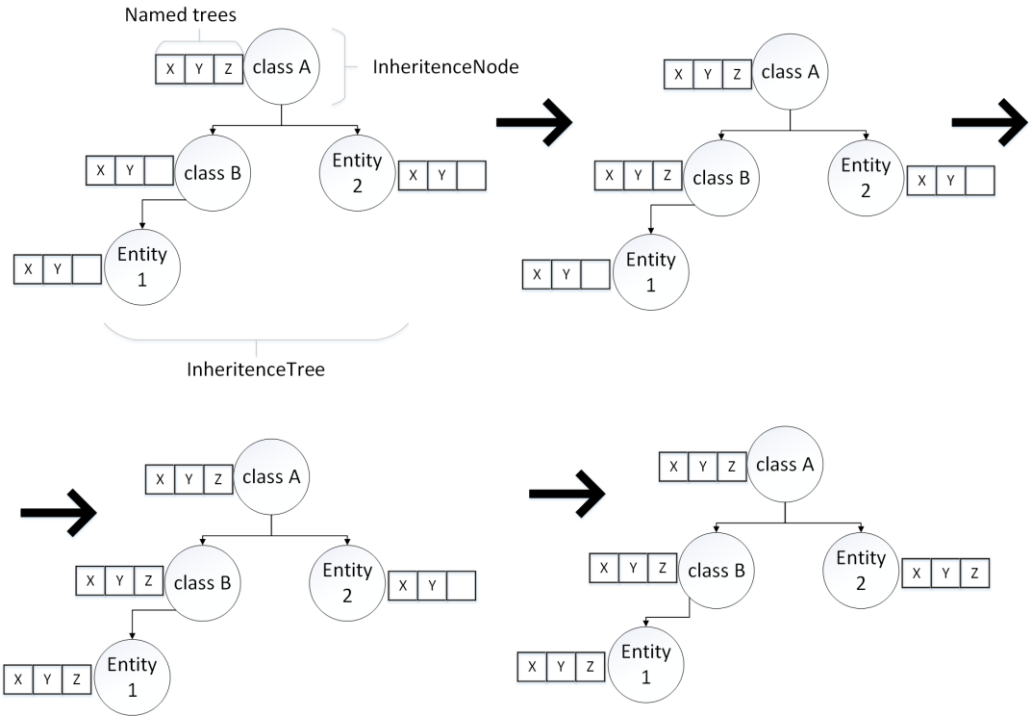
RootContainer (třída)

`RootContainer` používá k držení referencí na Named tree `List<Root>`. Nabízí metody pro tvorbu, přidání, mazání a hledání Named tree podle jména.

Root (třída)

Tato třída si pamatuje instanci třídy `RepresentantRoot` (representuje kořen Named tree) a dále obsahuje textové pole s názvem stromu bez prefixu (tj. část před znakem „_“ včetně). Jméno bez prefixu bychom si klidně mohli vždy zjistit a třída `Root` by nebyla třeba, nicméně takto je to pohodlnější a přehlednější.

Na Named tree se nemusí odkazovat jen entita, ale i jiný Named tree. Získáme tak komplexní možnosti tvorby Behavior trees. Toto odkazování obstarává `LookUpRepresentant`, který si pamatuje textový klíč příslušného stromu. Abychom zjistili rychle jaké Named trees patřičná entita má, je třeba přidat to třídy reprezentující entity instanci `List<Representant>`, která drží kořeny Named trees, a lze se tak na ně odkázat. Při definici nového vrcholu Behavior tree na `InheritenceNode`, pak vrchol, na kterým definujeme Named tree, projde své děti a oznámí jim přidání nového Named tree. Děti zase pošlou informaci dál a tak se rekurzivně přidá Named tree celému podstromu. V případě mazání je postup stejný.



Obrázek 14: Postup při přidávání nového Named tree nazvaného „Z“ k vrcholu reprezentující třídu A. Přidávaný strom je postupně propagován celému podstromu.

4.3.2. *LookUp vrchol*

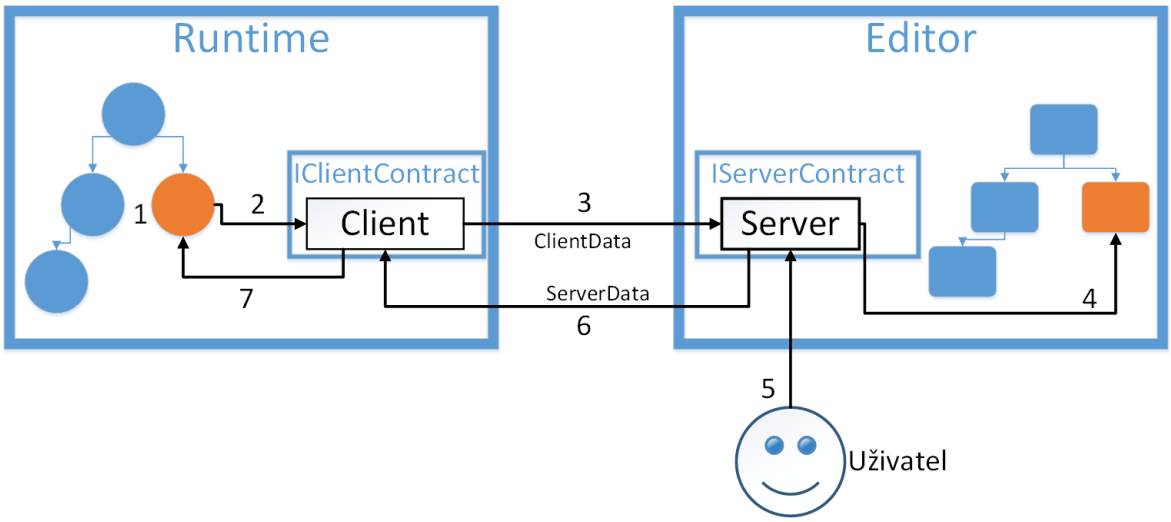
Účel tohoto vrcholu je vytvoření reference na strom, který chceme použít. Teď když už víme, jak princip znovu použití stromů funguje, je lehké navrhnout tento vrchol.

`LookUp` na straně Runtimu je implementován jako `LookUpNode`, který drží referenci na patřičný strom a v `Update` metodě `Task` vyrobený instancí třídy `LookUpNode` jen propaguje `Update` dolů a vrácený stav nahoru. Referenci na strom získá pomocí instance třídy `TreeLoader` (viz podkapitola 4.1.3), kterou Runtime vloží do `NodeParameters` a předá metodou třídy `Node` `OnNodeLoad`.
V editoru je `LookUp` implementován jako `LookUpRepresentant`, který si pamatuje textový klíč, který je použit stromem na který se odkazuje. Při exportování pak předá tento klíč vyrobené instanci třídy `Node` a tento `Node` využije `TreeLoader` k získání reference na daný strom s tímto klíčem jak bylo zmíněno.

4.4. Podpora ladění

Prvně je třeba si uvědomit, že máme dvě oddělené aplikace – hru a editor. Obě části jsou napsány v C#, a z kapitoly 2 víme, že musíme použít WCF a `NetNamedPipeBinding`. V roli serveru bude vystupovat editor a Runtime bude klient. Klidně by mohl být Runtime server a editor klient, jelikož zde máme vztah jedna ku jedné. Koncept editoru jako hlavní části, ke které se připojí hra nám přišel intuitivnější.

Hrubý princip bude následující. Uživatel nastaví breakpoint na vybrané entitě (entitách) v editoru, což signalizuje, že tyto entity chce uživatel ladit. Je volána `Update` metoda instance třídy `Task`, která reprezentuje vrchol stromu entity, na které je nastaven breakpoint. `Task` udělá callback pomocí klientské třídy, implementující `ICientContract`, do editoru a předá instanci třídy `ClientData` reprezentující data entity a `Task`u (bude rozebráno dále). Editor dostane informaci o vrcholu a zobrazí uživateli jaký vrchol je aktuálně volán. Uživatel má možnost potvrdit pokračování výpočtu vrcholu. `Task` v Runtimu zatím nepokračuje ve výpočtu, dokud nedostane potvrzení z editoru, který potvrzení odesílá pomocí instance třídy `Server` (implementující `IServerContract`). `Server` posílá data, která jsou reprezentována třídou `ServerData`, která může obsahovat další informace z editoru. Obrázek 15 ilustruje zmíněnou myšlenku. Pro jednoduchost zde není znázorněno zvláštní rozdělení na instance tříd `Task` a `Node`.



Obrázek 15: Princip fungování ladění. Kruhy znázorňují Nody, obdélníky se zaoblenými rohy Representanty

1. **Task dané instace třídy Node se dostane na řadu**
2. **Udělá callback do editoru pomocí klientské třídy**
3. **Klientská třída provede volání a vytvoří ClientData**
4. **Server dostane informaci o Tasku a zvýrazní příslušného Representanta**
5. **Uživatel potvrdí pokračování ve výpočtu**
6. **Server deleguje vytvoří ServerData a předá je klientovi**
7. **Klient povolí Tasku pokračovat**

Pojďme podrobněji rozebrat detaily a problémy.

4.4.1. *Nutnost dvou knihoven*

Volání do editoru z herní části frameworku se provádí jen v případě spuštění hry za účelem sledování výpočtu Behavior tree. Při spuštění hry víme, že hra poběží celou dobu jen v jednom režimu. Mohli bychom vytvořit dvě `Update` metody pro třídu `Task`, jednu pro ladění, druhou regulérní pro finální hru. Při spuštění by `RuntimeManager` dostal booleovskou proměnou říkající v jakém je režimu a podle toho by se celou dobu volaly správné metody. Nastanou tím ale dva problémy.

- Máme mnoho nevyužitého kódu (kromě nevyužitých metod je zde režie kolem volání do editoru a identifikace vrcholů), což není dobré pro paměť.
- Je to pomalejší, neboť v průběhu výpočtu musíme dělat kontroly toho, zda je zapnut ladící režim. I když je toto zpomalení minimální, lze se mu vyhnout.

Revive tento problém řeší použitím preprocesorové direktivy `#if DEBUG`. Kód mezi `#if DEBUG` a `#endif` je automaticky kompilován pouze pokud používáme Debug build. Tímto způsobem se můžeme obalit kód užitý jen v ladícím režimu. Toto řešení ale vyžaduje dvě různé knihovny. Vývojářskou, kde budou všechny ladící callbacky, a finální, která je bez veškeré režie spojené s laděním a užitím WCF.

4.4.2. *Definice kontraktu komunikace*

Veškerá definice tříd a rozhraní spojených s komunikací je definována v assembly *Revive.dll* v jmenném prostoru `Revive.Runtime.InterApplicationCommunication`. Tento jmenný prostor obsahuje tyto třídy a rozhraní.

IClientContract (rozhraní)

Definuje callback metody, které poskytuje klient, tj. (od)nastavení breakpointu a metoda `AllowContinue`, kterou volá server v případě udělení povolení v pokračování.

IServerContract (rozhraní)

Definuje metodu `RequestToContinue`, která je volána klientem a slouží k předání informace o tom, jaký vrchol je laděn.

ServerData (třída)

Reprezentuje data posílaná serverem ke klientovi. Obsahuje klíč vrcholu, který má povolení k pokračování. V budoucnu může obsahovat další informace sloužící například ke změně parametrů instance třídy `Task` za běhu.

ClientData (třída)

Reprezentuje data posílaná klientek k serveru. Obsahuje klíč instance třídy `Node`, jejíž `Task` je laděn a `Status`, který `Task` vrátil. Dále obsahuje instanci třídy `EntityDebugData`.

EntityDebugData (třída)

Tato třída reprezentuje specifické informace pro entitu, která ji může vytvořit. Při ladícím volání je klientské třídě předána entita a `DebugableTask` (o této třídě viz dále). Klient zavolá na entitě `GetDebugData` aby získal `EntityDebugData` a ty vloží do `ClientData`.

Účelem `EntityDebugData` je předat informace specifické pro danou entitu, neboť Behavior trees v editoru jsou definované pro typy entit a ne pro konkrétní instance. Data můžou být například identifikátor entity, zdraví entity, nesené předměty apod. Uživatel by pak nevěděl k jaké konkrétní entitě zvýrazněný vrchol během ladění patří. Tato třída současně obsahuje čtyři textové properties, jedno na nastavení jména entity a zbylé na definování případných dalších informací.

Client (třída)

Tato třída implementuje `IClientContract` a předává všechna volání třídě `Debug`.

Debug (statická třída)

Tato třída obsahuje samotnou logiku volání. Důležité jsou dvě metody

- `RequestToContinue`. Obstará volání do editoru. Tuto metodu volají instance třídy `DebugableTask` a je třeba, aby byla snadno dostupná a použitelná. Proto byl zvolen způsob udělat třídu `Debug` statickou. Jelikož statické třídy nemůžou implementovat rozhraní, bylo třeba udělat zvlášť třídu implementující rozhraní (`Client`), která předá volání třídě `Debug`.

- `Update`. `Debug` poskytuje vlastní funkci `Update`, která zajišťuje aktualizaci chování entit. Tato metoda je volána v `Update` metodě třídy `RuntimeManager` v případě použití ladící knihovny. Instance třídy `RuntimeManager` tak deleguje provádění aktualizace entit na třídu `Debug`. Je to z toho důvodu, že `RuntimeManager` neví nic o nastavení breakpointů apod. a proto se o aktualizaci stará `Debug`.

Blíže bude práce s touto třídou (a také `DebugableTask`) rozebrána v podkapitole představující detailně průběh ladění 4.4.6.

Server (třída)

Tato třída není jako jediná obsažena v *Revive.dll*, ale je obsažena v knihovně, kterou užívá editor *Revive.Editor.dll*. Třída je umístěna ve jmenném prostoru `Revive.Editor.InterApplicationCommunication`.

Hlavní účel třídy `Server` je podporovat ladění, neboť implementuje `IServerContract`. K vysvětlení jak toto rozhraní implementuje je třeba znát ViewModel (z MVVM) a proto je vysvětleno ve zvláštní podkapitole 5.3.8.

Teď když známe princip ladění, koukneme se na problém identifikace vrcholů.

4.4.3. *Jak identifikovat vrcholy*

Připomeňme, že `Node` je vrchol použitý hrou a `Representant` je vrchol v editoru (ve vrstvě Model z MVVM). `Node` i `Representant` musí mít stejný unikátní klíč, abychom dokázali jednoznačně určit, který `Representant` odpovídá instanci třídy `Node`, dělající callback a naopak. Otázkou je jak vytvořit unikátní klíč. Řešením je dvojúrovňový klíč, kde první úroveň je název entity a druhá úroveň bude obyčejné číslo pro rychlé hledání. Tato kombinace je unikátní. Takto zvolený klíč má výhody oproti například jen samým číselným kódům.

- Je čitelný a pro daný vrchol ihned poznáme vlastníka.
- Pokud je registr pamatující si vrcholy dobře navržen, je možné získat všechny vrcholy dané entity a nemusíme tvořit nadstavbu nad registrem.
- Je možné nechat druhotné vyhledávání podle číselného klíče na objektu vlastnícího vrcholy a mít tak možnost vlastní implementace.

Nutnost klíčů přináší dva problémy. Je nutné vytvoření registru, kde bude zaregistrován každý vrchol. V případě editoru je tento registr reprezentován třídou `RepresentantRegister`, v případě Runtimu v ladícím režimu `NodeRegister` (v případě užití finální *Revive.dll* registr netřeba, neboť se neladí). Dále už nelze tvořit jen tak instance třídy `Representant` a je třeba použít k tomu specializovanou factory třídu. Pro instance třídy `Node` factory nepotřebujeme, neboť se všechny vytváří v jednu chvíli při publikaci práce z editoru. Vyhledání instancí `Node` pak bylo popsáno v podkapitole zabývající se Runtimem 4.1.3. Zde je popsán i princip registrace instancí `Node` do `NodeRegister`. Proto se teď budeme zabývat třídou pro registraci instancí třídy `Representant` – `RepresentantRegister`.

4.4.4. *RepresentantRegister*

`RepresentantRegister` musí být v aplikaci jen jeden a je dostupný pro všechny třídy zajišťující logiku chodu editoru, konkrétně se jedná o assembly *Revive.Editor.dll* a *Revive.Editor.Application.exe* (*Revive.Editor.Application.exe* je závislá na *Revive.Editor.dll* a proto má také přístup k `RepresentantRegister`). Tato třída je umístěna ve jmenném prostoru `Revive.Editor.Model` (neboť patří do vrstvy Model v MVVM viz kapitola 5) a je obsažena v assembly *Revive.Editor.dll*. Jelikož se klíč instancí třídy `Representant` skládá ze dvou částí, je vhodné vytvořit třídu, hledající jen podle textového klíče a třídu hledající podle číselného klíče (takto například můžeme lehce změnit logiku hledání podle čísla při zachování logiky hledání podle textu). Schopnost zacházet s číselnými klíči reprezentuje rozhraní `IRepresentantKeyContainer`.

IRepresentantKeyContainer (rozhraní)

Reprezentuje schopnost zacházet s instancemi `Representant` podle číselného klíče, tj. registrování, odregistrování, hledání a vrácení všech instancí.

Rozhraní `IRepresentantKeyContainer` implementuje třída `DefaultKeyContainer`.

DefaultKeyContainer (třída)

K ukládání instancí třídy `Representant` používá pole. Tato třída přistupuje k registraci tímto způsobem:

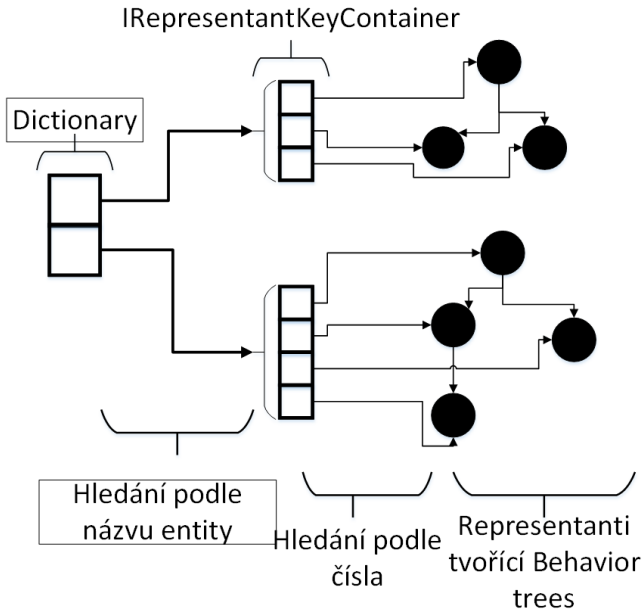
- Jakmile je jednou zaregistrována instance třídy `Representant`, její číselný klíč se už nikdy nezmění.

To zabrání aby měly instance třídy `Node` jiné číselné klíče než instance třídy `Representant`, které si navzájem odpovídají. Interně je tedy použito pole, kde se pamatují „díry“ vzniklé po mazání a ty se vyplňují jako první při přidávání.

S textovými klíči pak zachází přímo `RepresentantRegister`.

RepresentantRegister (třída)

Nabízí metodu pro vyhledání instance třídy `Representant` podle klíče. Dále umožňuje registrování instance `Representant` a registrování (odregistrování) `IRepresentantKeyContainer`. Interně obsahuje pro skladování instancí třídy Representant `Dictionary<string,IRepresentantKeyContainer>`. Pokud při registraci instance `Representant` neobsahuje `Dictionary` pod textovým klíčem `IRepresentantKeyContainer`, použije instanci `DefaultKeyContainer`. Při hledání instance třídy `Representant` je nejprve nalezen `IRepresentantKeyContainer` a ten je pomocí číselného klíče požádán o nalezení konkrétní instance. Znázorňuje to Obrázek 16.



Obrázek 16: Schéma hledání instance třídy Representant

4.4.5. *Centrální tvorba vrcholů*

Nutnost registrace vrcholů s sebou nese nutnost vytvoření factory třídy obstarávající tvorbu instancí třídy `Representant`. Pro instance třídy `Node` třídu nepotřebujeme, neboť každý `Reprezentant` ví, jaký `Node` je s ním spojený. Naopak v editoru tvoříme `Representanty` automaticky na základě parametrů na mnoha místech a je lepší mít jednu centrální třídu, která vytváří `Representanty` na základě parametru a obstará kriticky důležitou registraci. Takto můžeme centrálně řídit tvorbu a mít jistot, že jsou všechny instance zaregistrovány v `RepresentantRegister`. Tato factory třída se jmenuje `RepresentantFactory`.

RepresentantFactory (statická třída)

Hlavním účelem je vytvářet instance třídy `Representant`. Vytvořené instance jsou automaticky registrovány `RepresentantRegister`. Kromě toho poskytuje `RepresentantFactory` metody cílené na aktualizaci stromů, složených z `Representantů`, v případě změn. Změny můžou být:

- Zničení instance třídy `Representant`. Tato metoda přímo nezničí instanci, ale obstará uvolnění reference z `RepresentantRegister` aby mohla být garbage collectována.
- Změna typu, který `Representant` representuje (např. během synchronizace při změně jména třídy, kterou `Representant` zastupuje).
- Změna parametrů[1], které `Representant` obsahuje.

[1] Parametry, které jsou zobrazeny v editoru a jsou definovány pomocí atributů z Revive.dll

- Aktualizace instancí `LookUpRepresentant` (odkazují se na Named tree) při smazání Named tree.

4.4.6. *Runtime při čekání na potvrzení z editoru*

Jedním ze způsobů jak přistupovat ke krokování v ladícím režimu je následující. Entita, na které je nastaven breakpoint, se dostane do `Update` fáze. Runtime zavolá `Update` metodu jejího stromu. Jakmile se dostane nějaký `Task` na řadu, udělá callback do editoru, kde se zvýrazní příslušný `Reprezentant`. Vlákno vykonávající `Update` volání stojí, protože se čeká na potvrzení. Po potvrzení `Task` spustí výpočet (typicky zavolá `Update` metodu na dětech) po jehož dokončení zavolá znova do editoru, tentokrát s informací o tom jak výpočet dopadl. Příslušný `Representant` se zvýrazní a čeká se znova na potvrzení. Tímto způsobem se vykoná výpočet potřebné části Behavior tree a aktualizuje se chování entity.

Takový přístup má výhodu ve své jednoduchosti. Rozdíl mezi `Update` metodou v ladícím a finálním režimu je skoro neznatelný. Je zde jen pár řádků navíc dělajících callback. Na těchto řádcích se zasekne výpočet a netřeba dalších úprav. Nevýhodou je zamrznutí hry. Aby hra nezamrzla a dávala odezvu, bylo by třeba spustit Runtime ve zvláštním vlákně. Toto by přineslo technické komplikace uživatelskému programátorovi. Dále bychom nemohli aktualizovat chování entit, na kterých není nastavený breakpoint (Runtime je zaseknutý a čeká na potvrzení), což není až taková překážka. Hlavním problémem je zmíněná vyšší náročnost na implementování a problémy, které by přinesla nutnost vláknování. Toto je v přímém rozporu s konceptem frameworku Revive (konkrétně cíl C1).

Řešení použité Revive frameworkem je vytvoření speciálního stavu `InDebug`, který je přidán do `enumu Status` a který tak může být vrácen instancemi třídy `Task`. Kromě toho podědíme od třídy `Task` vlastní třídu obsahující položky důležité pro ladění – `DebugableTask`.

DebugableTask (abstraktní třída)

Tato třída dědí od `Task`, zachovává všechny abstraktní metody a je taktéž definovaná v *Revive.dll* v namespace `Revive.BehaviorTreeStructure`. Třídu `Task` obohacuje o 4 důležité properties (použití bude popsáno dále), všechny typu `bool`:

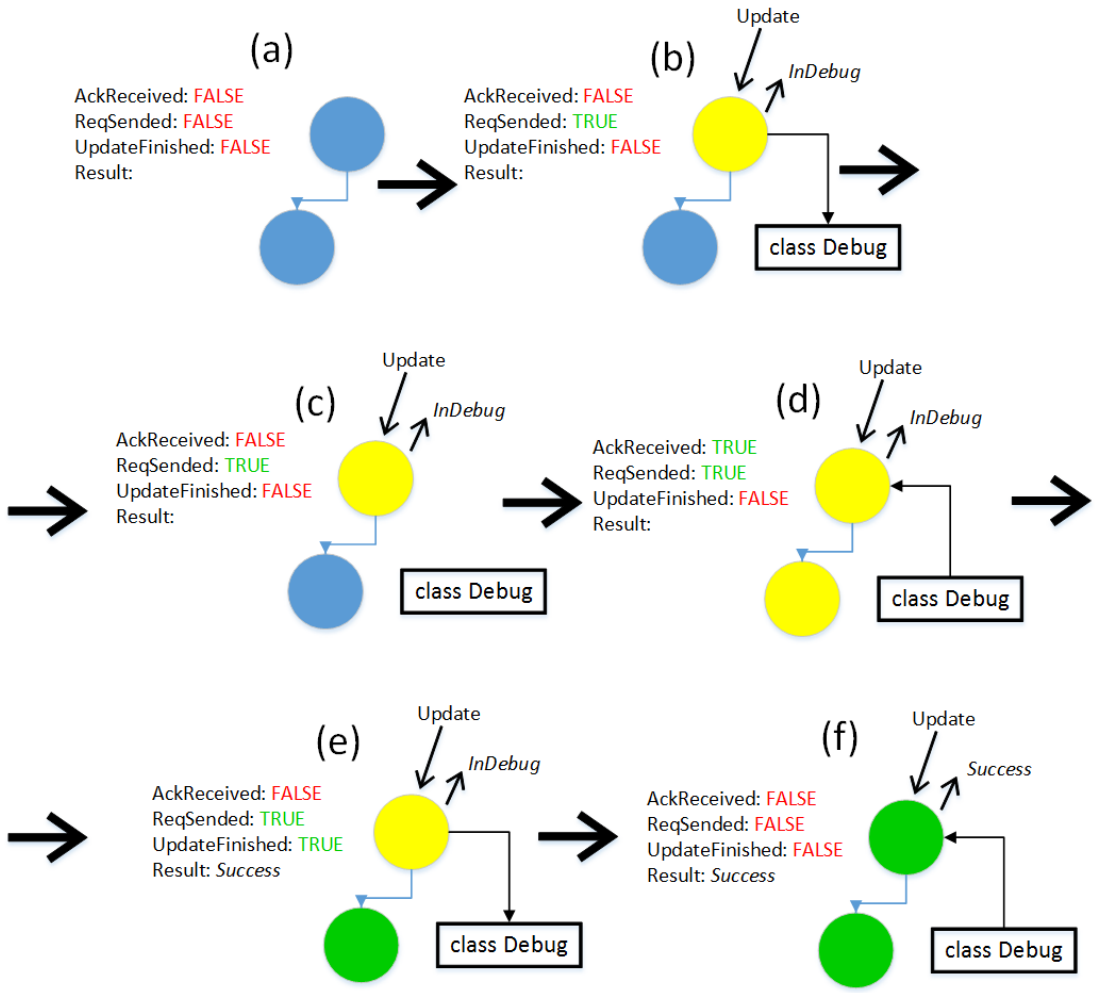
- `AckReceived` – říká, zda jsme dostali potvrzení od editoru.
- `ReqSended` – pamatuje si, zda jsme odeslali požadavek na pokračování ve výpočtu do editoru.
- `UpdateFinished` – indikuje, zda tento `Task` dokončil výpočet.
- `Result` – drží výsledek tohoto `Task`u v případě, že dokončil výpočet.

Kromě této třídy je nutné, aby každý `Task` byl schopný dělat ladící volání. Proto byla vytvořena statická třída `Debug`.

Debug (statická třída)

Tato třída byla představena v podkapitole 4.4.2. Funkcionalita bude uvedena v následujícím příkladu.

Dále si ukážeme jak celý průběh ladění probíhá. Vše shrnuje Obrázek 17.



Obrázek 17: Průběh ladění.

a) **Strom před voláním Update**

b) **Na prvním vrcholu bylo zavoláno Update, ten volá metodu třídy Debug, nastaví příznak a vrací Status.InDebug dokud nedostane potvrzení.**

c) **Task nedostal potvrzení a vraci Status.InDebug.**

d) **Task dostal potvrzení (byl mu nastaven příznak třídou Debug) a volá tak Update metodu syna.**

e) **Volání podstromu proběhlo a Task zná výsledek, nastaví příznak, uloží výsledek a volá Debug.RequestToContinue. Než dostane potvrzení vrací pořád Status.InDebug.**

f) **Task dostal potvrzení, vrací rodiči uložený stav a vynuluje příznaky.**

Až se `Task` dostane poprvé na řadu, zavolá metodu `RequestToContinue` statické třídy `Debug` a nastaví si příznak `ReqSended` na `true`. `Debug` udělá volání metody `RequestToContinue`, kterou poskytuje `IServerContract` a předá serveru `ClientData`, které editor použije k zobrazení informací o ladění (toto je probíráno v podkapitole 5 o MVVM). Dále si `Debug` uloží `Task`, který dělal volání a entitu vlastnící `Task`. Dokud nedostane `Task` potvrzení, nebude provádět žádný výpočet a v každém `Update` cyklu vrátí `Status.InDebug`.

V případě, že uživatel dá povolení pokračovat, propaguje toto povolení `Server` (implementuje `IServerContract`) na straně editoru ke klientovi (třída `Client`, která volání dál deleguje na třídu `Debug`) spolu s instancí `ServerData`. `Debug` po obdržení potvrzení nastaví instanci třídy `Task`, která je laděna, příznak `AckReceived` na `true` a tak v dalším `Update` cyklu může pokračovat. Vše se opakuje s podstromem instance třídy `Task` (nazvěme tento konkrétní `Task` A), které bylo uděleno pokračovat.

Po dokončení výpočtu podstromu ví instance třídy `Task` (instance pojmenovaná A) jaký vrátí stav rodiči. Před vrácením je ale nutno zopakovat celý proces volání (aby uživatel viděl jaký byl vrácen stav a mohl udělit povolení k pokračování). `Task` A tedy uloží výsledek volání do proměnné `Result`, zavolá `Debug.RequestToContinue`, nastaví `AckReceived` na `false` a čeká na povolení. V případě udělení povolení pak propaguje stav uložený v proměnné `Result` nahoru rodiči a vynuluje příznaky.

Takto řešený koncept ladění nevyžaduje od programátorského uživatele frameworku žádnou práci. Z hlediska hry se entity také chovají normálně a jsou pravidelně aktualizovány (ačkoliv některé nedělají nic).

Nevýhodou je značně složitější implementace `Nodů`. Jelikož se nám výpočet „nezasekne“ na určité řádce kódu, je nutné si držet příznaky a pomocí podmínek procházet výpočtem instance třídy `Task`. Zároveň je třeba počítat s novým stavem. Celý `Update` cyklus v ladícím režimu je tedy pomalejší než při použití finální verze knihovny Revive.dll. Dále vyvstává problém co dělat s entitami, na kterých není nastavený breakpoint a dostanou se do `Update` fáze. Pokud nemáme dokončené ladění stromu nějaké entity nesmíme tuto novou entity aktualizovat. Toto není z hlediska technického, ale uživatelského. Pokud bychom aktualizaci entit na kterých je nastaven breakpoint povolili, uživatel by v editoru vyděl potenciálně mnoho entit žádajících o potvrzení. Toto není moc přehledný přístup a proto je lepší mít nejvýše jednu entitu v ladícím režimu v jednu konkrétní chvíli. Proto si třída `Debug` pamatuje jen jednu entitu, která je laděna a v případě požadavku na ladění entity, na které je nastavený breakpoint (a nejedná se o onu uloženou) s entitou nedělá nic. V případě entity bez breakpointu ji normálně aktualizuje.

Tím jsme dokončili část dokumentace zabývající se částmi spojené hlavně se sdílenou knihovnou a teď probereme druhou hlavní část dokumentace – editor.

5. Vývojová dokumentace B – editor a synchronizace

5.1. MVVM

Jelikož editor frameworku využívá technologie WPF je vhodné použít návrhový vzor pro tuto konkrétní technologii – MVVM (11).

MVVM je zkratka pro Model – View – ViewModel. Toto jsou tři základní komponenty návrhového vzoru. Hlavním účelem MVVM je oddělení dat a vzhledu aplikace.

Model

Je vrstva obsahující data aplikace. Můžeme si ji představit například jako databázi s daty aplikace. Model neví o ViewModel ani o View.

ViewModelu

Hlavním účelem je zveřejnění dat z Modelu pro View takovým způsobem, aby je View mohl lehce konzumovat. Kromě dat zveřejňuje ViewModel i funkcionalitu užitím commands a binding. Dále obsahuje veškerou logiku obstarávající interakce s uživatelem a propaguje změny do Modelu. ViewModel si není vědom existence View.

View

Je vrstva, která zajišťuje zobrazení uživateli. Jako `DataContext` má nastaven ViewModel, a tak může konzumovat jeho funkcionalitu a data pomocí binding a commands. Pomocí data templatů definuje vzhled pro třídy z ViewModelu. View vrstva by neměla obsahovat code behind a měla by být napsána čistě v XAMLu.

V následujících podkapitolách popíšeme jednotlivé vrstvy.

5.2. Model

V této vrstvě se odehrává drtivá část veškeré logiky editoru. Zahrnuje načítání a parsování definujícího souboru, komunikaci s hrou během ladění a obsahuje veškerá data pro editor a metody pro práci s těmito daty. Model je v celé aplikaci jen jeden a je zveřejněn skrz statickou třídu `ModelManager`, která drží referenci právě na tento model. Samotný Model pak reprezentuje třída `ModelLayer`. Zveřejnění této třídy skrz statickou třídu umožňuje snadný přístup k základním částem z celého programu. Dále je třeba mít zvlášť třídu pro Model (a ne jen statickou třídu, která by byla Modelem) kvůli Synchronizaci (viz podkapitola 5.5).

Všechny třídy spojené s Modelem jsou umístěny ve jmenném prostoru `Revive.Editor.Model` a obsahuje je assembly *Revive.Editor.dll*. Pojďme se nejdříve podívat na `ModelLayer` a `ModelManager`.

ModelLayer (třída)

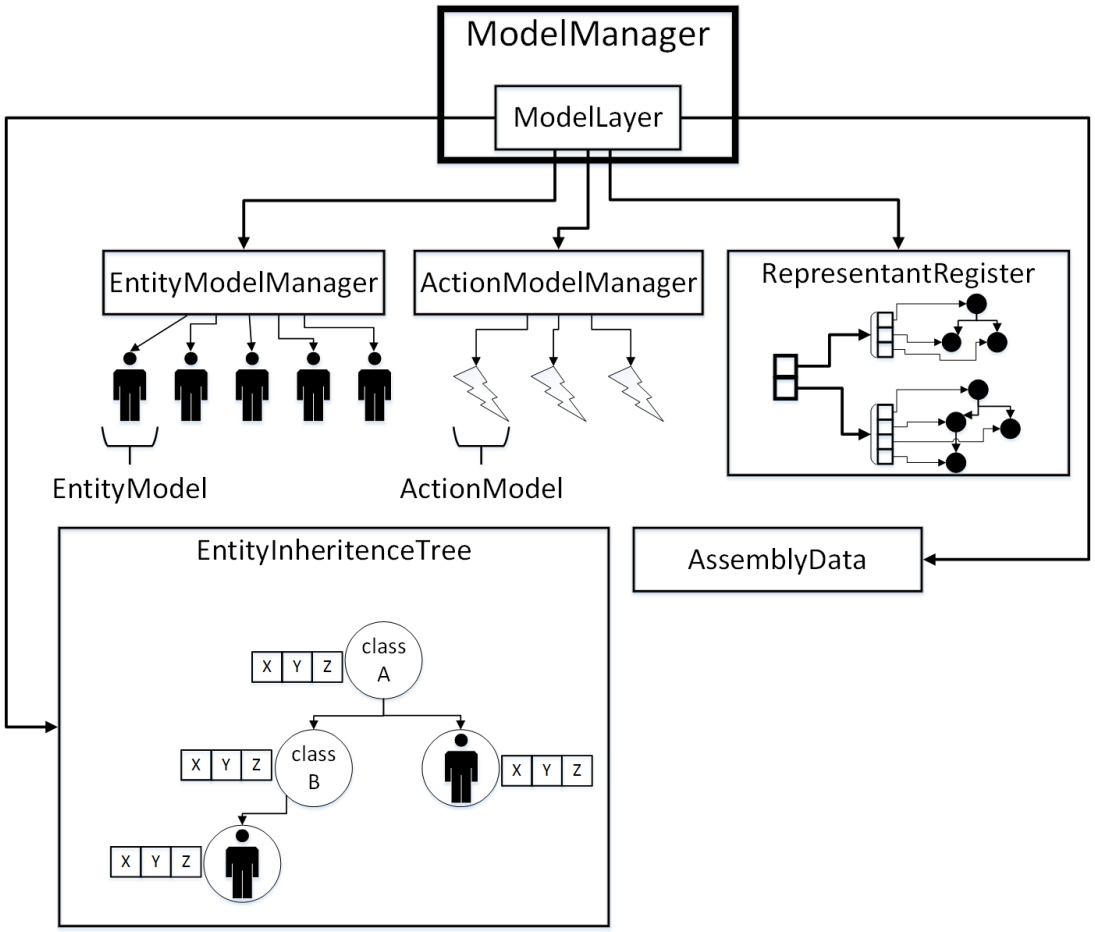
Reprezentuje Model a drží reference na všechny důležité třídy patřící do Modelu tj. `EntityModelManager`, `RepresentantRegister`, `EntityInheritenceTree`, `ActionModelManager`, `AssemblyData` (třídy budou rozebrány dále).

Při vytváření je možné říci, zda má být použit Model k synchronizaci, či se jedná o Model aplikace. `ModelLayer` se vytváří z definující assembly a v případě úspěchu vrací `AssemblyData` reprezentující data z definující assembly. V případě neúspěchu je vrácen `null` a seznam chyb zabraňující vytvoření v `out` parametru.

ModelManager (statická třída)

Drží referenci na jednu instanci třídy `ModelLayer` a zveřejňuje hlavní třídy zmíněné v `ModelLayer`. `ModelManager` obsahuje metody pro tvorbu shodné s konstruktory `ModelLayer`. Pomocí těchto metod vytvoří `ModelLayer`. Dále nabízí metodu na výměnu Modelu.

Jak vypadá Model ilustruje Obrázek 18. Třídy z obrázku budou dále rozebrány.



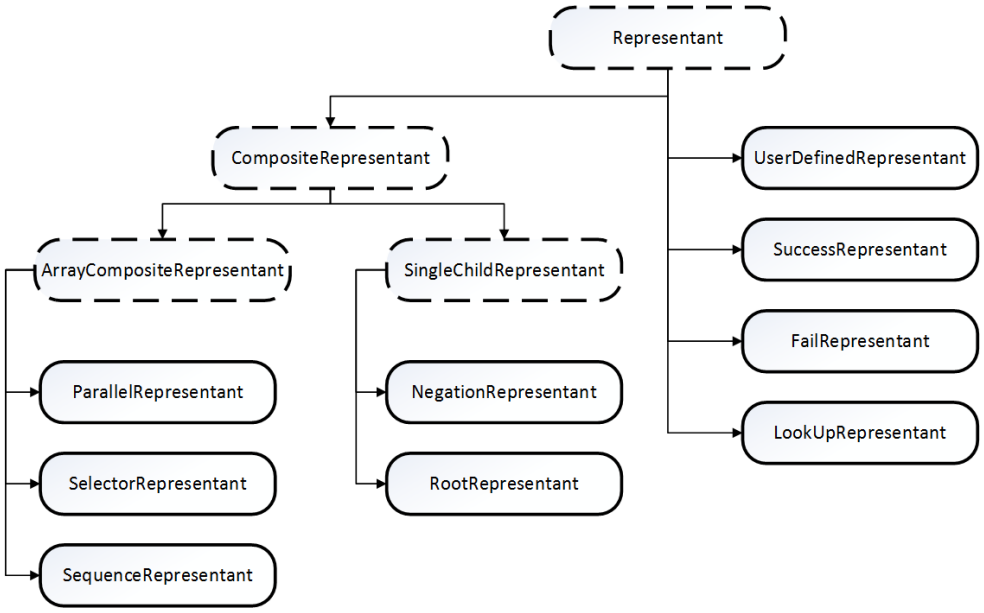
Obrázek 18: Model

Nejdříve se podíváme na tři základní třídy, které tvoří nejmenší stavební bloky Modelu – `EntityModel`, `ActionModel` a `Reprezentant`.

Representant (abstraktní třída)

Tvoří vrchol stromu jak již bylo zmíněno. Hlavním účelem (mimo toho být základním stavebním prvkem Behavior tree) je tvořit `Node` během publikování práce (proces kdy jsou exportovány stromy vytvořené v editoru do podoby použitelné Runtimem). Dále `Representant` drží `NodeParameters`. Do `NodeParameters` jsou zapisovány informace o tom jaký parametr má jakou hodnotu. Změny do ViewModelu propaguje pomocí eventů. Jedná se o to, když přijde `Representant` na řadu během ladící fáze. V tom případě spustí event, který říká, že je `Representant` zrovna laděn. Tento event odchytí `RepresentantViewModel` (jak se chová ViewModel viz další podkapitola).

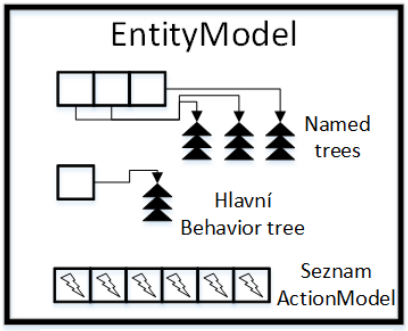
Potomci třídy Representant jsou v namespace Revive.Editor.Model.Representants a dědičnost znázorňuje Obrázek 19.



Obrázek 19: Dědičnost Representantů. Abstraktní třídy jsou ohraničeny přerušovanou čarou.

EntityModel (třída)

Zastupuje entitu, jméno entity drží v textové položce. Hlavním účelem je držet instanci třídy `Representant`, která je kořenem Behavior tree. `EntityModel` si udržuje reference na instance třídy `ActionModel`, která reprezentují akce entity, kterou `EntityModel` reprezentuje. Kromě toho má i seznam Named trees (zmíněno v podkapitole 4.3.1). `EntityModel` ilustruje Obrázek 20. Název třídy, kterou `EntityModel` reprezentuje, je uložen v textové Property `ClassName`. Každá instance `EntityModel` si také drží reference na všechny instance `ActionModel`, které může použít. Ty získá z `AssebmlyData` během konstrukce.



Obrázek 20: EntityModel

ActionModel (třída)

Reprezentuje akci. Důležité je si uvědomit, že reprezentuje akci obecně a ne akci konkrétní entity ve stromě. Akci ve stromě reprezentuje `Representant` vytvořený právě instancí třídy `ActionModel`. `ActionModel` si drží parametry (seznam instancí třídy `ParameterWrapper` viz podkapitola 4.1.1) akce získané z definující assembly, ze kterých tvoří `NodeParameters` pro `Representanta`.

Název třídy, kterou `ActionModel` reprezentuje, je uložen v textové property `ClassName`.

Dále si popíšeme základní části Modelu. Každá část se vyskytuje v Modelu právě jednou a Model obstarává jejich inicializaci při tvoření modelu z definujícího souboru. Tyto části už byly znázorněny a popisuje je Obrázek 18.

RepresentantRegistr (třída)

Registr již byl popsán v podkapitole 4.4.4. Drží reference na všechny instance třídy `Representant` v aplikace.

ActionModelManager (třída)

Tato struktura drží reference na všechny `ActionModely` (pomocí `Dictionary`) a nabízí jejich vyhledání na základě jejich jmen (klíč je `ClassName`, takto lehce dohledáme `ActionModel` k třídě, kterou reprezentuje). Toto je důležité při tvorbě entit, kdy jsou v `AssemblyData` obsaženy jen názvy akcí k příslušné entitě a my potřebujeme postupně získávat `ActionModely` pro entity. Zároveň je nutné mít všechny akce pohromadě kvůli synchronizaci (viz následující podkapitola po MVVM 5.5) a ViewModelu (když staví ViewModel akce, je nutné je hledat).

EntityModelManager (třída)

Tato struktura drží všechny instance třídy `EntityModel` pod jejich jmény a nabízí jejich hledání. Pro ukládání používá `EntityModelManager` instanci třídy `Dictionary<string,EntityModel>`, kde je klíčem `ClassName` property instancí třídy `EntityModel`. Entity je nutné držet centralizovaně kvůli požadavkům ViewModelu (při stavění je nutno hledat entity) a synchronizaci (viz následující podkapitola 5.5).

InheritenceTree (třída)

Tato struktura si strom dědičnosti, kde jsou všechny třídy, které buď definují entitu, nebo jsou předky třídy definující entitu. Skládá se z `InheritenceNode`, na kterých jde definovat Named tree, což je strom, na který se lze odkázat ze všech vrcholů v podstromu určeným daným `InheritenceNode` (včetně). Blíže již struktura byla popsána v podkapitole 4.3.1.

Toto byly základní části Modelu, následuje podkapitola zabývající se ViewModelem.

5.3. ViewModel

ViewModel je v aplikaci také jen jeden, je reprezentován třídou `ApplicationViewModel` (bude rozebrána v podkapitole 5.4.3 po dokončení částí ViewModelu) a konkrétní instance je zveřejněna skrz statickou proměnnou `CurrentViewModel` definovanou na třídě `ApplicationViewModel`, aby byl přístupný všem částem aplikace. Je vytvořený na základě Modelu a `AssemblyData` z definující assembly. Hlavním účelem je vytvoření tříd nad Modelem, které tak umožní použití Modelu ve View vrstvě a zajištění veškeré funkcionality aplikace. Tyto třídy typicky obsahují třídy Modelu a Properties s commandama pro možnost vytvoření bidingu ve View a možnost konzumace funkcionality ViewModelu. Nejdříve se budeme zabývat základním stavebním kamenem Behavior tree ve ViewModel vrstvě – třídou `RepresentantViewModel`. Jako první si probereme řešení jak zobrazit parametry a pak probereme samotnou třídu `RepresentantViewModel`.

5.3.1. *Zobrazení parametrů v editoru*

Celý návrh práce s parametry musí splňovat jednu důležitou podmínku a to být dobře rozšiřitelný. Je důležité počítat s tím, že se objeví náročnější uživatel, kterému nestačí vytvořené parametry. Proto nemůžeme pro každý parametr bez rozmyšlení vytvořit speciální třídu.

Druhá důležitá věc je skrytí určitých parametrů na základě hodnot jiných parametrů (tento požadavek byl formulován v analýze v 3.10). Proto potřebujeme třídu, která bude spravovat všechny parametry daného vrcholu a zviditelní patřičné parametry.

Všechny níže popsané třídy jsou v assembly Revive.Editor.dll v namespace `Revive.Editor.ViewModel.ParametersVM`. Pozn. při čtení textu je dobré se dívat na přiložený Obrázek 21 na konci této kapitoly na str. 49, který znázorňuje strukturu a logiku volání metod.

Nejdříve je třeba vytvoření základní třídy pro parametr `ParameterViewModel`.

ParameterViewModel (třída)

Tato třída poskytuje metody na přidání vizuální závislosti a na zkontrolování svojí viditelnosti na základě svých závislostí na hodnotách jiných parametrů. Každý ViewModel parametru, zobrazeného v editoru, musí dědit od této třídy. Dále zveřejňuje metody `GetValue` a `SetValue` pro nastavení hodnoty daného parametru.

Dále je třeba hlavní třídy `ParameterManager`.

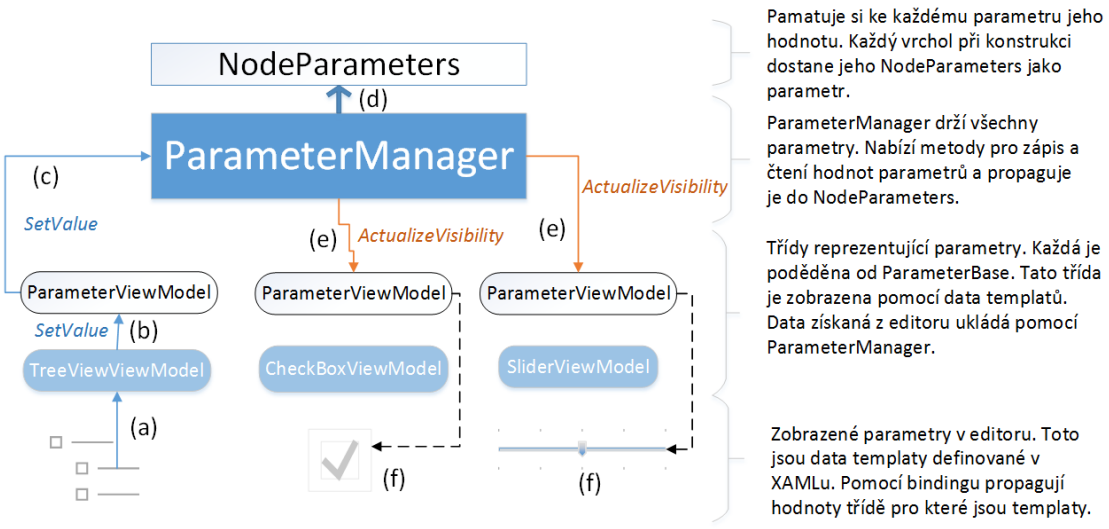
ParameterManager (třída)

Vlastní instanci `NodeParameters` k ukládání hodnot parametrů a zveřejňuje metody pro zápis (`SetValue`) a získání hodnoty určitého parametru (`GetValue`), kde interně zapisuje/čte z `NodeParameters`. Každý `RepresentantViewModel` vlastní jednu instanci této třídy. Dále `ParameterManager` udržuje seznam všech instancí `ParameterViewModel` (pro danou instanci RepresentantViewModel) reprezentující parametry. Při zápisu pak požádá instance tříd ParameterViewModel, aby upravili svou viditelnost.

Každý `ParameterViewModel` dostává `ParameterManager` při své konstrukci a k zápisu a čtení používá právě `ParameterManager`. Změny na případnou viditelnost jsou tak propagovány všem parametrům. `ParameterViewModel` obsahuje Property `IsVisible`, která indikuje viditelnost a při požádání o kontrolu své viditelnosti ji patřičně upraví. `IsVisible` používá View od daného parametru a pomocí binding tak mění svojí viditelnost.

Konkrétní parametry se pak implementují vytvořením vlastní třídy poděděné od `ParameterViewModel`. Zápisy a čtení hodnot parametrů jsou prováděny skrz metody poděděné od třídy `ParameterViewModel GetValue` a `SetValue`, a tak se není třeba starat o viditelnost (tu automaticky aktualizuje `ParameterViewModel`). Po vytvoření třídy reprezentující parametr je třeba dodat vzhled. Ten se provede vytvořením data templatu v XAML souboru. Příklad jak se propaguje zápis, aktualizuje viditelnost a

vypadá celá struktura ilustruje Obrázek 21. Tuto strukturu má každý `RepresentantViewModel`.



Obrázek 21: Příklad propagace zápisu a viditelnosti. Přerušované šipky znázorňují aktualizaci díky binding.

a) **Komponenta TreeView je změněna uživatelem a tato změna je propagována ViewModelu komponenty - TreeViewViewModel.**

b) **TreeViewViewModel získá novou hodnotu a uloží ji pomocí SetValue poděděné od ParameterViewModel.**

c) **ParameterViewModel při zavolání SetValue zavolá SetValue instance třídy ParameterManager**

d) **ParameterManager uloží hodnotu do instance NodeParameters.**

e) **ParameterManager zavolá ActualizeVisibility na všech instancích ParameterViewModel.**

f) **Tyto instance zkontrolují viditelnost a případně upraví IsVisible. Tato změna je promítnuta do viditelnosti komponent pomocí binding.**

5.3.2. *RepresentantViewModel*

Tato třída slouží jako `DataContext` pro vrcholy Behavior trees zobrazených v editoru. Drží referenci na příslušnou instanci třídy `Representant`, pro kterou je ViewModelem. Dále musí obsahovat již zmíněnou strukturu pro parametry `ParameterManager`. Kromě toho je zveřejněna booleovskou proměnná `IsCurrentlyDebugged`, říkající zda je vrchol laděn. Nastavením této proměnné signalizujeme View vrstvě, že je tento vrchol laděn a ta se podle toho může zařídit (například zvýraznit vrchol). Pro nastavení `IsCurrentlyDebugged` slouží statická metoda `HighlightRepresentant`. Ta nastaví `IsCurrentlyDebugged` naposled laděné instance `RepresentantViewModel` (pokud taková je), kterou si drží ve statické proměnné a do této proměnné přiřadí nově laděnou instanci (spolu s nastavením `IsCurrentlyDebugged`). Jelikož je v jednu chvíli laděn jen jeden vrchol, je toto řešení přehledné a rychlé.

`RepresentantViewModel` také obstarává handler pro event. od instance třídy `Representant`, pro kterou je ViewModelem. Chování závisí na typu události. Jsou implementované dvě důležité události:

- Ladící volání. V tomto případě handler zavolá `HighlightRepresentant` a uloží Status, který mu Representant vrátil.
- Smazání Named tree. V tomto případě handler oznámí uživateli, že smazání Named tree ovlivnilo LookUp vrchol příslušné entity.

Toto byl základní popis `RepresentantViewModel`, s vrcholy v editoru budeme dále chtít:

- Hýbat. Pro hýbání nabízí WPF třídu zvanou `Thumb`. Tato třída má event zvaný `DragDelta`. Tento event je zavolán pokud se pokusíme táhnout `Thumb` a tím ho přemístit. Jako parametry dostaneme horizontální a vertikální změnu pozice myši, pomocí kterých patřičně upravíme pozici `Thumb`. Abychom zachovali MVVM bylo by ideální mít handler pro `DragDelta` ve ViewModelu. Toto bohužel nejde a hýbání nejde vyřešit přes commandy. Proto je hýbání řešeno pomocí code behind pro View vrcholu (třída `RepresentantView`) a tvoří tak výjimku z pravidla. Hlavní účel MVVM je udržovatelnost a možnost změny. Toto naším řešením neztrácíme. Možnost být hýbán a obsahovat souřadnice X a Y reprezentuje rozhraní `IDraggable`, které `RepresentantViewModel` implementuje.
- Navzájem je spojovat a rozpojovat. Pro propojení dvou bodů potřebujeme dvě instance třídy, které si budou pamatovat souřadnice (tato třída je `ConnectPoint`) a třídu, která bude reprezentovat hranu mezi nimi (`Edge`).

IDraggable (rozhraní)

Rozhraní je definováno v *Revive.Editor.dll* ve jmenném prostoru `Revive.Editor.ViewResources`. Toto rozhraní rozšiřuje `INotifyPropertyChanged` (definované v `System.ComponentModel`) o dvě properties X a Y udávající souřadnice objektu implementující toto rozhraní. Tyto properties musí být implementovány tak, aby při změně použily `INotifyPropertyChanged` k oznámení nových souřadnic.

ConnectPoint (třída)

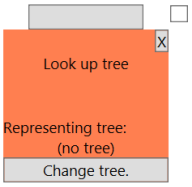
Tato třída je definována ve jmenném prostoru `Revive.Editor.ViewModel.RepresentantVMParts` a pamatuje si souřadnice X a Y.

Edge (třída)

Tato třída je definována ve jmenném prostoru `Revive.Editor.ViewModel.RepresentantVMParts` a pamatuje si souřadnice dvě instance třídy `ConnectPoint`.

Každý `RepresentantViewModel` má sadu `ConnectPoint`, přičemž jeden bude sloužit na propojení s rodičem a ostatní na propojení s dítětem (pokud se jedná o vrchol, který může mít děti).

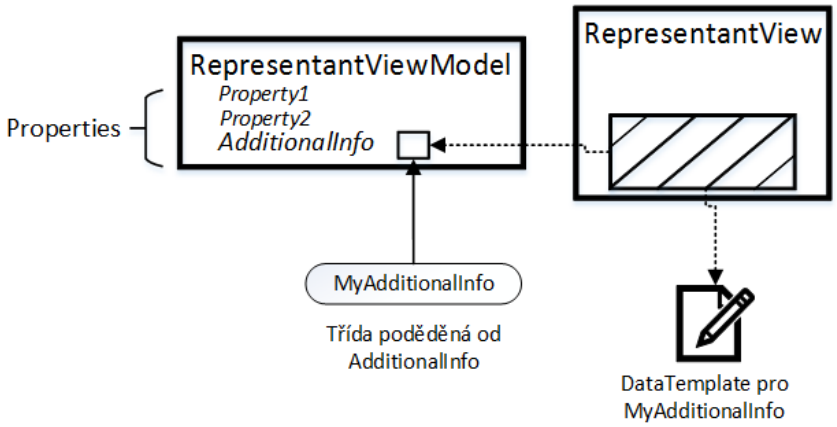
Kromě zmíněných možností může nastat případ, kdy potřebujeme zveřejnit pro View vrstvu netriviální informace. Například LookUp vrchol (Obrázek 22) ve svém View obsahuje tlačítko, které po kliknutí spustí dialog pro vytvoření Named tree. Zároveň vrchol zobrazuje na který Named tree se v současné chvíli odkazuje. Proto bychom chtěli aby:



Obrázek 22: LookUp vrchol.

- Každý `RepresentantViewModel` měl možnost vytvoření netriviální funkcionality.
- Nechceme, aby `RepresentantViewModel`, který nepotřebuje dodatečnou funkcionalitu, byl zatížen výsledným návrhem.

Proto je vytvořena třída `AdditionalInfo`. `RepresentantViewModel` má Property, kam jde přiřadit instance `AdditionalInfo`. Pro implementaci vlastní funkcionality stačí vytvořit vlastní třídu, poděděnou od `AdditionalInfo`. V této třídě si vytvoříme vlastní logiku a přiřadíme ji do zmíněné property. Toto přiřazení obstarává `TreeCreator` (konkrétně metoda `CreateRepresentant`, která má na starost tvorbu `RepresentantViewModel`, viz dále). Zobrazení třídy poděděné od `AdditionalInfo` pak obstará `DataTemplate` definovaný ve View vrstvě. Koncept zachycuje Obrázek 23.



Obrázek 23: AdditionalInfo. Čárkované šipky značí binding. Programátor definuje třídu odvozenou od AdditionalInfo a DataTemplate pro tuto třídu. RepresentantView automaticky zveřejňuje třídu z Property AdditionalInfo a DataTemplate je aplikován automaticky díky tomu, že má DataType nastavený na MyAdditionalInfo.

Tímto byl dokončen popis třídy RepresentantViewModel. Následuje dokumentaci dalších tříd patřících do ViewModelu.

5.3.3. *Akce ve ViewModelu*

Každý `RepresentantViewModel` je vyroben třídou `ActionViewModel`.

ActionViewModel (třída)

Třída je definována ve jmenném prostoru `Revive.Editor.ViewModel`. Hlavním účelem této třídy je zpřístupnit `ActionModel`, zveřejnění commandu pro tvorbu `RepresentantViewModel` a properties obsahující vizuální data tj. jméno akce, které má být zobrazeno v editoru a kategorie, do které akce patří. Vizuální data

nejsou předávány přímo v konstruktoru, ale je pro ně vytvořena speciální třída `ActionVisualData`, která je předána v konstruktoru. Toto řešení je výhodnější, neboť v případném přidání vizuálních dat (což je v budoucnu při případném rozšíření pravděpodobné) stačí upravit jednu třídu a konstruktor `ActionViewModel` zůstane nezměněn. Instance třídy `ActionViewModel` jsou seskupovány do kategorií odpovídající definicím kategorií pomocí atributů.

ActionVisualData (třída)

Definována ve jmenném prostoru `Revive.Editor.ViewModel`. Obsahuje informace o vizuálních datech pro `ActionViewModel`.

Category (třída)

Definována ve jmenném prostoru `Revive.Editor.ViewModel`. Obsahuje seznam instancí `ActionViewModel`. Instance `ActionViewModel` jsou seskupovány do kategorií podle definice atributu k třídám poděděných od `UserNode` v definující assembly.

Všechny instance `ActionViewModel` jsou umístěny v centrální třídě `ActionViewModelManager`.

ActionViewModelManager (třída)

Tato třída drží reference na všechny instance `ActionViewModel` a umožňuje hledání podle jména. Tato funkce je důležitá při stavbě ostatních částí aplikace, kde je třeba nahrávat akce (např. `TreeCreator`) a zároveň obstará tvorbu všech intancí `ActionViewModel` ze všech instancí `ActionModel` v Modelu.

`ActionViewModel` tvoří instance `RepresentantViewModel`, nicméně `ActionViewModel` není použit přímo k bindingu v hlavním okně. Třída, s kterou převážně interaguje uživatel je `TreeCreator`. Tuto třídu rozebereme v následující podkapitole.

5.3.4. *ViewModel pro editační plochu a menu s akcemi*

Hlavní část aplikace, tj. editační plocha a seznam dostupných akcí pro právě zobrazenou entitu, používá jako `DataContext` třídu `TreeCreator`.

TreeCreator (třída)

Tato třída je umístěna ve jmenném prostoru `Revive.Editor.ViewModel` a je schopná poskytovat funkcionalitu k vytváření Behavior tree. Na tvorbě Behavior trees se na úrovni Modelu podílejí instance `EntityModel` nebo `InheritenceNode`. Hlavním účelem `TreeCreator` je zveřejnění kolekce `RepresentantViewModel`, `Edge`, kategorií s `ActionViewModel`, Named trees a jméno pro zobrazení v editoru. Všechny tyto kolekce jsou pak použity View vrstvou pro zobrazení.

Pro tyto účely by stačilo vytvoření rozhraní, řekněme `ITreeCreator`, které by definovalo tento kontrakt. Výhoda by byla, že by toto rozhraní mohla implementovat libovolná třída. My ale potřebujeme zveřejnit i commandy, které si sami vytvoříme. Dále potřebujeme eventy pro:

- žádost o zavření `TreeCreator`

- informování, že byl nastavený breakpoint
- upozornění o žádost o zobrazení

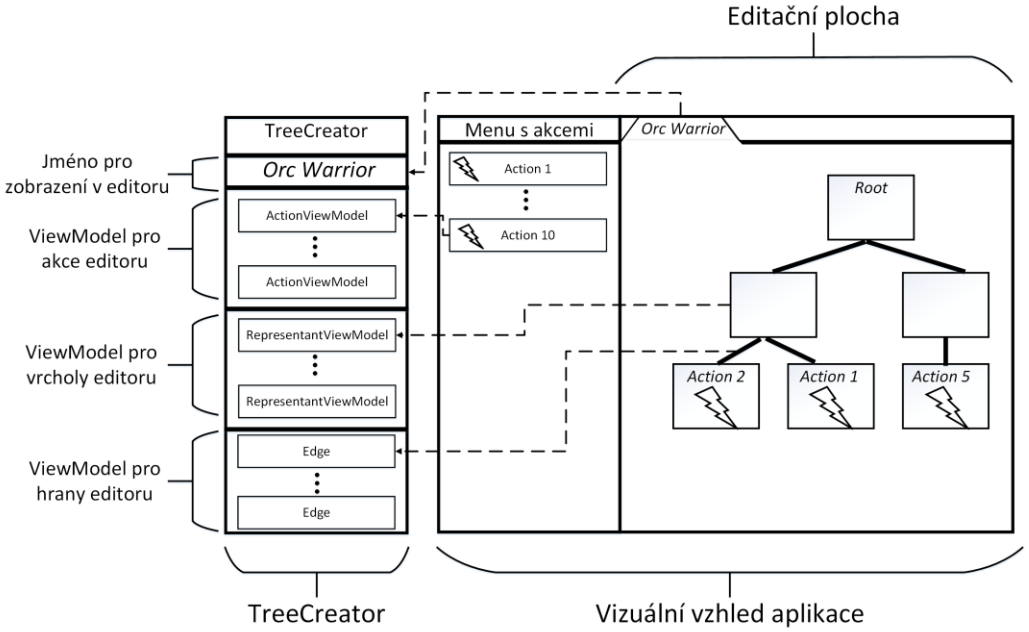
Žádost o zavření je použita hlavním ViewModelem (`ApplicationViewModel`) pro odstranění `TreeCreator` z aktivních `TreeCreators`.

Informování o nastavení breakpointu je použito serverem.

Žádost o zobrazení je použito `EntityCategoriesViewModel` (viz dále) pro zobrazení či zrušení zobrazení v editoru.

Poslední důležitou věcí, kterou `TreeCreator` obstarává je tvoření `RepresentantViewModel` (jako parametr dostane `ActionViewModel` a s jeho pomocí vytvoří `RepresentantViewModel` a `TreeCreator` ho zařadí do své kolekce) a přiřazení `AdditionalInfo`.

Jak slouží `TreeCreator` jako `DataContext` pro drtivou část vzhledu aplikace zobrazuje Obrázek 24.



Obrázek 24: TreeCreator slouží jako DataContext pro hlavní část zobrazené aplikace. Binding je znázorněn přerušovanýma šipkama (Named tree nejsou zobrazeny seznam s nimi je užit při přidávání nového Named tree).

Z výše zmíněných důvodů jde vidět, že potřebujeme třídu s určitou základní funkcionalitou, která umožní podědění a případné upravení, než jen kontrakt definovaný rozhraním. Od `TreeCreator` dědí dvě třídy – `EntityViewModel` a `InheritenceNodeTreeCreator`.

EntityViewModel (třída)

Tato třída je umístěna ve jmenném prostoru `Revive.Editor.ViewModel` a je poděděna od `TreeCreator`, s jehož funkcionalitou si vystačí, jen přidává property s instancí `EntityModel`. Reprezentuje entitu ve ViewModel vrstvě.

InheritenceNodeTreeCreator (třída)

Tato třída je umístěna ve jmenném prostoru `Revive.Editor.ViewModel` a je poděděna od `TreeCreator`. Třídu `TreeCreator` obohacuje o seznam commandů, které jsou použity k možnosti zobrazení a mazání Named tree.

Všechny instance `EntityViewModel` jsou drženy v centrální tříde `EntityViewModelManager`.

EntityViewModelManager (třída)

Tato třída drží reference na všechny instance `EntityViewModel` a umožňuje hledání podle jména. Hledání podle názvu je důležité například pro zobrazení entity při volání z Runtimu během ladění, kde dostaneme jen název entity. Dále `EntityViewModelManager` obstarává vytvoření všech instancí `EntityViewModel` ze všech instancí `EntityModelů` v Modelu.

Jelikož je možno entity zařadit do kategorií a tyto kategorie do dalších kategorií, je třeba třídy držící tuto informaci a zveřejňující ji View vrstvě. Tato třída se jmenuje `EntityCategoriesViewModel`.

EntityCategoriesViewModel (třída)

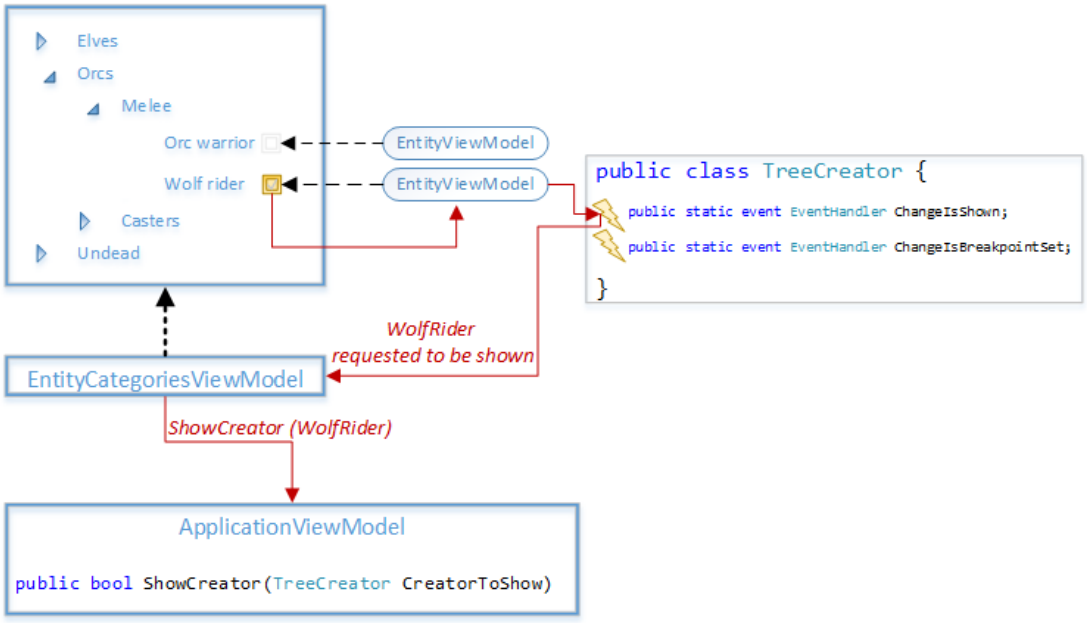
Třída je umístěna v namespace `Revive.Editor.ViewModel`. Entity je možné zařadit do kategorie a kategorie lze řadit do dalších kategorií. Dostáváme tak stromovou strukturu. `EntityCategoriesViewModel` zveřejňuje právě tuto stromovou strukturu a má na starosti zobrazení (resp. skrytí) `EntityViewModel` pomocí eventu spouštěného instancí třídy `TreeCreator` (viz začátek této podkapitoly na str. 52). Jak se zobrazí entita ilustruje Obrázek 25. Strom je složen ze dvou tříd – `EntityCategoryLeafNode` a `EntityCategoryNode` (obě umístěny ve stejném namespace).

EntityCategoryNode (třída)

Tato třída umožňuje definovat další instance `EntityCategoryNode` jako děti a tím umožňuje tvořit stromovou strukturu. View vrstvě zveřejňuje svoje jméno kategorie.

EntityCategoryLeafNode (třída)

Jednoduchá třída dědící od `EntityCategoryNode`, která obsahuje pouze referenci na instanci `TreeCreator` (reprezentující entitu). Pomocí této reference jednoduše zobrazí instanci `TreeCreator` zavoláním metody `ShowCreator` instance `ApplicationViewModel` (viz podkapitola 5.3.6).



Obrázek 25: Princip zobrazení entity. Binding je znázorněn černou přerušovanou šipkou. Červené šipky znázorňují postupně akce při zaškrtnutí entity „Wolf rider“ (aby se zobrazila). Wolf rider je list, jako data kontext má tedy instanci EntityCategoryLeafNode, která obsahuje instanci EntityViewModel (dědí od TreeCreator) s daty entity Wolf rider. Po kliknutí je změněna Property IsShown (definuje TreeCreator), to vyvolá spuštění statického eventu ChangeIsShown. Tento event poslouchá instance EntityCategoriesViewModel, použitá jako DataContext View části, která zobrazuje kategorie a entity. EntityCategoriesViewModel dostane v parametru instanci EntityViewModel a tu zobrazí pomocí metody ShowCreator, kterou definuje ApplicationViewModel.

Před popisem finální třídy `ApplicationViewModel`, která reprezentuje celý ViewModel je třeba probrat poslední část ViewModelu a to je části starající se o dědičnost a vytváření Named tree.

5.3.5. *Dědičnost a Named tree*

Začneme třídou, od které dědí každý vrchol stromu, reprezentující dědičnost v definující assembly. Všechny třídy v této podkapitole jsou definovány ve jmenném prostoru `Revive.Editor.ViewModel`.

InheritenceNodeViewModel (třída)

Tato třída je základem pro každý vrchol účastnící se dědičnost a poskytuje společné informace pro všechny typy vrcholů. Jméno a seznam dětí. Dále poskytuje metody pro tvorbu stromu z instance `InheritenceNode`, která reprezentuje kořen stromu dědičnosti a je definovaná v Modelu. Dále `InheritenceNodeViewModel` slouží jako `DataContext` pro View komponentu, která zobrazující dědičnost a Named trees. Dědičnosti se účastní dva typy vrcholů:

InheritenceLeafNode (třída)

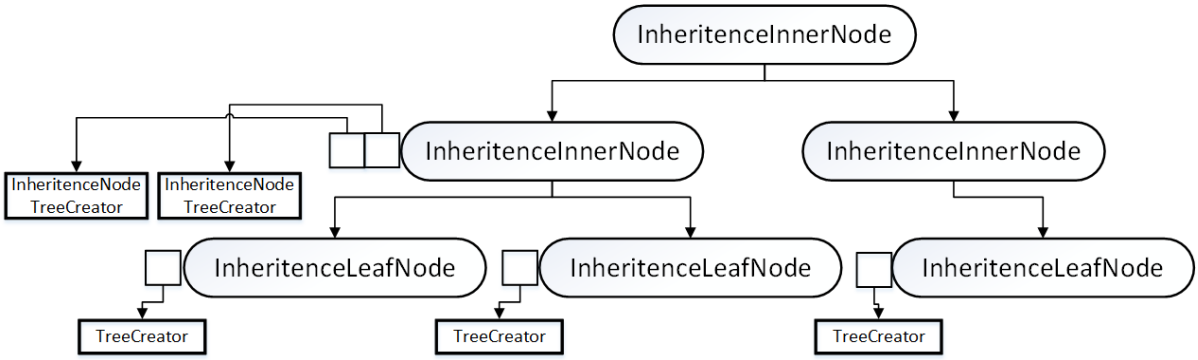
Tato třída reprezentuje listy. Obohacuje `InheritenceNodeViewModel` o proměnnou pamatující si instanci `TreeCreator`, kterého tento list reprezentuje a command, který umožní zobrazení této instance `TreeCreator`.

InheritenceInnerNode (třída)

Tato třída reprezentuje vnitřní vrcholy. Hlavním účelem je tvorba Named trees. Named trees si musí tento vrchol pamatovat. K tomu si pamatuje kolekci instancí třídy `InheritenceNodeTreeCreator` (popsána v předchozí podkapitole 5.3.4). Tyto instance pak zobrazí v případě, že uživatel požádá. V případě vytvoření nového Named tree, přidá novou instanci `InheritenceNodeTreeCreator` do kolekce a v případě mazání příslušnou instanci odstraní.

K tvorbě Named tree používá dialog, kterému předá seznam dosud definovaných stromů. Dialog vrátí jméno nového Named tree, které vepsal uživatel, přičemž pomocí kolekce dosud definovaných Named tree neumožní definovat strom, jehož jméno už `InheritenceInnerNode` obsahuje. Tvorbou dialogů se zabývá podkapitola 5.3.7.

Strom dědičnosti a instance třídy `TreeCreator` (resp. `InheritenceNodeTreeCreator`) reprezentuje Obrázek 26.



Obrázek 26: Stromová struktura reprezentující dědičnost.

Dědičnost byla poslední hlavní část ViewModelu. Zbývá popsat ViewModel pro hlavní okno aplikace.

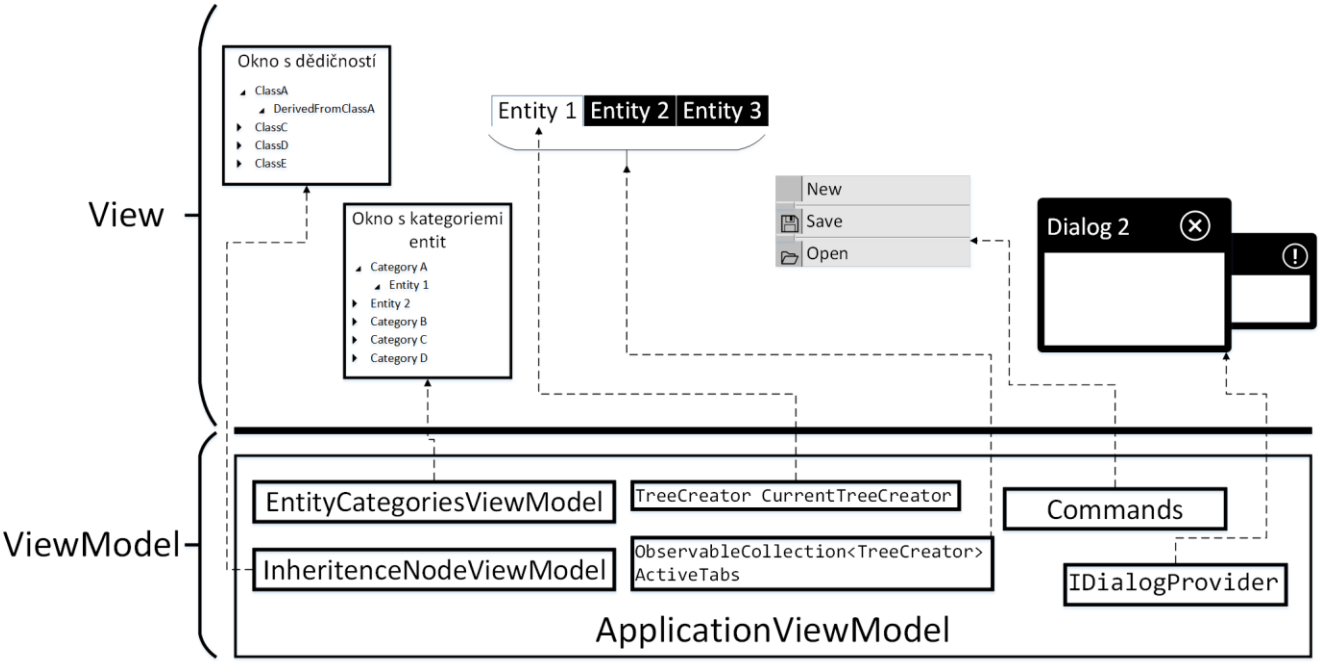
5.3.6. *ApplicationViewModel*

`ApplicationViewModel` je třída definovaná v `Revive.Editor.ViewModel` a tvoří hlavní ViewModel aplikace. Obsahuje statickou proměnou držící instanci `ApplicationViewModel`, která je použita jako `DataContext` pro hlavní okno. Toto umožňuje snadný přístup ze všech částí aplikace. Obsahuje všechny popsané části `ViewModelu` – `ActionViewModelManager`, `EntityViewModelManager`, `EntityCategoriesViewModel`, `InheritenceNodeViewModel`. Dále obsahuje seznam instancí `TreeCreator`, které jsou otevřené a aktuální instanci `TreeCreator`, kterou má otevřenou uživatel a je editována. Během ladění také obsahuje seznam instancí `TreeCreator`, na kterých je nastaven breakpoint. Poslední důležitou částí je umožnění tvorby dialogů (viz následující podkapitola). Všechny tyto části jsou zveřejněny skrz property, aby je šlo využít ve View vrstvě. Jsou zde i další property, jako jméno definující assembly, kterou používá editor, proměnná indikující zda je spuštěn režim ladění apod. Jmenování všech property by akorát snížilo přehlednost dokumentace.

Kromě properties zveřejňuje ApplicaitonViewModel mnoho commands, obstarávající tvorbu nového projektu, synchronizaci, načítání, ukládání apod.

Na závěr zmiňme důležité metody pro zobrazení (či odebrání ze zobrazených) instancí třídy `TreeCreator` – `ShowCreator`, `CloseCreator` a `CloseAllTabs`.

Pro znázornění funkce ApplicationViewModel jakožto hlavního ViewModelu je přiložen Obrázek 27.



Obrázek 27: ApplicationViewModel. Binding je zobrazen černou přerušovanou šipkou.

5.3.7. *Tvorba dialogů*

Kromě těchto základních částí je třeba, aby byl ViewModel schopný tvořit dialogy[1], neboť účel View je jen prezentovat data a ViewModel se stará o funkcionalitu. Jelikož tvorby dialogů je možná jen v code behind v hlavním okně ve View vrstvě (dialogu musíme nastavit rodičovské okno), nelze vše udělat ve ViewModelu. Zároveň samotná vizuální stránka do ViewModelu nepatří. Řešení je vytvořit rozhraní IDialogProvider.

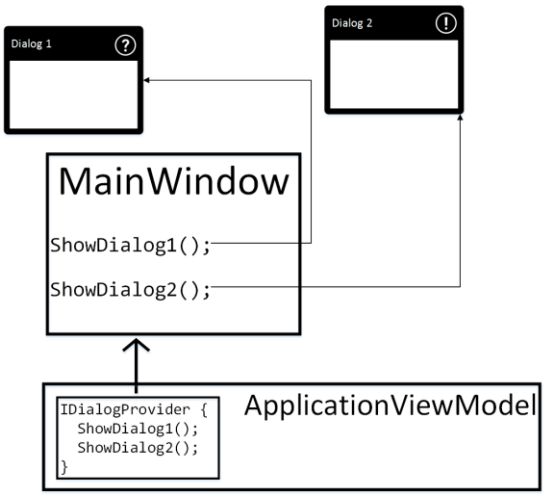
IDialogProvider (rozhraní)

Umístěno v *Revive.Editor.dll* ve jmenném prostoru `Revive.Editor.ViewResources`. Definuje metody pro tvorbu dialogů (každá může mít různou signaturu).

Toto rozhraní pak implementuje hlavní okno aplikace, které se postará o tvorbu dialogových oken. Když pak hlavní okno vytvoří ViewModel, předá mu samo sebe a tím pádem ViewModel dostane objekt, na kterém může volat metody pro tvorbu

[1] Tím myslíme vlastní okno, které zobrazíme jako dialog.

dialogů. Vytváření dialogu z ViewModelu je pak propagováno Modelu, jak zachycuje Obrázek 28.



Obrázek 28: Vytváření dialogů pomocí IDialogProvider

5.3.8. *Jak funguje server.*

Teď, když je jasná logika ViewModelu, je jednoduché ve stručnosti ukázat fungování třídy Server. Ladění bylo rozebíráno v podkapitole 4.4

Server (třída)

Tato třída je umístěna v *Revive.Editor.dll* ve jmenném prostoru `Revive.Editor.InterApplicationCommunication` a implementuje `IServerContract` (viz podkapitola 4.4.2). Aktuální instance je zveřejněna skrz statickou proměnou `CurrentServer`. Při vytvoření uloží všechny instance `TreeCreator` (tj `EntityViewModel` a `InheritenceNodeTreeCreator`) do `Dictionary<string,TreeCreator>`. Proč viz dále „Přijímání volání od klienta.“.

Server má dvě základní funkce:

- Přijímání volání od klienta.
- Nastavení breakpointů.

Funkce si rozebereme zvlášť.

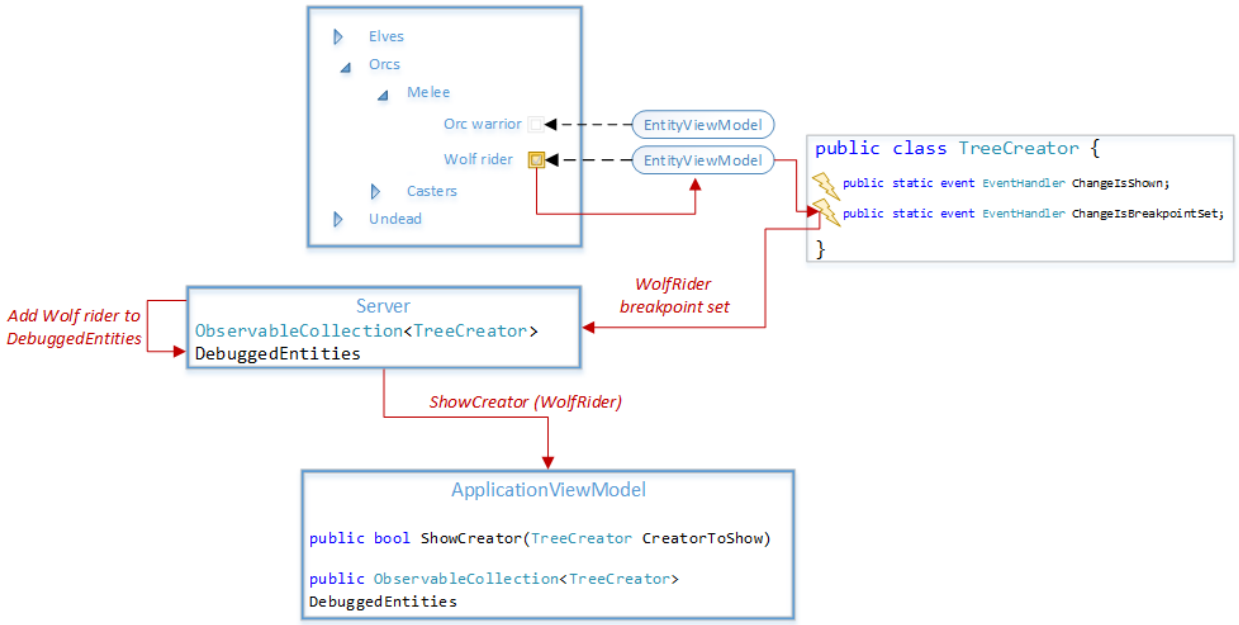
Přijímání volání od klienta.

Po přijmutí instance `ClientData` od klienta zjistí `Server` jaké instanci `TreeCreator` laděný vrchol patří pomočí klíče obsažený v instanci `ClientData` a `Dictionary`, kterou si `Server` vytvořil při konstrukci. Instanci třídy `TreeCreator` pak zobrazí pomocí `ShowCreator` metody třídy `ApplicationViewModel`. Po zobrazení zjistí pomocí třídy `RepresentantRegister` a klíčů z `ClientData` pak dohledá správnou instanci třídy `Representant`. Na té zavolá metodu `DoDebugNotification`, která spustí event, který odchytí `RepresentantViewModel` dané instance. Ten využije statickou metodu `RepresentantViewModel.HighlightRepresentant` (popsána v podkapitole 5.3.2). Změna v `RepresentantViewModel` je propagována skrz

binding do View vrstvy. Jelikož je zobrazena správná instance `TreeCreator`, uživatel ihned vidí jaký vrchol je laděn.

Nastavení breakpointů.

Pro nastavení breakpointů slouží property třídy `TreeCreator IsBreakpointSet`. Princip propagace, jaká entita byla vybrána, je stejný jako v případě zobrazení entity (třída `EntityCategoriesViewModel` v pokapitole 5.3.4) a proto poskytneme jen Obrázek 29. Při nastavení breakpointů instancím třídy `TreeCreator` jsou tyto instance přidávány do seznamu laděných instancí, kterou vlastní instance třídy `Server`. Tento seznam obsahuje také třída `ApplicationViewModel` jak bylo popsáno podkapitole 5.3.6. `ApplicationViewModel` pak View vrstvě jako svůj seznam poskytuje právě seznam instance třídy `Server`.



Obrázek 29: Nastavení breakpointů. Binding znázorňuje černá přerušovaná šipka.

Tímto máme probrány všechny třídy pracující v pozadí při práci s editorem (resp. ještě zbývá synchronizace, která je zařazena až po podkapitole View, neboť nám toto členění přišlo přehlednější). Proto můžeme probrat vizuální část editoru.

5.4. View

Účelem View je prezentovat data z ViewModelu. Zároveň ViewModel poskytuje commandy a Properties, které slouží k bindingu. Takto konzumuje View funkcionalitu ViewModelu. View je umístěn v assembly *Revive.Editor.Application.exe*. K presentaci dat se používají vestavěné WPF komponenty nebo vlastní data templaty. View obstarává několik věcí.

5.4.1. *Ostatní okna*

Kromě hlavního okna obsahuje aplikace okno s entitami a jejich kategoriemi (jako `DataContext` má `EntityCategoriesViewModel`) a okno s dědičností (jako `DataContext` má `InheritenceNodeViewModel` reprezentující kořen dědičnosti), kde jdou definovat Named trees. Obě tyto okna zobrazují hierarchickou strukturu a tak je třeba použití `HierarchicalDataTemplate`. Viditelnost oken je bindována na property instance třídy `ApplicationViewModel EntitiesCategoriesShown`, resp. `InheritenceTreeShown`. Tím může ViewModel měnit zobrazení oken. O vytvoření samotných oken se postará `MainWindow`, třída hlavního okna aplikace.

Okna jsou obsaženy ve jmenném prostoru `Revive.Editor.Application.Windows`.

5.4.2. *Dialogy*

Jak již bylo zmíněno v konci předchozí podkapitoly o ViewModelu 5.3.7, dialogy vytváří hlavní okno. Pro kontroly prováděné na vstupu od uživatele je dobré použít validační pravidla, které poskytuje WPF (12). Všechny dialogy jsou umístěny ve jmenném prostoru `Revive.Editor.Application.Dialogs` a jsou obsaženy v definici rozhraní `IDialogProvider` (viz zmíněná podkapitola 5.3.7) Potřebujeme tyto typy dialogů (rovnou píšeme třídy, které dialogy představují):

CreateNamedTreeDialog (třída)

Vytváří Named trees. Tento dialog nabízí `TextBox` pro vepsání názvu nového Named tree. Na `TextBox` jsou použita validační pravidla. Dialog dostává seznam instancí třídy `TreeCreator`, reprezentující definované Named tree a pomocí nich ověřuje vstup uživatele.

CreateNewProjectWindow (třída)

Vytváří nový projekt. Nový projekt se vytváří z definující assembly. V případě nějakých chyb zobrazí `CreateNewProjectWindow` chyby získané z metody pro vytváření nového Modelu `CreateModel` definované statickou třídou `ModelManager`.

ReferenceNamedTreeDialog (třída)

Tento dialog používají ViewModely od třídy `LookUpRepresentant` a slouží k tomu, aby se uživateli zobrazili všechny Named tree pro konkrétní entitu či vrchol ve stromu dědičnosti. Jako `DataContext` má nastavený tento dialog seznam instancí třídy `RootRepresentant`, které reprezentují stromy v dědičnosti. Uživatel pak vidí `ListBox`, který má jako `ItemsSource` právě tyto stromy, a proto může označit strom, na který se chce odkázat.

SynchronizingWizard (třída)

Tento wizard je použit při synchronizaci. Proč nestačí jednoduché okno je rozebráno ve zvláštní podkapitole 5.5

Toto jsou všechny dialogy. Následuje pasáž o hlavním vzhledu aplikace.

5.4.3. *Hlavní okno*

Hlavní okno se skládá ze tří částí:

- Levé menu. Toto menu obsahuje dostupné akce dané instance TreeCreator. Více to bylo popsáno a ilustrováno v podkapitole 5.3.4.
- Editační plocha, zabírající většinu plochy okna. Jak je tato plocha implementována je rozebráno v následující podkapitole 5.4.4.
- Horní menu a tlačítka. Práce s menu a tlačítky je rozebrána v kapitole 6 o práci s editorem.

Hlavní okno je reprezentováno třídou `MainWindow` a je umístěno ve jmenném prostoru `Revive.Editor.Application`. Jako `DataContext` používá `ApplicationViewModel`. Levé menu a editační plocha používají property `CurrentTreeCreator` instance `ApplicaitonViewModel` držící aktuálně editovanou instanci třídy `TreeCreator`. Horní menu pak využívá commands zveřejněných instancí `ApplicationViewModel`.

Následuje kapitola o editační ploše.

5.4.4. *Editační plocha*

K tvoření Behavior tree je vhodné použít komponentu `Canvas`, jelikož chceme volně pohybovat s vrcholy. Zároveň nechceme `Canvas` pevné délky a chtěli bychom podporu oddalování a přibližování. Pro tyto účely se velmi hodí `ZoomableCanvas` (13) vyvinutý Kaelem Rowanem, vývojářem v Microsoftu. `ZoomableCanvas` byl primárně vyvinut pro podporu virtualizace, jak datové (v paměti jsou jen data, kterých je potřeba), tak grafické (v paměti jsou jen objekty, které jsou zobrazeny). `ZoomableCanvas` bohužel obsahuje bug, kdy při použití virtualizace a odstranění objektu z kolekce (z jiného místa než konce) způsobí chybu. Naštěstí jde virtualizace vypnout (nebude ji třeba, neboť naše stromy budou obsahovat desítky vrcholů, v extrémním případě stovky).

Další problém při použití `ZoomableCanvasu` je nutnost použít `ListBox` místo obecnějšího `ItemsControl` jako kontejner pro data, pokud chceme, aby se zobrazily posuvníky v případě, že posuneme objekt mimo `ZoomableCanvas` (tím získáme nekonečnou designovou plochu). Pokud použijeme `ListBox` a jako `ItemsPanelTemplate` nastavíme `ZoomableCanvas`, `ListBox` nepropaguje `MouseDown` event do komponenty pod ním. Proto nemůže použít zvlášť `ZoomableCanvas` pro hrany a zvlášť pro vrcholy a oba `ZoomableCanvasy` navršit na sebe (toto je přirozená konstrukce, kterou umožňuje `ItemsControl`).

Jako řešení se naskýtá použít `ListBox` a jako zdrojovou kolekci předat `CompositeCollection`. `CompositeCollection` umožňuje definovat více kolekcí jako zdroj. Problém je, že verze WPF, která je použita frameworkem obsahuje nahlášený bug, kdy záleží na pořadí, ve kterém definujeme kolekce do `CompositeCollection`. Jedna z kolekcí se nezobrazí.

Revive framework použil pro vrcholy `ListBox` a pro hrany `ItemsControl` (obě komponenty mají jako `ItemsPanelTemplate` nastavený `ZoomableCanvas`). Kvůli `MouseDown` problému musí být `ItemsControl` umístěný nad `ListBoxem`.

Editační plocha je pak reprezentována vlastní třídou `NodeCanvas`.

NodeCanvas (třída)

Je umístěna ve jmenném prostoru `Revive.Editor.Application.View` a jako `DataContext` má instanci třídy `TreeCreator`. Jakými komponentami zobrazuje `NodeCanvas` vrcholy a hrany je popsáno výše.

Jelikož byla popsána editační plocha, je důležité popsat dvě třídy v ní zobrazené – `RepresentantView` a `Edge`. Před popisem tříd si rychle popíšeme jak je tvořen vzhled parametrů vrcholů (`RepresentantView`) v editoru.

Parametry

Styl fungování parametrů byl již popsán v podkapitole 5.3.1. Účel View je tvorba data templatů pro parametry. Tyto templaty jsou umístěny v `ResourceDictionary` nazvaném `ParameterViewResources. Dictionary` je umístěn ve jmenném prostoru `Revive.Editor.Application.Resources`.

Následuje popis dvou zmíněných tříd.

RepresentantView (třída)

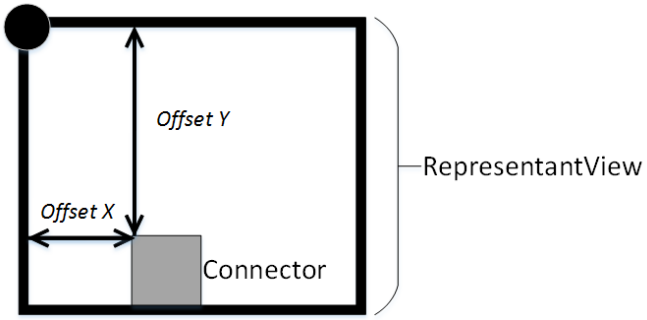
Jako `DataContext` slouží `RepresentantViewModel`. Pro `RepresentantView` je vhodné vytvoření `UserControl`, kterou nazveme právě `RepresentantView`. Aplikace pak bude přehlednější, neboť jako data template pro položky `ListBoxu` použijeme tento `UserControl`. V budoucnosti pak i snáze vyměníme vzhled vrcholu, neboť stačí vyměnit `UserControl`. Jak již bylo zmíněno u `RepresentantViewModel`, komponentou vhodnou pro pohyb je `Thumb` se svým `DragDelta` eventem. `RepresentantView` je umístěn ve jmenném prostoru `Revive.Editor.Application.View`.

Nejdříve musíme vyřešit data template pro `ConnectPoint` a jak bude probíhat spojování a rozpojování. Pro spojení a rozpojení potřebujeme komponentu `Connector`, která půjde mačkat a bude si pamatovat stav (zmáčknuta/nezmáčknuta). Pokud budou zmáčknuté dva `Connectory`, spojíme je hranou. Požadovanou funkcionalitu dobře splňuje `ToggleButton`, vestavěná komponenta WPF. Proto naše třída `Connector` dědí od `ToggleButton`.

Connector (třída)

Tato třída je v assembly *Revive.Editor.dll* v jmenném prostoru `Revive.Editor.ViewResources. Connector` si pamatuje objekt, kterému patří. Tento objekt musí implementovat `IDraggable` (rozhraní definuje schopnost hýbat se a informovat o změně souřadnic, viz podkapitola 5.3.2). `Connector` musí aktualizovat souřadnice `ConnectPoint` při pohybu. Jelikož `ConnectPoint` neví o View vrstvě, slouží jako držitel souřadnic, na které se bindují hrany. Změna souřadnic je záležitostí View. Proto se `Connector` stará o správné souřadnice `ConnectPoint`. Pro výměnu dat oběma směry je třeba vytvoření dvou dependency property se souřadnicemi (*x* a *y*) a vytvoření

bindingu s módem `TwoWay` mezi `ConnectPoint` a `Connector`. Dále si každý `Connector` musí pamatovat relativní souřadnice (nazvané *offset x* a *offset y*) vůči souřadnicím `RepresentantView`. Offsety ilustruje **Obrázek 30.**



Obrázek 30: Princip fungování offsetů. Kruh značí souřadnice rodičovské komponenty (RepresentantView), vůči které Connector počítá relativní souřadnice.

Celý proces aktualizace pak probíhá následovně. Uživatel pohne `RepresentantView`. Je spuštěn `DragDelta` event, handler pro tento event upraví souřadnice `RepresentantViewModel` (je nastavený jako `DataContext`). Tato změna způsobí, že se o změně souřadnic dozví `ConnectPoint`. `ConnectPoint` si nastaví nové souřadnice shodné s `RepresentantViewModel`. Toto je zachyceno `Connectorem` pomocí `TwoWayBinding`, který opraví souřadnice `ConnectPoint` podle offsetů. Instance `ConnectPoint` pak obsahuje správné souřadnice.

Kromě pohybu musí `RepresentantView` obstarat vhodný vzhled pro vrcholy a možnost editace a zobrazení parametrů vrcholu. Jako vzhled byl vybrán jednoduchý obdélník s názvem vrcholu. Pod obdélníkem zobrazíme `Connectory` pro propojení s dětmi a nad obdélníkem `Connector` pro propojení s rodičem. Jelikož jsou data templaty pro parametry aplikovány automaticky, netřeba tento problém řešit. Jen budeme chtít umožnit zobrazit a skrýt parametry. Pro to stačí použít `CheckBox` říkající zda chceme či nechceme zobrazit parametry. Parametry pak zobrazíme v komponentě `StackPanel` a `IsVisibility` `StackPanelu` bude bindován na `IsChecked` atribut `CheckBoxu`.

Vzhled `AdditionalInfo` (viz podkapitola 5.3.2) také nemusíme řešit, neboť je template aplikován automaticky a tak stačí `AdditionalInfo` zveřejnit v `ContentPresenteru` uvnitř `RepresentantView`.

Zbývá dořešit, jak zobrazit hrany, hrana je reprezentována třídou `Edge`.

Edge (třída)

Pro hrany potřebujeme ve WPF vykreslit čáru. Pro to nabízí WPF dvě komponenty – `Line` a `LineGeometry`. `LineGeometry` je jednoduchá čára, těchto čar můžeme mít stovky a aplikace bude rychlá. `Line` je plnohodnotná WPF komponenta s podporou data bindingu, stylů, routed eventů, apod. Proto je použití `Line` omezené na desítky nebo malé stovky objektů. Jelikož Potřebujeme data binding (pro `ConnectPoint`) a chceme mít možnost mazat hrany, jako data template pro `Edge` použijeme `Line`. Jelikož Behavior tree nebude obsahovat stovky hran, nevadí nám použití `Line`, výhodou je přehlednější kód a rychlejší práce.

Třídou `Edge` jsme dokončili celý editor, od logiky obsažené v Modelu a ViewModelu po vzhled ve View vrstvě. Zbývá dořešit ukládání, načítání aplikace a synchronizaci. První probereme synchronizaci.

5.5. Synchronizace

Synchronizace je jedna z nejdůležitějších částí aplikace a je jeden z hlavních cílů vytyčených v úvodu (cíl C6). Při synchronizaci je třeba řešit následující problémy:

1. Změna jména entity (přejmenování či vymazání).
2. Změna akcí entity.
3. Změna kategorie entity.
4. Změna jména akce (vymazání či přejmenování).
5. Změna parametrů akce (parametr přibyl, ubyl, změnil se typ nebo se změnila počáteční hodnota).
6. Změna kategorie akce.
7. Změna hierarchie tříd.

Nejhorším případem je změna hierarchie tříd, neboť nám poškodí celou strukturu Named trees a odkazování se na ně. Pojďme rozebrat řešení těchto problémů.

5.5.1. *Původní návrh*

Původní myšlenka jak řešit synchronizaci byla následující. Při vytvoření nového projektu se inicializuje Model. Synchronizace pak bude probíhat pomocí eventů. V případě změny 1 – 6 bychom provedli změnu (tj. přímo vymazali entitu, přidali parametr apod.) a celá aplikace by se synchronizovala pomocí eventů. Případ 7 by byl velmi složitý, neboť dědičnost se může změnit libovolně a bylo by třeba procházet strom dědičnosti a provádět změny, které by pouštěly další eventy. Od tohoto způsobu bylo rychle upuštěno právě kvůli složitosti implementace. Zároveň by byl kód hůře udržitelný, neboť provázaný systém eventů je složitý a hůře se upravuje. Také zde není možnost jednoduše pozastavit synchronizaci, vše je spouštěno automaticky. Hůře by se řešil i vzhled aplikace, kdy by ViewModel i View musely být připraveny reagovat na libovolnou změnu ze zmíněných bodů. Pojďme probrat návrh použitý Revive frameworkem.

5.5.2. *Synchronizace dvou modelů*

Behavior trees složené z instancí třídy `Representant` jsou navrženy tak, že jsou lehce přenositelné. Nejsou zde konkrétní vazby na určitou entitu či akci a to, jakou třídu vrchol reprezentuje, je řešeno pomocí textové reprezentace. Proto je idea taková, že vytvoříme nový Model a stromy, definované ve starém modelu, do nového překopírujeme. Samozřejmě, že to nejde tak jednoduše a bude třeba přenesené stromy doopravit. Než se pustíme do další analýzy, pojďme si probrat výhody tohoto přístupu:

- Jedná se o jednoduchou ideu. Kód je přehlednější, pokud je postaven na jednoduchém principu.

- Pomocí obou modelů lehce zobrazíme uživateli případné rozdíly mezi oběma Modely.
- Nemusíme „stavět synchronizaci na zelené louce“ a zužitkujeme kód tvořící model. Synchronizace se pak týká provedení změn mezi oběma modely.
- ViewModel vrstva je stavěna z Modelu. Až bude synchronizace nového modelu dokončena, stačí vytvořit nový ViewModel. Tím pádem netřeba řešit co dělat s oběma vyššími vrstvami po synchronizaci a stačí nové vytvoření.

Začneme představením toho, jak prezentovat synchronizaci uživateli.

5.5.3. *Synchronizační wizard*

Po vytvoření nového modelu získáme nové entity a akce. Pokud je srovnáme se starými akcemi a entitami nastane jedna z těchto možností:

- Entitu (akci) v novém Modelu najdeme pod stejným jménem.
- Entitu (akci) v novém Modelu nenajdeme, neboť je přejmenovaná, nebo je smazaná.

S entitami (resp. akcemi), které jsme našli nic dělat nemusíme (z hlediska uživatele). Jelikož z nových nepřiřazených entit (akcí) nepoznáme, které jsou opravdu nové a které jsou jen přejmenované, je třeba tuto možnost dát uživateli. Dále je třeba zobrazit informace o tom, jak synchronizace proběhla (které stromy byly změněny, které entity vymazány apod.). Proto pro synchronizaci nestačí jednoduché okno a je třeba vytvoření wizardu. Jako první část je vybrání definující assembly (na ní provedeme stejné kontroly jako při tvorbě nového projektu), dále umožnění určení které třídy jsou smazány a které jen přejmenovány a jako finální okno je informace o proběhlé synchronizaci.

Po této představě o tom, jak má synchronizace vypadat z uživatelského hlediska, můžeme přistoupit k implementaci.

5.5.4. *Synchronizace jednotlivých částí*

Všechny třídy související se synchronizací jsou umístěny v assembly *Revive.Editor.dll* (kromě wizardu, jenž je dialog viz 5.4.2). Dále jsou pak třídy obstarávající samotnou synchronizaci ve jmenném prostoru `Revive.Editor.Model.Synchronizing`. Jedná se o třídy probírané v této kapitole a tak to nebudeme explicitně zmiňovat. Synchronizaci má na starost třída `Synchronizer`.

Synchronizer (třída)

Instance této třídy je tvořena pomocí factory třídy `SynchronizerFactory`, která zajistí, že má instance `Synchronizer` správná data. Instanci třídě `Synchronizer` jsou předány dvě instance třídy `ModelLayer` reprezentující starý a nový Model. `Synchronizer` pak vykoná synchronizaci.

Další třídy mají spojitost s třídou `Synchronizer`, neboť ta je používá k provedení synchronizace.

Jelikož můžou být entity a akce přejmenovány, je dobré si vytvořit třídu, která drží tuto informaci.

RenameRules (třída)

Tato třída ukládá všechna přejmenování do `Dicitonary<string,string>` a tak můžeme lehce dohledat k dané akci (entitě) zda byla přejmenována.

Jako další struktura se hodí `EnumerableDifference`.

EnumerableDifference<T> (třída)

Tato třída je navrhnuta genericky, kde parametr T musí implementovat `INamable`. Účelem této třídy je vytvořit z dvou kolekcí (staré a nové), které jdou enumerovat (rozhraní `IEnumerable<T>`) a s pomocí instance `RenameRules` (mezi třídami ze staré a nové kolekce) vytvoření seznamů s rozdíly mezi kolekci. Aby tato třída zjistila, zda si dvě instance odpovídají, porovná jejich jména pomocí definovaného rozhraní `INamable`. Vytvořené seznamy jsou:

- Seznam instancí, které měli odpovídající jméno (`Name` definované `INamable`)
- Seznam instancí, které mají být smazány, tj. staré instance, jejichž jméno se nevyskytlo mezi jmény nových instancí a ani neexistuje přejmenování staré instance na nějakou novou definované v `RenameRules`.
- Seznam instancí, které jsou přidány, tj. seznam nových instancí, jejichž jméno se nevyskytuje mezi starými a ani neexistuje přejmenování staré instance, na novou
- Seznam instancí určených k přejmenování, tj. staré instance, jejichž přejmenováním získáme nějakou novou instanci. Tento seznam obsahuje vždy páry.

INamable (rozhraní)

Definuje jednoduchou textovou property `Name`.

Třída `EnumerableDifference` se hodí, neboť přesně vystihuje to, co se děje s akcemi a entitami.

Dále popíšeme princip synchronizace, než rozebírání jednotlivých tříd (ty jen zmíníme). Idea synchronizace není složitá, nicméně obnáší k realizaci hodně technické práce, jejíž rozebírání by znepřehlednilo koncept.

Synchronizace akcí

Akce musí být synchronizovány jako první, neboť je třeba správných akcí k následnému pročištění[1] Behavior trees. Synchronizace akcí je jednoduchá, jen

[1] Přijde nám to jako vhodný termín. Myslíme tím úpravu Behavior tree tak, aby obsahoval instance třídy Representant, které reprezentují akce, které patří dané entitě. Dále aby se instance LookUpRepresentant odkazovaly jen na stromy, které jsou definovány na předkovi entity vlastnící Behavior tree.

uložíme veškeré akce, které byly nahrazeny, vymazány a u kterých došlo ke změně parametru. Tyto data pak použijeme k pročištění Behavior trees.

Čištění stromů

Jelikož budeme často pročišťovat stromy na základě `EnumerableDifference<ActionModel>` dvou subjektů (entita či vrchol v dědičnosti), je vytvořena speciální statická třída `ActionDifferenceExecutor`, která pročistí stromy na základě `EnumerableDifference`. K čištění stromů používá metody statické třídy `RepresentantFactory` (viz podkapitola 4.4.5).

Dále je třeba řešit odkazování instancí `LookUpRepresentant` na Named trees. Zde je situace jednoduchá, neboť oba subjekty mají správné Named trees (u starého je to jasné a u nového to vyplývá z principu synchronizace dědičnosti viz dále) a nový subjekt mohl Named trees jen ztratit. Stačí tedy zjistit, jestli nový subjekt neobsahuje Named trees, které nejsou definovány ve starém. Pokud ano, jména těchto stromů si uložíme a projdeme Behavior tree nového subjektu, kde informujeme všechny instance `LookUpRepresentant` o zmizení těchto stromů a instance `LookUpRepresentant` si smažou informaci o tom, na jaký strom se odkazují (toto provádí metoda `ActualizeLookUpOnTreeDeletion` zmíněné třídy `RepresentantFactory`). Takto získáme pročištěný strom.

Synchronizace dědičnosti

Pro synchronizaci `InheritenceTree` uložíme všechny vrcholy starého stromu do `Dictionary`, kde jako klíč použijeme `ClassName` property příslušného vrcholu. Poté procházíme nový strom a pro každý vrchol kontrolujeme zda je ve starém stromu (pomocí `Dictionary`). Pokud ano, překopírujeme staré Named trees a pročistíme je. Připomeneme, že přidávání Named trees do `InheritenceNode` způsobí přidání do celého podstromu dědičnosti. Tím se nám dostanou správné Named trees do všech vrcholů podstromu, speciálně do entit (instance `EntityModel`). V průběhu procházení stromu dědičnosti odstraníme všechny použité `InheritenceNode` v `Dictionary`. Takto nám zbydou po synchronizaci v `Dictionary` jen `InheritenceNode`, jejichž Named tree musíme uvolnit z registru, neboť ten na ně drží reference. Toto je důležitý krok, neboť bychom ho nepoznali v aplikaci a měli bychom memory leak.

Synchronizace entit

Synchronizace entit probíhá podobně, jako synchronizace dědičnosti. Použijeme instanci `RenameRules` a překopírujeme staré stromy do nových entit a pročistíme.

Informace o proběhnuté synchronizaci

Po proběhnutí synchronizace je důležité zobrazit informace o případných změnách v editoru. Proto všechny metody obstarávající synchronizaci vracejí informace v třídách specifických pro danou změnu. Všechny tyto informace jsou seskupeny a na závěr synchronizace vráceny v seznamech.

Takto instance třídy `Synchronizer` provede synchronizaci a zveřejní nový model ve své property `NewModel`. Synchronizace je záležitost ViewModelu. Jak bylo zmíněno,

uživatel synchronizaci provádí pomocí synchronizačního wizardu. Proto je nutné vytvořit ViewModel pro synchronizaci.

5.5.5. *ViewModel pro synchronizaci*

Synchronizační wizard je spuštěn ViewModelem pomocí rozhraní `IDialogProvider`, které implementuje `ApplicationViewModel`. Tento wizard dostane skrz `IDialogProvider` instanci `SynchronizationViewModel`, kterou použije jako `DataContext`. Všechny třídy spojené se synchronizací na úrovni ViewModelu jsou obsaženy v assembly *Revive.Editor.dll* a ve jmenném prostoru `Revive.Editor.ViewModel.SynchronizingVM`. Wizard byl vytvořen na základě tutoriálu na stránkách CodeProject (14), kde je spoluautorem Josh Smith, expert na WPF.

Pro lepší představu je dobré přečíst podkapitolu 6.2.4 o synchronizaci v uživatelské dokumentaci či zkusit spustit editor a synchronizaci.

Každý krok wizardu má vlastní třídu, která slouží jako `DataContext` pro okno příslušného kroku a obstarává funkcionalitu. Třídy pro jednotlivé kroky používají jednu stejnou instanci `PageCommunicationChannel`, která obsahuje jméno assembly použité k synchronizaci a instanci `Synchronizer`, do které zapisují jednotlivé třídy pravidla pro přejmenování. Finální stránka pak spustí synchronizační proces, aktualizuje ViewModel a zobrazí informaci o změnách provedených synchronizací. Konkrétněji se nejedná o složitý proces a je popsán ve zmíněném tutoriálu nebo je přehledně srozumitelný ze zdrojových souborů.

Tímto je dokumentace synchronizace dokončena a zbývá poslední podkapitola zabývající se ukládáním aplikace.

5.6. Ukládání a načítání aplikace

Bez ukládání a načítání práce by se neobešla téměř žádná reálně použitelná aplikace. MVVM nám umožní snadné ukládání. Stačí uložit ViewModel a při nahrání nahrát ViewModel a View se pomocí bindingů postará o správné zobrazení. Jelikož ale ViewModel obsahuje `Connector`, který dědí od `ToggleButton`, nelze ho serializovat binárním serializérem. To nevadí, neboť ViewModel lehce postavíme z Modelu a tak stačí serializovat Model. Toto řešení může být potenciálně rychlejší, neboť se serializuje jen Model a nemusí se serializovat složitá struktura ViewModelu, která se jen postaví při nahrání, což je rychlejší. Binární serializace pro ukládání dat je vhodná, neboť se jedná o rychlý a pohodlný způsob jak uložit aplikaci. Nabízí se možnost serializovat do XML či použít nějaký vlastní způsob zapsání stavu aplikace. Obojí je ale zbytečné komplikování.

6. Práce s editorem

Tato kapitola popisuje práci s editorem. Je předpokládáno, že čtenář má přístup k aplikaci nebo alespoň ví, jak editor vypadá. Práci s editorem si lze vyzkoušet na hotovém projektu, který je přílohou této práce a je obsažen v souboru Appendix na přiloženém CD.

6.1. Start

Začneme tvorbou projektu, nový projekt získáme buď vytvořením, nebo nahráním.

6.1.1. *Vytvoření nového projektu*

Abychom mohli vytvořit nový projekt, je třeba assembly, kde jsou definovány entity a jejich akce. Této assembly říkáme *definující assembly*. Pro vytvoření nového projektu klikneme na **Menu → New Project**. Zobrazí se dialog pro vybrání definující assembly. Pokud nelze vytvořit projekt z vybraného souboru, je zobrazen seznam chyb zabraňující vytvoření.

6.1.2. *Načtení existujícího projektu*

K načtení projektu potřebujeme dříve uložený projekt. Projekty jsou ukládány s koncovkou `.revive` a jmenují se stejně, jako definující assembly (bez přípony). Projekt je defaultně uložen ve složce s názvem shodným s definující assembly (bez přípony) v adresáři, kde se nachází tato aplikace. Projekt je nahrán kliknutím na **Menu → Load**.

Následuje popis struktury aplikace.

6.2. Možnosti menu a horních tlačítek

Tato podkapitola popisuje funkcionalitu menu a tlačítek umístěných v horní části editoru.

6.2.1. *Okno s entitami*

Okno s entitami zobrazíme pomocí horního tlačítka s názvem **Entities**. Zobrazí se okno se stromovou strukturou. Jednotlivé uzly reprezentují kategorie a uzly, které nejdou dále expandovat konkrétní entity. Entity u sebe mají `CheckBox`, zatržením `CheckBoxu` signalizujeme editoru, že danou entitu chceme zobrazit. Po vybrání entit k zobrazení tento výběr potvrdíme kliknutím na **Confirm changes**.

6.2.2. *Strom dědičnosti*

Okno se stromem dědičnosti zobrazíme pomocí horního tlačítka **Inheritence tree**. Zobrazí se okno se stromovou strukturou. Tato struktura reprezentuje dědičnost tříd z definující assembly. Dědičnosti se účastní jen třídy, které mají jako potomka (ne nutně přímého potomka) entitu. Zopakujeme terminologii z kapitoly Analýza.

Named tree je Behavior tree, který není definován entitou, ale nějakou třídou od které třída, reprezentující entitu, dědí. Na tento strom se pak všechny entity, jež

jsou potomky (přímými či nepřímými) dané třídy mohou odkazovat. Toto umožňuje znovu použití jednoho Behavior tree více entitami.

Na každý uzel v oknu se dá kliknout pravým tlačítkem myši. Menu, které se zobrazí, závisí na tom na jaký uzel jsme klikli:

- Vnitřní uzel. Tento uzel jde dál expandovat. Pravé kliknutí na tento uzel ukáže dvě možnosti:
  - **Defined named trees**. Pokud tento uzel definuje nějaké Named trees, toto menu pokračuje dál a zobrazí jejich jména. Každý strom pak jde smazat (možnost **Delete tree**) či otevřít pro editování (**Show tree**). Každý uzel, na kterém je definován alespoň jeden Named tree je zobrazen tučným písmem.
  - **Create new tree**. Tato možnost umožní tvorbu nového Named tree. Zobrazí se dialogové okno, kde se napíše název stromu. Strom nesmí mít stejné jméno jako jiný strom definovaný tímto uzlem a nesmí obsahovat značky: **/ \\ : * ? \ < > | _**. Pokud strom nelze definovat, je formulář červený a zobrazí patřičný důvod. Vytvořený má název XXX_YYY kde XXX je název uzlu a YYY je název stromu.
- Koncový uzel. Tento uzel nejde dále expandovat a jedná se o entity. Menu pak zobrazí jen jednu položku **Show behavior tree**, která zobrazí strom entity v editoru.

6.2.3. *Zarovnání zobrazeného stromu*

Aktuálně zobrazený strom se zarovná pomocí tlačítka **Align tree.** Tato funkcionalita umožňuje přehledné zobrazení stromu.

6.2.4. *Synchronizace*

Synchronizace se spouští pomocí **Menu → Synchronize**. Synchronizace slouží k aplikování změn v definující assembly na prováděnou práci. Synchronizaci obstarává wizard. První strana požaduje vybrání definující assembly, vůči které se synchronizujeme. Následuje okno s akcemi. Zde jsou zobrazeny staré akce, které nebyly nalezeny v definující assembly a budou vymazány. V případě, že staré akce byly pouze přejmenovány, je zde možnost vybrat z nových akcí tu, která odpovídá staré. Dále je okno s entitami se stejným principem. Následuje poslední okno, které informuje o případných změnách, které synchronizace provedla s prací v editoru.

6.2.5. *Publikování práce*

Publikování se spouští přes **Menu → Publish** a požádá o vybrání definující assembly, která byla použita editorem. Publikování pak vytvoří složku Trees v adresáři, kde je definující assembly a do této složky nahraje všechny stromy vytvořené v editoru. Stromu jsou pak použity hrou.

6.2.6. *Ukládání*

Ukládání se provádí kliknutím na **Menu → Save**. Při ukládání se vytvoří složka v adresáři, kde je umístěn editor nazvaná stejně, jako definující assembly užita

editorem. Do této složky se pak uloží data editoru pod názvem již zmíněné definující assembly s příponou .revive.

6.2.7. *Ladění Behavior trees*

Ladění spouští tlačítko **Start debug** a ukončuje tlačítko **End debug**. Po spuštění ladění se čeká až se připojí hra. Po připojení hry jsou zpřístupněny možnosti ladění. Pojďme si je představit.

Nastavení breakpointu

Breakpointy se nastavují v okně s entitami a kategoriemi kliknutím na tlačítko **Entities.** V ladícím režimu je zde navíc pravý `CheckBox`, který slouží k nastavení breakpointů. Všechny entity, na kterých je nastaven breakpoint jsou zobrazeny v levém menu.

Laděná entita

Jméno laděné entity se zobrazuje vedle tlačítek v horní části obrazovky. Kromě toho levé menu obsahuje dodatečné informace o laděné entitě (pokud jsou definovány). Laděnou entity lze rychle zobrazit stisknutím tlačítka **Show debugged entity**.

Ladění stromů

V ladícím režimu nejde měnit struktura stromů. Vrchol, který je právě laděn je zobrazen výraznější barvou a ohraničen červeným rámečkem. Abychom umožnili pokračovat ve výpočtu laděnému vrcholu, musíme stisknout tlačítko **AllowContinue**. Po stisknutí se provede další krok výpočtu Behavior tree. Každý vrchol má jednu z pěti barev.

- Šedá. Značí, že nejsou známy informace o vrcholu.
- Žlutá. Vrchol se dostal právě do výpočtu a není zatím znám jaký stav vrátí.
- Zelená. Vrchol dokončil výpočet a vrátil stav „Úspěch“.
- Červená. Vrchol dokončil výpočet a vrátil stav „Neúspěch“.
- Oranžová. Vrchol dokončil výpočet a vrátil stav „Běžím“.

S průběhem výpočtu se postupně obarví Behavior tree. Aby bylo možné obarvení zrušit (pro přehlednost), je zde tlačítko **Clear nodes**. Toto tlačítko obarví všechny vrcholy vybraného stromu šedivou barvou.

Na závěr zmiňme, že v ladícím režimu nejdou vytvářet ani mazat Named tree a ladící režim nekontroluje, zda všechny struktury stromu vzájemně odpovídají v editoru a ve hře.

Následuje podkapitola o tvorbě Behavior trees.

6.3. Tvorba Behavior tree

Popíšeme si jak tvořit Behavior trees. Nejdříve si probereme editační plochu, kde jsou stromy tvořeny.

6.3.1. *Editační plocha*

Editační plocha zabírá většinu aplikace. Nad editační plochou jsou všechny otevřené entity zobrazeny jako záložky, přičemž jen jedna může být aktivní. Samotná editační plocha jde oddalovat a přibližovat pomocí scroll tlačítka na myši (či jiném tlačítku scrollování zastupující). Dále lze s editační plochou hýbat tak, že se podrží levé tlačítko myši na místě, kde není vrchol a táhne se. Poslední vlastností editační plochy je upravení velikosti podle umístěných vrcholů. Pokud je nějaký vrchol mimo zobrazitelnou část, jsou zobrazeny scrollbary.

6.3.2. *Vytvoření vrcholu*

Pro vytvoření vrcholu je třeba otevřít nějakou entitu, či Named tree. Levé menu pak obsahuje vrcholy pro danou entitu. Vrchol vytvoříme pomocí kliknutí na tlačítko, které ho reprezentuje. Vrchol se vytvoří na souřadnicích, které reprezentují počátek editační plochy. Tento počátek lze poznat zarovnáním stromu tlačítkem **Align tree**, neboť ten se podle něj orientuje.

Popíšeme si typy vrcholů

6.3.3. *Typy vrcholů*

Editor poskytuje kategorie, ve kterých jsou zobrazeny akce, které vytváří vrcholy. Vrcholy můžou být buď bez možnosti definice dítěte (typicky tyto vrcholy tvoří programátor v definující assembly) nebo s možností definicí dítěte. Dále se zde vyskytují vrcholy, které jsou společné všem Behavior trees. Jedná se o:

- **Sequence**. Tento vrchol uspěje, pokud uspějí všechny děti a neuspěje při prvním neúspěchu dítěte. Dědi pouští postupně po řadě. Další dítě přijde vždy na řadu, pokud to předchozí uspěje.
- **Selector**. Tento vrchol uspěje, pokud uspěje alespoň jedno dítě. Neuspěje při neúspěchu všech dětí. Dědi pouští postupně po řadě. Další dítě přijde vždy na řadu, pokud to předchozí neuspěje.
- **Parallel**. Tento vrchol pouští všechny své dědi paralelně. Nejedná se o úplný paralelismus, o spuštění všech dětí postupně za sebou v jednom `Update` kroku. Tento vrchol uspěje, pokud všechny vrcholy uspějí, selže při prvním selhání. Parallel se obzvlášť hodí pro použití s podmínkami, neboť pokud je daný vrchol podmínka, typicky uspěje v prvním `Update` kroku. V dalších by se nedostal na řadu. Podmínky ale Parallel spouští v každém `Update` kroku, a tak můžeme vytvořit jako děti sadu podmínek a strom, který vykonává nějakou činnost. Jakmile nějaká podmínka neplatí, strom vykonávající činnost pak přeruší výpočet.
- **Negation**. Tento vrchol vrátí „Neúspěch“ v případě úspěchu dítěte a „Úspěch“ v případě neúspěchu dítěte.
- **Look up tree.** Tento vrchol umožní se odkázat na nějaký Named tree.
- **Always fail**. Tento vrchol vždy selže.
- **Always succeed**. Tento vrchol vždy uspěje.

Tímto byly popsány typy vrcholů a můžeme přejít k poslední podkapitole týkající se práce se stromem.

6.3.4. *Práce se stromem*

Pokud máme na editační ploše nějaké vrcholy, lze využít jejich tlačítek ke spojení. Pro spojení je třeba vždy kliknout na jedno tlačítko v horní části vrcholu a jedno tlačítko v dolní části jiného vrcholu, takto se provede spojení. Pokud by vznikla kruhová závislost, editor nepovolí na tlačítko kliknout.

Vrcholy jsou pak spojeny hranami, hrany lze mazat držením klávesy Ctrl a levým kliknutím myši. Vrcholy se mažou tlačítkem obsahující křížek umístěném v pravém horním rohu. Pohyb vrcholů se pak provádí držením levého tlačítka myši nad vrcholem a táhnutím.

Kromě spojování, rozpojování a hýbáni lze zobrazit parametry vrcholů. To se provádí pomocí malého `CheckBoxu` zobrazeného vedle vrcholu.

Tímto jsme popsali veškerou práci s editorem. Následuje kapitola zabývající se implementaci Revive frameworku v kódu.

7. Práce s knihovnou Revive.dll

Účelem této kapitoly je ukázat jak implementovat Revive framework na úrovni kódu. Hlavním cílem je vytvoření assembly, kde budou definovány všechny entity a akce. Je důležité, aby taková assembly byla jen jedna. Tato assembly je pak nahrána editorem a tím je umožněna tvorba Behavior trees. Všechny důležité třídy jsou obsaženy v knihovně *Revive.dll*. Můžeme si představit, že *Revive.dll* má dvě části:

- Kontrakt pro entity a jejich akce.
- Runtime, který obstarává běh AI ve hře.

Knihovna se vyskytuje ve dvou verzích, vývojářská a finální. Vývojářská verze je použita v případě, že je potřeba sledování výpočtu Behavior trees. Nejdříve začneme popisem kontraktu a pak Runtimu. Pro příklad, jak vypadá implementace Revive frameworku, je dobré se podívat na přiložený program v souboru Appendix na přiloženém CD, který obstarává simulaci třech entit.

7.1. Co musí splňovat entity

Každá třída reprezentující entitu musí implementovat rozhraní `IEntity` definované ve jmenném prostoru `Revive.BehaviorTreeStructure`. Dále pokud implementuje třída rozhraní `IEntity`, nesmí od ní dědit žádná další třída. Toto rozhraní definuje dva kontrakty:

- Property `EntityData` vracející instanci `EntityData`. `EntityData` je třída použitá frameworkem. Tato property se implementuje poskytnutím proměnné `EntityData` pro zápis a čtení.
- Metoda `GetDebugData` vracející instanci `EntityDebugData`. Tato metoda je volána při ladění. Třída `EntityDebugData` obsahuje čtyři `string`ové property, do kterých je možno přiřadit `string`, který se pak bude zobrazovat v editoru. Jedna property je pojmenována `Name` a slouží k zobrazení konkrétního identifikátoru entity (aby bylo jasné, jaká entita je laděna). Tato metoda může vrátit `null` v případě nezájmu předávat data (toto ale není vhodné řešení pro případné ladění). `EntityDebugData` jsou umístěny ve jmenném prostoru `Revive.Runtime.InterApplicationCommunication`.

Následuje příklad implementace psa:

```
public class Dog : IEntity {
       private EntityData d;
       private int id;

       public EntityData EntityData {
           get {
               return d;
           }
           set {
               d = value;
           }
       }

       public EntityDebugData GetDebugData() {
```

```
        EntityDebugData debugData = new EntityDebugData();
        debugData.Name = "Dog " + id;
        return debugData;
    }
}
```

Další důležitá věc je tvorba akcí pro entity.

7.2. *Jak vytvořit akce entit*

Akce entit jsou listy v Behavior tree. Pro akci je třeba vytvořit vrchol, který ji vytváří (třída poděděná od `UserNode`) a pak třídu poděděnou od `UserTask`, kterou vrchol vytvoří. Třída poděděná od `UserTask` obstarává samotnou logiku výpočtu a je specifická pro každou instanci entity. Obě třídy jsou obsaženy ve jmenném prostoru `Revie.BehaviorTreeStructure`.

7.2.1. *Tvorba akce*

Pro vytvoření akce je nejprve nutné vytvořit vrchol. Ten musí dědit od třídy `UserNode`. Nazvěme naši poděděnou třídy `MyNode`. Je důležité, aby `MyNode` obsahoval konstruktor, přijímající instanci `NodeParameters`. Toto potřebuje interně framework. `NodeParameters` jsou pro získání parametrů z editoru (popíšeme dále v podkapitole 7.2.3). `MyNode` musí implementovat dvě metody:

- `CreateTask`. Tato metoda vrací instanci poděděnou od `UserNode`, která reprezentuje výpočet `MyNode`. Nazvěme tuto instanci `MyTask`. `MyTask` musíme implementovat (později).
- `OnNodeLoad`. Tato metoda je volána frameworkem, když je načten do paměti. Slouží k předání instancí vlastních tříd hry. Bude rozebráno dále.

Poslední velmi důležitou věcí je označit `MyNode` a všechny položky jako `Serializable` pomocí atributu `[Serializable]`. Implementace vrcholu tak může vypadat takto:

```
[Serializable]
public class MyNode : UserNode {
    public MyNode(NodeParameters p) : base(p) { }

    public override UserTask CreateUserTask(IEntity Entity) {
        return new MyTask(this, Entity);
    }

    public override void OnNodeLoad(NodeParameters Parameters) {
    }
}
```

Po implementaci `MyNode` je třeba implementovat `MyTask`, obstarávající logiku výpočtu. `MyTask` dědí od `UserTask` konstruktor, který přijímá instanci `IEntity` která vlastní instanci `MyTask` a `Node`, jenž reprezentuje vrchol, který instanci této třídy vytváří (v našem případě `MyNode`). Dále musí `MyTask` implementovat tři abstraktní metody definované třídou `UserTask`:

- `OnUpdate`. V této metodě se odehrává hlavní výpočet v každé `Update` smyčce hry. Jako parametr dostane instanci třídy `Blackboard`. Tato třída bude rozebrána dále v části zabývající se parametry. Slouží k předávání objektů

mezi různými instancemi tříd poděděných od `UserTask` ve stromu dané entity. Vrací enum `Status`, který signalizuje jak volání metody skončilo. Může být:
    - `Success` – volání skončilo úspěšně a výpočet skončil.
    - `Failure` – výpočet selhal (selhal ve smyslu terminologie Behavior tree).
    - `Running` – výpočet ještě neskončil a bude pokračovat v dalším volání.
    - `Invalid` – výpočet skončil s chybu.
- `OnInitialize`. Sem patří inicializační logika uzlu.
- `OnTerminate`. Sem patří logika zajišťující úklid po ukončení (zavření otevřeného souboru apod.). Jako parametr dostane od rodiče `Status` indikující v jakém stavu byl při ukončování.

Od `UserTask` dědí `MyTask` tři properties:

- `CreatorOfTheTask` – pamatuje si `Node`, který vytvořil `Task`, je instance třídy `Node`, kterou dostane `MyTask` v parametru při konstrukci.
- `EntityOwningTask` – pamatuje si instanci `IEntity`, které tento `Task` patří, jedná se o instanci, kterou dostane `MyTask` v parametru při konstrukci.
- `IsCondition` – indikuje zda `MyTask` reprezentuje podmínku, to je důležité pro některé vrcholy frameworku `Revive`, neboť ví, že se jedná o podmínku a například pak mohou testovat podmínku v každém `Update` kroku (viz Parallel v podkapitole 6.3.3)

Ještě je třeba přiřadit takto vytvořené vrcholy entitám, aby bylo jasné jaké akce má jaká entita. To se dělá přes atributy a je rozebráno v další podkapitole.

Příklad implementace `MyTask`, který ve druhé fázi výpočtu vypíše „Hello world“ může vypadat takto:

```
public class MyTask : UserTask {
        public MyTask(Node n, IEntity e)
            : base(n, e) {
        }
        int step;

        public override Status OnUpdate(Blackboard Blackboard) {
            step++;
            if (step == 1) return Status.Running;
            else {
                Console.WriteLine("Hello world");
                return Status.Success;
            }
        }

        public override void OnInitialize() {
            step = 0;
        }

        public override void OnTerminate(Status Status) {
        }
    }
```

Tímto jsme probrali základní strukturu a můžeme se věnovat atributům.

7.2.2. *Použití atributů*

Atributy jsou umístěny ve jmenném prostoru `Revive.Attributes` a slouží ke definici dalších informací pro třídy poděděné od `UserNode`. Konkrétní použití každého atributu je v komentářích u třídy, budeme se věnovat co vše lze definovat.

1. Jméno. Pokud nechceme aby se u náš vrchol zobrazoval stejně, jako jméno třídy, použijeme tento atribut. Stejně můžeme jméno použít u entity[1]. Příklad je:

```
[Serializable]
[Name("Cast fireball")]
public class CastFireballNode : UserNode
```

2. Přiřazení akcí [2] k entitám. Na toto slouží atribut `DefineAciton`, který v parametru dostane seznam Typů, reprezentující třídy poděděné od `UserNode`. Tento atribut nemusí být definován na třídě implementující `IEntity`, ale na libovolné třídě. Pak akce, které takto definujeme automaticky dědí všechnu odvozené třídy. Uveďme příklad, kdy se definují akce a využije se dědičnosti. Předpokládáme, že jsme si naimplementovaly tři třídy `MoveNode` (reprezentuje pohyb entity), `BiteNode` (reprezentuje akci kousnutí) a `CastFireballNode` (reprezentující vykouzlení ohnivé koule). Třída Dog pak má nadefinované akce `BiteNode` a `MoveNode` a třída `Wizard` pak `MoveNode` a `CastFireballNode`.

```
/// <summary>
/// Represents all objects with ability to move.
/// </summary>
[DefineAction(typeof(MoveNode))]
public abstract class Movable {

}

[DefineAction(typeof(BiteNode))]
public class Dog : Movable, IEntity {

}

[DefineAction(typeof(CastFireballNode))]
public class Wizard : Movable, IEntity {

}
```

3. Zařazení akce do kategorie. Pokud chceme vizuálně zařadit některé akce k sobě, můžeme ke každé z nich definovat kategorii pomocí `DefinaActionCategory` atributu. Jako parametr dáme jméno kategorie a všechny akce stejné kategorie pak budou vizuálně spolu.

---

[1] Třída implementující IEntity. Dále pro přehlednost budeme používat toto názvosloví.

[2] Akcí myslíme třídu poděděnou od UserNode. Dále pro přehlednost budeme používat toto názvosloví.

4. Zařazení entity do kategorie. Pokud nechceme uživateli editoru zobrazit všechny entity jen tak vedle sebe, je možné definovat entitám stromovou strukturu pomocí atributu `DefineEntityCategory`. Atribut v parametru dostane `string`, který reprezentuje zařazení do kategorie. Tento parametr je ve tvaru názvů jednotlivých kategorií a podkategorií oddělených znakem „/“. Parametr nezačíná znakem „/“. Příklad ukazuje, jak zařadíme entitu `OrcWarrior` do stromové struktury Orcs → Melee → Infantry a entity `Shaman` do Orcs → Casters. Uživatel editoru pak uvidí příslušně zařazené entity.

```
[DefineEntityCategory("Orcs/Casters")]
public class Shaman : IEntity {

}

[DefineEntityCategory("Orcs/Melee/Infantry")]
public class OrcWarrior : IEntity {

}
```

5. Můžeme definovat parametry a vizuální závislosti mezi nimi. Tomuto tématu věnujeme samostatnou podkapitolu.

7.2.3. *Parametry pro akce*

Akcím jdou definovat parametry. Tyto parametry jsou pak zobrazené v editoru a uživatel s nimi může pracovat. Jelikož atributy neumožňují předávat instance ve svých konstruktorech, všechny parametry musí být v textové podobě. Parametr se skládá z:

- Názvu. Název parametru se zobrazuje v editoru.
- Typu. Typ parametru určuje, jak editor daný parametr zobrazí (`CheckBox`, `ComboBox` apod.)
- Hodnoty. Hodnota parametru je uložena v textové podobě.
- Povolené hodnoty. Toto nemá každý parametr, ale některé parametry dovolené hodnoty potřebují ke svému fungování (například `ComboBox`, který zobrazuje kouzla entity, zde nadefinujeme seznam kouzel a uživatel nemůže zvolit nic jiného, než co jsme nadefinovali)
- Výchozí hodnota. Tato výchozí hodnota je použita před tím, než uživatel hodnotu změní. Pokud parametr má jen seznam povolených hodnot, jako výchozí se použije ta první.

Parametr se definuje pak pro danou akci atributem `DefineParameter`, který dostane název parametru, typ parametru a buď výchozí hodnotu, nebo seznam povolených hodnot. Typ parametru určuje `enum ParameterType` a sémantika parametrů je jasná z typu nebo ji obsahuje komentář.

Když jsme parametr definovaly, ještě bychom rádi získali hodnotu parametru při spuštění hry. Na to slouží instance třídy `NodeParameters`, která pod textovým klíčem, který je jméno parametru, skladuje hodnoty parametrů. Tuto instanci získáme v konstruktoru naší akce.

Kromě definice parametrů je zde atribut `DefineDependency`. Pro účel si představme tento příklad:

Mějme parametr „typ kouzla“ s hodnotami „ohnivá koule“ a „ledový blesk“. Dále mějme parametr „rozšířit oheň po dopadu“ s hodnotami „true“ a „false“. Požadujeme mít možnost nadefinovat aby se parametr „rozšiř oheň po dopadu“ zobrazil v editoru jen pokud je vybrána možnost „ohnivá koule“ u parametru „typ kouzla“.

K tomu slouží DefineDependency. Atribut dostane jako parametry:

- Jméno toho parametru, jehož viditelnost chceme měnit. (Kdo)
- Jméno na jakém parametru závisí.
- Možné hodnoty, které způsobí, že „Kdo“ bude viditelný.

Při definici závislosti je kontrolováno, zda parametry existují, ale ne, zda existuje výchozí hodnota, proto je potřeba dávat pozor na chyby. Shrňme tuto kapitolu komplexním příkladem entity `Shaman`, která má definovanou akci `DoSomething` s těmito parametry (toto je jen ilustrační příklad, v reálné implementaci je to špatný přístup):

- `PlayAngryAnimantion`. Má hodnoty `true` nebo `false` a říká, zda se má přehrát animace, která reprezentuje naštvaného `Shamana`.
- `CastSpell` s hodnotami `SelfHeal`, `LightingBolt`. Vykouzlí buď sebe léčení nebo blesk.
- `ChooseEnemy` s hodnotami `FirstNear`, `LowestHp`. Udává na koho má `Shaman` seslat kouzlo, nejbližší entita nebo entita s nejmenším počtem HP.

Dále chceme, aby `ChooseEnemy` bylo zobrazeno jen pokud je vybrán `LightingBolt`. Také chceme vědět, zda máme přehrát animaci.

```
    [Serializable]
    [DefineParameter("PlayAngryAnimation",ParameterType.CheckBox,"false")]
    [DefineParameter("CastSpell",ParameterType.ComboBox,"SelfHeal,LightingBolt")]
    [DefineParameter("ChooseEnemy",ParameterType.ComboBox,"FirstNear","LowestHp")]
    [DefineDependency("ChooseEnemy","CastSpell","LightingBolt")]
    public class DoSomething : UserNode {
        public DoSomething(NodeParameters p)
            : base(p) {
            object parameter;
            if (p.TryGetParameter("PlayAngryAnimation",out parameter)){
                string value=parameter as string;
                if (value==null) throw new ArgumentException("Parameter is not a
string.");
                if (value == "true") isAngry = true;
                else isAngry = false;
            }
            else throw new ArgumentException("Parameter PlayAngryAnimation was not
found")
        }

        private bool isAngry;

        public override UserTask CreateUserTask(IEntity Entity) {
            ...
        }
```

```
        public override void OnNodeLoad(NodeParameters Parameters) {
            ...
        }
    }
```

7.2.4. *Komunikace akcí mezi sebou*

Mohou nastat případy, kdy potřebujeme aby byly instance třídy Task schopny komunikace mezi sebou. Motivaci uvedeme na příkladu z podkapitoly 3.11.

Mějme akci, která zjistí jaké nepřátelské entity jsou poblíž. Tato akce má jako parametr v editoru možnost filtrace pro výběr entity s nejmenším počtem zdraví, výběr nejbližší, výběr entity s nejvyšším útokem apod. Dále máme sadu různých akcí, které odpovídají různým typům útokům. Chtěli bychom mít možnost nadefinovat „Použij ohnivou kouli na nejbližší nepřátelskou entitu s nejmenším počtem zdraví“. Toto je jednoduše proveditelné pomocí sekvence, která bude mít dvě děti – akci zajišťující výběr entity a akci odpovídající vyčarování ohnivé koule.

K realizaci tohoto problému slouží třída `Blackboard` (ve jmenném prostoru `Revive.BehaviorTreeStructure`). Instance této třídy poskytuje možnost uložit libovolný objekt pod číselným klíčem v rozsahu 0 – 49. Každá entita pak má jednu instanci `Blackboard`, která je předávána skrz `OnUpdate` metodu akce. Proto sem instance třídy `Task` může něco zapsat a jiná instance přečíst. Typicky se předpokládá, že se vytvoří `enum`, který reprezentuje různé položky v instanci `Blackboard` (např. položka na ukládání entity, položka na uložení předmětu apod.) a tímto `enum`em se pak indexuje do instance `Blackboard`.

Poslední věc, kterou bychom chtěli, je předávání vlastních objektů svým akcí. Tím se zabývá následující kapitola.

7.2.5. *Předání vlastních objektů akcím*

Předávat vlastní objekty je velmi požadovaná vlastnost. Uvažme například objekt ve hře, schopný najít pro danou entitu cestu. Pokud máme akci, která umožňuje pohyb (a například v parametrech zobrazí uživateli místa kam může entita jít), potřebuje tato akce k vykonání své činnosti objekt obstarávající počítání cesty.

Objekt `RuntimeManager`, který obstarává `Update` entit (je rozebrán v další podkapitole) definuje property `OnNodeLoadParameters` typu `NodeParameters`. Tuto property využijeme k uložení objektu, který chceme předat a v metodě `OnNodeLoad` v naší třídě, poděděné od `UserNode` pak objekt načteme a uložíme. Protože všechny instance třídy `Task` mají odkaz na `Node` co je vyrobil, lze získat objekty ze hry. Ilustruje to jednoduchý příklad.

```
    ...
RuntimeManager manager = RuntimeManager.CreateManager();
manager.OnNodeLoadParameters["PathPlanner"] = pathPlanner;
    ...

public class MyNode : UserNode {

        ...
```

```
    PathPlanner planner;

    public override void OnNodeLoad(NodeParameters Parameters) {
        planner = Parameters["PathPlanner"];
    }
}
```

7.3. Použití ve hře v Update smyčce

Tato podkapitola se zabývá tím, jak aktualizovat entity za běhu hry. Vše obstarává třída `RuntimeManager` umístěná v `Revive.Runtime`. Tato instance může být ve hře jen jedna a je tvořena skrz statickou metodu `CreateManager`. Jak bylo řečeno v uživatelské dokumentaci v podkapitole 6.2.5, stromy z editoru jsou publikovány do složky *Trees*. Defaultně je složka vytvořena ve stejném adresáři, jako je assembly s entitami a akcemi. `RuntimeManager` pak potřebuje cestu k této složce, pokud se cesta nezadá, implicitně předpokládá, že je složka ve stejném adresáři, jako assembly vytvářecí instanci `RuntimeManager`.

`RuntimeManager` poskytuje metodu `Update`, která vykoná všechnu potřebnou aktualizaci chování entity. Dále obsahuje metody pro před nahrání určitých stromů do paměti či jejich odstranění z paměti. To je kvůli urychlení prvotního načtení když je potřeba. Nebudeme psát konkrétní metody a odkážeme se na vygenerovanou dokumentaci či nápovědu k třídám v IDE Visual Studio.

Poslední kapitola se zabývá laděním.

7.4. Ladění

Jak bylo zmíněno na začátku této kapitoly, knihovna se vyskytuje ve dvou verzích. Pro podporu ladění je třeba použít tu vývojářskou. Ladění probíhá samo a jediná nutná podmínka je spuštění editoru a spuštění ladění. Bez spuštěného ladění v editoru RuntimeManager vyhodí výjimku `EditorNotInDebugModeException`.

Jak bylo popsáno v podkapitole 7.1, pro zobrazení dalších informací o entitě je třeba implementovat metodu `GetDebugData`.

Laděním jsme dokončili celou kapitolu zabývající se implementací Revive frameworku v kódu. Ještě jednou zmíníme, že pro praktický příklad implementace slouží přiložená simulace.

V následující kapitole se probírá obohacení zdrojových kódů frameworku.

8. Obohacujeme Revive framework

Tato kapitola se zabývá obohacením zdrojových kódů frameworku o nový parametr pro akce a tvorbou nového vrchol pro framework.

8.1. Tvorba nového parametru pro akce

Celý proces tvorby nového parametru si probereme na příkladu vytvoření TextBoxu, který od uživatele vezme jako vstupy jen kladná čísla.

8.1.1. *Nutné body pro implementaci*

Pro implementaci jakéhokoliv nového parametru je nutné splnit následující body (nejlépe postupně):

1. Obohatit enum `ParameterType` o nový parametr v projektu **Revive** v namespace `Revive.Attributes` v souboru *AttributesCollection.cs*.
2. Vytvoření ViewModelu pro parametr v projektu **Revive.Editor** ve jmenném prostoru `Revive.Editor.ViewModel.ParametersVM`. K implementaci je nutné použít třídu dědící od `ParameterViewModel`. Více je popsáno v dokumentaci v podkapitole 5.3.1.
3. Obohatit metodu `CreateParameterViewModels` třídy `ParameterManager` v projektu **Revive.Editor** ve jmenném prostoru `Revive.Editor.ViewModel.ParametersVM`. Zde ve `switch` podmínce je třeba dodat novou větev, která vytvoří nový ViewModel pro nový parametr a ten přidat do kolekce (z kódu je jasné co dělat).
4. Implementovat `DataTemplate` pro parametr v projektu **Revive.Editor.Application** ve složce *Resources* v souboru *ParameterViewResources.xaml*. `DataTemplate` musí mít definovaný `DataType` na typ třídy, kterou jsme vytvořili a která tvoří ViewModel.
5. V případě, že nejde na parametr klikat, je třeba zkontrolovat metodu `OnPreviousMouseMove` u `NodeCanvas` a upravit podmínku.

Pojďme si ukázat implementaci zmíněného parametru.

8.1.2. *Příklad implementace*

Implementovaný TextBox nazveme PositiveIntegerBox.

Nejdříve obohatíme enum.

```
public enum ParameterType {
        /// <summary>
        /// Values are true of false. Use only one of these for default value.
        /// </summary>
        CheckBox,
        ComboBox,
        /// <summary>
        /// TextBox, where can be writen only positive integers.
        /// </summary>
        PositiveIntegerBox
    }
```

Dále implementujeme ViewModel. Pro View si zveřejníme property, která obsahuje text od uživatele. Všimněme si, že pro získání a zapsání hodnot se používají metody poděděné od třídy `ParameterViewModel`.

```
public class PositiveIntegerBoxViewModel : ParameterViewModel {
        public PositiveIntegerBoxViewModel(ParameterManager Manager,
                                           ParameterWrapper Wrapper,
                                           RepresentantViewModel Owner)
            : base(Manager, Wrapper, Owner) {
            if (Wrapper.ParameterType != ParameterType.PositiveIntegerBox)
                throw new ArgumentException(String.Format("IntegerBoxViewModel
                got parameter which is not of type IntegerBox and is type
                of {0}", Wrapper.ParameterType));
        }

        public string Value {
            get { return GetValue(Name); }
            set {
                SetValue(Name, value);
                OnPropertyChanged("Value");
            }
        }
    }
```

Obohatíme `switch`.

```
       switch (w.ParameterType) {
                    case ParameterType.CheckBox:
                        Parameters.Add(new CheckBoxViewModel(this, w, owner));
                        break;
                    case ParameterType.ComboBox:
                        Parameters.Add(new ComboBoxViewModel(this, w, owner));
                        break;
                    case ParameterType.PositiveIntegerBox:
                        Parameters.Add(new PositiveIntegerBoxViewModel(this, w,
owner));
                        break;
                    default:
                        break;
                }
```

A nakonec definujeme vzhled (code behind ukazovat nebudeme).

```
    <DataTemplate DataType="{x:Type p:PositiveIntegerBoxViewModel}">
        <DockPanel Visibility=
"{Binding Path=IsVisible,Converter={StaticResource ResourceKey=boolToVisibility}}">
            <TextBlock Text="{Binding Name}" DockPanel.Dock="Left" />
            <r:PositiveNumberTextBox Text="{Binding Value}" MinWidth="30" />
        </DockPanel>
    </DataTemplate>
```

Následuje podkapitola o přidání nového vrcholu.

8.2. Tvorba nového vrcholu pro framework

V této kapitole nebudeme uvádět příklad, neboť tvorba vrcholu vyžaduje mnoho kódu a nechceme ho sem psát. Místo toho odkážeme na zdrojové kódy Revive, případně vložíme ukázku krátkého kódu. Tyto zdrojové kódy je dobré projít společně s čtením bodů důležitých pro implementaci. Zdrojové kódy dobře navádí co udělat.

8.2.1. *Nutné body pro implementaci*

1. Vytvořit novou třídu a přidat ji do jmenného prostoru `Revive.BehaviorTreeStructure.InnerNodes` v případě, že se jedná o vrchol s dětmi či `Revive.BehaviorTreeStructure.LeafNodes` v případě vrcholu bez dětí. Obě namespace jsou v projektu Revive.
2. Označit daný `Node` atributem `[Serializable]`
3. Upravit enum `RepresentantType` (přidat novou položku) v projektu **Revive.Editor** ve jmenném prostoru `Revive.Editor.Model` v souboru *Representant.cs*.
4. Aby byl vrchol použitelný všemi entitami, je třeba upravit metodu `AddActionsNotDefinedInAssembly` třídy `AssemblyData` v projektu `Revive.Editor.csproj` ve jmenném prostoru `Revive.Editor.Loading`. Je třeba zaregistrovat nový vrchol k třídě `Object` (tak ho zdědí všechny třídy) a vytvořit parametry. Pro tvorbu parametrů používá Revive konvenci takovou, že třídy poděděné od Node definují dvě statické metody, které vrací parametry a závislosti mezi nimi. Pak vypadá kód přehledně takto.

```
// Sequence
ActionWrapper wrapper = new ActionWrapper(typeof(SequenceNode));
wrapper.Parameters = SequenceNode.GetParameters();
wrapper.Dependencies = SequenceNode.GetVisualDependencies();
wrapper.Category = "Composites";
wrapper.RepresentantType = Model.RepresentantType.Sequence;
wrapper.Name = "Sequence";

Actions.Add(wrapper);
ActionRegister["SequenceNode"] = wrapper;
ClassRegister[typeof(Object).Name].Actions =
     new Type[] {typeof(SequenceNode), …}
```

5. Přidat třídu poděděnou od třídy `Representant`, reprezentující nový vrchol do jmenného prostoru `Revive.Editor.Model.Representants` v projektu **Revive.Editor**. Při implementaci metody `CreateNode` je důležité nezapomenout překopírovat klíče z instance `Representant` do tvořené instance `Node` jako třeba zde:

```
public override BehaviorTreeStructure.Node CreateNode(NodeFactory Factory) {
      FailNode node = new FailNode(this.NodeParameters);
      node.StringKey = this.StringKey;
      node.ID = this.ID;
      return node;
}
```

6. Označit tuto třídu poděděnou od `Representant` atributem `[Serializable]`
7. Implementovat vytváření nové třídy poděděné od `Representant` v `RepresentantRegister` v projektu `Revive.Editor.csproj` ve jmenném prostoru `Revive.Editor.Model`.

To byly všechny kroky k vytvoření vlastního vrcholu.

Tímto jsme dokončili celou tuto kapitolu a následuje srovnání s existujícími nástroji.

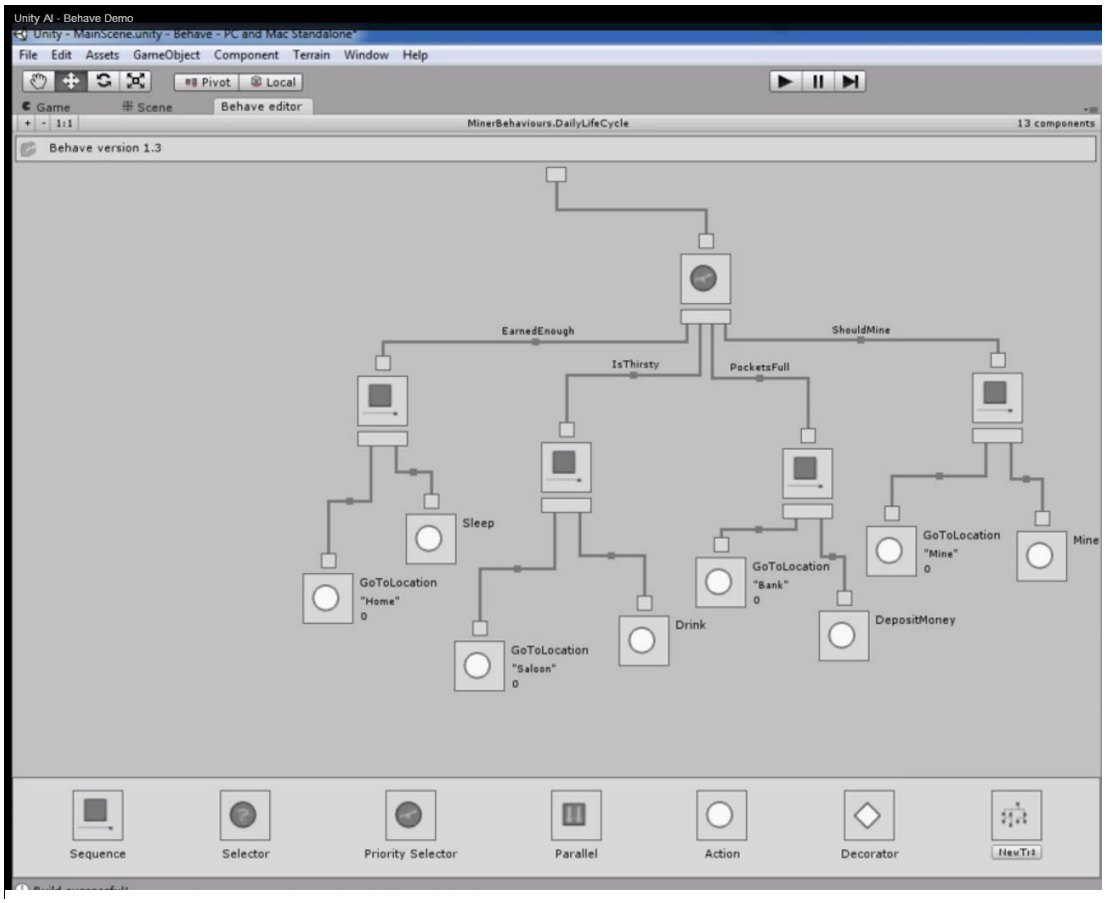
9. Srovnání s existujícími nástroji

V době psaní této bakalářské práce neexistuje moc editorů pro tvorbu AI založené na konceptu Behavior trees. Existují knihovny umožňující použití z kódu, ale nejedná se o kompletní framework, který byl cílem této práce. Jako příklad takovéto knihovny lze použít **TreeSharp** (*https://code.google.com/p/treesharp/*).

Jako editory pro srovnání nám přišly vhodné následující tři.

Behave *(http://angryant.com/behave/)*

Tento editor je plug-in do relativně nového a velmi se rozvíjejícího Unity enginu (5), který podporuje všechny důležité herní platformy současnosti. Ukázku tohoto editoru znázorňuje Obrázek 31. Stejně jako Revive nabízí tento editor možnost tvořit strom a umístit vrcholy na libovolné místo plátna. Dále je zde také možnost definovat strom a v jiném stromu se na něj jen odkázat. Oproti Revive jsou akce metody entit a nedefinují se zvlášť ve třídách. Tento způsob má určitě své výhody v přehlednosti a možná i rychlosti, nicméně neumožňuje znovu použití nějaké akce jinou entitou. Dále neposkytuje takové možnosti jako plnohodnotná třída s dědičností a možností definování vlastních metod a proměnných. Při exportu Behavior tree v programu Behave je možnost vybrat release nebo debug export, v závislosti na typu exportu je pak spuštěn příslušný režim (release/debug). Revive používá dvě knihovny, jednu debug a druhou release, debug mód je pak automaticky spouštěn u debug knihovny. Tato možnost pak umožňuje použít

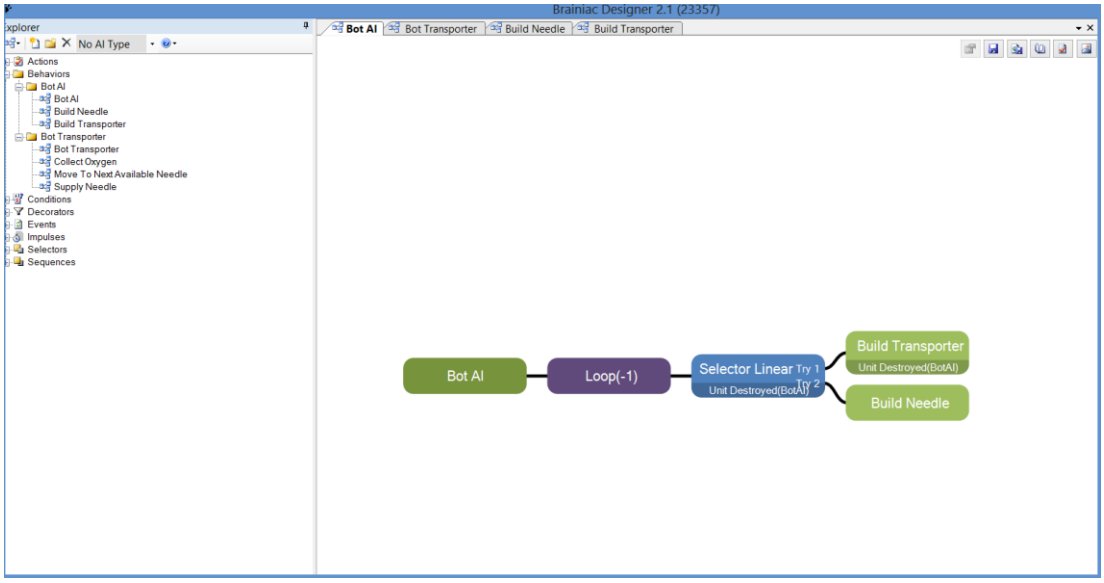


Obrázek 31 Plocha pro tvorbu stromu v programu Behave.

jedou vyexportované Behavior trees, jak pro debug, tak pro release běh hry bez změny exportovaných dat oproti Behave. Na rozdíl od Revive Behave obsahuje Priority Selector, který oproti normálnímu selektoru poskytuje výběr ze svých dětí na základě definovaných funkcí. Jelikož je Behave komerční projekt je dále jak po stránce designu, tak po stránce funkcionality.

Brainiac designer *(http://brainiac.codeplex.com/)*

V době psaní této práce je nejspíše alespoň 2 roky mrtvým projektem. Oproti Revive poskytuje kromě klasického pohledu na strom na plátně zobrazení v podobě stromu odkazů (`TreeView` komponenta ve WPF). K tomuto editoru není dokumentace a tak se hůře analyzuje. Pro akce také používá zvlášť třídu. Pro export se vytvoří přímo kód tříd v C#. Brainiac poskytuje podobnou funkcionalitu jako Revive. Každý vrchol má možnost parametry, které lze zobrazit a vrcholy, které lze použít jsou umístěny v pravém menu. Hezkou věcí je možnost „sbalit" strom do jednoho vrcholu a okomentovat co daný podstrom dělá, také možnost exportovat vytvořený Behavior tree jako obrázek je hezká. Nelíbila se nám nemožnost pohybovat s vrcholy stromu, které jsou umístěny deterministicky. Jak tento editor vypadá popisuje Obrázek 32.



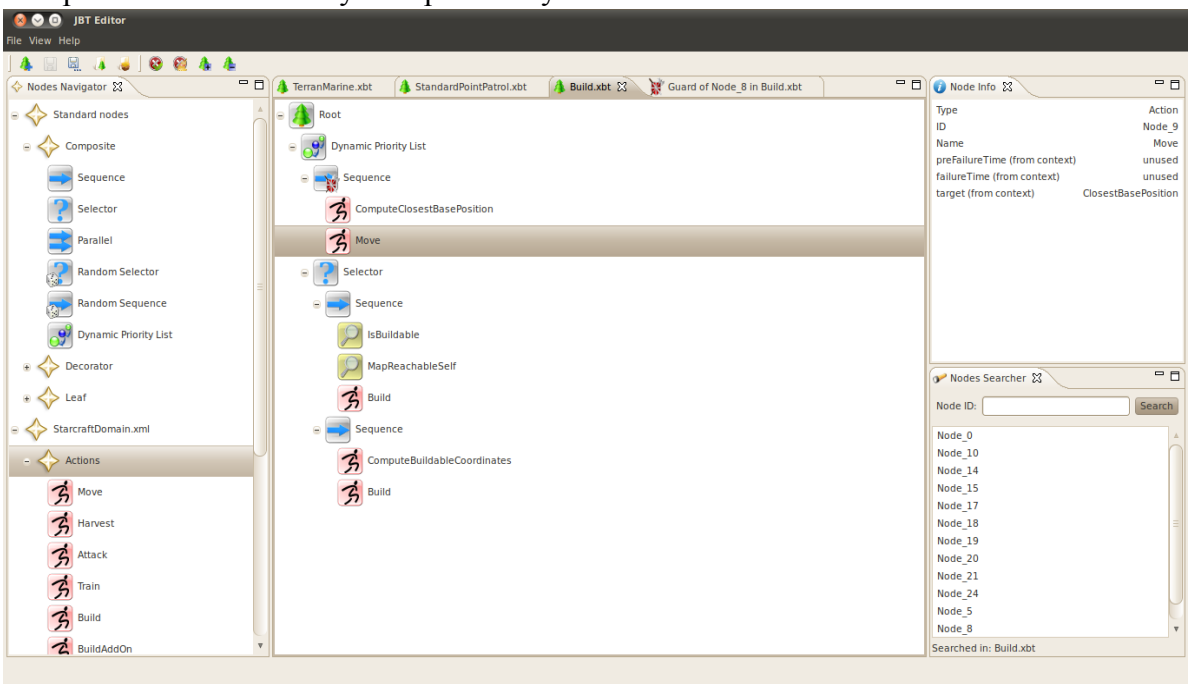
Obrázek 32 Výřez hlavní části programu Brainiac designer

Java Behavior Trees – JBT *(https://github.com/gaia-ucm/jbt)*

Tvoření stromu je zde děláno pomocí TreeView (viz Obrázek 33), takže editor stejně tak jako Brainiac designer neumožňuje interaktivní práci s vrcholy. Za zmínku stojí export dat reprezentujících strom ve formě XML. Revive tuto možnost nemá, neboť při runtimu je třeba co nejrychlejšího nahrání stromu. Proto jsou stromy uložené jako binární serializovaná data a v případě uložení jako XML by tedy stejně bylo nutné nějaké předzpracování. Nástroj pro vygenerování XML dat ze serializovaných binárních dat by nebyl těžký úkol vzhledem k acyklické povaze dat a stromové struktuře. Definice akcí a podmínek v JBT probíhá také přes XML kde nadefinujeme akce s podmínkami, které daná entita má a dále nadefinujeme typ těchto akcí a podmínek (např. jaké typy parametrů vrchol

dostává). Dále je pak nutné implementovat akce a podmínky v kódu. Naštěstí JBT nabízí možnost vygenerovat třídy podle definice v XML souboru. Přesto je zde dvojí definice entit a akcí, která se nám nelíbí a proto považujeme řešení používající reflection za lepší.

Vzhledově je JBT podobný Revive a obsahuje pravé menu s akcemi a podmínkami a záložky s rozpracovanými entitami.



Obrázek 33 JBT editor

Všechny editory se ukázali jako vhodné nástroje pro tvorbu Behavior trees a lišili se jen v designu a přístupem ke zpracování entit a akcí definovaných uživatelským programátorem. Z hlediska funkcionality poskytují editory stejné možnosti.

10. Závěr

10.1. Zhodnocení práce

Tvorba Revive frameworku se ukázala jako práce nad rámec jednoho člověka. Jde to vidět ve vzhledu frameworku a sadě před připravených vrcholů. Framework byl ale navržen tak, aby oba tyto problémy šlo v budoucnosti vyřešit bez úpravy logiky samotného frameworku. Jelikož se nám podařilo splnit všechny naše vytyčené cíle C1 – C7 i E1 – E5, je framework plnohodnotným nástrojem pro tvorbu AI do her. Vzhled není z hlediska tvorby AI překážkou a funkcionalita editoru je navržena tak, aby ji bylo možné lehce využívat a potřebná funkce byla rychle dostupná bez zbytečné posloupnosti kliknutí. Práce s editorem je tak pohodlná a nebrání hlavnímu účelu editoru – tvorbě AI.

Výbornou vlastností Revive frameworku je jeho jednoduchá implementace ve hře. Pro použití stačí implementovat pár tříd, případně o anotovat třídy atributy a vše ostatní zpracuje framework automaticky. O podporu sledování výpočtu stromů se programátor, implementující framework, nemusí starat a jediné co je třeba, je použití ladící verze knihovny *Revive.dll*.

Vytvořený framework lze reálně použít k tvorbě hry, jak je probráno v příloze. Architektura frameworku pak v budoucnu umožní obohacení o další typy vrcholů pro editor.

10.2. Budoucí práce

Jak již bylo popsáno, hlavní slabinou frameworku vzhled. Naštěstí je logika frameworku implementována a tak zbývá řešit novou definici vzhledu v XAMLu. Pro další obohacení pak bude třeba změn v kódu, nicméně dobrá architektura umožňuje inkrementální změny a netřeba velkých úprav.

V budoucnosti bychom chtěli implementovat:

1. Možnost označit více vrcholů a hýbat tak celým stromem.
2. Možnost přidat obrázky k jednotlivým akcím a entitám.
3. Mini mapa zobrazující celou editační plochu s vyznačením pro aktuálně zobrazenou část.
4. V ladění výpočtu stromu umožnit měnit parametry. Pro to je třeba doimplementovat data posílaná editorem do hry a umožnit Runtime části frameworku předání informace instancím třídy `Task`.
5. Při ladění vypočtu stromu umožnit měnit vlastnosti laděné entity (např. snížit zdraví). K tomu je třeba vytvořit novou třídu pro posílaná data a upravit rozhraní `IEntity` pro příjem těchto dat.
6. Umožnit bohatší předání informací o entitě během ladění výpočtu stromu. Bylo by vhodné celý design navrhnout tak, jak je navržen systém parametrů. Získali bychom tak bohaté možnosti implementace informací o entitě pomocí XAML data templatů. Spolu s bodem 5 by pak framework poskytoval velmi pěkné možnosti pro ladění.

7. Úprava vzhledu vrcholů. Bylo by dobré vytvořit speciální data templaty pro každý typ vrcholu. K tomu nám dobře poslouží enum `RepresentantType`. Na základě tohoto enumu pak rozhodneme
8. Kontrola, zda třída implementující `IEntity` v definující assembly nemá potomka.
9. Bylo by pohodlné, aby se na jméno Named tree, na který se `LookUp` vrchol v editoru odkazuje, dalo kliknout a zobrazil by se Named tree. Jelikož `LookUpRepresentantInfo` dostává v parametru instanci `TreeCreator`, která reprezentuje Named tree, nejedná se o složitou implementaci.
10. Velmi dobrá funkcionalita pro zpřehlednění práce s Behavior trees je možnost „sbalit“ celý podstrom do jednoho vrcholu a tento vrchol nějak pojmenovat.
11. Vrcholy tvořit v místě na plátně, kde je uživatel. Způsob zvolený editorem je vytvoření vrcholu na pevně daných souřadnicích (s malým rozptylem). To je nevhodný způsob když nejsme v místě, kde se tvoří vrcholy. Jediné co stačí udělat, je změnit souřadnice tvořeného vrcholu. To by mohlo probíhat například tak, že by ViewModel při vytváření spustil event a jako parametr by předal vytvářenou instanci třídy `RepresentantViewModel`. View vrstva pak ví, v jaké části editační plochy se uživatel nachází, a tak může upravit patřičně souřadnice.
12. Možnost řadit abecedně akce v levém menu pro přehlednost.
13. Lepší ukládání. Tj. upozornění uživatele při zavření editoru a neuložení práce, při načítání práce zobrazit entity, které byly otevřené v době ukládání.
14. Nahrání definující assembly do zvláštní aplikační domény a po získání informací tuto doménu od nahrát spolu s assembly. Toto umožní, aby editor nahrál informace z definující assembly a ta mohla být později změněna a překompilována během běhu editoru. V tuto chvíli Revive framework pracuje tak, že nahraje assembly, získá data, ale .NET neumožňuje od nahrání assembly. Tím pádem pokud se pracuje s editorem, nelze překompilovat definující assembly.

11. Zdroje

1. WPF. *MSDN*. [Online] [Citace: 30. 04 2013.] http://msdn.microsoft.com/en-us/library/ms734712.aspx.

2. Možnost komunikace mezi procesy v .NETu. *MSDN*. [Online] [Citace: 30. 04 2013.] http://msdn.microsoft.com/en-us/library/8119f66k(v=vs.100).aspx.

3. .NET Remoting. *MSDN*. [Online] [Citace: 30. 04 2013.] http://msdn.microsoft.com/en-us/library/72x4h507(v=vs.100).aspx.

4. WCF. *MSDN*. [Online] [Citace: 30. 04 2013.] http://msdn.microsoft.com/en-us/library/ms734712.aspx.

5. *Unity engine.* [Online] [Citace: 30. 04 2013.] http://unity3d.com/.

6. **Buckland, Mat.** *Programming Game AI by Example*. Texas : Wordware Publishing, Inc., 2005.

7. **Champandard, Alex J.** Článek o predikci AI v roce 2008. *AIGameDev.* [Online] [Citace: 01. 05 2013.] http://aigamedev.com/open/editorial/retrospective-game-ai-2008/.

8. Komunita zabývající se tvorbou AI do her. *AIGameDev.* [Online] [Citace: 30. 04 2013.] http://aigamedev.com/.

9. Úvodní článek o Behavior trees. *AIGameDev.* [Online] [Citace: 30. 04 2013.] http://aigamedev.com/open/article/bt-overview/.

10. .NET Reflection. *MSDN*. [Online] [Citace: 05. 05 2013.] http://msdn.microsoft.com/en-us/library/f7ykdhsy.aspx.

11. MVVM. *MSDN*. [Online] [Citace: 30. 04 2013.] http://msdn.microsoft.com/en-us/magazine/dd419663.aspx.

12. Validace ve WPF. *Codeproject.* [Online] [Citace: 30. 04 2013.] http://www.codeproject.com/Articles/15239/Validation-in-Windows-Presentation-Foundation.

13. **Rowan, Kael.** ZoomableCanvas. *MSDN*. [Online] [Citace: 30. 04 2013.] http://blogs.msdn.com/b/kaelr/archive/2010/07/29/zoomablecanvas.aspx.

14. **Josh Smith, Karl Shifflett.** Tutorial na vytvoření wizardu ve WPF. *CodeProject.* [Online] [Citace: 05. 05 2013.] http://www.codeproject.com/Articles/31837/Creating-an-Internationalized-Wizard-in-WPF.

15. **Nathan, Adam.** *WPF 4 Unleashed.* místo neznámé : Sams Publishing, 2010.

16. *Blender*. [Online] [Citace: 30. 04 2013.] http://www.blender.org/.

12. Seznam použitých termínů

AI – Umělá inteligence (z anglického artificial intelligence).

Entita – Objekt ve hře, který je schopný reakce na podněty, například pes, voják, robot či past . Entity jsou schopny nějaké akce.

Akce – Činnost, které je schopna entita. Například pohyb, seslání kouzla, počátek dialogu apod.

Definující soubor – Soubor, kde jsou implementovány entity a jejich akce.

Definující assembly – Assembly, kde jsou třídy pro entity a akce.

Node – Vrchol Behavior tree v kódu.

Representant – Vrchol Behavior tree použitý v editoru.

LookUp – Vrchol, který se odkazuje na jiný Behavior tree.

Named tree – Behavior tree definovaný na třídě, jenž není entita a je předkem nějaké entity. Na tento typ stromu se dá odkázat pomocí LookUp a tím strom využít na více místech.

13. Přílohy

Přílohou k této práci je reálné nasazení frameworku Revive k jednoduché simulaci. Tato simulace je přiložena na CD a je umístěna ve složce Appendix. Složka Appendix obsahuje dvě složky:

Demo executable

Obsahuje dvě složky `Editor` a `Simulation`. Složka Editor obsahuje binární soubor (*Revive.exe*), který spustí editor. Složka `Simulation` pak obsahuje binární soubor (*Simulation.exe*), který spustí simulaci. Dále obsahuje obrázky pro simulaci.

Po spuštění editoru je třeba kliknout na **Menu → Load**. A vybrat soubor *Simulation.revive* (složka `Simulation`, která se otevře v dialogu při kliknutí). Tím otevřeme Behavior trees, které používá simulace. Pro publikování Behavior trees je pak třeba vybrat soubor *Simulation.exe* (`Appendix/Demo executable/Simulation`). Další informace o práci s editorem je v kapitole 6. Samotná simulace pak bude rozebrána dále.

Demo project

Tato složka obsahuje zdrojové soubory k simulaci. Obsahuje jednu složku `Simulation`. Složka `Simulation` pak obsahuje:

- Debug Revive library – obsahuje ladící verzi *Revive.dll*.
- Final Revive library – obsahuje finální verzi Revive.dll.
- Simulation – obsahuje zdrojové soubory simulace.
- *Simulation.sln* – solution pro simulaci.

Dále rozebereme stručně co simulace dělá a hrubou dokumentaci zdrojových kódů simulace.

Simulace

Simulace obsahuje 3 entity, které dělají následující věci.

Miner Bob

Bob kope zlato v dolu. V případě, že nasbírá 5 zlatých nuggetů, jde si koupit pití. Vše pořád opakuje do té doby, než k němu nepřijde Eliza a oznámí, že je hotové jídlo. Bob v tom případě přeruší práci a půjde se najíst.

Eliza

Elizy hlavní úkol je uvařit jídlo pro Boba. K tomu potřebuje jednu mrkev, bramboru a cibuli. Jakmile nasbírá suroviny, jde ke ohni a uvaří polívku, kterou položí na stůl. Po položení polívky jde oznámit Bobovi, že je polívka připravena a jde si odpočinout k ohni. Po sněyení polívky jde znovu vařit.

Pes

Pes běhá po různých místech či za Elizou nebo Bobem.

Zdrojové kódy simulace

Nebudeme zde rozebírat implementaci a místo toho odkážeme prohlídnutí zdrojových kódů. Zmíníme jen základní strukturu. Projekt obsahuje následující soubory:

- *TemplatesForEntities.xaml* – obsahuje data templaty pro entity.
- *TemplatesForObjects.xaml* – obsahuje data templaty pro neživé objekty.
- *Entities.cs* – obsahuje definici entit.
- *Actions.cs* – obsahuje definici akcí.
- *StuffConnectedWithMovement.cs* – obsahuje definici tříd potřebných k pohybu.
- *GameDataStructure.cs* – obsahuje třídu `MovementCalculator`, která obstarává pohyb entit. Tuto třídu bychom nepotřebovali, ale demonstrujeme na ní předání instancí vlastních tříd v metodě `OnNodeLoad`.
- *NotifyingObject.cs* – obsahuje třídu s implementací `INotifyPropertyChanged`.
- *NotMovingObjects.cs* – obsahuje neživé objekty (obchod, pole apod.).

Simulace je psána s hlavním účelem demonstrovat použití Revive frameworku. Proto je kód psán hlavně jednoduše a srozumitelně.

Projekt používá finální verzi knihovny *Revive.dll*. Pro vyzkoušení si ladění je třeba tuto knihovnu vyměnit za ladící (složka `Debug Revive library`). Jen připomeneme, že v případě použití ladící verze knihovny je třeba nejprve zapnout editor, nahrát projekt a spustit ladění. Poté můžeme spustit simulaci.