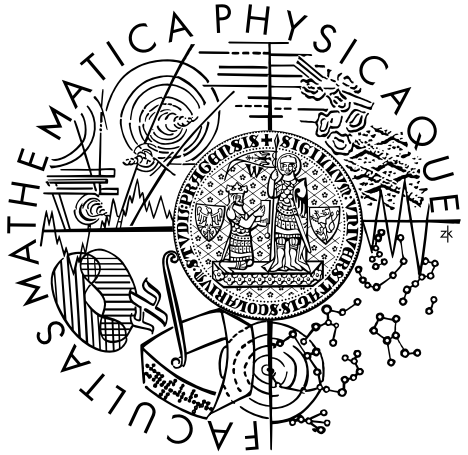
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

BAKALÁŘSKÁ PRÁCE



Petr Geiger

Hra DungeonMaster pro platformu .NET

Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Jezek Pavel Ph.D.
Studijní program: Informatika
Studijní obor: Programovaní a softwarové systémy

Praha 2016

Prohlašuji, že jsem tuto bakalářskou práci vypracoval(a) 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 ..... dne ..... Podpis autora

Název práce: Hra DungeonMaster pro platformu .NET

Autor: Petr Geiger

Katedra: Katedra distribuovaných a spolehlivých systémů

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

Abstrakt: Cilem bakalarske prace je reimplementovat hru Dungeon Master. V soucasne dobe existuje jiz nekolik klonu teto zname hry. Nicmene oproti nim se tato prace zameruje predevsim na nasledujici aspekty. Hra bude naprogramovana v jazyce C# s vyuzitim platformy .NET. Dale cely engine je navrhnuty smerem k udrzitelnosti a rozsiritelnosti. Tzn. s vyuzitim tohoto enginu bude mozne naprogramovat i jinnou hru zalozenych na podobnych zakladech. Ale predevsim by melo byt jednoduche pridavat do enginu nove funkce nad ramec teto prace. Engine by mel take byt priparveny na rozdilne vstupni formaty map. Dale by mela byt kompletne oddelena zobrazovaci vrstva. Vzhledem k povaze projektu by engine mel slouzit jako ukazkovy priklad pouzitelny pro vzdelavani.

Klíčová slova: RPG Dungeon Master vzdělávání

Title: Dungeon Master Game for the .NET Platform

Author: Petr Geiger

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Jezek Pavel Ph.D., Department of Distributed and Dependable Systems

Abstract: Abstract.

Keywords: RPG Dungeon Master education

Děkuji svému vedoucímu práce Mgr. Pavlu Ježkovi Ph. D. jak za pomoc s výběrem práce, tak za cenné rady při její tvorbě.

Obsah

1. Úvod

Originální hra Dungeon Master pochází z dob , kdy její autoři byli značné limitování tehdejším hardwarem i softwarem. Jedná se především o nedostatek výkonu, paměti a absence vyšších programovacích jazyků.Zejména kvůli poslednímu bodu je celá původní hra implementována staticky. Z čehož plyne, že s jakýmkoli rozšířením enginu je zapotřebí upravovat stávající kód. Což dnes již víme, že je značně neefektivní. Datový soubor, který obsahuje data o levelech originální hry, obsahuje následující prvky. Především obsahuje jednotlivé mapy, kdy jedna mapa odpovídá vždy jedné hladině a čím vyšší level tím hlouběji se dostáváme. Každá mapa je složena z dlaždic a má danou výšku a šířku. Celá plocha je vyplněna dlaždicemi různých typů. Je to například zeď, podlaha, teleport, jáma, schody, iluze zdi, otevírací zeď, dveře atp. Na každé dlaždici mohou být objekty. Mohou to být věci, které se dají sbírat a nějak používat. Dále pak příšery, hráč, dveře, teleport a přepínače, které tvoří základní mechaniky dungeonu. Pro každé tyto objekty, je ještě definovaný jejich konkrétní typ. Tzn. může být mnoho druhů přepínačů, příšer, dveří atp.

Jelikož je tato hra velmi známá, existuje již několik klonů, oproti nim se ale tato práce snaží zejména o vyřešení problému s rozšiřitelností a udržitelnosti enginu. Z toho důvodu je celý engine naprogramovaný v jazyce c#, který obsahuje dostatečné silné koncepty pro dosažení tohoto cíle.

Dalším cílem je samozřejmě to, aby byla reimplementace co nejpodobnější originální hře. Avšak již není kladen důraz na t, aby obsahovala veškeré detaily. A to protože z předchozího odstave vyplývá, že by mělo být jednoduché dodělat detaily kdykoliv jindy jako rozšíření. Jelikož se jedná pouze o další rozšíření, na které by měl být engine připravený.

Z předchozích cílů vyplývají následující podcíle, které je třeba vyřešit.

- Samotné rozparsovaní a interpretace dat obsahující herní mapy.
- Reprezentace dungeonu jako celku.
- Reprezentace samotných levelů.
- Reprezentace dlaždic v jednotlivých levelech.
- Reprezentace přepínačů
- Sestavování herních map a inicializování objektů
- Renderování a interakce s objekty.
- Reprezentace herních entit.
- Uspořádaní entit na dlaždicích
- Reprezentace vlastností a schopností entit
- Reprezentace relací mezi entitami
- Reprezentace těl entit a jejich inventářů

- Reprezentace věcí objevujících se v dungeonu
- Reprezentace akcí s věcmi.
- Reprezentace kouzel.

2. Related works

3. Analýza

Tato kapitola pojednává o postupech a řešeních použitých pro jednotlivé části enginu. Jsou zde popsány jak slepé a špatné návrhy, tak návrhy, které se ukázaly jako nejlepší.

3.1 Parsování binárních dat

Aby bylo celé dílo vůbec proveditelné, bylo nutné zajistit, že bude možné rozparsovat data map originální hry. K tomuto účelu dobře posloužila existující technická dokumentace (viz. **?**). Zmíněná dokumentace obsahuje popis binárního formátu herních map. Není však vždy úplně jasný vztah konkrétních věcí vzhledem k celému projektu. Z toho důvodu se při interpretaci dat muselo projít mnoho slepých cest metodou pokus omyl.

I z předchozího důvodu se ukázalo lepší vytvořit samostatnou vrstvu, která data rozparsuje do datových objektů, ke kterým je přistupováno z vrstev vyšších. Formát herních map obsahuje některá specifika, kvůli kterým, bylo vhodné nad samotnými rozparsovanými daty provést ještě post-processing. Například se jedná o uložení objektů, na dlaždicích. Každá taková dlaždice má v původním formátu spojový seznam tzv. objektových identifikátorů. Tyto identifikátory obsahují typ věci, na který odkazují a potom index do datové struktury daného typu. V rámci post-processingu je v každé dlaždici vytvořen seznam pro konkrétní typ. Tyto seznamy jsou pak naplněny daty ze spojového seznamu. Tento přístup ulehčí a zpřehlední práci vyšší vrstvě používající tato data.

Výše popsaná data obsahují pouze popis herních map. Samotné vlastnosti daných objektů, které nejsou ovlivnitelné při návrhu herních map neobsahují. Tyto data jsou obsažena v dalších binárních souborech. Nicméně tuto práci již odvedlo mnoho jiných a jsou například vystavená na webových stránkách v HTML formátu (viz. **?**). Proto jsem se rozhodl tuto práci využít a rozparsovat přímo HTML data do datových objektů. Tato fáze je opět přidána do parsovací vrstvy. K usnadnění práce s HTML jsem použil knihovnu **?**. Některé části HTML souborů jsou manuálně upraveny, pro usnadnění parsování. Z těchto souborů jsem také získal některé textury, například pro věci, přísery, a portréty šampionů. Další potřebné textury jsou buď z vlastni tvorby nebo získané pomocí extraktoru (**?**).

Tato vrstva je navíc oddělena do speciálního projektu a to z toho důvodu, že v době její tvorby ještě nebylo rozhodnuto jaký framework bude použit pro přístup ke grafické kartě.

3.2 Použití frameworku pro práci s grafickou kartou

Celý projekt není nikterak extrémně vázaný na konečně vybranou platformu. Samozřejmě pro použití jiné platformy by bylo třeba provést portaci. Ale jelikož použitý rendering je pouze ve fázi proof of concept, neobsahuje žádné pokročilejší akce s grafickým hardwarem. A je tedy závislý pouze na několika strukturách.

Troufám si tedy říct, že případná portace by s dostatkem času nebyl až takový problém.

I z předchozího důvodu jsem se nakonec rozhodl přistoupit k výběru frameworku, s kterým už některé zkušenosti mám. Jedná se tedy o XNA Framework od firmy Microsoft. Nicméně protože tento framework již není firmou nadále podporován a vyvíjen, použil jsem konečně jeho klon MonoGame (**?**). Velkou výhodou je, že framework je multiplatformní. Jeho tvůrci tvrdí, že podporuje platformy iOS, Android, MacOS, Linux, Windows , OUYA, PS4, PSVita, Xbox One.

3.3 Reprezentace dungeonu jako celku

Dungeon je logicky rozdělen do jednotlivých levelů, o tyto levely se pak stará samotné jádro enginu. Jádro enginu se ve výsledku stará pouze o načítání levelů a vykreslování správné oblasti dle pozice hráče. Pro načítání levelů se používají buildery, které jsou zároveň bodem rozšiřitelnosti. Naprogramováním různých builderů je dosaženo toho, že engine dokáže načítat mapy v různých formátech.

Při každé změně pozice hráče se vyhledají viditelné dlaždice. Pokud je některá z těchto dlaždic označená jako spojnice levelů, engine propojí tyto dlaždice s odpovídajícími dlaždicemi v cílovém levelu. V originální hře jsou takové dlaždice například schody, teleport nebo jámy. Nicméně je opět na programátorovi, jaké dlaždice tak označí. Viditelné dlaždice jsou potom aktualizované a renderované.

3.4 Reprezentace levelů

Samotné struktury levelů obsahují pouze seznam dlaždic na dané úrovní dungeonu, identifikátor levelu a seznam živých entit. Engine vždy pracuje se třemi posledně načtenými levely.

3.5 Reprezentace dlaždic

V originální hře jsou dlaždice vždy uloženy v nějakém obdélníkovém poli. To znamená, že dlaždice, které nejsou využity pro chodby , jsou vždy vyplněny zdmi. Za účelem udělat engine co nejdynamičtější, nerozhodl jsem se ukládat dlaždice do obdélníkového pole, ale reprezentovat vazby mezi nimi jako obousměrný orientovaný graf. Jednak pro hodně řídké mapy to může ušetřit nějaké místo. Nicméně důležitějším důvodem bylo, že při takovéto reprezentaci lze jednoduše získat sousedy pouze ze znalosti konkrétní dlaždice. Kromě toho je takto celý engine obecnější. Například by takto bylo jednoduší rozšířit celý engine tak, aby nemusely byt jednotlivé dlaždice na mřížce nebo aby mohli mít dlaždice více sousedů. Byla by to sice velký zásah do celého enginu, ale mnoho částí by se takto dalo znovu použít nebo jen drobně upravit.

Předchozí rozhodnutí má následující důsledek. Například již nejsou potřeba dlaždice typu zeď. Což je dost výrazná změna oproti originální hře, kde dekorace, přepínače, věci atp. jsou uložené právě v dlaždicích typu zeď. Namísto originálního přístupu jsem se rozhodl, že zdi boudu definovány v samotných dlaždicích typu podlaha. Toto rozhodnutí vedlo také k některým problémům. Například

pro vytváření podlahy bylo nutné číst i další dlaždice kolem té právě vytvářené. Zejména je pak nutné posílat zprávy původně cílené zdím na odpovídající dlaždice podlahy. Ve výsledku si myslím, že tato reprezentace je vhodnější například kvůli možnému rozšíření popsaném v předchozím odstavci. Zdá se být také intuitivnější a tedy srozumitelnější reprezentací.

Zdi jsou tedy součástí dlaždic. Nyní přichází otázka, jak tyto zdi tedy reprezentovat. Buď přímo v samotných dlaždicích nebo zeď udělat jako zvláštní objekt, který je součástí dlaždic. Ze začátku byl v enginu použit první zmiňovaný způsob. Ukazovalo se ale, že tato reprezentace není příliš vhodná, protože potom samotná dlaždice řeší věci, které k ní přímo nenáleží. Nakonec tato reprezentace byla změněna a v enginu bylo použita druha zmiňovaná možnost. Vedlo k ní zejména předělání oddělené grafické vrstvy od té logické. Výhoda tohoto přístupu byla kromě samotného oddělení kódu i možnost znovu použití kódu za pomocí dědičnosti. Například zeď s nějakým přepínačem může dědit z normální prázdné zdi. Nicméně ukázala se i nevýhoda tohoto přístupu. A to že komunikace směrem ze zdi k dlaždici je trochu těžkopádná. Buď by bylo zapotřebí předat zdi referenci na jejich rodičovské dlaždice nebo se museli použít události. Pokud to bylo v některých případech třeba, přiklonil jsem se k použití událostí.

3.6 Přepínače

3.7 Popis funkce a složení

Základní mechaniky map dungeon Masteru tvoří tzv. přepínače. Jsou to objekty, které lze nějakým způsobem aktivovat. V originální hře je to možné kliknutím myši na dekoraci přepínače, přesunem na dlaždici s přepínačem nebo aktivací jiným přepínačem. Ve skutečnosti se každý přepínač může skládat z několika senzorů, kde poslední z nich určuje dekoraci přepínače. Každý senzor přepínače obdrží aktivaci a případně provede svoji akci. Každý sensor může provést lokální nebo vzdálenou akci. Pro lokální akce je to zarotování sekvencí senzorů přepínače nebo přidání zkušeností hráči. Pro vzdálené akce je to odeslání zprávy, která se skládá ze samotné akce, tj. deaktivace, aktivace, přehození stavu. A dále obsahuje směr, který může být interpretován jako strana dlaždice typu zdi, která má být aktivována. Nebo pro speciální přepínače může určovat jinou akci. Kromě toho obsahuje další vlastnosti jako typ, zda je opakovatelný, zda je použitelný pouze jednou, jakou ma dekoraci, doba, po které se provede akce.

Výše popsaný systém přepínačů přináší hned několik problémů: Jakým způsobem efektivně a přehledně rozparsovat data senzorů. Z technické dokumentace není zcela zřejmé, jak se mají jaké vlastnosti senzorů použit. Jak vůbec reprezentovat přepínače. Jakým způsobem provádět vzdálené akce.

3.7.1 Reprezentace přepínač

První reprezentace

Jako první způsob jsem se rozhodl vždy pro sekvenci senzorů originální hry vytvořit objekt, který měl stejné nebo o něco obecnější vlastnosti dané sekvence. Ze začátku se totiž zdálo, že jen výjimečně jsou sekvence senzorů větší či složitější.

Proto jsem se rozhodl vytvořit v novém enginu objekty, které mají obecnější vlastnosti pro několik sekvencí. A určitá sekvence senzorů si poté inicializuje objekt podle svých vlastností. Vlastnosti senzorů jsem určoval podle textového popisu v dokumentaci, která nebyla úplně dostačující. I proto čím více přepínačů jsem naimplementoval, tím více ukazovalo problémů. Nakonec jsem dospěl k závěru, že tento způsob reprezentace je nemožný.

Druha reprezentace

Jiná varianta se stále spoléhala na typ objektů z předchozího případu. Parsování sekvence senzorů jsem se rozhodl udělat pomocí konečného automatu. Tedy tak, že pro každý objekt existovaly předdefinované sekvence senzorů. A tak se pro každou sekvenci senzorů našel správně reprezentující objekt. Později se ale ukázalo, že i malá změna v sekvenci senzorů může generovat podobný objekt. Takže by bylo třeba spoustu předdefinovaných šablon a jednoduše by mohl nastat případ, že pro některou sekvenci neexistuje žádný objekt.

Třetí reprezentace

Nezbylo teď jiné možnosti než se přiblížit více k implementaci podobné originální hře. Za tímto účelem jsem se začal poohlížet po po zdrojových kódech originálu. Originální zdrojový kód pravděpodobně nebyl a nebude nikdy publikovaný. Naštěstí se ale našel člověk, který celou hru dekompiloval, takže po zkompilování tehdejším kompilátorem výsledná binárka odpovídala té originální (viz. **?**). Kód to je sice těžko čitelný, ale obsahuje přesný popis, jak které senzory mají fungovat. Vytvořil jsem tedy objekt přepínač, který má v sobě sekvenci senzorů, tak jako tomu je v originální hře. S využitím zdrojových kód jsem vytvořil odpovídající objektově orientovaný kód v jazyce c#. Je tedy například možné dovytvořit další nové senzory. V bylo toto rozšiřovaní jen značně omezené, jednotlivé type senzorů se odlišovali číselným identifikátorem. Jelikož jsem ale nechtěl případné rozšiřitele enginu limitovat tímto způsobem tvorby herních mechanik. Je samozřejmě možný, vytvořit jiný přepínač, který se bude chovat tak, jak si programátor zvolí. Takže případné nové přepínače nemusí vůbec senzory používat. To jak funguje senzor uvnitř je tedy čistě na programátorovi.

Reprezentace zprávy přepínače

Jak již jsem psal, celý systém přepínačů je závislý na posílání zpráv. V originální hře je cíl odeslané zprávy vázán na souřadnice dlaždice, na kterou se má zpráva odeslat. Jelikož jsem se v enginu nechtěl na plno vázat na tyto pevné souřadnice, rozhodl jsem se to i zde udělat trošku jinak. Například jsem zmiňoval, že je principiálně možné, aby dlaždice měla více sousedů, u kterých by pak souřadnice nešli přesně nastavit. Z tohoto důvodu jsem se rozhodl přepínačům při jejich inicializaci předat referenci na jejich cílovou dlaždici. To řeší problémy, které jsem popsal v předchozích větách, nicméně přináší pár dalších (viz. inicializace). Jak jsem již zmiňoval, v originální hře jsou zprávy opět reprezentovány pevnou datovou strukturou. V této části jsem chtěl dát také programátorovi větší svobodu. A to tak, aby bylo možné posílat vlastní zprávy alespoň vlastně vytvořeným dlaždicím. Toho je nakonec možné dosáhnout tak, že vlastní dlaždice bude poděděna

z generické dlaždice, která bere jako typový parameter typ dané zprávy. Zpráva pak musí alespoň dědit ze základní zprávy, která odpovídá zprávě v původní hře.

3.8 Builder map

Builder map je objekt, který je schopen dodávat levely samotnému enginu. A stojí na něm, aby vytvořil potřebné mapy z dostupných dat. Vytvoření builderu je samozřejmě opět na programátorovi. Builder by se měl také start o případně kešování, již načtených levelů. Tato práce obsahuje pouze builder schopny rozparsovat mapy originální hry.

Jelikož jsem si původně myslel, že některé zdi nebude vůbec nutné parsovat, rozhodl jsem se v prvotních fázích procházet data map pomocí algoritmu depth first search. Nakonec se ale ukázalo, že i některé nepřístupně části map jsou používané. Například pomocí teleportu, nebo vazbami mezi přepínači. Proto jsem později zvolil techniku, že projdu všechny dlaždice od shora dolů. A podle nich vytvořím odpovídající objekty v novém enginu.

3.9 Inicializace a sestavení herních map a objektů

V celém projektu jsem se snažil držet paradigmatu defenzivní programovaní. Tedy jde o to nechávat veřejně pouze takové položky, u kterých je to nezbytně nutné. Na to navazuje problém s inicializací takových tříd. jelikož například přepínače potřebují referenci na cílovou dlaždici, ale zároveň jsou obsahem dlaždice, je třeba oddělit fázi inicializace dlaždic od inicializace přepínačů. Potom ale nejde přepínače předat do dlaždic, aniž by to narušovalo cíl držet se defenzivního programovaní. Kromě toho mají konstruktory takových tříd mnoho parametrů. Nejprve jsem inicializaci řešil právě těmito konstruktory, ale z předchozích důvodu jsem přišel s tzv. inicializátory.

Inicializátor je datová třída obsahující vlastnosti, které by normálně, byly parametry konstruktoru inicializovaného objektu. Místo těchto parametrů je do konstruktoru předán inicializátor. Inicializátor má také události vyvolané při inicializaci a při dokončení inicializace. Třída beroucí inicializátor jako parametr konstruktoru se zaregistruje na tyto události. Tím pádem není nutná žádná přebytečná položka ve třídě pro inicializátor. události jsou pak volané skrze metodu v inicializátoru. Inicializátorem inicializovaná třída si z něj nakopíruje parametry a odhlásí událost. Od té chvíle už není možné třídu modifikovat. Data inicializátoru se tedy mohou postupně naplňovat a po jejich naplnění se inicializace provede zavoláním jejich metody. S využitím a asynchronních metod, je navíc možné vyvolat inicializaci prvku (např. přepínačů, které potřebují reference na dlaždice) již při vytváření dlaždic. Což vede k přehlednějšímu kódu a celkově k inicializaci cyklických struktur, bez porušení našeho požadavku defenzivního programování.

S využitím dědičnosti inicializátorů je možné nasimulovat volání rutin konstruktorů, tak jak je v c# běžné. Na každé úrovní dědičnosti se využijí některé vlastnosti inicializátoru. Nechť inicializátor B je potomek inicializátoru A v hiearchii dědičnosti. A nechť C a D jsou třídy, které chceme inicializovat a zároveň D

je potomkem C. A také platí, že konstruktor třídy D bere inicializátor třídy B a konstruktor třídy C bere inicializátor A. potom inicializátor B můžeme použít pro oba konstruktory tříd C i D. Přičemž na každé úrovni dědičnosti inicializátoru může být zvláštní inicializaci oznamující událost. Pokud tedy inicializátor volá inicializační události v e správném pořadí, může to nasimulovat pořadí inicializace běžné u konstruktorů. Při inicializaci dlaždic je právě tento způsob používán.

3.10 Renderování a interakce

Interakcí je myšleno reakce kliku myší na herní objekt. Z počátku si renderovaní řešil každý objekt sám v sobě. Pozicování těchto objektů bylo z pravidla absolutní. Naopak interakce byla zase řešena pro všechny objekty stejně pomocí bounding boxu. Obojí se později ukázalo jako špatný přístup.

Namísto toho vznikla nová vrstva, oddělená od samotných objektů zajišťující jak renderování tak interakci. Každý objekt který to vyžaduje má tak vlastní renderer-interactor na míru. Jak takový objekt vypadá uvnitř je zpravidla na programátorovi. Nicméně obvyklý přístup je takový, že renderer má referenci na konkrétní renderovaný objekt. Podle jeho zpravidla readonly vlastnosti potom určuje chování vykreslovaní nebo interakce. Pokud se jedná o renderery pro statické objekty (např. dlaždice), tak se zpravidla pozicování určuje relativně vůči rodiči. Takže interakční či renderovací funkce dostane jako parametr dosavadní transformační matici.

Celá výhoda tkví v tom, že za takovýchto podmínek je možné renderovací-interakční vrstvu libovolně přepracovat. Oproti originální hře , je právě v tomto projektu odlišně provedené renderování. Nicméně nic nebraní tomu napsati s vlastní vrstvu, tak aby vpadala jako originál.

3.11 Herní entity

3.11.1 Úvod

V originální hře existují pouze dva typy entit. Jsou to nepřátelské příšery a hráčovi šampioni. Každé tyto entity se mohou lišit v závislosti na jejich vlastnostech. Tedy ve hře existuje několik druhů šampionů a několik druhů příšer. V originální hře jsou tyto dva světy naprosto oddělené. Já jsem se nicméně rozhodl udělat celou situaci o něco obecnější.

Entitou v tomto enginu může být téměř cokoliv, s čím je potřeba nějak interagovat ve smyslu bojů. Tedy entita je cokoliv co má nějaké vlastnosti, které jsou ovlivnitelné například útokem. Entity si dále definují tzn. uspořádaní na dlaždici. Takže například dveře jsou entita, která má vlastnosti jako zdraví, odolnost atp. Tento příklad je ukázkovou věcí, která je v originální hře řešena staticky. S využitím entit v těchto případech je možné dosáhnout větší obecnosti a tím pádem je se možné vyhnout zbytečně specializovanému a podobnému kódu.

Kromě entit jsou v enginu ještě tzv. živé entity. Což jsou entity s tím rozdílem, že kromě vlastností mají také schopnosti. Dále mají definovanou pozici dle uspořádaní, relaci vůči ostatním živým entitám(zda jsou přátelští či nepřátelští) a konečně části těla a inventáře.

Z hlediska enginu jsou teď oproti originální hře šampioni i příšery na rovnocenné úrovni.

3.11.2 Vlastnosti a schopnosti entit

Cílem nového enginu je opět poskytnout možnost rozšíření vlastností a schopností. Za následujících podmínek je teď možné na tomto enginu postavit hru, která si sama definuje, jaké vlastnosti a schopnosti, které entity budou mít. Dále je možné nadefinovat akce, které tyto vlastnosti a schopnosti budou využívat. Tim je dosaženo opět další dynamičnosti enginu.

Vlastnosti

V původní hře jsou vlastnosti opět pevně definované. Jsou to například vlastnosti jako zdraví, mana, výdrž, síla atp. Šampioni a příšery mají většinu vlastnosti odlišných, tím pádem i způsob jejich použití je odlišný. Jak jsme si ale řekli v předchozí sekci, tak příšery i šampioni v novém enginu jsou nyní na stejné úrovni. Zároveň některé entity některé vlastnosti mít nemusí a tím pádem jsou u nich tyto vlastnosti v základní hodnotě. Z toho důvodu jsem se rozhodl nechat seznam vlastnosti na každé entitě zvlášť zcela dynamicky. Pokud daná entita některou vlastnost nemá, vrátí svoji základní hodnotu. Potom při konání některé akce ovlivňující vlastnosti dané entity se akce sama rozhodne, které vlastnosti nějakým způsobem použije. Například pokud provádíme akci útok magickou ohnivou koulí a entita nemá žádné vlastnosti odolnost proti magii či odolnost proti ohni. Jsou tyto vlastnosti v základním stavu, čili nijak neovlivní damage způsobené touto akcí.

Schopnosti

Schopnosti jsou stejně jako vlastnosti v originální hře pevně definovány. I v tomto bodě se nový engine snaží o větší dynamičnost. Opět ne všechny entity musí mít všechny schopnosti. Pokud určitou schopnost entita nemá, vrátí se její hodnota s levelem nula. Podle schopností se určuje, zda je entita danou akci schopna provést. Tedy každá akce má definovaný minimální levely pro sadu schopností. Pokud pak levely vyhovují, může být daná akce provedena.

V originální hře jsou dva typy schopností. Jsou to základní schopnosti a skryté schopnosti. Pokud některá akce vylepšuje schopnost skrytou rozdělí se získané zkušenosti mezi obě schopnosti. Tento koncept schopností není v enginu vyžadován. Je možné si udělat jednoúrovňové schopnosti nebo klidně několika úrovňové schopnosti. Vše záleží na tom jaké chování programátor daným schopnostem naprogramuje.

3.11.3 Relace mezi entitami

Další funkcí kterou oproti originálu engine umožňuje je definovat pro entity jejich nepřátelé. Každá živá entita má token identifikující skupinu. Entity v této skupině jsou mezi sebou přátelské. Každá entita si může nadefinovat svoje nepřátelské tokeny. Podle vzoru originálu jsou v této konkrétní hře pouze dva relační tokeny. Jeden pro šampiony a druhý pro příšery. Nicméně celý tento system je

navržen právě pro stanovení obecnějších vztahů mezi entitami. To může být aplikováno v libovolném rozšíření hry.

3.11.4 Tělo a inventáře

Každá živá entita má definované její části těla. Oproti originálu, kde se řeší pouze části těla šampionů, zde je možné definovat tělo pro každou entitu. Je tak možné pracovat s částmi těla entit, které nemají humanoidní formu. Některé části těla se dají použít jako úložiště věcí. Věci potom mají nadefinováno s jakým typem úložiště jsou kompatibilní. Tato reprezentace dává dobrý smysl. Například pokud máme lidskou část těla - nohy. A medvědí část těla - nohy. Tak lidské kalhoty by měli jít nasadit pouze na lidské nohy, nikoliv medvědí. Naopak medvědí chrániče na nohy půjdou dát pouze na nohy medvědí. Takto pokročilejší věci v originální hře nejsou použity, věci tam mohou sbírat pouze šampioni. Nicméně pro případné rozšiřitele je principiálně možné navrhnout hru, v které se tyto techniky budou používat.

Kromě částí těla jsou zde ještě definovány další externí úložiště, které slouží jako hlavní úložný prostor. O velikosti a typu těchto úložišť opět rozhoduje tvůrce konkretních entit. Nicméně stejně tak jako v originální hře, i tato instance má implementované úložiště pouze pro šampiony. Mezi taktové úložiště patří například batoh, kapsa, bedna, toulec atp.

3.12 Věci

Tato sekce pojednává o věcech, které se objevují v mapách hry Dungeon Master. Myšleno věci, které jdou sebrat ze země, vložit do ruky hráče či šampiona, vložit do inventáře či úložiště kdekoliv v dungeonu(např. do výklenků). Takovéto věci mohou zejména definovat akce, které s nimi lze provádět. Věci v originální hře se dělí do několika skupin tj. zbraně, zbroje, lektvary, truhly a další. Každá věc může mít několik akcí a je na uživateli enginu, jakou akci dané věci přiřadí.

V originální hře má každá věc unikátní číselný identifikátor, který používá pro jednoznačnou identifikaci typu předmětu. Nikoliv typu jako typ proměnné, ale jako například jeden konkrétní typ zbraně. Tyto identifikátory jsou používány především u různých druhů klíčů, ale není to podmínkou. Identifikátory jsou obecně používané sensory, například tak, že konkrétní senzor se aktivuje pouze pokud má hráč v ruce věc s určitým identifikátorem. Není tedy pravda, že každá instance, například klíče, má rozdílný identifikátor. Naopak, všechny instance klíče stejného druhu, mají stejný identifikátor.

Z počátku jsem pro stanovení identifikátoru zvolil stejnou strategii jako tvůrci originální hry, ale celou dobu se mi to zdálo jako dosti dřevní řešení. Tento problém nakonec vyřešil problém jiný a to jakým způsobem přesně reprezentovat věci. Tak aby neměli zbytečně položky uložené několikrát a aby se dali snadno vytvářet. Ze začátku měli všechny věci uložené všechny jejich vlastnosti u sebe. Bylo to tak i s položkami, které byly vždy pro instanci jednoho typu stejné. Tak jsem se tyto položky rozhodl delegovat do zvláštních tříd, odpovídající jednotlivým identifikátorům. Reference těchto tříd tedy není slouží jako identifikátory. Samotné instance věcí potom obsahují právě reference na tyto třídy.

Navíc třídy popsané v předchozím odstavci slouží jako továrny na věci jejich typu jakožto identifikátoru. Se znalostmi továrny lze buď vytvořit věc se základními hodnotami jejich vlastností. A nebo se znalostmi konkrétního typu(myšleno jazykového) továrny, lze předáním inicializátoru vytvořit novou věc s hodnotami dle inicializátoru. Každá továrna má také následující vlastnosti: hmotnost, jméno, akce, které lze s věcmi provádět, a definice míst v inventáři , kam lze věc uložit. Specializované typy, myšleno jako zbraně, lektvary, atp. pak mají některé další vlastnosti.

3.13 Akce

3.14 Kouzla

4. Programátorská dokumentace

Následující sekce této kapitoly jsou určeny především pro lidi, kteří chtějí porozumět více celému enginu. Především však dobře poslouží lidem, kteří by tento engine chtěli použít pro svoji hru nebo by chtěli jen rozšířit některé části tohoto enginu Dungeon Masteru.

4.1 Jádro enginu

Jádro enginu tvoří třída **DungeonBase**, jak již bylo řečeno, stará se zejména renderování, aktualizování herních objektů, inicializaci hráče, načítání a propojování herních map. Pokud má čtenář zájem o úpravu některých z těchto věcí, je tu správně.

4.1.1 Renderovaní

Tato sekce slouží pro čtenáře, kteří mají zájem o reimplementaci následujících věcí:

- Způsob výběru dlaždic, které se mají používat pro aktualizaci a rendering.
- Pořadí v jakém se jednotlivé dlaždice renderují.
- Použití osvětlení a celkově nastavení grafického zařízení.
- Dosah viditelnosti
- atp.

Výběr a vykreslování použitých dlaždic

V základní verzi vždy existuje kolekce právě používaných dlaždic. Tato kolekce se znovu naplní vždy, když hráč změní pozici. dlaždice se zde vyhledávají algoritmem breath first search.

Vlastní algoritmus výběru dlaždic je možný přidat poděděním třídy **DungeonBase** a overridnutím metody **UpdateVisibleTiles**. Metoda uloží do proměnné **currentVisibleTiles** vybraný seznam dlaždic. Tato proměnná je pak využívána v metodě **Draw**, kde se ale renderují v opačném pořadí kvůli průhlednosti.

Nastavení grafického zařízení

Nastavení osvětlení a celkově efektu, textury a batcheru pro vykreslovaní minimapy, se provádí v metodě **InitializeGraphics**. Jejím overridnutím je možné provést modifikace. Pokud nehodláte upravovat všechny popsané věci, je doporučené zavolat nejprve funkci rodiče která data zinicializuje na základní hodnotu. Samotná minimapa se vykresluje funkcí **DrawMiniMap**, která se volá klasicky ve funkci **Draw**. Overridnutím této funkce je tedy možné například minimapu úplně odstranit, tak jako to je v originální hře. V této sekci je ještě možná dobré

zmínit proměnou **FogHorizont**, která je v základu používaná jednak jako maximální vzdálenost, po kterou jsou dlaždice hledány, a za druhé slouží v efektu jako hodnota FogHorizontu.

4.1.2 Inicializace hráče

O abstrakci různých způsobů načítaní levelů se stará interface **IDungeonBuilder**. Instanci implementace tohoto rozhraní je nutné předa Dungeonu v konstruktoru. V základu se nové mapy načítají v momentu, kdy některá ze zobrazených dlaždic je dlaždice navazující na další level. Takové dlaždice jsou právě dlaždice implementující rozhraní **ILevelConnector**. Poslední tři levely získané skrze **IDungeonBuilder** jsou uložené v kolekci **ActiveLevels**. Pokud je nutné změnit strategii načítání levelů, je třeba overridovat metody **SetupLevelConnectors** popř. **ConnectLevels**. Pro změnu strategie ukládání a mazání aktivních levelů, je třeba podědit třídu **LevelCollection**. Kterou je pak nutno nastavit do vlastnosti **ActiveLevels**.

Následující sekce popisují jaké objekty lze jakým způsobem rozšířit či upravit. O tom jak takové nové objekty potom používat pojednává pozdější sekce.

4.2 Rozšiřitelnost dlaždic

4.2.1 Popis dlaždic

Nejobecnější strukturu dlaždice definuje interface **ITile**. Dlaždice tedy musí obsahovat pozici vzhledem k mřížce dlaždic a potom level, ve kterém se nachází. Tyto vlastnosti používají příšery nebo hráč k určení své pozice. Dále obsahuje vlastnosti sloužící k rozhodovaní pohybu entit, věcí a orientaci příšer. Další částí je již zmiňovaný **LayoutManager**, který slouží k rozdělení prostoru mezi entitami na dlaždici. Podle něj se příšera či hráč rozhoduje, zda může na danou dlaždici vstoupit a obsadit, tak část jejího prostoru. Má také definované soused, podle kterých se zase entity rozhodují při pohybu mezi dlaždicemi. Další část API slouží k modifikaci stavu dlaždice. Stav jde buď přímo nastavit pomocí volaní odpovídajících funkcí nebo předáním zprávy. Poslední část obsahuje metody a události pro vstup a odchod věcí z dlaždice. O volání těchto funkcí se musí starat samy objekty, které chtějí vstoupit či odejít. Takové objekty jsou poté skrze dlaždici aktualizovány, pokud implementují rozhraní **IUpdateable**. To však již musí zařizovat jednotlivé implementace dlaždic, o niž bude řeč v další sekci.

4.2.2 Inicializace dlaždic

Jak již bylo zmíněno v analýze, k inicializaci dlaždic jsou použity tzv. inicializátory. Inicializátor obsahuje vlastností, které by normálně byly předány jako parametry v konstruktoru. Ovšem v moment předávaní inicializátoru do konstruktoru, inicializátor ještě nemusí být plný. Namísto toho má události **Initializing** resp. **Initialized**, které jsou vyvolány při resp. po inicializaci. Na tuto událost je třeba zaregistrovat inicializační funkci, která zkopíruje data z inicializátoru do samotného objektu. Pro každou úroveň hierarchie dědičnosti jsou určeny zvláštní vlastnosti a zvláštní inicializační události. Rodičovské události inicializátoru jsou

vždy po zdědění rodičovského inicializátory zakryty novými inicializačními událostmi pro danou proveň dědičnosti. Tento způsob je použit pouze pro inicializaci dlaždic. Pro jiné objekty, může inicializátor sloužit pouze jako objekt udržující parametry. Parametry, které jsou hned v konstruktoru inicializované. To vše záleží na konvice kterou si programátor zvolí.

4.2.3 Implementace dlaždic

O částečnou implementaci rozhraní **ITile** se stará třída **Tile**. Definuje základní layout manager, nicméně je ho popřípadě možné overridovat vlastní implementaci. Zejména se pak ale stará o inicializaci pozic na mřížce, levelu a sousedů skrze **TileInicializator**. Dále abstraktně deklaruje vlastnost **SubItems**, na jejichž objektech, které implementují **IUpdateable** volá metodu **Update** skrze metodu **Update** na dlaždici. Abstraktně deklaruje také vlastnosti **Sides**, kde těmto stranám přeposílá případné zprávy, které přišli na tuto dlaždici skrze metodu **AcceptMessageBase**. Dále implementuje také funkce pro vstup/odchod objektů tím, že vyvolá odpovídající události na dlaždici. Pokud tedy chcete volání těchto událostí zachovat, je při případném overridování metod nutné zavolat implementaci rodiče. Všechny předchozí popsané funkce je možné overridovat v potomkovi a tak přizpůsobit prováděné akce.

Přímý generický potomek **Tile⟨TMessage⟩** poskytuje již zmiňovanou možnost přijímat v potomcích vlastní zprávy. Poděděním tohoto typu, specifikováním typu zprávy v typovém parametru a následnou implementací metody **AcceptMessage** lze definovat rutinu při příjmu zprávy. Tato třída overriduje a zároveň uzavírá metodu rodič **AcceptMessageBase**, která deleguje zprávy typu **TMessage** do metody **AcceptMessage**. Naopak základní implementace metody **AcceptMessage** je zavolání právě rodičovské implementace **AcceptMessageBase**. Takže při případném overridování této funkce stojí za zamyšlení, zda-li je třeba volat implementaci rodiče či nikoliv. Všechny dlaždice, které dědí ze třídy popsané v tomto odstavci vytváří potomky dva. Jednoho pro případně rozšiřitele, která ponechává typový parametr.A druhá, která definuje typový parametr na obecný typ **Message**. Ještě stojí z zmínku, že každá zpráva musí dědit právě ze třídy **Message**.

Dlaždice podlaha

Za zmínku ještě stojí přímý potomek třídy **FloorTile⟨TMessage⟩** a to **FloorTile⟨TMessage⟩**. Tato třída se stará jak o interakce, tak rendering zdi a podlahy. Stará se tady například o možnost pokládaní věcí na zem, či u zdí do případných výklenků. Dále se stará o zobrazování a aktivování případných přepínačů. Z toho důvodu, pokud se chystáte vytvářet nějakou novou dlaždici, je dobré se zamyslet, jestli nevyužít již tuto implementaci. Nicméně nic nebrání tomu podědit přímo z **Tile⟨TMessage⟩**, pokud by tato implementace z nějakého důvodu nevyhovovala. Při drobných úpravách je zase možné třídě podstrčit pozměněné-poděděné implementace jejich stran, ať už zdí nebo podlahy. Třída deleguje veškeré akce vstupu/odchodu do samotné podlahy typu **FloorSide**. Stejně tak implementace kolekce **SubItems** se deleguje na podlahu.

Další dlaždice

Další dlaždice již převážně využívají dědičností právě již zmíněnou předchozí podlahu. Výjimkou jsou například schody. Schody jsou příkladem dlaždice, který implementuje rozhraní **ILevelConnector**, které požaduje implementaci následujících vlastností

- cílový level
- pozice cílové dlaždice v cílovém levelu

Jak už bylo zmíněno engine potom při načtení levelu nastaví poslední položku tohoto interface **NextLevelEnter** na odpovídající dlaždici. A je už na samotné dlaždici, aby při změně této vlastnosti udělala potřebné akce. U schodů je to například nastavení sousedů vedoucích do dalšího levelu. Právě zde se například hodí overridování vlastnosti **Neighbors** vlastní třídou starající se o sousedy. Tato možnost je použita ještě u jámy, kvůli propadu do nižšího levelu. Rozhraní spojující levely je pak ještě u teleportu, kde při vstupu na teleportační dlaždici a splnění požadavku teleportu se tento objekt teleportuje na jinou dlaždici. Za zmínku ještě stojí existence rozhraní **IHasEntity**, které obsahuje vlastnost typu entity. Implementuje ho například dlaždice typu dveře, která vrací jako entity dveře. Tímto způsobe lze pak například útokem dveře rozbít, protože jsou definované jako entita s vlastností zdraví, odolnost atp. Nikde jinde v enginu toto využito není, tím více prostoru mají případní rozšiřité.

4.2.4 Strany dlaždic

Jak již bylo naznačeno v předchozích sekcích, dlaždice mohou mít strany. To mohou být ať už zdi, tak i podlaha nebo strop. Každou tuto stranu reprezentuje samostatný objekt, který má samostatný renderer-interactor. Poděděním z těchto hotových tříd si lze tedy ušetřit nějakou práci. Není to však nutnost, je možné si pro své dlaždice implementovat vlastní strany implementací rozhraní **ITileSide**. Toto rozhraní vyžaduje pouze položku pro renderer a z kompatibilních důvodů pro starou verzi požaduje, aby byla schopna přijímat zprávy **Message**.

Jednotlivé implementace

Třída **TileSide** slouží jako rodič všech zdí implementovaných ve hře. Nereaguje nijak na interakci hráče a její renderer pouze renderuje zdi texturu na správné místo, popř. dekoraci.

Jejím přímým potomkem je třída **ActuatorTileRenderer**,ta navíc obsahuje přepínač a její renderer se navíc stará o jeho vykreslování a interakci.

Dalším potomkem je třída **TextTileSide**, jejíž renderer navíc zobrazuje na zdi text, v závislosti na tom, zda-li je viditelný. Viditelnost textu lze změnit zasláním zprávy s odpovídajícími informacemi této straně.

Předposlední a nejsložitější stranou je třída **FloorTileSide**. Tato strana obsahuje čtyři úložiště na věci, kam může uživatel pokládat věci. Je to jedno z míst, kde je potřeba komunikovat s rodičovskou dlaždicí, k tomu slouží událost **SubItemsChanged**, která se vyvolá vždy, když byla nějaká věc odebrána nebo přidána. Pomocí enumerátoru tohoto objektu je potom možné vyenumerovat obsažené věci. Věci lze pak na podlahu přidávat dvěma způsoby, jednak přímo z

ruky hráče pomocí renderer-interactoru do úložného prostoru na podlaze. Toto přidání opět pomocí události informuje přes podlahu rodičovskou dlaždici. Další způsob je zavolání metody **OnObjectEnter** resp. **OnObjectLeft**, které také vyvolají notifikační událost o změně a přidají objekt do nějakého prostoru. Tento způsob je použit například u teleportu, kde se vědci přidávají do dané dlaždice voláním metody **OnObjectEnter**. Tato metoda pak volá odpovídající metodu podlahy. Poslední stranou je třída **ActuatorFloorTileSide**, která pouze navíc přidává nášlapný přepínač.

4.3 Renderery

Jelikož každá dlaždice a každé strany dlaždic mají renderery, je jím čas věnovat tuto sekci. Všechny tyto objekty a mnoho dalších k tomuto účelu implementují rozhraní **IRenderable**, které pouze vyžaduje položku typu **IRenderer**. Stěžejní funkce tohoto rozhraní jsou funkce **Render** a **Interact**.

Nejdůležitější z parametrů těchto funkcí je dosavadní transformace. Tj. obsahuje složeninu transformací složenou z jednotlivých transformací na cestě z kořene stromu reprezentující závislosti renderer na sobě. Takže například kořen strumu bude renderer dlaždice, jeho syn bude strana dlaždice, její syn bude výklenek ve zdi a její syn bude věc ve výklenku(list). Při takovéto reprezentaci se pak musí všechny renderované objekty posunout či jinak transformovat vůči jejich rodiči. Tzn. každý renderer si musí zvolit pozicovací konvenci, kterou pak musí závislé renderery dodržovat. Tento způsob renderování je požíván u statických objektu jako jsou dlaždice, zdi, výklenky a podobně. Pro pohybující objekty je používaná absolutní pozice a renderery těchto objektů transformují objekty přímo na jejich pozici. Volba mezi těmito dvěma případy je na programátorovi.

Renderery jsou vždy dělány na míru objektu, který mají renderovat. Tzn. často se renderer v konstruktoru inicializuje instancí s konkretním typem. Třída této instance pak má zpravidla readonly vlastnosti, podle kterých renderer určuje, co má vykreslovat. Jelikož implementovaná grafická vrstva je pouze ve formě proof of concept, je co nejjednodušší a neobsahuje například animace. Nicméně ze zde popsané povahy renderer je jasné, že toho může být dosaženo vytvořením událostí na renderovaných objektech, které si renderer zaregistruje. Dovedu si představit, že by šla s takovýmto návrhem velmi jednoduše udělat grafická vrstva, která bude velmi pěkná, bude mít kvalitní šD modely animace. Zabralo by to jen spoustu času a byla by k tom u potřeba horda grafiků.

Při podědění nějakého rendereru je třeba zjistit dosavadní transformaci, která je normálně vypočítána v rodiči. K tomuto záměru slouží funkce **GetCurrentTransformation**, která bere jako parametr dosavadní transformaci. Podděním jež existujícího rendereru lze ušetřit mnoho práce a opakujícího se kódu. Proto je takový přístup v této implementaci často použit. Funkce popsané na začátku sekce jsou samozřejmě virtuální a jdou overridovat, obsahují taky jeden parametr typu **object** pro případné předávaní dodatečných dat mezi renderery. Tohoto parametru není nikde v této implementaci využito.

Jelikož renderery zásadně mění vzhled a pozici všech věcí, je nutné tuto vrstvu propojit i s interakcí. Interakční funkce má skrze parametr typu **ILeader** přístup k položce interactor, která je typu **object**. Daný renderer musí vědět, pro jaký typ interactoru je a na ten si ho musí vhodně přetypovat. V této implementaci je

použit paprsek (**Ray**). Celý tento koncept interactoru by šel sice udělat genericky, ale prolínal by se celou strukturou rendererů a z toho důvodu jsem se rozhodl v tomto místě ustoupit. A udělat situaci jednoduší na úkor kontroly za překladu.

4.4 Přepínače

4.4.1 Úvod

Jak již bylo řečeno, přepínače se skládají z jednotlivých senzorů. Senzory pak mohou provádět následující akce:

- změna stavu dlaždice
- proházení senzorů přepínače
- přidání zkušeností hráči

Senzory mohou být aktivovány:

- Kliknutím na dekoraci senzoru, pokud je senzor na zdi a je to poslední senzor přepínače.
- Zprávou od vyslanou jiným senzorem.

Tento typ přepínačů odpovídá přepínačům v původní hře.

Dále engine povoluje nové přepínače, který nemusí používat senzory. Je tedy čistě na programátorovi, jakým způsobem se bude přepínač aktivovat.

4.4.2 Implementace přepínačů se senzory

Za přepínačem stojí navenek interface **IActuatorX**. Toto rozhraní pouze vyžaduje API pro příjem zpráv typu **Message** pro dodržení kompatibility s původní hrou. Jelikož ostatní aktivace záleží na typu přepínače. Protože například nášlapný přepínač se aktivuje jiným způsobem než přepínač na zdi.

Implementace senzorů

Senzory provádějí samotné akce přepínačů popsané v úvodu. Odpovídající akce jsou provedeny pouze pokud byl daný senzor aktivován. Po kliknutí na přepínač, dojde postupně u všech senzorů o pokus jejich aktivace. Každý typ senzoru má jinou podmínku aktivace. Senzory se dělí na senzory použité na podlaze, na zdi a na speciální tzv. logické senzory.

Hlavní třídou reprezentující přepínač je třída **SensorX**. Tato třída obsahuje některé společné funkce, které může využít každý potomek. Dále take definuje vlastnosti, které lze rozdělit do tří skupin. Tyto vlastnosti jsou inicializovány inicializátorem, který slouží pouze jako datový nosič. Jde o inicializátor typu **SensorInitializerX**. Pro použití v potomcích se můžou použít poděděné verze tohoto inicializátoru. Následuje seznam vlastností.

Obecné vlastnosti:

- Opoždění akce - čas v milisekundách, po kterém se provede případná akce

- Informace, zda-li je efekt určen lokálně pro rodičovskou dlaždici, či pro nějakou jinou.
- Flag opakovatelnost - každý senzor ji používá trochu jinak, ale většinou pokud je true, tak se obrátí cílová akce
- Na jedno použití - senzor se po první aktivaci zablokuje, a není dále používán.
- Flag určující, zda po aktivaci dojde k přehrání zvuku. (v tomto enginu není použit)

Vlastnosti pro lokální akci:

- Flag určující, zda-li se má seznam senzorů při aktivaci zarotovat.
- Flag určující, zda-li se mají hráči po aktivaci přidat zkušenosti.

Vlastnosti pro vzdálenou akci:

- Efekt - Určuje jaký efekt má odeslaná zpráva po aktivaci. (Akce může být obrácená pomocí flagu popsaného v obecných vlastnostech) Hodnoty efektu jsou následující:
  - Aktivace
  - Deaktivace
  - Přepnutí stavu
  - Drž - Tento stav nelze odeslat ve zprávě a je na senzoru, aby ho interpretoval pomocí aktivace či deaktivace.
- Směr použitý v odeslané zprávě. Směr může být interpretován jako číslo. (Lze získat pomocí **MapDirection.Index**)
- Reference na cílovou dlaždici.

Kromě předchozích vlastností obsahuje tato třída funkce pro vykonání akcí senzorů. Funkce **TriggerEffect** odešle zprávu podle vlastností senzoru na danou pozici, pokud je cíl vzdálený. Jinak provede akci pro lokální efekt. Lokální efekt lze také vykonat přímo zavoláním funkce **TriggerLocalEffect**. Tyto funkce lze použít pouze v potomcích.

Dále se senzory dělí na senzory použité na zemi resp. na zdi. Tomu odpovídají třídy **FloorSensor** a **WallSensor**. Obě tyto třídy mají funkci **TryTrigger**, která se stará o provedení případného efektu. Zda-li se efekt provede stanoví funkce **TryInteract**. Pro každou variantu mají funkce odlišné parametry. Pokud tedy chcete vytvořit vlastní senzor, implementujte funkci **TryInteract**. Tato funkce bude volána z funkce **TryTrigger** a bude tedy dělat obecnou funkcionalitu odeslání zprávy a určení výsledného efektu zprávy. Pokud chcete změnit i tuto funkcionalitu, poskytněte vlastní implementaci i pro tuto funkci.

Kromě předchozích senzorů ještě existují tzv. logické senzory. První z nich je **LogicGateSensor**, který funguje jako logické hradlo. Celkem má k dispozici osm bitů. Čtyři nich jsou nastaveny na nějaké hodnoty a ostatní na nuly. S druhou

čtveřicí lze manipulovat pomocí zpráv. Index směru zprávy určuje kolikátý bit se má ovlivnit. A tento bit je pak ovlivněn akcí zprávy. Pokud se pak stane, že jsou obě čtveřice stejné, je vyvolaný efekt senzoru. Dalším senzorem je **CounterSensor**, který má dané číslo. Aktivační zprávy toto číslo poté navyšují a deaktivační zprávy ho snižují. Pokud je číslo nula, je vyvolaný efekt.

Implementace přepínačů používající senzory

Rodiče všech přepínačů kompatibilních originální hře tvoří třída **Actuator**. Tato třída poskytuje pouze společnou rutinu pro zarotování senzory přepínače. Lze vyvolat pouze z potomků. Zda-li se mají senzory zarotovat určuje dle vlastnosti **Rotate**, která se poté nastaví na false. Tato funkce se volá z potomků pokud proběhl pokus o aktivaci nějakého senzoru.

Prvním potomkem předchozí třídy je třída **FloorActuator**, která může obsahovat pouze senzory určené na podlahu tj. **FloorSensor**. Pokus o její aktivaci se provádí funkcí **Trigger**. Jako první parametr se jí předává objekt, který vstupuje/odchází z dlaždice. Potom seznam všech objektů na dlaždici a naposled informace zda objekt vstupuje či vystupuje. Pro přepínač je vytvořena speciální podlaha **ActuatorFloorSide**, která se stará o volání zmíněné funkce.

Dalším potomkem je třída **WallActuator**, která může obdobně obsahovat pouze senzory určené na zeď tj. **FloorSensor**. Pokus o její aktivaci se provádí opět funkcí **Trigger**, která bere jediný parametr typu **ILeader**. Pro přepínač již není vytvořena žádná speciální strana dlaždice, ale je použita strana přijímající obecné přepínače **ActuatorWallTileSide**. Funkce **Trigger** je pak volána skrze renderer dané strany.

Poslední speciální implementací je třída **LogicActuator**, která může obsahovat již zmíněné logické senzory. O komunikaci s jejími senzory se stará rodič, pomocí zasílání zpráv. Pro tento přepínač je vytvořená speciální dlaždice, která není odnikud přístupná. Tato dlaždice je reprezentovaná třídou **LogicTile**.

4.4.3 Implementace obecných přepínačů

Tato sekce pojednává přepínačích, které ke své funkci nepoužívají senzory. Jejich vznik byl podnícen snahou, dát programátorovi větší svobodu při tvorbě přepínačů, nicméně zároveň řeší následující problém. Kromě standardních dekorací v se v dungeonu vyskytují dekorace, které provádějí nějakou funkci po kliknutí na ně. K těmto dekoracím však v originální hře nenáleží žádné senzory. Jedná se o tzv. náhodné dekorace, kdy každá zeď má definováno, zda může obsahovat náhodné dekorace. Při generování mapy se pak s určitou pravděpodobností dekorace na zeď dá či nikoliv. Z toho důvodu tento engine neobsahuje stejné dekorace na stejných místech, protože není znám náhodný generátor k tomu použitý. Nicméně zpět k problému akcí po kliknutí na dekoraci. Studii dekompilovaného kódu originální hry (viz. **?**) se ukázalo, že některé akce jsou fixovány na konkrétní typy dekorací. Takže například v dekorace s výklenky mají napevno implementovaný kód umožnující vkládaní věcí do výklenku. Dalším příkladem jsou fontánky, které naplňují lahve s vodou. Posledním příkladem speciální dekorace je Vi Altair výklenek, který dokáže po vložení kostí oživit šampiona. Jelikož tyto dekorace mohou být i součástí senzorů, bylo zapotřebí udělat kolem samotných dekorací

wrapery, které zajišťují případné akce. Tyto wrapery jsou tedy implementací přepínačů tj. rozhraní **IActuatorX**, kdy dekorace bez akce neprovádějí žádnou akci a dekorace s nějakou akcí danou akci provádějí. O samotné vyvolání interakce se opět musí postarat renderery. Takže zmiňované senzory jsou vlastně přepínače v přepínačích.

Obecné přepinače tedy nemají nikterak napevno zvolený formát, je čistě na programátorovi, jak implementuje uvnitř jejich funkci a jakým způsobem bude s přepínači prováděna interakce. Existující implementace tedy jsou **DecorationItem**, která neprovádí žádnou interakci, ale pouze rendering o který se stará **DecorationRenderer**. Dále jde o třídu **Alcove**, která reprezentuje výklenek a má odpovídající **AlcoveRenderer**. A poslední z nich je **ViAltairAlcove**, který dědí ze třídy **Alcove** a stará se o již zmiňovanou reinkarnaci šampionů.

4.5 Herní entity

Za deklarací neživých entit stojí rozhraní **IEntity**. Definuje pouze funkci pro získání vlastnosti dané entity. Za živými entitami stojí naopak rozhraní **ILiveEntity**, která navíc disponuje funkcí pro získání schopností. Dále deklaruje:

- tělo entity
- způsob rozmístění entity na dlaždici
- relace s dalšími entitami

4.5.1 Implementace vlastností entit

Vlastnost deklaruje rozhraní **IProperty**. Nejprve obsahuje vlastnost **Value**, která stanovuje nynější hodnotu dané vlastnosti. Tato hodnota je jako jediná možná měnit z venčí. V jejím getteru a setteru se typicky provádějí kontroly okrajových hodnot a případné reakce na ně. Další vlastností je **BaseValue**, což je nejvyšší hodnota, které může vlastnost nabýt, pokud není nějak modifikovaná. Maximální hodnotu včetně modifikací určuje vlastnost **MaxValue**. Poslední vlastností je **AdditionalValues**, která shromažďuje právě dané modifikace. Je to sekvence typu **IEntityPropertyEffect**, která definuje hodnotu a typ hodnoty. Poslední vlastností je již zmíněný typ vlastnosti. Jednotlivé typy vlastností jsou definovány jako reference na třídy implementující rozhraní IPropertyFactory. Entita na základě tohoto typu potom vrací příslušné vlastnosti. Typem může být jakákoliv třída implementující toto rozhraní, nicméně typická taková třída nemá žádné položky. Proto existuje generická třída **PropertyFactory**, která bere jako typový parametr právě typ alespoň **IProperty**. Tato třída se řídí vzorem singleton, jejíž jediná instance lze získat přes statickou položkou **Instance**. Takto lze pak generovat typy pomocí samotné třídy reprezentující vlastnosti, takže není nutné deklarovat stále nové třídy, které nic nedělají.

Za zmínku ještě stojí částečná implementace rozhraní **IProperty** a to třída **Property**. Tato třída definuje maximální hodnotu jako součet základních hodnoty a sumu modifikujících hodnot. Dále Při nastavení konkrétní hodnoty ořezává hodnotu do povoleného intervalu to je mezi nulou a maximální hodnotou. Také definuje událost vyvolanou při změně hodnoty. A naposledy definuje kolekci

modifikujících hodnot jako **HashSet⟨T⟩**. Všechny vlastnosti jsou deklarovány abstraktně nebo virtuálně. Tato třída lze použít k usnadnění implementace vlastností.

Některé vlastnosti mohou požadovat přístup k jiným vlastnostem či přístup k samotné rodičovské entitě. V takovém případě je čistě na programátorovi, jak takového cíle dosáhne. Typicky se v takovém případě předá v konstruktoru potřebná vlastnost nebo se vytvoří událost, která danou závislost deleguje vně.

Seznam předdefinovaných vlastností je možné najít ve jemném prostoru **DungeonMasterEngine.DungeonContent.Entity.Properties**. Jak již bylo zmíněno, může nastat, že daná entita vlastnost neobsahuje. V takovém případě je dobré kvůli dodržení konzistentnosti vracet předdefinovanou instanci třídy **EmptyProperty**. Nicméně rozhodně to není povinností při všech použití enginu.

4.5.2 Implementace schopností entit

Schopnosti deklaruje rozhraní **ISkill**. První vlastností tohoto rozhraní je **SkillLevel**, která udává úroveň, na které je daná schopnost. Do této hodnoty jsou započítány i úrovně získané například kouzelnými předměty. Hodnotu bez těchto extra úrovní určuje vlastnost **BaseSkillLevel**. Dále obsahuje vlastnosti **Experience** a **TemporaryExperience** určující dosavadní hodnotu zkušeností. Dále obsahuje vlastnost **BaseSkill**, což je reference na základní(viz. analýza) schopnost. Pokud je daná schopnost již základní, je hodnota **null**. Přidat schopnosti zkušenosti je možné zavoláním funkce **AddExperience**. Konkrétní algoritmus přepočítávání zkušeností na levely záleží vždy na konkrétní implementaci. Stejně jako u vlastností i zde existuje vlastnost **Type**, tentokrát ale typu **ISkillFactory**. Stejně jako u vlastností, je možné tyto reprezentanty vytvářet pomocí generické třídy **SkillFactory**.

Třída **SkillBase** implementuje získávání zkušeností schopností jako v originální hře. Všechny schopnosti využívající tuto třídu jsou ve jemném prostoru **DungeonMasterEngine.DungeonContent.Entity.Skill**. Stejně jako u vlastností i zde entita nemusí mít dotazovanou schopnost. V takovém případě vrací instanci třídy **EmptySkill**.

4.5.3 Tělo a inventáře entity

Následující API umožňuje vytvořit různé druhy těl pro různé entity. Každá část těla může sloužit jako úložiště. Dále pak existují externí úložiště. API je natrhnuté tak, aby dovolovalo definovat různé druhy úložišť a těl. V základu obshuje definice pouze pro lidské tělo.

Inventář

Inventář je definován rozhraním **IInventory**. Tak především každý inventář má definované jakého je typu. Typ inventáře musí implementovat rozhraní **IStorageType**. Toto rozhraní definuje velikost inventáře. Dále definuje samotné úložiště jako **ReadOnlyList**. A v poslední řadě obsahuje funkce pro přidávaní a odebírání věcí z úložného prostoru.

Třída **Inventory** implementuje všechny funkce inventáře. Je jí tedy možno přímo použít nebo rozšířit.

Část těla

Část těla je definovaná rozhraním **IBodyPart**, které je potomkem rozhraní **IInventory**. Navíc definuje TODO

Obecnou implementací rozhraní je tříd **BodyPart**. Typ konkrétního úložiště předaný v konstruktoru určuje typ části těla.

Tělo entity

Tělo entity je definováno rozhraním **IBody**. Obsahuje seznam částí těla, seznam všech úložišť včetně částí těla a funkce pro vyhledání úložiště či části těla dle typu.

Engine obsahuje pouze implementaci pro lidské tělo tj. třída **HumanBody**

4.5.4 Rozmístění entity na dlaždici

Definice prostorů a cest na dlaždici

Rozhraní **IGroupLayout** definuje obecně možné rozmístění na dlaždici. Nejprve obsahuje všechny možné prostory, které lze využít. Jednotlivě prostory reprezentují rozhraní **ISpace**. Dale API musí poskytovat funkce pro nalezení cesty z libovolného prostoru ,definovaného layoutem, na libovolnou sousední dlaždici nebo k libovolné ze stran dlaždice. Jednotlivé články cesty jsou reprezentovány rozhraním **ISpaceRouteElement**. Poslední funkce musí umět vytvořit z prostoru a dlaždice článek cesty, tedy **ISpaceRouteElement**.

Rozhraní **ISpace** musí definovat, jakým stranám dlaždice je prostor přilehlý. Dále musí definovat prostor, který využívá na dlaždici, pomocí obdélníku. Jak již bylo řečeno obdélník musí zabírat prostor z 1000x1000 pole. Přičemž souřadnice rostou shora dolů a zleva doprava. Nahoře je pak sever, vpravo východ, dole jih a vlevo západ. Kromě toho také musí definovat sousední prostory, k čemuž využívá již známé generické rozhraní **INeighborable**.

Rozhraní **ISpaceRouteElement** se potom skládá pouze s prostoru, dlaždice a absolutně pozicované lokace, na které má stát daná entita. Pro výpočet pozice se může použít pozice dlaždice a samozřejmě daný prostor. Při definici vlastního layoutu je možné využit předem připravený hledač nejkratších cest pro prostory **GroupLayoutSearcher**. Pro reprezentaci sousedů prostorů je tu zase tříd **FourthSpaceNeighbors**.

Celé toto rozhraní je readonly struktura, která pouze definuje prostory na dlaždici a způsob pohybu mezi nimi. Z toho důvodu dává dobry smysl instance tříd reprezentující toto rozhraní implementovat jako singletony. Engine definuje tři layouty a to pro čtyři, dvě a jednu entitu. Ostatní možné

Řízení obsazeného prostoru na dlaždici

K předchozímu mechanismu je ještě třeba další část, která bude zaznamenávat samotné využité pozice na dlaždici. K tomuto účelu existuje třída **LayoutManager**. Lze pomocí ní získat seznam entity na dlaždici a seznamy využitých pro-

storů dlaždic. Dále poskytuje API pro přidání entity na daný prostor na dlaždici, odebrání prostoru a získání entit, které využívají alespoň část nějakého prostoru. Z toho vyplývá, že entity s různým layoutem mohou být na stejné dlaždici, pokud je na ní dostatek prostoru pro oba prostory definované layoutama.

4.5.5 Relace s dalšími entitami

Každá živá entita musí definovat vlastnost typu **IRelationManager**. Toto rozhraní musí definovat relační token typu **RelationToken** pro danou entitu. Dále definuje funkce, která pro daný token vrátí, zda entity odpovídající danému tokenu je nepřátelská. Jednoduchou implementací tohoto rozhraní je třída **DefaultRelationManager**, která má si při svém vzniku definuje své neměnné nepřátelé. Nicméně pokud daná implementace nevyhovuje je čistě na programátorovi, aby si vytvořil implementaci vlastní. K dispozici je ještě generátor unikátních tokenů a to statická třída **RelationTokenFactory**.

4.5.6 Implementace entit

Jak již bylo zmiňováno, jsou v tomto enginu obsažené dvě implementace živých entit a jedna neživá.

Šampion

Šampion je první živá entita a reprezentuje jej třída **Champion**. Šampion neobsahuje žádnou umělou inteligenci, je ovládán hráčem skrze třídu **Theron**, která reprezentuje hráčovu skupinu šampionů. Inicializace vlastností a schopností probíhá přes datový initializer definovaný rozhraním **IChampionInitializer**. Resp. jeho předáním do konstruktoru, dále je nutné specifikovat relační token a seznam nepřátel. Posunout daného šampiona je možné přiřazením do vlastnosti **Location**.

Zde je dobré zmínit generickou třídu **Animator**, která bere dva typové parametry. První z nich je typ posouvaného objektu, který musí implementovat alespoň rozhraní **IMovable**. Toto rozhraní definuje vlastnosti, pomocí kterých lze měnit pozice daného objektu. Dalším parametrem je typ prostorů, mezi kterými se objekt pohybuje. Ten musí implementovat alespoň rozhraní **IStopable**, které definuje pozici, na kterém se má objekt postavit. Animátor poskytuje API pro plynulý posun objektů mezi prostory.

U šampiona je tento animátor použit automaticky při změně lokace.

Příšera

Další a poslední živou entitou ve hře je příšera, kterou reprezentuje třída **Creature**. Tato třída reprezentuje všechny příšery ve hře. Vlastnosti jednotlivých typů příšer jsou ve třídách typu **CreatureFactory**. Každá konkrétní instance příšery má referenci na tuto třídu a její chování je ovlivněné vlastnostmi v ní obsažené.

Tato třída definuje pro příšery jednoduchou umělou inteligenci. Za zmínku stojí, že ke zjednodušení implementace jsou zde opět využity asynchronní funkce. Ty jsou především využity pro vytváření zpoždění akcí pomocí **Task.Delay**,

bez nutnosti složitého počítání času a vracení se zpět do funkce po jeho uplynutí. Asynchronní funkce jsou však prováděny v jednom vlákně a to vždy ve funkci **Update**. V každém volaní funkce je vyprázdněna celá fronta asynchronních funkci, proto je třeba dávat pozor na její přehlcení, které může vést až k neresponzivnosti aplikace. Následuje seznam funkcí a jejich popis použitých pro simulování inteligence příšer. Všechny tyto funkce je možné overridovat.

- **Live** - V této funkci je nekonečný cyklus, který volá obslužné rutiny podle stavu příšery. Jsou to:
  - Lov - příšera spatřila nepřítele a pronásleduje ho
  - Cesta domů - příšera pronásledovala nepřítele, kterého následně ztratila, proto jde domů tj. na místo svého vzniku
  - Hlídkování - příšera hlídkuje v okolí oblasti svého vzniku
- **FindEnemies** - Pomocí prohledávání do šířky se pokusí najít entity s nepřátelským tokenem. Maximální hledanou oblast je určena dle vlastností příšery. Pokud je nepřítel nalezen nastaví proměnou **hountingPath**.
- **MoveToSpace** - Přesune příšeru mezi prostory pomocí animátoru a nastaví správně použité prostory pomocí **LayoutManageru**.
- **MoveThroughSpaces** - Přesune příšeru přes prostory dlaždice až na cílovou dlaždici pomocí funkce **MoveToSpace**. Při každém pohybu hledá nepřátele pomocí funkce **FindEnemies** pokud je tak nastaveno parametrem funkce. Pokud nějaké najde, přeruší pohyb.
- **MoveToNeighbourTile** - Posune příšeru na určenou dlaždici přes prostory dlaždice pomocí funkce **MoveThroughSpaces**. Pokud je cesta nepřístupná vrátí **false**.
- **GoHome** - Příšera jde domů podle cesty uložené v proměnné **homeRoute** a poté ji nastav na **null**. Mezi dlaždicemi se posunuje pomocí funkce **MoveToNeighbourTile**.
- **FindNextWatchLocation** - Pomocí prohledávání do šířky z místa objevení příšery nalezne dlaždici, na které dlouho příšera nebyla a vrátí k ní cestu.
- **WatchAround** - Pomocí funkce **FindNextWatchLocation** nalezne cestu na další místo k hlídkování. Poté jde na dané místo pomocí funkce **MoveToNeighbourTile**.
- **Fight** - Provede útok na nepřítele.
- **PrepareForFight** - Dostane se co nejblíže k dlaždici, na který je nepřítel a zahájí útok pomocí funkce **Fight**.
- **GetPathHome** - Pokusí se nalézt cestu domů. Pokud ji nalezne nastaví **homeRoute**.
- **EstablishNewBase** - Nastaví výchozí dlaždici na nynější.

- **Hount** - Příšera pronásleduje nepřítele na poslední místo, kde ho spatřila. Pokud je nepřítel na sousední dlaždici připraví se k útoku pomocí funkce **PrepareForFight**. Pokud nepřítele ztratí, pokusí se najít cestu domů pomocí **GetPathHome**. Pokud cestu nenalezne založí si domov tam, kde je pomocí funkce **EstablishNewBase**.

Závěr

Přílohy