UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Adrian Pieper
nr albumu: 243 677

Adventure Maker - język dla terenowych gier RGP

Praca magisterska na kierunku:
INFORMATYKA
Promotor:
dr W. Bzyl

Gdańsk 2017

Streszczenie

W ramach pracy zaprojektowałem i zaimplementowałem Adventure Maker - język domenowy[1] służący do opisu terenowych gier RPG[2]. Do implementacji wykorzystałem technologię Xtext [1]. Wykorzystanie narzędzia Xtext umożliwiło stworzenie podstawowej infrastruktury języka zawierającej między innymi linker i kompiler. Dodatkowo powstały język zintegrowany jest z IDE Idea InteliJ[3] co pozwala na podświetlanie, autouzupełnianie oraz automatyczne sprawdzanie składni.

Wygenerowane przez *Adventure Maker* gry wykorzystują technologię NFC[4] oraz GPS[5] w celu lokalizacji graczy w pomieszczeniach, jak i na otwartym terenie.

W projekcie wykorzystałem popularne narzędzia i biblioteki, z których najważniejsze to Gradle[2], JUnit[3] i Dagger 2 [4]. Pisząc kod starałem się stosować zasady czystego kodu [5].

Praca składa się z dwóch części: implementacji języka Adventure Maker oraz szablonu aplikacji, wraz z przykładowym kodem gry. Całość kodu dostępna jest w publicznym repozytorium GIT (pod adresem `https://github.com/adrpieper/magisterka`). Przykładowa aplikacja oraz wtyczka do IDE Idea InteliJ zostały udostępnione na stronie `https://adrpieper.github.io/magisterka`.

Słowa kluczowe

Android, DSL, Xtext, Location-Based Game, RPG

[1] Domain Specific Language - DSL - język programowania służący do rozwiązywania jednej klasy problemów

[2] Role Playing Game - Gra fabularna `https://pl.wikipedia.org/wiki/Gra_fabularna`

[3] Zintegrowane środowisko programistyczne `https://www.jetbrains.com/idea/`

[4] Near Field Communication - wysokoczęstotliwościowy standard komunikacji krótkiego zasięgu `https://pl.wikipedia.org/wiki/Near_Field_Communication`

[5] Global Positioning System - globalny system nawigacji satelitarnej `https://pl.wikipedia.org/wiki/Global_Positioning_System`

Spis treści

Wprowadzenie

Sukces niedawno wydanej gry ”Pokemon GO” pokazał, że pomysł by gracz musiał poruszać się po fizycznym świecie jest strzałem w dziesiątkę. Okazuje się, że na gry tego typu znajduje się całkiem spora rzesza odbiorców, czego nie dało się nie zauważyć, gdyż grupy poszukiwaczy tytułowych pokemonów, spotkać można było niemal na każdym kroku. Gry terenowe, czyli aplikacje rozrywkowe, basujące głównie na fizycznej pozycji gracza, stały się nowym trendem w dziedzinie rozrywki elektronicznej.

Pomimo sukcesu ”Pocemon GO”, gry terenowe nadal stanowią niewielką część rynku rozrywki elektronicznej. Moim zdaniem niewielka ilość aplikacji tego typu, wynika z braku odpowiednich narzędzi do ich tworzenia. Napisanie tego typu gier w tradycyjny sposób jest dość skomplikowane, a przez to kosztowne. Wykorzystanie technologii typu GPS, czy NFC wymaga od programisty wiedzy i znajomości specjalistycznego API oraz sprawia, że testowanie aplikacji jest utrudnione.

Zauważyłem, że spora część elementów gier terenowych jest wspólna, szczególnie jeśli ograniczyć się jedynie do gier RPG. Pomyślałem, że skoro gry tego typu opierają się na podobnych zasadach, powinny one zostać zdefiniowane tylko raz, w myśl zasady - ”nie powtarzaj się”. W ten sposób można by odciążyć projektanta gry od szczegółów implementacyjnych, pozwalając mu skupić się w całości na tym co ważne, czyli zasadach i scenariuszu rozgrywki. Stwierdziłem, że napisanie tego typu gry powinno sprowadzać się jedynie do zdefiniowania miejsc, postaci oraz zasad obowiązujących w wirtualnym świecie. Postanowiłem więc sprawdzić, czy osiągnięcie takiego ideału jest w ogóle możliwe. Okazuje się, że tak, o ile tylko zastosuje się do tego odpowiednie podejście i narzędzia.

Sam pomysł umieszczenia warstwy wspólnej dla wielu aplikacji nie jest nowy. Istnieje wiele takich rozwiązań. Warstwę tą nazywa się frameworkiem lub silnikiem gry. Większość powstających obecnie gier osadzonych jest właśnie na tego typu rdzeniu. Brak jest natomiast narzędzia, wyspecjalizowany do tworzenia konkretnego typu gier, jakimi są terenowe gry RPG.

Moim celem nie było stworzenie niesamowicie grywalnej, ani popularnej gry.

Chciałem pokazać, na ciekawym moim zdaniem przykładzie gier terenowych, jak można rozwiązać problem ”wynajdywania koła na nowo” wykorzystując własny język domenowy. Adventure Maker pozwala na implementacje gry przez osobę niebędącą programistą, ponieważ wszystkie niezbędne w grze elementy są już zaimplementowane. Składnia języka jest przyjazna dla osoby zaznajomionej z tematyką gier RPG i nie wymaga znajomości, ani żadnego języka programowania ogólnego przeznaczenia, ani żadnych dodatkowych technologii.

Aby osiągnąć ten cel skorzystałem z popularnych i sprawdzonych narzędzi i rozwiązań. Do implementacji samego języka użyłem frameworka XText oraz języka Xtend [6]. Framework dostarcza, na podstawie gramatyki języka, całą infrastrukturę języka, zawierającą elementy takie jak: parser, linker, typechecker, compiler. Dodatkowo pozwala na integracje zaprojektowane języka z popularnymi IDE: eclipse oraz IntelliJ Idea.

Xtend jest językiem programowania wywodzącym się i składniowo zbliżonym do JAVA. Został zaprojektowany, aby wyeliminować kilka wad tego języka. Zawiera elementy niedostępne w JAVA oraz upraszcza niektóre struktury pozwalając na pozbycie się niepotrzebnego kody. Kodem wynikowym dla języka Xtext jest Java, a nie kod binarny. Mamy tu więc do czynienia z transpilacją, czyli przetłumaczeniem jednego kodu źródłowego w inny. Wykorzystanie Xtend jest zalecanym sposobem implementacji generatorów dla języków domenowych tworzonych przy użyciu Xtext. Główną zaletą języka Xtend, w kontekście pisania generatorów kodu, są szablony, które nie są dostępne w języku JAVA.

Aplikację napisałem za pomocą AndroidSKD [7]. Jest to zestaw standardowych narzędzi programisty umożliwiający tworzenie aplikacji na platformę Android w języku Java. Przy wykorzystaniu AndroidSDK zalecane jest użycie Gradle [2] - narzędzia służącego do automatyzacji budowania projektów informatycznych. Do jego zadań należą między innymi:

- Zarządzanie zależnościami
- Automatyzacja testów
- Budowanie i instalacja aplikacji

Wraz ze specjalnie zaprojektowanym pluginem, Gradle stanowi obowiązkowy element każdego projektu związanego z systemem Android.

Do testów wykorzystałem JUnit [3] oraz Mockito [8]. JUnit to framework służący do testowania programów napisanych w języku Java. Początkowo zaprojektowany przez Erich Gamma i Kent Beck, dziś rozwijany w formie open source. Biblioteka Mockito jest ściśle związana z frameworkiem JUnit. Umożliwia tzw. Mockowanie[1]. Główną zaletą Mockito jest bardzo wygodne API, tworzące swego rodzaju wewnętrzny DSL.

Aby móc zapanować nad zależnościami pomiędzy klasami wykorzystałem bibliotekę Dagger 2. Istnieje wiele bibliotek umożliwiających DI[2], jednak Dagger 2 cieszy się największą popularnością wśród programistów Androida. Jego główną zaletą jest rozstrzyganie zależności w czasie kompilacji oraz generacja kodu. Dzięki temu możemy wykryć wady projektu, takie jak zależności cykliczne jeszcze przed uruchomieniem aplikacji. Dodatkowo wygenerowany kod jest o wiele szybszy niż typowe rozwiązanie czasu wykonania - czyli refleksja. Jest to bardzo ważna cecha w przypadku aplikacji Androidowych, gdyż urządzenia mobilne mają stosunkowo ograniczone zasoby.

[1] Technika testowania polegająca na podmianie obiektu na jego atrapę https://pl.wikipedia.org/wiki/Atrapa_obiektu

[2] Dependency Injection - wstrzykiwanie zależności https://en.wikipedia.org/wiki/Dependency_injection

ROZDZIAŁ 1

Przegląd istniejących rozwiązań

W pracy postawiłem sobie na celu stworzenie języka domenowego pozwalającego na pisanie terenowych gier RPG. Cele postawione przed tym językiem są typowe dla frameworku - udostępnić narzędzie do przyjaznego rozwiązywania problemów określonej klasy. Nowy jest natomiast obszar w jakim działać będzie język, czyli terenowe gry RPG.

Tematyka pracy obejmuje więc, cztery pozornie niezwiązane ze sobą tematyki jakimi są gry terenowe, gry RPG, frameworki oraz języki domenowe. Ten rozdział ma na celu zaprezentowanie istniejących rozwiązań w każdej z tych dziedzin.

1.1. Gry terenowe

Rozwój technologii mobilny sprawił, że od paru lat na rynku rozrywki elektronicznej pojawił się całkiem nowy pomysł - gry terenowe. W grach tych zrezygnowano ze znanego z tradycyjnych gier wirtualnego świata na rzecz tzw. rzeczywistości rozszerzonej. W grach terenowych gracz nie steruje już postacią za pomocą myszki, czy klawiatury, lecz jest zmuszony do fizycznego przemieszczania się po rzeczywistym świecie. Lokalizacja gracza zostaje przeniesiona do świata gry za pomocą technologii takiej jak np. GPS. Mapa po której porusza się gracz jest więc mapą znaną z lekcji geografii. Elementami które sprawiają, że świat gry nazywamy rzeczywistością rozszerzoną, są pojawiające się w grze, a nie istniejące w rzeczywistości, obiekty lub postaci, które wpływają w jakiś sposób na przebieg rozgrywki.

Doskonały przykładem gry terenowej jest Pokemon GO. Jest to gra, której premierę ciężko było przeoczyć. Gra zyskała olbrzymią popularność w zaledwie kilka dni, czego efektem były tłumy graczy szturmujących parki i place. Tam właśnie ukrywały się tytułowe Pokemony, których szukanie i kolekcjonowanie, są kluczowymi elementami rozgrywki.

Inny pomysł na wykorzystanie lokalizacji gracza mieli twórcy gry Landlord. Gra przenosi koncepcje znale z popularnej gry Monopoly do świata rzeczywistego. Cel gry pozostał niezmieniony, jest nim oczywiście inwestowanie w nieruchomości. Nie znajdziemy tam natomiast planszy, pionków, ani kostki. Gracze Landlord, w celu dokonywania wirtualnych zakupów, zmuszeni są do podróżowania po rzeczywistym świecie.

1.2. Gry RPG

Gry RPG, inaczej gry fabularne, są to gry w których gracze wcielają się w rolę fikcyjnych postaci poruszających się po fikcyjnym świecie. Celem graczy jest zazwyczaj ukończenie jakiegoś scenariusza, bądź też po prostu osiągniecie określonego celu np. zdobycie jakiegoś przedmiotu, danej ilości złota lub rozwój postaci do konkretnego poziomu. Istnieją też gry otwarte, w których gracz nie ma żadnego narzuconego celu, a jedynie przemierza fikcyjny świat ze znanej tylko sobie motywacji. Tradycyjnie grę tego typu rozgrywa się w wyobraźni graczy. Jeden z graczy wciela się wtedy w tzw. mastera gry. Zadaniem mastera jest prowadzenia graczy przez świat gry poprzez opowiadanie pewnej historii oraz zadawanie pytań. Master przedstawia graczom jak wygląda sytuacja, w której znajdują się ich postacie oraz prosi ich o podjęcie decyzji, dotyczącej zachowania się postaci w danej sytuacji. Gracze podejmują decyzję, po czym master gry określa z jakimi skutkami się ona wiąże i przechodzi do dalszej opowieści. Aby zachować pewną spójność gry, master podejmuje decyzję w oparciu o ustalony zbiór zasad (tzw. mechanikę gry), zazwyczaj efekt podjętej decyzji zależy też od rzutu kością.

Oprócz tradycyjnej odmiany gier fabularnych powstały tej ich planszowe oraz komputerowe odmiany. W grach tych nie już mastera, a jego rolę przejmuje z góry narzucony scenariusz oraz ścisłe zasady.

Jednym z najpopularniejszych przykładów planszowych gier RPG jest Magia i Miecz. Gracze wybierają w niej jedną z kilkudziesięciu postaci. Celem gry jest dostanie się do tzw. Korony Władzy. Aby móc tego dokonać muszę rozwinąć umiejętności swoich postaci. Jest to możliwe poprzez wyciąganie kart przygód, które zawierają opis sytuacji, w której znalazł się gracz.

W świecie gier komputerowych, RPG osiągnęły niekwestionowany sukces. Wydanych tytułów są całe dziesiątki i nie sposób ich tutaj wymienić. Dodatkowo, warto zwrócić uwagę, że typowe elementy tych gier takie jak rozwój i statystyki postaci przedostały się już do prawie każdego gatunku gier komputerowych.

1.3. DSL

DSL [9], czyli języki domenowe, to języki programowania zaprojektowane z myślą o ściśle określonym i z reguły bardzo wąskim zastosowaniu. W odróżnieniu od języków programowania ogólnego przeznaczenia, języki domenowe nie nadają się do rozwiązania większości problemów informatycznych. Sprawdzają się natomiast świetnie w dziedzinie, do której zostały zaprojektowane Dzięki ograniczeniu się jedynie do wąskiej grupy zastosowań, możliwe jest tworzenie języków, które są zrozumiałe dla osób będących ekspertami w danej dziedzinie. Języki domenowe należą zazwyczaj do języków deklaratywnych, gdyż skupiają są wokół tego co, a nie w jaki sposób, chce osiągnąć programista.

Języki domenowe ze względu na sposób ich implementacji można podzielić na dwie grupy:

- Języki wewnętrzne (Internal DSL)
- Języki zewnętrzne (External DSL)

1.3.1. Internal DSL

Internal DSL to język stworzonych w ramach innego istniejącego już języka ogólnego przeznaczenia. Technicznie rzecz biorąc jest to zazwyczaj zbiór klas udostępniających wygodny dla programisty, dający wrażenie pisania w innym języku zbiór metod. Klasy te umieszcza się w bibliotece, którą możemy użyć do rozwiązania ściśle określonego problemu. Główną cechą takich bibliotek jest wyraźne nastawienie na udostępniany interfejs, a nie samą implementacje. O jakości takiego rozwiązania świadczy nie tyle wydajność jego działania, lecz łatwość użycia. Biblioteki takie pozwalają programiście na bardziej czytelne, prostsze i zwięzłe wyrażenie je-

go intencji. Przykładami taki języków są np. język asercji z biblioteki AssertJ lub biblioteka Mockito.

1.3.2. External DSL

External DSL to język domenowy z prawdziwego zdarzenia. Język taki posiada ściśle określoną gramatykę i od początku jest projektowany w określonym celu. Nie stanowi on części innego języka, chodź często z potrafi z nim współpracować. Przykładem takiej współpracy może być np. komunikacja z bazą danych, gdzie kod programu (napisany np w języku JAVA), wywołuje pewne zapytanie w języku SQL. Przykładami zewnętrznych języków domenowych są:

- SQL - język służący do obsługi relacyjnych baz danych
- CSS - język służący do definiowania stylu stron internetowych
- HTML - język służący do definiowania struktury strony internetowej

1.4. Frameworki

Frameworki są doskonałym przykładem korzyści jakie niesie ze sobą popularna zasada Clean Code [5] - DRY (Don't Repeat Yourself), czyli nie powtarzaj się. Twórcy frameworków wychodzą z założenia, że projekty informatyczne można z powodzeniem podzielić na pewne grupy. Projekty informatyczne należące do takiej grupy bywają do siebie tak podobne, że zazwyczaj łatwiej jest wskazać cechy wspólne niż różnice. Ideą frameworka jest implementacja wszystkich elementów wspólnych w jednym miejscu i udostępnienie programistom-użytkownikom frameworka przyjaznego interfejsu do implementacji tych różnic. Programista jest więc, ograniczony pewnymi ramami, w których musi mieścić się jego aplikacja, stąd nazwa takiego narzędzia.

Dobrym przykładem takiej grupy są aplikacje webowe.Wszystkie udostępniają pewien interfejs w postaci stron html, przechowują dane w bazach (zazwyczaj relacyjnych) i komunikują się z użytkownikiem za pomocą protokołu HTTP. Elementem różniącym te aplikacje jest ich wygląd i funkcjonalność.

Z uwagi na fakt, że gry RPG są bardzo popularne, a jednocześnie do siebie bardzo podobne, naturalnym wydaje się stworzenie oprogramowania pozwalające na łatwe tworzenie takiego typu gier. Narzędzie RPG Maker [10] pozwala na proste wytwarzanie dwuwymiarowych gier RPG. Według twórców, tworzenie gier przy pomocy RPG Maker, jest możliwe bez jakiejkolwiek wiedzy na temat programowania, jednocześnie dając bardzo duże możliwości doświadczonym użytkownikom. Oprogramowanie udostępnia przyjazne GUI, dzięki któremu można tworzyć w pełni funkcjonalne gry i to na wiele różnych platform. Jedynym, ale bardzo istotnym ograniczeniem jest ściśle narzucony gatunek gier. Jednak właśnie to ograniczenie pozwoliło na stworzenie narzędzia jednocześnie tak prostego i funkcjonalnego.

Warto wspomnieć tu również o istnieniu narzędzi pozwalających na łatwe tworzenie nawet zaawansowanych gier mobilnych i to dowolnego typu. Takim oprogramowanie jest np. Unity [11]. Unity udostępsnia przyjazne GUI, które pozwala na tworzenie świata gry. Elementy fizyki takie jak grawitacja, są już w pełni zaimplementowane w silniku gry. Programista musi natomiast jedynie pamiętać o nadaniu obiektom odpowiednich cech takich jak np. masa. Logikę gry można zaimplementować w jednym z dwóch języków UnityScript oraz C#. UnityScript jest językiem o składni bardzo zbliżonej do JavaScript, natomiast C# to popularny obiektowy język programowania czerpiący wszystko co dobre z JAVA i C++. Użycie powszechnie znanych języków oraz wieloplatformowość z pewnością przyczyniły się do olbrzymiej popularności, którą cieszy się Unity.

ROZDZIAŁ 2

Wymagania

Jednym z pierwszym kroków jakie należy podjąć przez przystąpieniem do projektowania oprogramowania jest analiza wymagań. Andventure Maker umożliwia tworzenie gier terenowych o narzuconych z góry, dość wąskich i specyficznych ramach, typowych dla tradycyjnych gier RPG. Napisana w nim gra polega głównie na rozwoju postaci, poprzez odwiedzanie lokacji, podejmowanie decyzji, a przede wszystkim pokonywanie przeciwników. Postać sterowana przez gracza zdobywa punkty doświadczenia, dzięki którym gracz może rozwijać postać w wybranym przez siebie kierunku.

Wymagania, jakie postawiłem przed Adventure Makerem można podzielić na funkcjonalne i niefunkcjonalne.

2.1. Funkcjonalne

Do wymagań funkcjonalnych należą:

- Język DSL zawierający elementy takie jak:
    - Klasy Postaci
    - Przedmioty
    - Lokacje
    - Przygody
    - Przeciwnicy
- Integracja ze środowiskiem IntelliJ Idea
- Kompilacja aplikacji na podstawie pliku DSL

Zbiór elementów wchodzących w skład zaprojektowanego języka DSL został przeze mnie wybrany arbitralnie, w taki sposób, żeby zawierał minimum niezbędne do zaprojektowania gry. Chciałem, aby zaprojektowany język był jednocześnie prosty, ale i pozwalający na implementacje nawet dość zawiłej logiki. Oczywiście RPG z prawdziwego zdarzenia powinien zawierać nieco bardziej rozbudowany. Uznałem jednak, że dodanie elementów takich jak np. misje zbędnie skomplikowały by język. Elementy te prawdopodobnie pojawią się, w przypadku wydania kolejnej wersji Adventure Maker. Nie mniej jednak implementacja stworzona w ramach tej pracy obejmować będzie jedynie wyszczególnione punkty.

U celu ułatwienia edycji kodu warto udostępnić użytkownikowi przyjazne IDE. Na szczęście narzędzia takie już istnieją i nie musiałem projektować ich od zera. Popularne narzędzia programistyczne pozwalają na rozszerzanie ich funkcjonalności poprzez tzw. wtyczki, czyli oprogramowanie, które doinstalowuje się do już istniejącego IDE. Postanowiłem więc, że udostępnię obsługującą język Adventure Maker wtyczkę do IDE IntelliJ Idea. Postanowiłem wybrać to środowisko z kilku powodów:

- jest darmowy w wersji Community,
- udostępnia integracje z Android SDK,
- jest wspierany przez Xtext i Xtend.

Przyjazna edycja kodu to za mało, aby język był funkcjonalny. Użytkownikowi należy udostępnić możliwość kompilacji kodu. W tym przypadku będzie to transkompilacja, ponieważ kodem wyjściowym nie będzie kod binarny, a kod JAVA. Na podstawie wygenerowanego oraz dostarczonego kodu możliwa musi być kompilacja gotowej aplikacji, bez konieczności dopisywania ani jednej linii dodatkowego kodu.

2.2. Niefunkcjonalne

Do wymagań niefunkcjonalnych należą:

- Tworzenie gier RPG bez znajomości technologii i języków programowania ogólnego przeznaczenia

- Wykorzystanie darmowych narzędzi
- Ukrycie szczegółów implantacyjny GPS i NFC przed użytkownikiem języka

Podstawowym założeniem Adventure Maker jest udostępnienie możliwości tworzenia gier osobom będącym laikami jeśli chodzi o programowanie i technologie IT. Abym mógł w ogóle udostępnić im ten język ważne jest wykorzystanie darmowych narzędzi.

Język został zaprojektowany tak, aby twórca gry mógł traktować technologie NFC i GPS w sposób bardzo zbliżony, nie zważając na ich całkowicie odmienną techniczną implementację. Z jego punktu widzenia zarówno pomieszczenie oznaczone tagiem NFC, jak i współrzędne geograficzne, stanowią po prostu lokację, w której gracza mogą spotkać dowolnego typu przygody.

ROZDZIAŁ 3

Projekt Interfejsu Aplikacji

Ważnym elementem każdego współczesnego oprogramowania jest jego graficzny interfejs użytkownika [1]. W przypadku języka do tworzenia gier można mówić o dwóch rodzajach GUI: interfejsie edytora i powstałych w języku gier.

Jako, że Adventure Maker został zrealizowany jako wtyczka do istniejącego IDE, sprawa jego GUI jest rozwiązana. IDE Idea IntelliJ dostarcza bardzo wygodny i dopracowany interfejs użytkownika, a dostarczona przeze mnie wtyczka wykorzystuje jego możliwości.

Język nie pozwala twórcy gry na dostosowanie interfejsu do swoich potrzeb. Tak więc, jako jego projektant, postanowiłem przygotować interfejs prosty, a zarazem uniwersalny i neutralny względem klimatu gry. GUI jest podzielone na trzy części:

- Menu początkowe
- Tryb zwiedzania
- Tryb przygody

3.1. Menu początkowe

Ekran pozwala na wybór typu postaci, jaką dany gracz ma zamiar kontrolować. Został podzielony na dwie części: listy klas postaci oraz szczegółów wybranej klasy. Szczegóły zawierają nazwę klasy oraz informacje o jej statystykach.

[1] GUI - Graphical User Interface - Graficzny interfejs użytkownika

3.2. Tryb zwiedzania

Zwiedzanie stanowi główny ekran gry. Jest wyświetlany przez cały czas, w którym gracz porusza się pomiędzy lokacjami w poszukiwaniu przygód.

Tryb zwiedzania zawiera dosyć dużo informacji dlatego też został podzielony na następujące części:

- Przegląd postaci
- Umiejętności
- Przedmioty

Pomiędzy widokami gracz może nawigować swobodnie, przeciągając je w lewo lub w prawo.

Przegląd postaci

Przegląd postaci zawiera informacje o ilości i poziomie doświadczenia sterowanej przez gracza postaci. Dodatkowo wyświetla też statystyki uwzględniające bonusy otrzymane z noszonych przedmiotów.

Umiejętności

Ekran udostępnia drzewko umiejętności oraz dwa przyciski ”Reset” oraz ”OK”. Drzewko służy zarówno do przeglądania posiadanych, jak i odblokowywania nowych umiejętności. Przycisk reset przywraca graczowi rozdane punkty umiejętności, natomiast przycisk ”OK” zatwierdza ich wybór.

Przedmioty

Sekcja przedmiotów, zawiera wszystkie posiadane przez gracza rzeczy. Przedmioty używane aktualnie przez gracza znajdują się w górnej części ekranu, natomiast pozostałe w dolnej, zwanej plecakiem. Dodatkowo na środku wyświetlana jest suma bonusów do statystyk uzyskanych z noszonych przedmiotów.

3.3. Tryb przygody

Gracz nie ma swobodnego dostępu do tej części interfejsu. Jest ona wyświetlana dopiero w momencie znalezienia się gracza w lokacji zawierającej jakąś przygodę.

Tryb przygody zawiera kilka ekranów, z których każdy odpowiedzialny jest za wyświetlenie konkretnego zdarzenia. Gracz nie może swobodnie nawigować pomiędzy nimi. Dostępne GUI zależne jest sytuacji, w której znajduje się postać gracza.

Decyzja

Ekran jest bardzo prosty. Zawiera jedynie pole wyświetlające pytanie skierowane do gracza oraz listę decyzji w postaci przycisków.

Walka

Widok walki odpowiedzialny jest za wyświetlanie oraz sterowanie przebiegiem starcia pomiędzy postacią gracza, a dowolnym przeciwnikiem. Interfejs wyświetla nazwę przeciwnika, kolorowe paski zawierające informacje o punktach życia (tzw. hit points) postaci i przeciwnika oraz o punktach many gracza. W dolnej części znajdują się przyciski akcji pozwalające na użycie standardowego ataku lub dowolnej umiejętności.

Komunikat

Ekran komunikatu składa się z dwóch części. Wiadomości oraz przycisku ”OK”, którym gracz potwierdza, że zapoznał się z komunikatem. Wiadomość najczęściej wyświetlana jest w prostym oknie tekstowym, jednak w niektórych przypadkach konieczne jest użycie elementów bardziej rozbudowanych. Jest tak na przykład w przypadku okna wyświetlanego pod koniec przygody, zawierającego między innymi wykaz zdobytych przez gracza przedmiotów w postaci listy.

ROZDZIAŁ 4

Architektura Języka i Aplikacji

Adventure Maker składa się z dwóch projektów. Pierwszy odpowiedzialny jest za implementacje języka. Drugi projekt, to szkielet aplikacji na system Android, będący bazą pod wszystkie gry pisane w Adventure Maker.

4.1. Adventure Maker DSL

Projekt powstał w celu implementacji gramatyki i generatora dla języka Andventure Maker. Projekt bazuje na narzędziu Xtext[1], a produktem wyjściowym jest wtyczka do IDE IntelliJ Idea obsługująca zaprojektowany DSL.

4.1.1. Definicja Gramatyki

Gramatyka języka Adventure Maker znajduje w pojedynczym pliku o nazwie ”AML.xtext”. Definicja w nim zawarta zawiera zarówno opis modelu syntaktycznego i semantycznego tworzonego języka. Innymi słowy zdefiniuje jaki tekst należy do języka oraz jak będzie on reprezentowany w pamięci komputera. Xtext na podstawie tego pliku generuje między innymi parser, który sprawdzi, czy dany tekst poprawnym programem oraz zwróci jego reprezentacje w postaci drzewa obiektów.

4.1.2. Generator Kodu

Generator odpowiedzialny jest za wygenerowanie kodu (w tym przypadku kodu Java) na podstawie modelu semantycznego zwróconego przez parser w postacie drzewa obiektów. Generator został napisany w języku Xtend i składa się z głównej klasy implementującej metodę ”doGenerate” oraz klas pomocniczych utworzonym w celu dekompozycji problemu na mniejsze części zgodnie z zasadą pojedynczej odpowiedzialności [5].

4.2. Aplikacja Android

Ten projekt to standardowa aplikacja Androidowa wykorzystująca AndroidSDK rozbudowana o język Adventure Maker, na podstawie którego generowana jest cześć kodu aplikacji.

Architektura projektu jest dość rozbudowana i można z niej wydzielić następujące moduły:

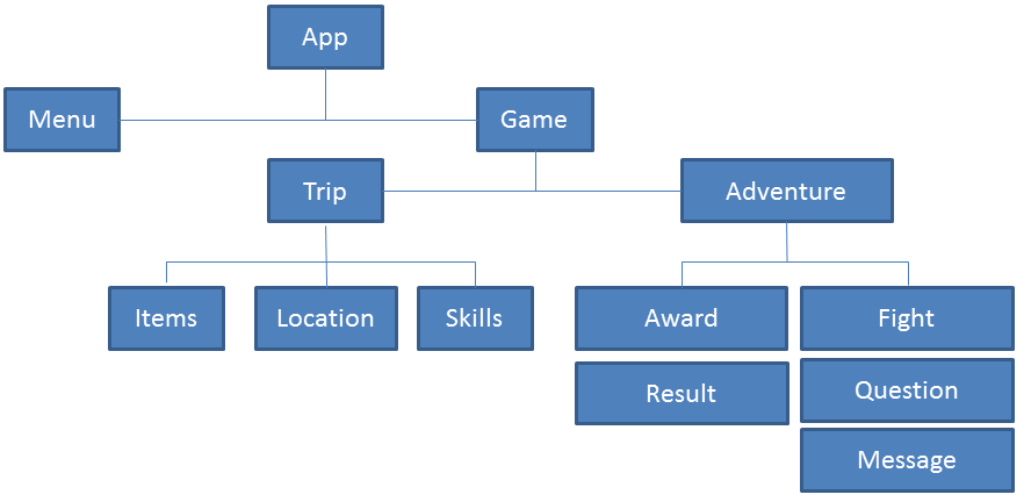
- Standardowy Kod Aplikacji
- Zasoby
- Plik manifestu
- Pliki Grandle
- Testy
- Wewnętrzy DSL
- Plik game.aml
- Wygenerowany Kod Implementujący Zasady Gry
- Testowy Kod Implementujący Zasady Gry

Pierwsze pięć elementów jest typowych dla aplikacji android, natomiast pozostałe są specyficzne i wynikają z zastosowania języka Adventure Maker.

Standardowy kod aplikacji

Jest to bardzo rozbudowana, ale i najbardziej tradycyjna i oczywista część aplikacji. W jego skład wchodzą pakiety ”edu.ug.inf.am.*” umieszczone w folderze ”app/src/main/java”. Moduł jest odpowiedzialny za sposób działania i wygląd aplikacji. Korzysta z kodu Implementujący Zasady Gry oraz Zasobów. Została w nim umieszczona logika związana z systemem Android taka jak komunikacja z czujnikami NFC i GPS, czy obsługa interfejsu użytkownika. Dodatkowo kod odpowiada za elementy stałe dla każdej gry takie jak mechanizm walki i rozwoju postaci.

Kod został podzielony na warstwy i komponenty, co zostało odzwierciedlone w strukturze pakietów. Komponenty to zbiory klas realizujące wspólne zadanie. Ich struktura jest hierarchiczna tzn. w skład konkretnego komponentu wchodzą kolejne, odpowiedzialne za zadania bardziej sprecyzowane. Kompletną strukturę komponentów aplikacji przedstawie drzewo 4.1.



Rysunek 4.1. Struktura komponentów aplikacji

Źródło: Własne opracowanie

Wszystkie komponenty zostały zrealizowane w postaci pakietów o podobnej strukturze. W nich skład wchodzi po kilka pakietów-warstw zawierających bezpośrednio klasy realizujące funkcjonalność tych warstw. Dodatkowo pakiety, nie będące liśćmi w drzewie, zawiera pakiety-komponenty znajdującymi się niżej w hierarchii.

W kodzie wydzieliłem następujące warstwy i odpowiadające im nazwy pakietów:

- Widok (view)
- Contoler (controller)

- Model (model,state)
- Logika (logic)
- Komponent (dagger)

Widok to warstwa odpowiedzialna za interfejs użytkownika. W jej skład wchodzą przede wszystkim klasy dziedziczące po Activity, Fragment oraz View. W większości przypadków zadaniem klasy jest połączenie danych z widokiem za pomocą dostępnego od niedawna w Adroidzie mechanizmu Data Bindingu. Dana pobierane są z warstwy kontrolera.

Warstwa kontrolera odpowiada przede wszystkim za obsługę zdarzeń przychodzących z interfejsu użytkownika, oraz dostarczanie modelu danych dla widoku. Zajmuje się też modyfikacją danych. W prostych przypadkach modyfikuje je bezpośrednio, a w bardziej zaawansowanych deleguje to zadanie do warstwy logiki. Inną ważną odpowiedzialnością tej warstwy jest przełączanie pomiędzy widokami.

Model odpowiada za przechowywanie danych w uporządkowany sposób. Hermetyzuje dane udostępniając zestaw metod dostępowych (gettery i settery). Ta warstwa została podzielona na dwa podtypy: model i stan. Modelem nazywam wszystkie klasy, których stan jest obserwowany przez widok (wzorzec obserwator). Stanem (State) nazywam klasy typu POJO, których stan nie może być obserwowany, a ich jedynym zadaniem jest przechowywanie stanu aplikacji.

Logika to warstwa odpowiadająca za wszelkie operacje wykonywane na danych, które nie powinny znaleźć się w warstwie kontrolera ze względu na zbyt duże skomplikowanie, bądź wagę kodu. Wydzielenie tego kodu do osobnych klas w połączeniu z testowaniem jednostkowym pozwala stworzyć kod, co do którego mamy pewność, że jego kluczowe aspekty działają prawidłowo. Usunięcie tego kodu z kontrolera jest zgodne z zasadą pojedynczej odpowiedzialności [5] i pozwala na jego użycie w wielu miejscach programu.

Pakiet ”dagger” to miejsce w którym integrowane są wszystkie klasy należące do danego komponentu. W projekcie wykorzystano technikę wstrzykiwania zależności (DI), która pozwala na usunięcie bezpośrednich zależności pomiędzy klasami, aż do momentu ich integracji. Za wstrzykiwanie zależności w moim projekcie

odpowiedzialna jest biblioteka Dagger 2, stąd nazwa pakietu w którym to zadanie jest realizowane.

Zasoby

Zasoby są elementem wspólnym dla każdej aplikacji. Tradycyjnie zostały umieszczone w folderze "app/scr/main/res". Tutaj definiuje się elementy takie jak layouty, ikonki, animacje itp. Większość zasobów stanowią pliki xml. Dostęp do zasobów możliwy jest z poziomu kodu aplikacji poprzez klasę R. Klasa ta jest generowana automatycznie, a jej skład wchodzi zestaw stałych, będących identyfikatorami odpowiednich zasobów. Android SDK posiada wbudowany system zarządzania zasobami, który pozwala na przygotowanie konkretnego zasobu w kilku wersjach (np. różny layout dla kilku wielkości ekranów). Tak przygotowany zasób, jest dostępny pod pojedynczym numerem id, a o wybór odpowiedniej wersji dba system.

WiększođC zasobów stanowią layouty, z których znaczna część wykorzystuje tzw. Data Binding. Mechanizm ten pozwala opisać logikę odpowiedzialną za wyświetlanie danych w sposób deklaratywny, przy pomocy języka xml. AndroidSDK generuje na jego podstawie specjalny kod, który będzie realizował to zadanie.

Plik manifestu

Plik manifestu jest elementem niezbędnym w każdej aplikacji androidowej. Znajduje się on w folderze "app/scr/main". W nim zadeklarowane są elementy takie jak identyfikator aplikacji, zbiór dostępnych aktywności, potrzebne uprawnienia, ikonę startową itp.

Gradle

Gradle jest narzędziem służącym do budowania projektów, który stał się standardem w aplikacjach androidowych. W projekcie znajdują się 2 pliki Gradle, jeden dla modułu aplikacji, drugi dla całego projektu. Zostały umieszczone odpowiednio w folderze "/app" i "/". W plikach Gradle zdefiniowane jest struktura projektu, zadania związane z jego budowaniem oraz potrzebne zależności.

Testy

Moduł testów odpowiedzialny jest za sprawdzenie, czy kod aplikacji działa prawidłowo. Ich kod został umieszczony w folderze ”app/scr/test/java”. W jego skład wchodzą klasy, implementujące testy jednostkowe wszystkich ważnych elementów aplikacji. Kod testów wykorzystuje zewnętrzne biblioteki JUnit, Mockito i AssertJ.

Wewnętrzy DSL

W projekcie wykorzystałem zarówno wewnętrzny, jak i zewnętrzny język domenowy. Zastosowanie zewnętrznego DSL pozwala twórcy gry na opisanie świata gry w sposób przyjazny dla osoby znającej tematykę gier RPG. Wewnętrzny DSL jest odpowiedzialny za dostarczenie wygodnego API do opisu zasad gry w języku Java, co znacznie uprościło zaimplementowania generatorów kodu.

Technicznie rzecz biorąc moduł stanowi on interfejs do komunikacji pomiędzy modułami Standardowy Kod Aplikacji, a Kodem Implementujący Zasady Gry. Zarówno dostarczone API, jak i implementacja modułu zostały zrealizowana w języku JAVA. Dlatego też nazywam go wewnętrznym językiem domenowym. W skład modułu wchodzą wszystkie klasy pakietów ”pl.aml.*”.

Istotną częścią tego modułu są klasy pozwalające na wygodne budowanie obiektów (tzw. Buildery) oraz metody statyczne tworzące instancje tych klas. Dzięki ich zastosowaniu możemy tworzyć kod przyjazny dla programisty.

Różnicę pomiędzy kodem pisany z i bez wykorzystania wewnętrznego DSL zaprezentowałem w dodatku A.

Plik game.aml

Plik znajduje się w folderze ”app/src/main/java”. Jest kluczowym elementem całego projektu, bo to właśnie w nim definiuje się w języku Adventure Maker elementy związane z konkretną grą.

Wygenerowany Kod implementujący zasady gry

Ten moduł znajduje się folderze ”app/aml-src” i zawiera kod wygenerowany automatycznie na podstawie pliku ”game.aml”. Kod ten implementuje szczegóły dotyczące konkretnej gry. Wykorzystuje API dostarczone przez Wewnętrzny Język DSL. Wygenerowany tu kod użyty jest w module Kod Aplikacji.

Testowy Kod implementujący zasady gry

Ten moduł zawiera kod analogiczny do Wygenerowanego Kodu implementujący zasady gry, jest jednak napisany ręcznie. Kod został umieszczony w folderze ”app/test-src”. Moduł ten zastępuje kod generowany podczas testowania i rozwijania aplikacji i nie jest wykorzystywany w produkcie finalnym.

ROZDZIAŁ 5

Testy

W celu sprawdzenia poprawności działania stworzonego oprogramowania musiałem je przetestować. Na pierwszym miejscu postawiłem testy automatyczne, ponieważ są najbardziej praktyczne i wiarygodne. Testy automatyczne zaimplementowałem w postaci mockowanych testów jednostkowych za pomocą bibliotek JUnit i Mockito.

Niestety z przyczyn technicznych nie dało się pokryć testami automatycznymi wszystkich funkcjonalności. Z tego powodu część kodu została przetestowana w sposób manualny. Elementami przetestowanymi ręcznie są działanie modułu NFC oraz GPS. Do każdego z tych elementów przygotowałem scenariusz testowy, który następnie wykonałem. Sposób przeprowadzenia oraz wyniki tych testów zostały przedstawione poniżej.

5.1. Testy automatyczne

5.1.1. Struktura testów

Podczas automatycznych testów jednostkowych testowałem z osobna działanie poszczególnych funkcji w oderwaniu od reszty systemu. Każdy test został podzielony na 3 sekcje, zawierające kod odpowiedniego typu:

- Given
- When
- Then

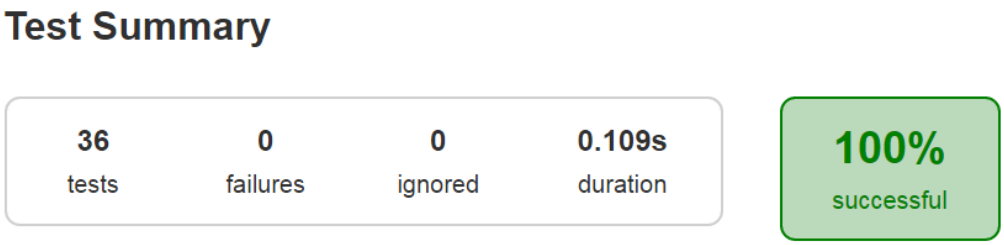
W sekcji *Given* został umieszczony kod implementujący warunki początkowe. Są to przede wszystkim linie kodu mockującego metody klas zależności klasy testowanej. Mogą to być też wywołania metod dostarczających jakieś dane.

Sekcja *When* składa się z pojedynczej linii. Jest to wywołanie testowanej metody na instancji testowanej klasy.

Sekcja *Then* odpowiedzialna jest za sprawdzenie, czy metoda zachowała się w oczekiwany sposób. Zazwyczaj jest to kod składający się z kilku linii zawierających asercje. Asercje służą sprawdzeniu czy, ile razy i z jakimi parametrami zostały wywołane metody klas-zależności. Jeżeli funkcja zwraca jakiś wynik, za pomocą asercji testuje się, czy pokrywa się on z oczekiwanym.

5.1.2. Wyniki

Testy jednostkowe zostały uruchomione przy pomocy narzędzia Gradle [2]. Poniższe zestawienie pokazuje, że wszystkie testy przeszły pomyślnie.



Rysunek 5.1. Podsumowanie testów

Źródło: Wygenerowane przez Gradle

5.2. NFC i GPS

Testy funkcjonalność NFC i GPS zostały przeprowadzone ręcznie według poniższego scenariusza:

- Przygotowanie pliku AML[1]

[1] Adventure Maker Language - rozszerzenie plików obsługiwane przez język Adventure Maker

- Kompilacja
- Sprawdzenie poprawności zachowania

5.3. Plik AML

Przygotowane pliki AML dla testu NFC i GPS są bardzo podobne. Oba składają się z przykładowej Klasy Postaci oraz dwóch Przygód i Lokacji. W obu przypadkach przygody polegają jedynie na wyświetleniu komunikatu.

Różnica pojawia się w przypadku definicji Lokacji. W pliku przygotowanym dla GPS, są to dwa znajdujące się niedaleko od siebie obszary o promieniu 10 metrów. W przypadku NFC Lokacje zdefiniowane są przez odpowiednie znaczniki "Test1" i "Test2".

Kod Adventure Maker przygotowany do przeprowadzenia obydwu testów znajduje się w załączniku.

5.3.1. Sprawdzenie poprawności zachowania

Poprawność działania modułów GPS i NFC została sprawdzona manualnie. W obu przypadkach sprawdziłem, czy wejście do odpowiedniej Lokacji uruchomi odpowiednie zdarzenie.

W przypadku testu GPS przeszedłem się z urządzenie wyposażonym w moduł GPS w odpowiednie miejsca. W przypadku NFC po prostu przyłożyłem telefon do odpowiednich znaczników.

Oba testy zakończyły się sukcesem.

ROZDZIAŁ 6

Pisanie gier w Adventure Maker

Pisanie własnej gry przy użyciu Adventure Maker wymaga przygotowania środowiska pracy. W tym rozdziale zawarłem instrukcje dotyczące instalacji, konfiguracji i korzystania z potrzebnych narzędzi.

6.1. Konfiguracja środowika

Aby móc korzystać z Adventure Maker należy pobrać i zainstalować IDE IntelliJ Idea, JRE[1], JDK[2] i Android SDK oraz wtyczkę AML.

JDK o raz JRE należy pobrać ze strony Oracje (https://www.java.com) w wesji 1.8 i zainstalować zgodnie z instrukcją producenta.

IDE IntelliJ Idea można pobrać ze strony producenta - JetBrains (https://www.jetbrains.com/idea). Wersja community jest darmowa i wystarczająca do obsługi Adventure Maker. Podczas instalacji należy pamiętać o włączeniu pluginu obsługującego aplikacje pisane na system Android.

Android SDK należy poprać ze strony (https://developer.android.com) i zainstalować w dowolnej lokalizacji, którą należy następnie skonfigurować w IntelliJ Idea.

Wtyczka AML jest odpowiedzialna za obsługę języka Adventure Maker, w tym generowanie kodu Java na podstawie pliku ".aml". Do prawidłowego działania wtyczki potrzebne są wtyczki Xtext i Xtend, które można pobrać ze strony domowej (https://eclipse.org/Xtext/download.html). Wtyczkę należy pobrać ze strony (https://adrpieper.github.io/magisterka) i zainstalować ręcznie.

[1] Java Runtime Enviroment - środowisko uruchomieniowe Java

[2] Java Development Kit - standardowy zestaw narzędzi wykorzystywanych do tworzenia aplikacji w technologii Java

6.2. Przygotowanie i kompilacja gry

Przed rozpoczęciem pracy na własnym projektem należy poprać repozytorium (https://github.com/adrpieper/magisterka). Znajdujący się w nim projekt ExampleAdventure, jest projektem przykładowej gry. Najłatwiejszym sposobem na jego wykorzystanie jest skopiowanie go i posługiwanie się nim jako szablonem. Należy pamiętać, aby zmienić id aplikacji w pliku Manifestu.

Implementacja własnej gry sprowadza się do edycji jednego pliku, mianowicie ”game.aml”. Należy w nim umieścić kod w języku Adventure Maker. Kod musi zawierać wszystkie niezbędne elementy gry. Pisząc ten kod można posłużyć się dokumentacją języka zamieszczoną w pracy. Warto też wzorować się na kodzie przykładowej gry, który równiej wchodzi w jej skład.

Projekt jest standardowym projektem Androidowym, tak więc możemy go też uruchomić przyciskiem play dostępnym w IDE IntelliJ Idea. Ważne jest, żeby korzystać z IDE, nawet w przypadku budowania narzędziem gradle, ponieważ kod, generowany na podstawie pliku AML, tworzony jest przez samo środowisko.

ROZDZIAŁ 7

Dokumentacja języka

Poniżej znajduje się dokumentacja języka Adventure Maker.

7.1. Plik AML

Plik definiuje wszystkie dostępne w grze elementy. Plik może zawierać dowolną ilość Klas Postaci, Typów Przedmiotów, Przeciwników, Lokacji oraz Przygód. Dodatkowo należy w nim umieścić Przygody Początkowe.

7.2. Przygody Początkowe

Przygody Początkowe definiują Przygody udostępnione graczowi na starcie gry. Definicja bloku Przygody Początkowe rozpoczyna się od słów kluczowych ”adventure on start”. Następnie w nawiasach klamrowych znajduje się dowolna liczba deklaracji przygód dostępnych na początku gry. Każda taka definicja składa się z nazwy Przygody, słówka kluczowego ”at” i nazwy Lokacji.

Poniższy przykład zawiera definicje składającą się z dwóch Przygód ”MysteriusMan” i ”Bandits”, ulokowanych odpowiednio w ”OldHouse” i ”Forest”.

```
adventure on start {
    MysteriusMan at OldHouse
    Bandits at Forest
}
```

7.3. Klasy Postaci

Definicja Klasy, inaczej typu postaci to informacje na temat statystyk oraz umiejętności dostępnych dla danej Klasy Postaci. Statystyki dzielą się na Początkowe, któ-

re postać otrzymuje przy starcie gry, oraz Poziomowe, które zwiększają całkowite statystyki postaci z każdym zdobytym poziomem. Umiejętności zostały zorganizowane w postaci drzewa. Aby odblokować daną umiejętność, gracz musi najpierw odblokować wszystkie prowadzące do niej umiejętności.

Definicja typu postaci rozpoczyna się od słów kluczowych "character type". Następnie należy podać nazwę typu za którą znajdują się nawiasy klamrowe. W nawiasach klamrowych należy umieścić kolejno:

- Statystyki Początkowe
- Statystyki Poziomowe
- Drzewo Umiejętności

Definicja Statystyk Początkowych rozpoczyna się od słów kluczowych "stats on start:". Za tymi słowami umieszczone są ilościowe parametry siły, inteligencji i zwinności oznaczone odpowiednio słowami kluczowymi "str", "int", "agi", będącymi skrótami od angielskich tłumaczeń nazw tych parametrów.

Definicja Statystyk Poziomowych jest bardzo podobna to tych Początkowych. Występują tu tylko dwie niewielkie różnice. Po pierwsze, definicja zaczyna się od swój kluczowych "stats per lvl:". Pod drugie przed każdym parametrem występuje znak plusa, który podkreśla fakt, że są to wartości, które zwiększają statystyki postaci.

Definicja Drzewa Umiejętności rozpoczyna się od słów kluczowych "skills tree:". Dalej znajduje się dowolna liczba gałęzi drzewa. Każdy gałąź składa się z nazwy umiejętności, której dotyczy oraz opcjonalnie zbioru gałęzi potomnych, które definiują umiejętności, dostępne po odblokowaniu umiejętności-rodzica. Zbiór gałęzi potomnych definiuje się w nawiasach klamrowych poprzedzonych symbolem "=>".

Poniżej znajduje się pełny przykład definicji postaci typu "Wizard":

```
character type Wizard {
   stats on start:
      10 str
      20 int
```

```
      15 agi
   stats per lvl:
      + 1 str
      + 2 int
      + 1 agi
   skills tree:
      Fireball => {
         Wirewall
         BlackMagic
      }
      Poisoning
}
```

Z definicji można odczytać, że postać typu "Wizard" otrzymuje 10 punktów siły, 20 inteligencji i 15 zwinności na początku gry. Jego statystyki zwiększają się o 1 punkt siły, 2 inteligencji i 1 zwinności z każdym zdobytym poziomem doświadczenia. Na początku może odblokować umiejętność "Fireball" lub "Poisoning". Zdobycie umiejętności "Fireball" udostępnia ponadto dwie kolejnych: "Wirewall" i "BlackMagic".

7.4. Typy Przedmiotów

Przedmiot to element, który stanowi nagrodę dla gracza i może być wykorzystany w celu podniesienia statystyk jego postaci. Definicja Typu Przedmiotu jest bardzo prosta. Wystarczy podać jego nazwę, kategorię oraz wartości premii do odpowiednich statystyk. Do wyboru mamy następujące kategorie (w nawiasie podano odpowiadające kategorii słówko kluczowe):

- Broń (weapon)
- Hełm (helmet)
- Zbroja (armor)
- Rękawice (gloves)

Wartości premii umieszczone są w nawiasach klamrowych i składają się z wartości liczbowej poprzedzonej znakiem plus oraz słówka kluczowego ”agi”, ”str” lub ”int” odnoszącego się do odpowiedniego typu statystyk. Przedmiot nie musi zawierać bonusu do każdego typu statystyk.

Poniżej znajduje się przykład definiujący przedmiot typu ”Sword”, należący do kategorii Broń, zwiększający siłę postaci o 4, a zwinność o 3 punkty.

```
item Sword (weapon) {
   + 4 str
   + 3 agi
}
```

7.5. Lokacje

Lokacja to miejsce do którego można się odwołać definiując Przygodę. Miejsce takie oznacza pewien obszar w fizycznym świecie np. pokój albo kawałek lasu. Lokacje można definiować na dwa sposoby: za pomocą technologii GPS lub NFC. W przypadku technologii GPS lokacje opisuje okrąg o środku składającym się z współrzędnych geograficznych i promieniu wyrażonym w metrach. Użycie tej technologii zalecane jest do opisu obszarów znajdujących się na otwartej przestrzeni. Opisując lokacje za pomocą technologii NFC wystarczy podać odpowiedni tag. Tą technologię należy wykorzystać do opisu przestrzeni zamkniętych jak np. pokoje. Trzeba pamiętać, że odpowiedni znacznik NFC musi znajdować się w tym pomieszczeniu.

Definicja Lokacji rozpoczyna się od słowa kluczowego ”location” i jej nazwy. Jeżeli lokacja ma być opisana przez znacznik NFC należy użyć słów kluczowych ”tagged as”, a następnie podać tekst, który stanowi zawartość tagu. W przypadku korzystania z technologii GPS używa się słów kluczowych ”in radius of” po którym podaje się odległość w metrach. Następnie po słowach ”meters from” należy umieścić szerokość i długość geograficzną.

Poniżej znajdują się przykłady definicji lokacji ”OldHouse” i ”Forest”. Lokacja ”OldHouse” oznaczona jest tagiem ”OldHouseLocation”, natomiast ”Forest”

to obszar o promieniu 100 metrów od punktu o współrzędnych 54°35'45.9"N, 18°14'42.7"E.

```
location OldHouse tagged as OldHouseLocation
location Forest in radius of 100 meters
^^Ifrom 54.596077, 18.245200
```

7.6. Przeciwnicy

Przeciwnik to byt, z którym gracz może stoczyć walkę podczas trwania Przygody. Jego definicja jest stosunkowo prosta i zbliżona do obiektu w formacie json. Definiując Przeciwnika należy użyć słowa kluczowego "opponent", podać jego nazwę oraz zbiór parametrów otoczonych nawiasami klamrowymi.

Pierwsze trzy parametry to "power", "hp", "exp" to parametry liczbowe ich wartość należy podać w postaci liczby. Oznaczają one odpowiednio moc przeciwnika, czyli jego zdolność bojowa, ilość posiadanych przez niego punktów życia oraz ilość doświadczenia, które gracz zdobywa w nagrodę za jego pokonanie.

Czwarty parametr - "loot", to lista przedmiotów, który może otrzymać gracz po pokonaniu przeciwnika, oddzielonych przecinkami. Każdy przedmiot na zawiera informacje o typie przedmiotu oraz prawdopodobieństwo jego otrzymania.

Poniższy przykład definiuje przeciwnika o nazwie "Dragon":

```
opponent Dragon {
   power : 100
   hp : 200
   exp : 100
   loot : MagicSword 50%, SteelArmor 10 %
}
```

Z definicji wynika, że "Dragon" posiada moc o wartości 100, 200 punktów życia. Za pokonanie tego Przeciwnika gracz otrzyma 100 punktów doświadczenia, oraz odpowiednio 50% i 10% szansy na zdobycie przedmiotów "MagicSword" oraz "SteelArmor".

7.7. Przygody

Przygody są najważniejszym i najbardziej rozbudowanym elementem języka Adventure Maker. Definicja każdej Przygody składa się z słowa kluczowego ”adventure”, nazwy Przygody, słów ”starts from” oraz Zdarzenia Początkowego.

Zdarzenie otoczone jest zawsze nawiasami klamrowymi i może zawierać jedną z definicji:

- Komunikat
- Walka
- Zdarzenie Warunkowe
- Pytanie
- Modyfikacja Przygód
- Blok Zdarzeń

Przed przejściem do przykładowej definicji Przygody, należy zapoznać się z dokumentacją poszczególnych Zdarzeń.

7.7.1. Komunikat

Komunikat to najprostsze ze Zdarzeń. Polega na zwykłym wyświetleniu komunikatu, z którym gracz ma się zapoznać.

Definicja Komunikatu składa się ze słowa kluczowego ”Show” oraz treści komunikatu.

Poniżej znajduje się przykład kodu wyświetlającego komunikat ”Hello World!”.

```
Show "Hello World!"
```

7.7.2. Walka

Walka oznacza Zdarzenie polegające na pojedynku z pojedynczym przeciwnikiem lub ich grupą. Zdarzenie tego typu zawiera informacje o ilości i typie przeciwników oraz Zdarzeniach wywoływanych w przypadku zwycięstwa jak i porażki.

Definicja walki rozpoczyna się od słów kluczowych ”Fight with”. Następnie należy wymienić nazwy wszystkich przeciwników oddzielając je przecinkami. Dalej można umieścić opcjonalnie Zdarzenie wywoływane w przypadku zwycięstwa oraz Zdarzenie wywoływane w przypadku porażki.

Definicja Zdarzeń wywołanych w przypadku zwycięstwa i porażki składa się ze słów kluczowych odpowiednio ”If win” i ”If lost” oraz samego zdarzenia.

Poniżej znajduje się przykładowa definicja Walki:

```
Fight with Troll, Orc
If win {
   {Remove VillageInDanger at Forest}
   {Get MagicSword}
}
If lost {
   {Remove VillageInDanger at Forest}
   {Show "Unfortunately, you have not saved the villagers"}
}
```

Z definicji wynika, że gracz będzie musiał się zmierzyć z dwoma przeciwnikami o nazwach ”Troll” i ”Orc”. Jeżeli ich pokona zdobędzie przedmiot - ”MagicSword”, w przeciwnym wypadku zostanie wyświetlony komunikat - ”Unfortunately, you have not saved the villagers”. W obu przypadkach Zdarzenie ”VillageInDanger” nie będzie już dłużej dostępne w Lokacji ”Forest”.

7.7.3. Zdarzenie Warunkowe

Zdarzenie Warunkowe służy do zadeklarowania Zdarzeń, które wywołane zostaną, tylko w przypadku spełnienia określonych warunków. Dodatkowo można zdefiniować Zdarzenie alternatywne, wywołane w przypadku niespełnienia warunków. W aktualnej wersji języka składnia Zdarzeń Warunkowych jest dość uboga i pozwala jedynie na sprawdzenie, czy postać sterowana przez gracza posiada odpowiednie statystyki.

Definicja zdarzenia warunkowego rozpoczyna się od słów kluczowych ”If you have” po których pojawia się Wyrażenie Warunkowe. Następnie należy podać de-

finicję Zdarzenia. Opcjonalnie można dodać definicje Zdarzenia alternatywnego poprzedzając je słowem ”else”.

Wyrażenia Warunkowe składają się z słówka kluczowego ”more” lub ”less”, skrótowej nazwy parametru (”str, ”agi” lub ”int”), słowa ”than” oraz wartości liczbowej. Dodatkowo wyrażenia warunkowe można łączyć za pomocą słówek ”and” lub ”or” oraz nawiasów. Wyrażenie warunkowe można też poprzedzić słowem ”no”, które wprowadza negacje.

Poniżej znajdują się dwa przykłady wyrażeń warunkowych.

```
If you have no more int than 30
If you have more agi than 20 or more str than 10
```

Pierwsze wyrażenie sprawdza, czy postać ma nie więcej niż 30 punktów inteligencji. Drugie sprawdza, czy posiada więcej niż 20 punktów siły lub więcej niż 10 punktów zwinności.

Poniżej znajduje się pełny przykład definicji Zdarzenia Warunkowego:

```
If you have more str than 30
   {Fight with Orc}
else
   {Show "The fight will not be necessary"}
```

Z powyższej definicji wynika, że postać o sile większej niż 30 będzie musiała walczyć z przeciwnikiem - ”Orc”, natomiast gracz kontrolujący postać słabszą zobaczy komunikat - ”The fight will not be necessary”.

7.7.4. Pytanie

Pytanie, jest Zdarzeniem nieco podobnym do Zdarzenia Warunkowego, z tą różnicą, że wybór Zdarzenia zależy nie od spełnienia, bądź niespełnienia warunków, a od decyzji podjętej przez gracza.

Definicja Pytania zawsze rozpoczyna się słowem kluczowym ”Ask”, po którym znajduje się treść wyświetlonego pytania, którą należy podać w cudzysłowie. Po pytaniu należy umieścić jeden lub więcej Odpowiedzi. Odpowiedź składa się ze słów kluczowych ”Answer” oraz ”to”, pomiędzy którymi należy umieścić treść odpowiedzi (również w cudzysłowie) oraz zdarzenia, do którego prowadzi dana odpowiedź.

Poniżej znajduje się przykład definicji Pytania:

```
Ask "Do you want a new armor?"
Answer "Yes" to { Get Armor }
Answer "No" to { Show "Your choise." }
```

Z powyższej definicji wynika, że graczowi zostanie wyświetlone pytanie - ”Do you want a new armor?”. Gdy odpowie ”Yes”, otrzyma przedmiot o nazwie ”Armor”. W przeciwnym wypadku ujrzy komunikat - ”Your choise.”.

7.7.5. Modyfikacja Przygód

Modyfikacja Przygód służy do dodawania lub usuwania Przygód do lub z Lokacji. Definicja Zdarzenia dodającego nową Przygodę rozpoczyna się od słowa ”Add”, a usuwającego ”Remove”. Następnie należy podać nazwę Przygody i opcjonalnie częstość jej występowania w nawiasach (domyślna częstość wynosi 1). Dalej należy podać lokację, której dotyczy edycja poprzedzając ją słowem kluczowym ”at”.

Poniżej znajduje się kilka przykładów Modyfikacji Przygód.

```
{ Add Bandits (4) at Forest }
{ Add VillageInDanger at Forest }
{ Remove MysteriusMan at OldHouse }
```

Pierwsze dwie linie dodają odpowiednio Przygody ”Bandits” i ”VillageInDanger” do Lokacji Forest. Przygoda ”Bandits” będzie występowała czterokrotnie częściej niż ”VillageInDanger”. Ostania linia usuwa Przygodę ”MysteriusMan” z Lokacji ”OldHouse”.

Blok Zdarzeń

Blok Zdarzeń to jedno lub więcej Zdarzeń umieszczonych w nawiasach klamrowych. Takie zdarzenia wywołują się sekwencyjnie. Poniższy przykład pokazuje blok składający się z 3 zdarzeń.

```
{
   {Remove VillageInDanger at Forest}
```

```
   {Get MagicSword}
   {Show "Unfortunately, you have not saved the villagers"}
}
```

7.7.6. Przykład

Poniżej znajduje się przykład kompletnej definicji przygody.

```
adventure MysteriusManAd starts from {
  Ask "Are you ready to fight?"
  Answer "No" to { Fight with MysteriusManTraining }
  Answer "Yes" to {
    If you have more str than 30
      or more int than 30
      or more agi than 30 {
        Fight with MysteriusMan
        If win {
          {Show "Great. Now, you are ready for fight with Dragon"}
          {Add BigDragon at Hills}
          {Remove MysteriusManAd at OldHause}
        }
      }
      else {Show "I don't think so"}
    }
}
```

Przygoda ”MysteriusManAd” rozpoczyna się od Pytania - ”Are you ready to fight?”. Odpowiedź ”No” prowadzi do Walki z ”MysteriusManTraining”. Odpowiedź ”Yes” prowadzi do efektu zależnego od statystyk postaci. Postać posiadająca przynajmniej jedną ze statystyk na poziomie powyżej 30 punktów, będzie musiała stoczyć Walkę z ”MysteriusMan”. Jeżeli gracz zwycięży w walce otrzyma komunikat - ”Great. Now, you are ready for fight with Dragon”, odblokuje Przygodę ”BigDragon” w Lokacji ”Hills”, a przygoda ”MysteriusManAd”, nie będzie dłużej dostępna

w ”OldHause”. Postać niespełniająca tych warunków otrzyma po prostu komunikat - ”I don’t think so”.

ROZDZIAŁ 8

Przykładowa gra

Aby ułatwić zrozumienie i korzystanie z Adventure Maker postanowiłem przygotować przykładową grę. Kod potrzebny do skompilowania gry znajduje się w załączniku.

8.1. Opis scenariusza

Ponieważ Adventure Maker powstał w ramach pracy dyplomowej postanowiłem, że tematem gry nie będzie oklepane ratowanie księżniczek i walka ze smokami w świecie fantasy, a studiowanie. Okazało się, że pomimo, że język nie został zaprojektowany z myślą o grach tego typu, bez problemu udało mi się ją napisać.

Akcja gry rozgrywa się na uniwersytecie, a celem gracza jest zdanie egzaminu magisterskiego. Aby to osiągnąć, gracz musi rozwijać umiejętności swojej postaci poprzez zdawanie pomniejszych egzaminów. Kiedy gracz będzie już gotowy do ostatecznej próby, musi udać się do specjalnej sali, w której może podejść do egzaminu i zdobyć uprawniony dyplom.

Gracz może osiągnąć cel obierając jedną z dwóch strategii. Na początku gry musi podjąć decyzję, czy chcę podążać szlachetną ścieżką dobrego studenta, czy też iść na łatwiznę i oszukiwać. Gra zawiera element moralizatorski, ponieważ wybierając ścieżkę oszusta, można co prawda zdać kilka pomniejszych egzaminów, ale ostatecznie gracz i tak skazany jest na porażkę. Gra zaimplementowana jest w taki sposób, żeby niewiedza nieuczciwego studenta w pewnym momencie wyżła na jaw, uniemożliwiając mu tym samym zakończenie studiów.

8.2. Uruchamianie gry

Grę można pobrać (https://adrpieper.github.io/magisterka) i zainstalować na dowolnym urządzeniu z systemem android w wersji powyżej 4.0, obsługującym komunikację NFC. Przed rozpoczęciem gry należy zaprogramować i umieścić znaczniki NFC zawierające tagi:

- ExamClassRoom
- PlainClassRoom

Znaczników ”PlainClassRoom” można przygotować dowolną ilość i rozmieścić je w kilku salach uczelni. Znacznik ”ExamClassRoom” powinien znajdować się w pojedynczej sali, w której odbywać się będą egzaminy dyplomowe.

Po takim przygotowaniu można już swobodnie rozpocząć rozgrywkę.

DODATEK A

Przykładowy kod Klasy Postaci

Poniżej znajdujące się kod prezentujący tworzenie obiektu reprezentującego taką samą klasę postaci na dwa sposoby: z i bez wykorzystania wewnętrznego DSL.

Kod niewykorzystujący wewnętrznego DSL

```
Stats statsOnStart = new Stats(10, 20, 30);
Stats statsPerLevel = new Stats(1, 2, 3);

SkillTree skills = new SkillTree(
  new SkillNode(
    BASIC_HIT,
    new SkillNode(POISON_HIT, BASIC_HIT),
    new SkillNode(SUPER_POISON_HIT, BASIC_HIT)
  ),
  new SkillNode(POISON_HIT)
);

new CharacterType("Knight",statsOnStart,statsPerLevel,skills);
```

Kod wykorzystujący wewnętrzny DSL

```
characterClass("Knight")
  .statsOnStart(10, 20, 30)
  .statsPerLevel(1, 2, 3)
  .skills(
    node(
```

```
      BASIC_HIT,
      node(POISON_HIT),
      node(SUPER_POISON_HIT)
  ),
  node(POISON_HIT)
)
```

DODATEK B

Kod przykładowej gry

```
skill BasicKnowledge {
  effect : take (3 * int) damage
  mp : 12
  cooldown : 1 turns
}

skill BrilliantIdea {
  effect : take (4 * int) damage
  mp : 20
  cooldown : 2 turns
}

skill Focusing {
  effect : take (12 * int + 13 * agi) damage
  mp : 60
  cooldown : 4 turns
}

skill AwesomeFocusing {
  effect : take (12 * int + 13 * agi) damage
  mp : 60
  cooldown : 4 turns
}

character type GoodStudent {
  stats on start:
    10 str
```

```
    20 int
    10 agi
  stats per lvl:
    + 1 str
    + 2 int
    + 1 agi
  skills tree:
    BasicKnowledge => {
      BrilliantIdea
      Focusing => {AwesomeFocusing}
    }
}

skill PaperCheatsheet {
  effect : take 2*agi damage
  mp : 10
}

skill Smartphone {
  effect : take 2*agi damage
  mp : 10
}

skill FriendHelp {
  effect : take 2*agi damage
  mp : 10
}

skill Luck {
  effect : take 2*agi damage
  mp : 10
}
```

```
character type Cheater {
  stats on start:
    10 str
    10 int
    20 agi
  stats per lvl:
    + 1 str
    + 1 int
    + 2 agi
  skills tree:
    Luck => {
      PaperCheatsheet
      FriendHelp
      Smartphone
    }
}

item FunnyHat (helmet) {
}

item GoodPen (weapon) {
  + 12 str
}

item GoodGrade (armor) {
  + 12 str
}

item Diploma (armor) {
  + 10 agi
  + 10 str
  + 10 int
}
```

```
item WarmGloves (gloves) {
  + 1 str
}

item BrandNewSmartPhone (weapon) {
  + 5 int
}

opponent SimpleExam {
  power : 10
  hp : 20
  exp : 100
  loot : GoodGrade 50%, FunnyHat 1 %, BrandNewSmartPhone 10%,
         GoodPen 20%,  WarmGloves 10%
}

opponent MasterDegreeExam{
  power : 50
  hp : 100
  exp : 1000
  loot : Diploma 100%
}

adventure NotAStudent starts from {
  Show "You are not a Student"
}

adventure SimpleExamAdventure starts from {
  Fight with SimpleExam
  If win {
    {
      If you have more int than 30 {
```

```
        {Show "You are ready for MasterDegreeExam" }
        {Remove SimpleExamAdventure at ClassRoom}
        {Add MasterExamAdventure at ClassRoom}
      }
      else
        { Show "You have to pass few more exams." }
    }
    {
      If you have more agi than 30 {
        {Show "You was catched on cheating." }
        {Add NotAStudent at ClassRoom}
        {Remove SimpleExamAdventure at ClassRoom}
      }
    }
  }
}

adventure MasterExamAdventure starts from {
  {Show "Have you passed all exams?"}
  {
    Fight with MasterDegreeExam
    If win {
      {Show "Great. You just have gradueted your Master Degree"}
      {Remove MasterExamAdventure at MasterExamRoom}
    }
    If lost
      { Show "You can try another one" }
    }
}

location ClassRoom tagged as ClassRoom
location MasterExamRoom tagged as MasterExamRoom
```

```
adventure on start {
  SimpleExamAdventure at ClassRoom
}
```

DODATEK C

Kod testu NFC

```
character type NFCTestCharacter {
  stats on start:
    10 str
    10 int
    10 agi
  stats per lvl:
    + 1 str
    + 1 int
    + 1 agi
  skills tree:
}
location TestLocation1 in radius of 10 meters
  from 54.601292, 18.269273
location TestLocation2 in radius of 10 meters
  from 54.600007, 18.269222

adventure TestAdventure1 starts from {
  Show "Test adventure 1"
}
adventure TestAdventure2 starts from {
  Show "Test adventure 2"
}
adventure on start {
  TestAdventure1 at TestLocation1
  TestAdventure2 at TestLocation2
}
```

DODATEK D

Kod testu GPS

```
character type NFCTestCharacter {
  stats on start:
    10 str
    10 int
    10 agi
  stats per lvl:
    + 1 str
    + 1 int
    + 1 agi
  skills tree:
}
location TestLocation1 tagged as Test1
location TestLocation2 tagged as Test2

adventure TestAdventure1 starts from {
  Show "Test adventure 1"
}

adventure TestAdventure2 starts from {
  Show "Test adventure 2"
}

adventure on start {
  TestAdventure1 at TestLocation1
  TestAdventure2 at TestLocation2
}
```

Bibliografia

[1] Strona domowa frameworku xtext. http://www.eclipse.org/Xtext/, 2016 (dostęp Kwiecień 21, 2017).

[2] Strona domowa gradle. https://gradle.org/, 2016 (dostęp Kwiecień 21, 2017).

[3] Strona domowa junit. http://junit.org/junit4/, 2016 (dostęp Kwiecień 21, 2017).

[4] Strona domowa dagger2. https://google.github.io/dagger/, 2016 (dostęp Kwiecień 21, 2017).

[5] Robert C. Martin. Czysty kod. Podręcznik dobrego programisty. Helion, 2015.

[6] Strona domowa języka xtend. http://www.eclipse.org/xtend/, 2016 (dostęp Kwiecień 21, 2017).

[7] Strona domowa systemu android. https://developer.android.com/, 2016 (dostęp Kwiecień 21, 2017).

[8] Strona domowa mockito. http://site.mockito.org/, 2016 (dostęp Kwiecień 21, 2017).

[9] Domainspecificlanguage. https://martinfowler.com/bliki/DomainSpecificLanguage.html, 15 May 2008 (dostęp Maj 19, 2017).

[10] Strona domowa rpgmaker. http://www.rpgmakerweb.com/, 2016 (dostęp Kwiecień 21, 2017).

[11] Strona domowa unity3d. https://unity3d.com/, 2016 (dostęp Kwiecień 21, 2017).

Spis tabel

Spis rysunków

Oświadczenie

Ja, niżej podpisany(a) oświadczam, iż przedłożona praca dyplomowa została wykonana przeze mnie samodzielnie, nie narusza praw autorskich, interesów prawnych i materialnych innych osób.

.....	.....
data	podpis