

UNIWERSYTET GDAŃSKI
Wydział Matematyki, Fizyki i Informatyki

Adrian Pieper

nr albumu: 243 677

AdventureMaker - język dla terenowych gier RGP

Praca magisterska na kierunku:

INFORMATYKA

Promotor:

dr W. Bzyl

Gdańsk 2017

Streszczenie

W ramach pracy został zaprojektowany i zaimplementowany *AdventureMaker* - DSL (ang. Domain Specific Language) służący do opisu terenowych gier RPG (ang. Role Playing Game). Implementacja wykorzystuje technologię *Xtext* [1], która umożliwiła stworzenie podstawowej infrastruktury języka zawierającej między innymi linker i kompilator. Dodatkowo powstały język zintegrowany jest z IDE *IntelliJ IDEA*¹, co pozwala na podświetlanie, automatyczne uzupełnianie i sprawdzanie składni.

Wygenerowane przez *AdventureMaker* gry działają na platformę *Android* [7]. W celu lokalizowania graczy zarówno w pomieszczeniach, jak i na otwartym terenie, wykorzystują technologię NFC² oraz GPS (ang. Global Positioning System).

W projekcie zostały wykorzystane popularne narzędzia i biblioteki, z których najważniejsze to *Gradle* [2], *JUnit* [3] i *Dagger 2* [4]. Kod został napisany zgodnie z zasadami Clean Code [5].

Praca składa się z dwóch części: implementacji języka *AdventureMaker* oraz szablonu aplikacji, wraz z przykładowym kodem gry. Całość kodu dostępna jest w publicznym repozytorium GIT (pod adresem <https://github.com/adrpieper/magisterka>). Zostały tam również udostępnione aplikacja oraz wtyczka do IDE *IntelliJ IDEA*.

Słowa kluczowe

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

¹Zintegrowane środowisko programistyczne firmy JetBrains. <https://www.jetbrains.com/idea/>

²Near Field Communication - wysokoczęstotliwościowy standard komunikacji krótkiego zasięgu. https://pl.wikipedia.org/wiki/Near_Field_Communication

Spis treści

Wprowadzenie	7
1. Przegląd istniejących rozwiązań	11
1.1. Gry terenowe	11
1.2. Gry RPG	12
1.3. Domain Specific Language (DSL)	13
1.3.1. Internal DSL	13
1.3.2. External DSL	14
1.4. Silniki gier	14
2. Wymagania języka <i>AdventureMaker</i>	17
2.1. Wymagania funkcjonalne	17
2.2. Wymagania нефункционалне	18
3. Interfejs Gry i Ogólny Przebieg Rozgrywki	21
3.1. Wybór Klasy Postaci	21
3.2. Tryb Zwiedzania	22
3.3. Tryb Przygody	23
4. Architektura implementacji języka <i>AdventureMaker</i>	25
4.1. <i>AdventureMaker</i> DSL	25
4.1.1. Definicja gramatyki	25
4.1.2. Generator Kodu	25
4.2. Aplikacja Android	26
5. Testy	33
5.1. Testy automatyczne	33
5.1.1. Struktura testów	33
5.1.2. Wyniki testów	34
5.2. Testy modułów NFC i GPS	34

5.3. Kod testów	35
5.4. Sprawdzenie poprawności zachowania aplikacji	35
6. Pisanie gier w <i>AdventureMaker</i>	37
6.1. Konfiguracja środowika	37
6.2. Przygotowanie i kompilacja gry	38
7. Dokumentacja języka	39
7.1. Klasy Postaci	39
7.2. Typy Przedmiotów	41
7.3. Lokacje	41
7.4. Przeciwnicy	42
7.5. Przygody	43
7.6. Przygody Początkowe	48
8. Przykładowa gra	49
8.1. Opis scenariusza	49
8.2. Uruchamianie gry	50
A. Przykładowy kod Klasy Postaci	53
B. Kod przykładowej gry	55
C. Kod testu NFC	61
D. Kod testu GPS	63
Bibliografia	65
Spis rysunków	67
Oświadczenie	69

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, bazujące głównie na fizycznej pozycji gracza, stały się nowym trendem w dziedzinie rozrywki elektronicznej.

Pomimo sukcesu *Pokemon 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 programowania - "nie powtarzaj się" [5]. 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 wyspecjalizowanego 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. Ponieważ wszystkie niezbędne w grze elementy są już zaimplementowane, *AdventureMaker* pozwala na implementację gry przez osobę niebędącą programistą. 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 integrację zaprojektowanego języka z popularnymi IDE: *Eclipse* oraz *IntelliJ IDEA*.

Xtend jest językiem programowania wywodzącym się i składniowo zbliżonym do Java, który został zaprojektowany, aby wyeliminować kilka wad tego języka. Zawiera elementy niedostępne w Java oraz upraszcza niektóre struktury pozwalając na pozbycie się niepotrzebnego kodu. Kodem wynikowym dla języka *Xtend* jest Java, a nie kod binarny. Mamy tu więc do czynienia z transpilacją, czyli przetłumaczeniem jednego kodu źródłowego na 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 *AndroidSDK* [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 biblioteki *JUnit* [3] oraz *Mockito* [8]. *JUnit* służy pisania testów jednostkowych dla programów napisanych w języku Java. *Mockito* umożliwia tzw. Mockowanie¹. 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 (ang. Dependency Injection), jednak *Dagger 2* cieszy się największą popularnością wśród programistów systemu *Android*. Jego główną zaletą jest rozstrzygnięcie zależności w czasie kompilacji oraz generacja kodu. Dzięki temu można wykryć wady projektu, takie jak zależności cykliczne jeszcze przed uruchomieniem aplikacji. Dodatkowo wygenerowany kod jest o wiele szybszy niż refleksja, czyli typowe rozwiązanie czasu wykonania. Jest to bardzo ważna cecha w przypadku aplikacji na urządzenia mobilne, ponieważ posiadają one stosunkowo ograniczone zasoby.

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

ROZDZIAŁ 1

Przegląd istniejących rozwiązań

AdventureMaker to język pozwalającego na pisanie terenowych gier RPG. Zastępuje on więc popularne rozwiązanie jakim jest korzystanie z silnika gier.

Tematyka pracy zahacza o cztery tematyki jakimi są gry terenowe, gry RPG, silniki gier 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 nazywa się rzeczywistością rozszerzoną, są pojawiające się w grze, a nie istniejące w rzeczywistości, obiekty lub postaci, które wpływają w jakiś sposób na przebieg rozgrywki.

Doskonałym przykładem gry terenowej jest wspomniana we wstępie *Pokemon GO*. 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 znalezione z popularnej gry Monopoly do świata rzeczywistego. Cel gry pozostał niezmieniony, jest nim oczywiście inwestowanie w nieruchomości.

ści. 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 osiągnięcie 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 też 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 muszą 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, war-

to zwrócić uwagę, że typowe elementy tych gier takie jak rozwój i statystyki postaci przedostały się już do prawie każdego gatunku gier komputerowych.

1.3. Domain Specific Language (DSL)

Domain Specific Languages [9], czyli języki domenowe, projektowane są z myślą o ściśle określonym i z reguły bardzo wąskim zastosowaniu. W odróżnieniu od języków programowania ogólnego przeznaczenia, języki domenowe nie nadają się do rozwiązywania 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 się 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 pisanie 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 implementację. O jakości takiego rozwiązania świadczy nie tyle wydajność jego działania, lecz łatwość użycia. Biblioteki takie pozwalają programiście na bardziej czytelne, prostsze i zwarte wyrażenie jego 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, choć 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. Silniki gier

Silniki gier są doskonałym przykładem korzyści jakie niesie ze sobą popularna zasada *Clean Code* [5] - DRY (Don't Repeat Yourself), czyli nie powtarzaj się.

Ideą silnika jest implementacja wszystkich elementów wspólnych dla gier w jednym miejscu i udostępnienie programistom-użytkownikom przyjaznego interfejsu do implementacji mechaniki ich własnej gry. Programista jest oczywiście ograniczony przez możliwości silnika, ale w zamian uzyskuje narzędzie, które pozwala mu skupić się tylko na elementach istotnych dla jego gry.

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 oprogramowaniem jest np. *Unity* [11]. Unity udostępnia 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 języka *AdventureMaker*

Jednym z pierwszymi kroków, jakie należy podjąć przed przystąpieniem do projektowania oprogramowania jest analiza wymagań. *AdventureMaker* 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 oraz pokonywanie przeciwników. Postać sterowana przez gracza zdobywa punkty doświadczenia, dzięki którym gracz może rozwijać postać w wybranym przez siebie kierunku.

Wymagania, jakie postawiłem przed *AdventureMaker* można podzielić na funkcjonalne i нефункционалне.

2.1. Wymagania funkcjonalne

Do wymagań funkcjonalnych należą:

- Język DSL zawierający elementy:
 - 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 implementację nawet dość zawilej logiki. Oczywiście RPG z prawdziwego zdarzenia powinien być bardziej rozbudowany. Uznałem jednak, że dodanie elementów takich jak np. misje zbędnie skomplikowałyby język. Elementy te prawdopodobnie pojawią się, w przypadku wydania kolejnej wersji *AdventureMaker*. Niemniej jednak implementacja stworzona w ramach tej pracy obejmować będzie jedynie wyszczególnione punkty.

W 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 *AdventureMaker* wtyczkę do IDE *IntelliJ IDEA*. Wybrałem 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 musi być możliwa kompilacja gotowej aplikacji, bez konieczności dopisywania ani jednej linii dodatkowego kodu.

2.2. Wymagania niefunkcjonalne

Do wymagań niefunkcjonalnych należą:

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

- Wykorzystanie darmowych narzędzi
- Ukrycie szczegółów implementacyjnych GPS i NFC przed programistą gry

Podstawowym założeniem *AdventureMaker* jest udostępnienie możliwości tworzenia gier osobom będącym laikami jeśli chodzi o programowanie i technologie IT. Aby móc w ogóle udostępnić im ten język, ważne jest wykorzystanie darmowych narzędzi.

Język *AdventureMaker* 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

Interfejs Gry i Ogólny Przebieg Rozgrywki

Pomimo, że ostateczny kształt gry będzie zdefiniowany przez jego programistę, przebieg każdej rozgrywki jest podobny. Ciężko byłoby go opisać w oderwaniu od jej graficznego interfejsu, jako, że stanowi on jej nieodzowny element.

Celem gry jest rozwój kierowanej przez gracza postaci. Jest to możliwe dzięki tzw. *Przygodom*, czyli wydarzeniom uruchamiającym się w zależności od fizycznej lokalizacji postaci. Miarą rozwoju postaci są *Poziomy Doświadczenia*, które zwiększają się po osiągnięciu określonego progu *Punktów Doświadczenia*.

Rozgrywkę można podzielić na 3 etapy:

- Wybór Klasy Postaci
- Tryb Zwiedzania
- Tryb Przygody

3.1. Wybór Klasy Postaci

Na początku gry gracz musi dokonać wyboru *Klasy Postaci*. GUI składa się pojedynczego ekranu podzielonego na dwie części: listy *Klas Postaci* oraz szczegółów wybranej *Klasy*. Lista pozwala na wybór jednej z pośród udostępnionych przez programistę *Klas*. Każda *Klasy Postaci* posiada nazwę, określone *Statystyki* oraz *Drzewo Umiejętności*, którymi powinien kierować się gracz dokonując wyboru.

Na *Statystyki* postaci składają się jego *Siła*, *Inteligencja* i *Zręczność*. Wartości te wpływają na przebieg *Przygody* oraz umiejętności bojowe postaci. *Statystyki* można podzielić na *Początkowe*, czyli takie, które gracz otrzymuje na początku rozgrywki oraz *Poziomowe*, które gracz otrzymuje z każdym nowym *Poziomem Doświadczenia*.

Drzewko Umiejętności zawiera informację na temat dostępnych dla danej *Klasy Postaci Umiejętności*. Gracz może korzystać z *Umiejętności* podczas walki, ale dopiero po ich odblokowaniu. Aby odblokować daną *Umiejętność*, musi posiadać wszystkie będące jej przodkami w drzewie *Umiejętności*. Za każdą nową odblokowaną umiejętność gracz musi zapłacić *Punktami Umiejętności*, które zdobywa wraz z nowym *Poziomem Doświadczenia*.

3.2. Tryb Zwiedzania

Gracz znajduje się w tym trybie od momentu wyboru postaci przez cały czas, w którym gracz porusza się w poszukiwaniu *Przygód*.

Tryb zwiedzania 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 *Punktów* oraz *Poziomie Doświadczenia* sterowanej przez gracza postaci. Dodatkowo wyświetla też *Statystyki* uwzględniające *Premie* otrzymane z używanych *Przedmiotów*.

Umiejętności

Ekran udostępnia *Drzewko Umiejętności* oraz dwa przyciski *Reset* i *OK*. Służy zarówno do przeglądania posiadanych, jak i odblokowywania nowych *Umiejętności*. Przycisk *Reset* przywraca graczowi rozdane *Punkty Umiejętności*, natomiast przycisk *OK* zatwierdza ostatecznie wybór odblokowanych *Umiejętności*.

Przedmioty

Przedmiot stanowi nagrodę dla gracza i może być wykorzystany w celu podniesienia *Statystyk* jego postaci o wartości przypisane mu jako *Premie do Statystyk*.

Sekcja zawiera wszystkie posiadane przez gracza *Przedmioty*. Te używane aktualnie znajdują się w górnej części ekranu, natomiast pozostałe w dolnej, zwanej *Plecakiem*. Dodatkowo na środku wyświetlana jest suma *Premii do Statystyk* uzyskanych z używanych *Przedmiotów*.

Przedmioty podzielone są na kategorie, a gracz może używać jednocześnie tylko po jednym *Przedmiocie* każdego typu.

3.3. Tryb Przygody

Gracz przechodzi do tego trybu z *Trybu Zwiedzania*, kiedy znajdzie się w miejscu zawierającego *Przygodę*. Każda przygoda składa się z serii *Zdarzeń* (min. jedno).

GUI trybu składa się kilku ekranów:

- Decyzja
- Walka
- Komunikat

Każdy odpowiedzialny jest za wyświetlenie konkretnych typów *Zdarzeń*. Gracz nie może swobodnie nawigować pomiędzy nimi, ponieważ wpływa na przebieg *Zdarzeń* jedynie w sposób pośredni, a dostępne GUI zależne jest od sytuacji, w której się aktualnie znajduje.

Decyzja

Ekran zawiera pole wyświetlające pytanie skierowane do gracza oraz listę decyzji w postaci przycisków. Poprzez wybór odpowiedniej decyzji, gracz ma wpływ na dalszy przebieg *Przygody*.

Walka

Widok *Walki* służy do wyświetlania oraz sterowania przebiegiem starcia pomiędzy postacią gracza, a dowolnym przeciwnikiem. Interfejs wyświetla nazwę przeciwnika, kolorowe paski zawierające informacje o *Punktach Witalnych* postaci i przeciwnika oraz o *Punktach Many* gracza.

Aby wygrać *Walkę* gracz musi zmniejszyć ilość posiadanych przez przeciwnika *Punktów Witalnych* do zera. Analogicznie, aby nie przegrać, musi utrzymywać ilość własnych *Punktów Witalnych* powyżej tego poziomu.

Gracz ma do dyspozycji zdobyte *Umiejętności* oraz *Atak Podstawowy*. Korzystanie z *Umiejętności* wymaga poniesienie kosztu w *Punktach Many*.

Skuteczność *Ataku Podstawowego* oraz ilość *Punktów Witalnych* i *Many* gracza zależy od *Statystyk* jego postaci.

Komunikat

Ekran komunikatu służy do wyświetlania wszelkiego rodzaju informacji dotyczącej przebiegu *Przygody*. Dzięki nim możliwe jest wprowadzanie narracji w grze, co stanowi istotny element gier RPG. GUI składa się z wyświetlanej informacji oraz przycisku *OK*, którym gracz potwierdza, że zapoznał się z komunikatem. Informacja najczęściej wyświetlana jest w prostym oknie tekstowym, jednak czasami konieczne jest wyświetlenie elementów bardziej rozbudowanych. Przykładem jest okno wyświetlane pod koniec *Przygody*, zawierającego listę zdobytych przez gracza *Przedmiotów* oraz *Punktów Doświadczenia*.

Architektura implementacji języka *AdventureMaker*

AdventureMaker składa się z dwóch projektów. Pierwszy odpowiedzialny jest za implementację języka. Drugi projekt to szkielet aplikacji na system Android będący bazą pod wszystkie gry pisane w *AdventureMaker*.

4.1. *AdventureMaker* DSL

Projekt powstał w celu implementacji gramatyki i generatora dla języka *Adventure 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 *AdventureMaker* znajduje w pojedynczym pliku o nazwie *AML.xtext*. Definicja w nim zawarta zawiera opis modelu syntaktycznego i semantycznego tworzonego języka. Innymi słowy definiuje, jaki tekst należy do języka oraz jak będzie on reprezentowany w pamięci komputera. *Xtext* na podstawie tego pliku generuje między innymi parser, który sprawdza, czy dany tekst jest poprawnym programem oraz zwraca jego reprezentację 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 postaci drzewa obiektów. Generator został napisany w języku *Xtend* i składa się z głównej klasy implementującej metodę *doGenerate* oraz klas pomocniczych utworzonych

w celu dekompozycji problemu na mniejsze części zgodnie z zasadą pojedynczej odpowiedzialności [5].

4.2. Aplikacja Android

Projekt to standardowa aplikacja androidowa wykorzystująca *AndroidSDK* rozbudowana o język *AdventureMaker*, na podstawie którego generowana jest część kodu aplikacji.

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

- Standardowy Kod Aplikacji
- Zasoby
- Plik manifestu
- Pliki *Gradle*
- Testy
- Wewnętrzny DSL
- Plik *game.aml*
- Wygenerowany Kod Implementujący Zasady Gry
- Testowy Kod Implementujący Zasady Gry

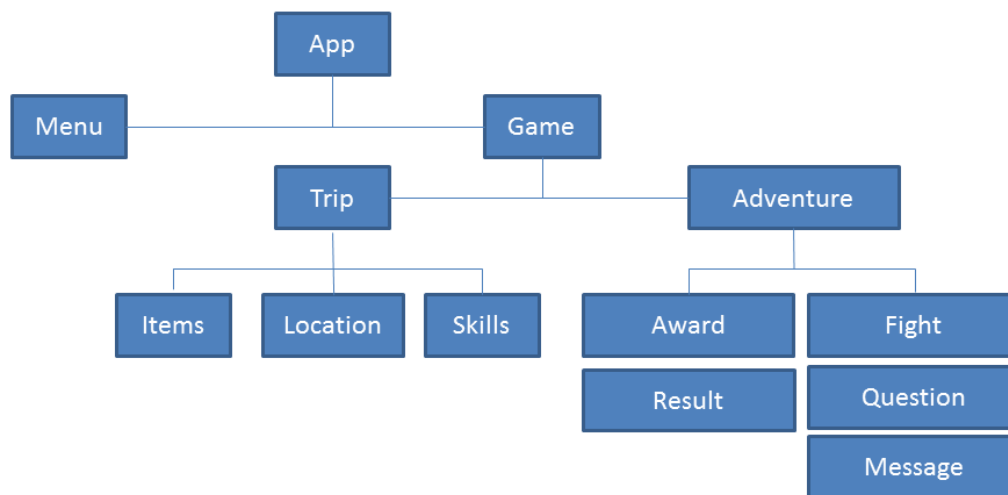
Pierwszych pięć elementów jest typowych dla aplikacji android, natomiast pozostałe są specyficzne i wynikają z zastosowania języka *AdventureMaker*.

Standardowy Kod Aplikacji

Jest to bardzo rozbudowana, ale i najbardziej tradycyjna i oczywista część aplikacji. W jego skład wchodzi pakiety *edu.ug.inf.am.** umieszczone w folderze *app/src/main/java*. Moduł jest odpowiedzialny za sposób działania i wygląd aplikacji. Korzysta z Kodu

Implementującego 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 implementuje elementy stałe dla każdej gry takie jak mechanizm walki i rozwoju postaci.

Kod modułu 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 wchodzi kolejne, odpowiedzialne za zadania bardziej sprecyzowane. Kompletną strukturę komponentów aplikacji przedstawia drzewo 4.1.



Rysunek 4.1. Struktura komponentów aplikacji

Źródło: Własne opracowanie

Wszystkie komponenty zostały zrealizowane w postaci pakietów o podobnej strukturze. W ich skład wchodzi po kilka pakietów-warstw zawierających bezpośrednio klasy realizujące funkcjonalność tych warstw. Dodatkowo pakiety, nie będące liśćmi w drzewie, zawierają pakiety-komponenty znajdujące 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 wchodzi przede wszystkim klasy dziedziczące po *Activity*, *Fragment* oraz *View*. W większości przypadków zadaniem klasy jest połączenie danych, udostępnianych przez warstwę kontrolera, z widokiem za pomocą dostępnego w systemie *Android* mechanizmu *Data Binding*[7].

Warstwa kontrolera odpowiada przede wszystkim za obsługę zdarzeń przychodzących z interfejsu użytkownika, 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 (*getter*y i *setter*y). 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 (ang. Plain Old Java Object), 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 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 w jej skład wchodzi zestaw stałych, będących identyfikatorami odpowiednich zasobów. *AndroidSDK* 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.

Większość zasobów stanowią layouty, z których znaczna część wykorzystuje *Data Binding*. Mechanizm 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, które zostały umieszczone odpowiednio w folderze */app* i */*. W plikach *Gradle* zdefiniowana 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/src/test/java*. W jego skład wchodzi klasy implementujące testy jednostkowe wszystkich ważnych elementów aplikacji. Kod testów wykorzystuje zewnętrzne biblioteki *JUnit*, *Mockito* i *AssertJ*.

Wewnętrzny 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 miało na celu uproszczenie implementacji generatorów kodu.

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

Istotną częścią tego modułu są klasy pozwalające na wygodne budowanie obiektów (tzw. *Buildery*¹) oraz metody statyczne tworzące instancje tych klas. Różnicę pomiędzy kodem pisanym z i bez wykorzystania wewnętrznego DSL zaprezentowałem w dodatku A.

¹Builder - Budowniczy - wzorzec projektowy [https://pl.wikipedia.org/wiki/Budowniczy_\(wzorzec_projektowy\)](https://pl.wikipedia.org/wiki/Budowniczy_(wzorzec_projektowy))

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 *AdventureMaker* elementy związane z konkretną grą.

Wygenerowany Kod Implementujący Zasady Gry

Ten moduł znajduje się w folderze `app/aml-src` i zawiera kod wygenerowany automatycznie na podstawie pliku `game.aml`. Zawarty w nim kod 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 Implementującego Zasady Gry, jest jednak napisany ręcznie. Kod został umieszczony w folderze `app/test-src`. Moduł 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 konieczne było przetestowanie go. Testy automatyczne zaimplementowane zostały w postaci mockowanych testów jednostkowych za pomocą bibliotek *JUnit* i *Mockito*.

Testy automatyczne, są najbardziej praktyczne i wiarygodne. Niestety z przyczyn technicznych nie dało się pokryć nimi 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 testowane są z osobna działania 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.

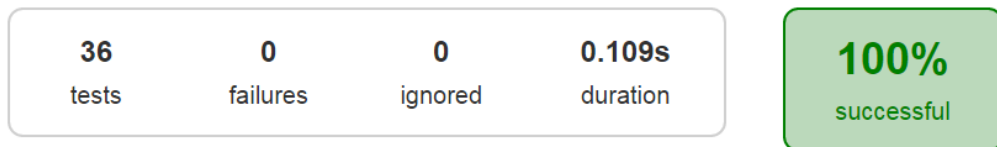
Sekcja *When* składa się z pojedynczej linii. Jest to wywołanie testowanej metody na instancji testowanej klasy.

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 testów

Testy jednostkowe zostały uruchomione przy pomocy narzędzia *Gradle* [2]. Zestawienie wyników (rys. 5.1) pokazuje, że wszystkie testy przeszły pomyślnie.

Test Summary



Rysunek 5.1. Podsumowanie testów

Źródło: Wygenerowane przez *Gradle*

5.2. Testy modułów NFC i GPS

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

- Przygotowanie pliku AML¹
- Kompilacja
- Sprawdzenie poprawności zachowania

¹AdventureMaker Language - rozszerzenie plików obsługiwane przez język *AdventureMaker*

5.3. Kod testów

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 obszary o promieniu 10 metrów. W przypadku NFC *Lokacje* zdefiniowane są przez odpowiednie tagi *Test1* i *Test2*.

Kod *AdventureMaker* przygotowany do przeprowadzenia obydwu testów znajduje się w dodatkach.

5.4. Sprawdzenie poprawności zachowania aplikacji

Poprawność działania modułów GPS i NFC została sprawdzona manualnie. W obu przypadkach zostało sprawdzone, czy wejście do odpowiedniej *Lokacji* uruchomi odpowiednie *Zdarzenie*.

W przypadku testu GPS urządzeniem wyposażone w moduł GPS zostało przeniesione w odpowiednie miejsca. W przypadku NFC telefon został przyłożony do odpowiednich znaczników.

Oba testy zakończyły się sukcesem.

ROZDZIAŁ 6

Pisanie gier w *AdventureMaker*

Pisanie własnej gry przy użyciu *AdventureMaker* 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 *AdventureMaker* należy pobrać i zainstalować IDE *IntelliJ IDEA*, JRE¹, JDK² i Android SDK oraz wtyczkę AML.

JDK o raz JRE należy pobrać ze strony *Oracle* (<https://www.java.com>) w wersji 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 *AdventureMaker*. Podczas instalacji należy pamiętać o włączeniu pluginu obsługującego aplikacje pisane na system Android.

Android SDK należy pobrać 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 *AdventureMaker*, 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 z repozytorium projektu (<https://github.com/adrpieper/magisterka>) i zainstalować ręcznie.

¹Java Runtime Enviroment - środowisko uruchomieniowe Java

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

6.2. Przygotowanie i kompilacja gry

Przed rozpoczęciem pracy nad własną grą należy pobrać znajdujący się w repozytorium (<https://github.com/adrpieper/magisterka>) projekt przykładowej gry (ExampleAdventure). 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 pliku *game.aml*. Należy w nim umieścić kod w języku *AdventureMaker*. Kod musi zawierać wszystkie niezbędne elementy gry. Pisząc ten kod można posłużyć się załączoną dokumentacją języka oraz wzorować się na kodzie przykładowej gry.

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.

Dokumentacja języka

Składnia języka przypomina nieco format *json*¹. Została jednak dostosowana do tego, aby była jak najbardziej czytelna i zrozumiała dla osób zaznajomionych z tematyką gier RPG.

Kod gier napisane w *AdventureMaker* umieszcza się w pojedynczym pliku z rozszerzeniem *.aml*. Plik definiuje wszystkie dostępne w grze elementy. Może zawierać dowolną ilość *Klas Postaci*, *Typów Przedmiotów*, *Przeciwników*, *Lokacji* oraz *Przygód*. Dodatkowo należy w nim umieścić sekcję *Przygody Początkowe*.

7.1. Klasy Postaci

Definicja *Klasy*, inaczej typu postaci to informacje na temat *Statystyk* oraz *Umiejętności* dostępnych dla danej *Klasy Postaci*. Definicja rozpoczyna się od słów kluczowych *character type*. Następnie należy podać nazwę *Klasy Postaci* 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 parametrów².

¹JavaScript Object Notation - lekki format wymiany danych komputerowych <https://pl.wikipedia.org/wiki/JSON>

²Strength, Intelligence, Agility

Definicja *Statystyk Poziomowych* jest bardzo podobna. Odróżniają je jedynie słowa kluczowe *stats per lvl*: zamiast *stats on start*: oraz występujący przed każdym parametrem znak plusa, który podkreśla, 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*. Zdobyte umiejętności *Fireball* udostępnia dwie kolejnych: *Wirewall* i *BlackMagic*.

7.2. Typy Przedmiotów

Definicja *Przedmiotu* składa się z nazwy, kategorii oraz wartości *Premii do 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*. W przypadku braku zdefiniowania, któreś z *Premii*, wynosi ona 0.

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  
}
```

7.3. 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* to okrąg o środku opisanym współrzędnymi geograficznymi 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 from 54.596077, 18.245200
```

7.4. Przeciwnicy

Przeciwnik to byt, z którym gracz może stoczyć *Walkę* podczas trwania *Przygody*. Definiując *Przeciwnika* należy użyć słowa kluczowego *opponent*, podać jego nazwę oraz zbiór parametrów otoczonych nawiasami klamrowymi.

Pierwsze trzy parametry: *power*, *hp*, *exp* to wartości liczbowe. Oznaczają one odpowiednio *Moc Przeciwnika*, czyli jego zdolność bojowa, ilość posiadanych przez niego *Punktów Witalnych* oraz ilość *Punktów 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 prawdopodobieństwo jego otrzymania.

Poniższy przykład definiuje *Przeciwnika Dragon*:

```
opponent Dragon {
    power : 100
```

```
hp : 200
exp : 100
loot : MagicSword 50%, SteelArmor 10 %
}
```

Z definicji wynika, że *Dragon* posiada *Moc* o wartości 100 oraz 200 *Punktów Witalnych*. Za pokonanie tego *Przeciwnika* gracz otrzyma 100 *Punktów Doświadczenia*, oraz odpowiednio 50% i 10% szansy na zdobycie *Przedmiotów MagicSword* oraz *SteelArmor*.

7.5. Przygody

Przygody są najważniejszym i najbardziej rozbudowanym elementem języka *AdventureMaker*. Dzięki nim możliwe jest napisanie scenariusza rozgrywki.

Definicja *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*.

Komunikat

Komunikat służy wyświetleniu informacji, z którym ma się zapoznać gracz. Definicja składa się ze słowa kluczowego *Show* oraz wyświetlanej informacji.

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

Show "Hello World!"

Walka

Walka oznacza *Zdarzenie* polegające na pojedynku dowolną liczbą przeciwników. Zawiera informacje o ilości i typie *Przeciwników* oraz *Zdarzeniach* wywoływanych w przypadku zwycięstwa jak i porażki.

Definicja 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*: *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*.

Zdarzenie Warunkowe

Zdarzenie Warunkowe służy do zadeklarowania *Zdarzeń*, które wywołane zostaną, tylko w przypadku spełnienia określonych warunków. Dodatkowo można zdefi-

niować *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 definicję *Zdarzenia*. Opcjonalnie można dodać definicję *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 poprzedzić słowem *no*, które wprowadza negację.

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".

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. Po pytaniu powinny się znaleźć jedna lub więcej *Odpowiedzi*. *Odpowiedź* składa się ze słów kluczowych *Answer* oraz *to*, pomiędzy którymi należy umieścić treść *Odpowiedzi* 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".

Modyfikacja Przygód

Modyfikacja Przygód służy do dodawania lub usuwania *Przygód 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*. Ostatnia 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"}
}
```

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 *OldHau-se*. Postać niespełniająca tych warunków otrzyma po prostu *Komunikat* - "I don't think so".

7.6. Przygody Początkowe

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

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

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


ROZDZIAŁ 8

Przykładowa gra

Aby ułatwić zrozumienie i korzystanie z *AdventureMaker* postanowiłem przygotować przykładową grę. Kod potrzebny do skompilowania gry znajduje się w dodatku.

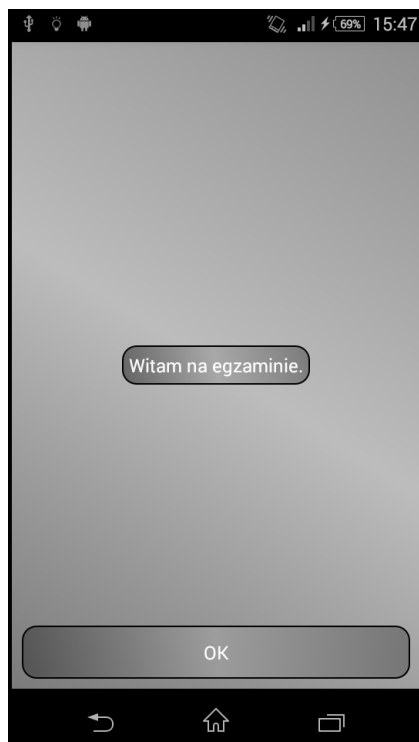
8.1. Opis scenariusza

Ponieważ *AdventureMaker* powstał w ramach pracy dyplomowej tematem gry nie jest ratowanie księżniczek i walka ze smokami w świecie fantasy, a studiowanie. Okazuje się, że pomimo, że język nie został zaprojektowany z myślą o grach tego typu, bez problemu udało 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 upragniony dyplom.

Gracz może osiągnąć cel obierając jedną z dwóch strategii. Na początku gry musi podjąć decyzję, czy chce 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 wyszła na jaw, uniemożliwiając mu tym samym zakończenie studiów.

Po przyłożeniu urządzenia do znacznika postać gracza przenosi się do sali egzaminacyjnej, a gracz powinien zobaczyć widoczny na zrzucie z ekranu (rys. 8.1) komunikat.



Rysunek 8.2. Wiadomość wyświetlana przed podejściem do testu

Źródło: Własne opracowanie

DODATEK A

Przykładowy kod Klasy Postaci

Poniżej znajduje 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 caught 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 graduated 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/>. (dostęp Kwiecień 21, 2017).
- [2] Strona domowa gradle. <https://gradle.org/>. (dostęp Kwiecień 21, 2017).
- [3] Strona domowa junit. <http://junit.org/junit4/>. (dostęp Kwiecień 21, 2017).
- [4] Strona domowa dagger2. <https://google.github.io/dagger/>. (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/>. (dostęp Kwiecień 21, 2017).
- [7] Strona domowa systemu android. <https://developer.android.com/>. (dostęp Kwiecień 21, 2017).
- [8] Strona domowa mockito. <http://site.mockito.org/>. (dostęp Kwiecień 21, 2017).
- [9] Martin Fowler. Domain specific language. <https://martinfowler.com/bliki/DomainSpecificLanguage.html>. 15.05.2008 (dostęp 19.05.2017).
- [10] Strona domowa rpgmaker. <http://www.rpgmakerweb.com/>. (dostęp Kwiecień 21, 2017).
- [11] Strona domowa unity3d. <https://unity3d.com/>. (dostęp Kwiecień 21, 2017).

Spis rysunków

4.1. Struktura komponentów aplikacji	27
5.1. Podsumowanie testów	34
8.1. Przykład umieszczenia znacznika NFC	50
8.2. Wiadomość wyświetlana przed podejściem do testu	51

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