WYŻSZA SZKOŁA
INFORMATYKI i ZARZĄDZANIA
z siedzibą w Rzeszowie

KOLEGIUM INFORMATYKI STOSOWANEJ

Kierunek: INFORMATYKA

Specjalność: Programowanie

Patryk Rzegocki
Nr albumu studenta w69840

Gra komputerowa akcji z walką turową i elementami RPG

Prowadzący: mgr inż. Ewa Żesławska

Praca projektowa programowanie obiektowe C#

Rzeszów 2024

Spis treści

Wstęp

W dzisiejszym świecie, w którym technologia i rozrywka odgrywają kluczową rolę w naszym życiu, wiele osób poszukuje sposobów na oderwanie się od codziennych obowiązków i stresów. Gry RPG maja na celu nie tylko dostarczenie rozrywki, ale także stworzenie unikalnej przestrzeni, w której ludzie mogą przeżywać przygody, rozwijać swoje umiejętności i zanurzać się w wirtualnym świecie, praktykując eskapizm. W świecie gry, gracze będą mieli możliwość wcielenia się w różnorodne postacie, które będą musiały stawić czoła wyzwaniom, zagadkom i potworom. Szczególnie przyciągająca jest wyjątkowa forma - niespotykany sposób przedstawienia gry i wciągające mechaniki potrafią zapewnić ludziom godziny zabawy. Z tego powodu, ważne jest, by znaleźć rozwiązanie na powyższe zagadnienie, co ten projekt stara się zrealizować.

Rozdział 1

Opis założeń projektu

1.1 Cele projetu

Projekt będzie dotyczył stworzenia gry komputerowej akcji z walką turową i elementami RPG, której głównym celem będzie dostarczenie użytkownikowi emocjonującej rozgrywki w walkach, eksploracji lochów, wykonywaniu zadań i walka z bossami. Problemem rozwiązywanym przez projekt jest deficyt gier komputerowych o tematyce roguelike, a także brak gier oferujących przystępny, ale angażujący system rozwoju postaci, który nie jest zbyt skomplikowany ani zbyt prosty. Wiele osób poszukuje gier, które nie wymagają skomplikowanego uczenia się, ale oferują angażującą rozgrywkę i możliwość rozwoju postaci w zależności od wybranej strategii. Prosty, ale wciągający system walki sprawi, że gra będzie dostępna dla graczy o różnych doświadczeniach z grami komputerowymi, co zwiększa jej potencjalną popularność. Analiza rynku gier pokazuje, że gracze cenią sobie gry z przystępnym poziomem trudności, po które można łatwo sięgnąć i równie łatwo odłożyć na później. Aby rozwiązać problem, wymagana jest wiedza na temat rynku gier, preferencji graczy, projektowania i programowania gier oraz testowania ich. W celu rozwiązania projektu należy: zaplanować i przygotować koncepcję gry; zaprojektować system walki i inne mechaniki; zaprojektować inne elementy takie jak przedmioty, umiejętności, lub przeciwnicy; zaimplementować mechaniki, funkcjonalności oraz interfejs użytkownika; przetestować oraz zoptymalizować grę; wydać finalną wersję gry. Wynikiem pracy będzie gotowy program komputerowy w postaci gry.

1.2 Wymagania funkcjonale i niefunkcjonalne

Wymagania funkcjonalne:
Na liście wymagań funkcjonalnych znajdują się,

- Tworzenie postaci.
- System statystyk, poziomów i umiejętności.
- Efekty pasywne, efekty statusu
- System walki.
- System eksploracji lochów: poruszanie się, pola z walką, ogniska, skrzynie, rośliny i pułapki
- Kupowanie, używanie, tworzenie i inne operacje na przedmiotach i ekwipunku.
- System ekwipunku i powiązanych mechanik (ulepszanie, przekuwanie, wstawianie ulepszeń w postaci galdurytów).
- System innych przedmiotów użytkowych: mikstur, galdurytów, torb z przedmiotami i innych.
- Generowanie przeciwników.

- System postaci niezależnych (NPC) i ich usług
- System misji
- Obsługa plików JSON jako baza danych NoSQL.
- Zapis stanu gry.
- Interfejs użytkownika (Spectre.Console)

Wymagania niefunkcjonalne:
Na liście wymagań niefunkcjonalnych znajdują się,

- Wydajność.
- Responsywność.
- Zgodność z platformą Windows.
- Dostępność.
- Optymalizacja pamięci.
- Możliwość rozbudowy.
- Użyteczność.
- Niezawodność.
- Przenośność.

Rozdział 2

Opis struktury projektu

Niniejszy rozdział przedstawia zaprojektowaną strukturę projektu, jego opis techniczny oraz kluczowe informacje dotyczące używanych technologii.

2.1 Język i narzędzia

Projekt został zrealizowany w języku C# w platformie .NET 8.0. Wykorzystano również następujące narzędzia:

- **JetBrains Rider** - IDE do programowania w C#.
- **Newtonsoft.Json** - biblioteka do obsługi JSON.
- **Spectre.Console** - biblioteka do tworzenia interaktywnych konsolowych aplikacji.
- **KanbanFlow** - narzędzie do planowania i zarządzania projektem metodą Kanban.
- **Microsoft Office** - wielozadaniowy pakiet aplikacji biurowych. W projekcie zostały konkretnie wykorzystane programy Word oraz Excel, do projektu gry i przechowywania informacji o mechanikach.

2.2 Minimalne wymagania sprzętowe

Aby uruchomić projekt, wymagane są następujące minimalne zasoby sprzętowe:

- Procesor: 2.0 GHz lub szybszy.
- Pamięć RAM: 4 GB.
- Miejsce na dysku: 10 MB wolnego miejsca.
- System operacyjny: Windows 10 lub nowszy.
- .NET: 8.0 lub nowszy.

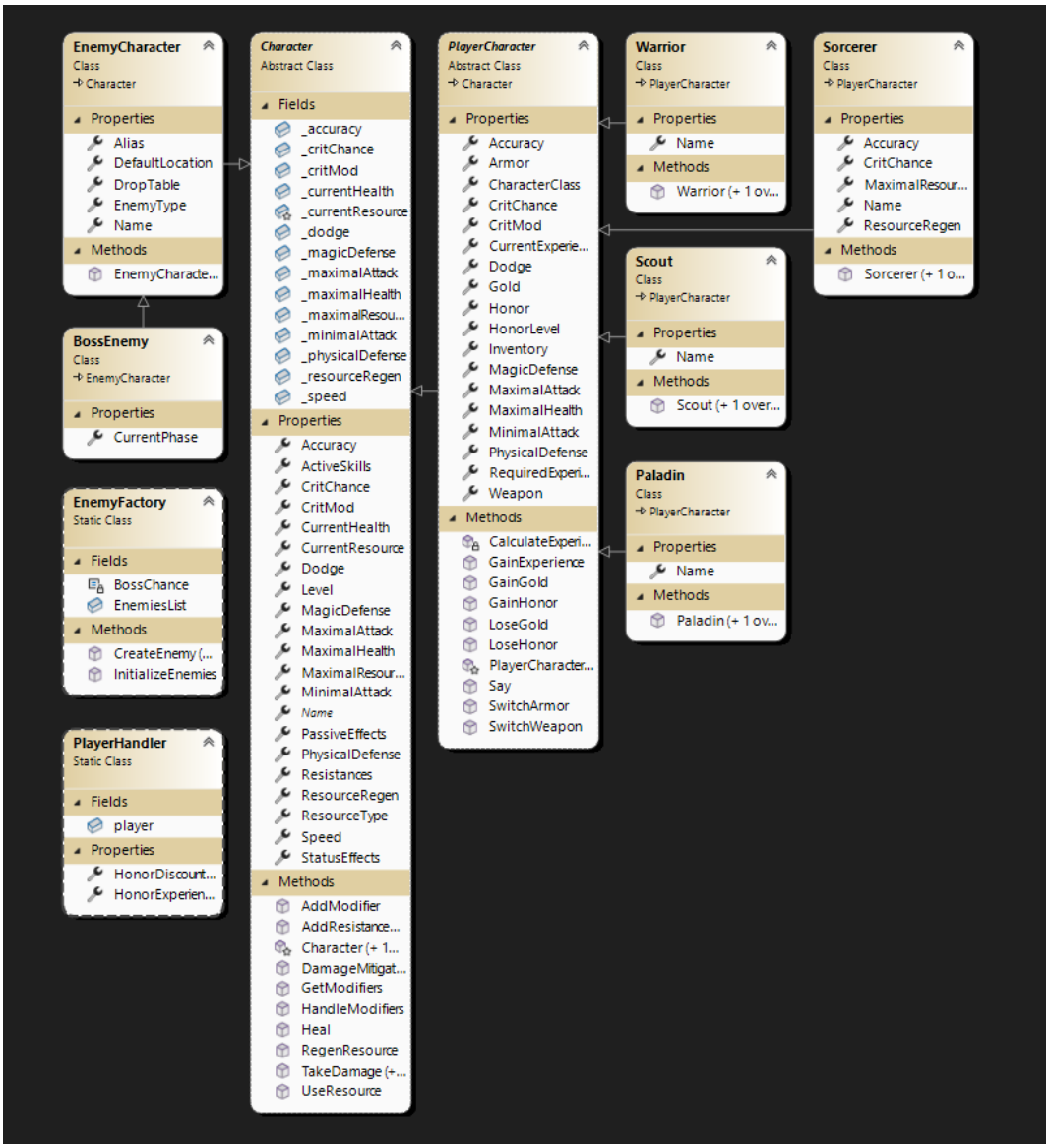
2.3 Zarządzanie danymi

Zarządzanie danymi w projekcie odbywa się za pomocą plików JSON, które przechowują informacje o postaciach, przedmiotach oraz lokacjach. Wykorzystano klasę `DataPersistanceManager` do odczytu i zapisu stanu gry, co umożliwia łatwe zarządzanie stanem gry. Ponadto, dane o obiektach stałych, takich jak przeciwnicy, przedmioty, czy tablice dropów przechowywane są w formacie JSON do deserializacji poprzez użycie metod zarządzających w klasach, takich jak `PlantDropManager` czy `ItemManager`. W celu deserializacji list obiektów używających interfejsów, stworzono specjalne konwertery przy użyciu Biblioteki Newtonsoft.Json.

2.4 Diagram klas

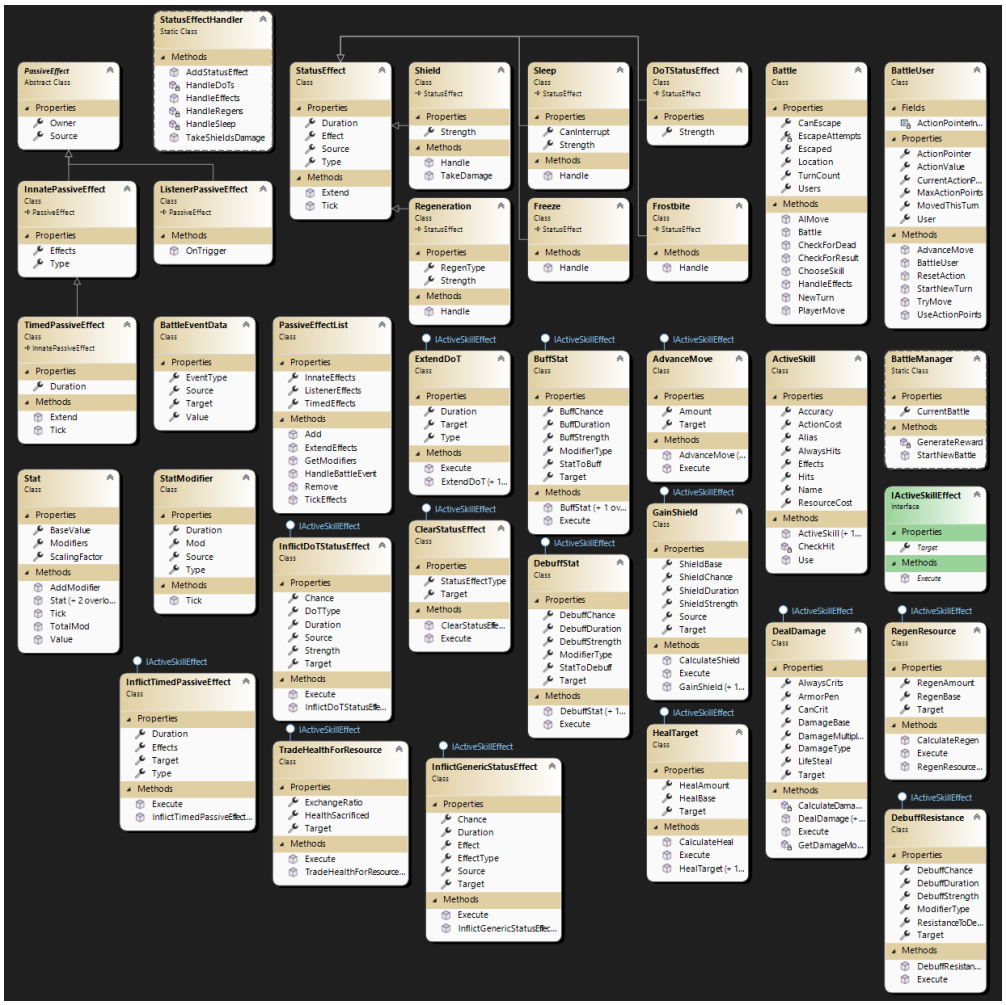
W tej sekcji przedstawiono diagram klas projektu, w celu wizualizacji struktury klas i ich pola, właściwości oraz metody. Ze względu na duży rozmiar projektu i powstałego diagramu klas, podzielono diagram na sekcje, każda odpowiadająca za osobny podfolder struktury.

- Na rysunku 2.1, przedstawiono sekcję Characters (Postacie), odpowiedzialną za klasy postaci gracza i przeciwników.



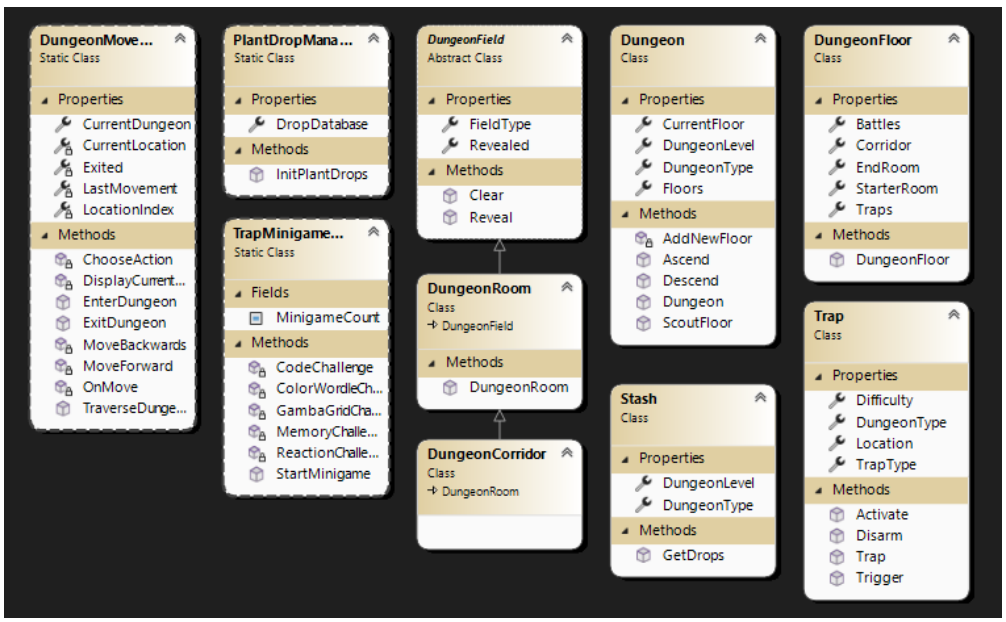
Rysunek 2.1: Diagram klas przedstawiający strukturę części ‘Characters’ projektu.

- Na rysunku 2.2, przedstawiono sekcję Combat (Walka), odpowiedzialną za klasy używane przez mechaniki systemu walki: umiejętności, statystyki, bitwy oraz efekty pasywne.



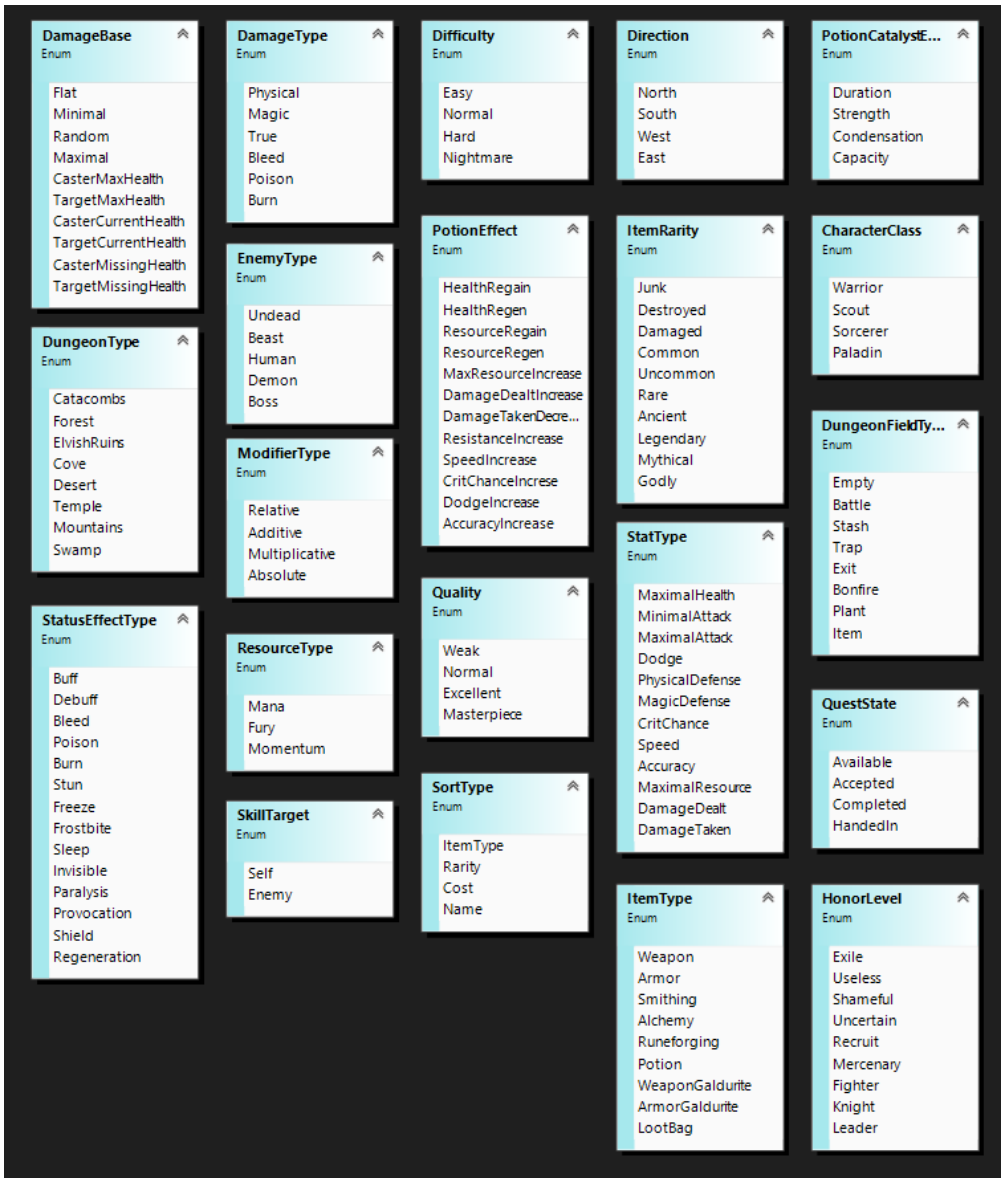
Rysunek 2.2: Diagram klas przedstawiający strukturę części ‘Combat’ projektu.

- Na rysunku 2.3, przedstawiono sekcję Dungeons (Lochy), odpowiedzialną za klasy umożliwiające eksplorację i generowanie lochów.



Rysunek 2.3: Diagram klas przedstawiający strukturę części ‘Dungeons’ projektu.

- Na rysunku 2.4, przedstawiono sekcję Enums (Enumy), odpowiedzialną za przechowywanie typów wyliczeniowych projektu.

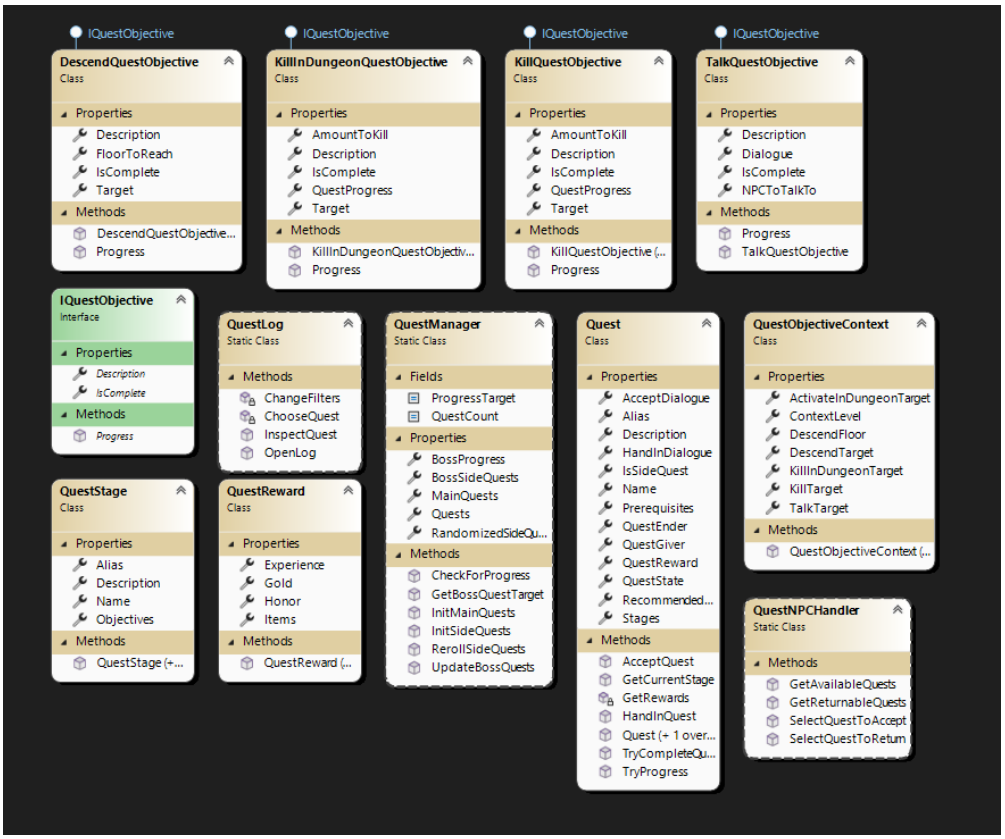


Rysunek 2.4: Diagram klas przedstawiający strukturę części ‘Enums’ projektu.

- Na rysunku 2.5, przedstawiono sekcję Items (Przedmioty), odpowiedzialną za klasy implementujące typy przedmiotów i zarządzające nimi.

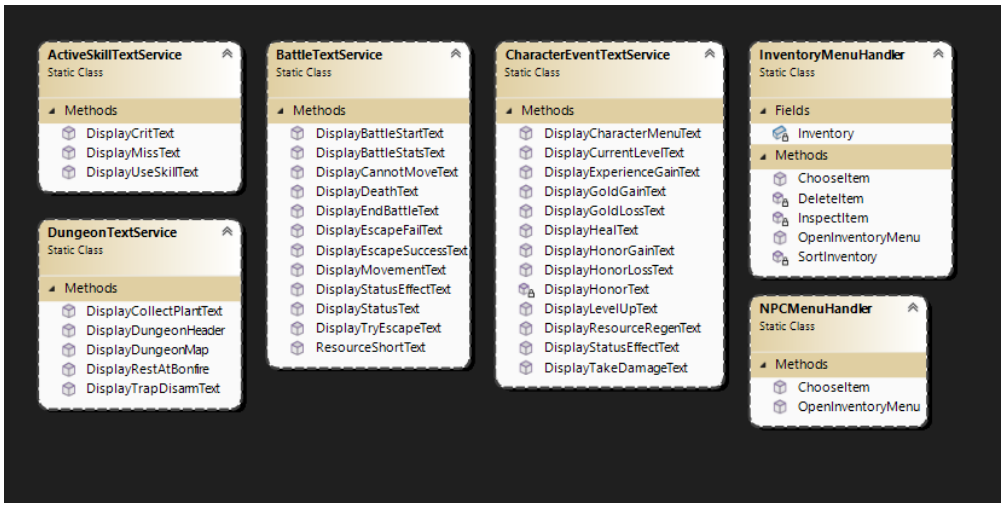
Rysunek 2.5: Diagram klas przedstawiający strukturę części ‘Items’ projektu.

- Na rysunku 2.6, przedstawiono sekcję Quests (Zadania), odpowiedzialną za klasy implementujące strukturę zadań i zarządzające nimi.



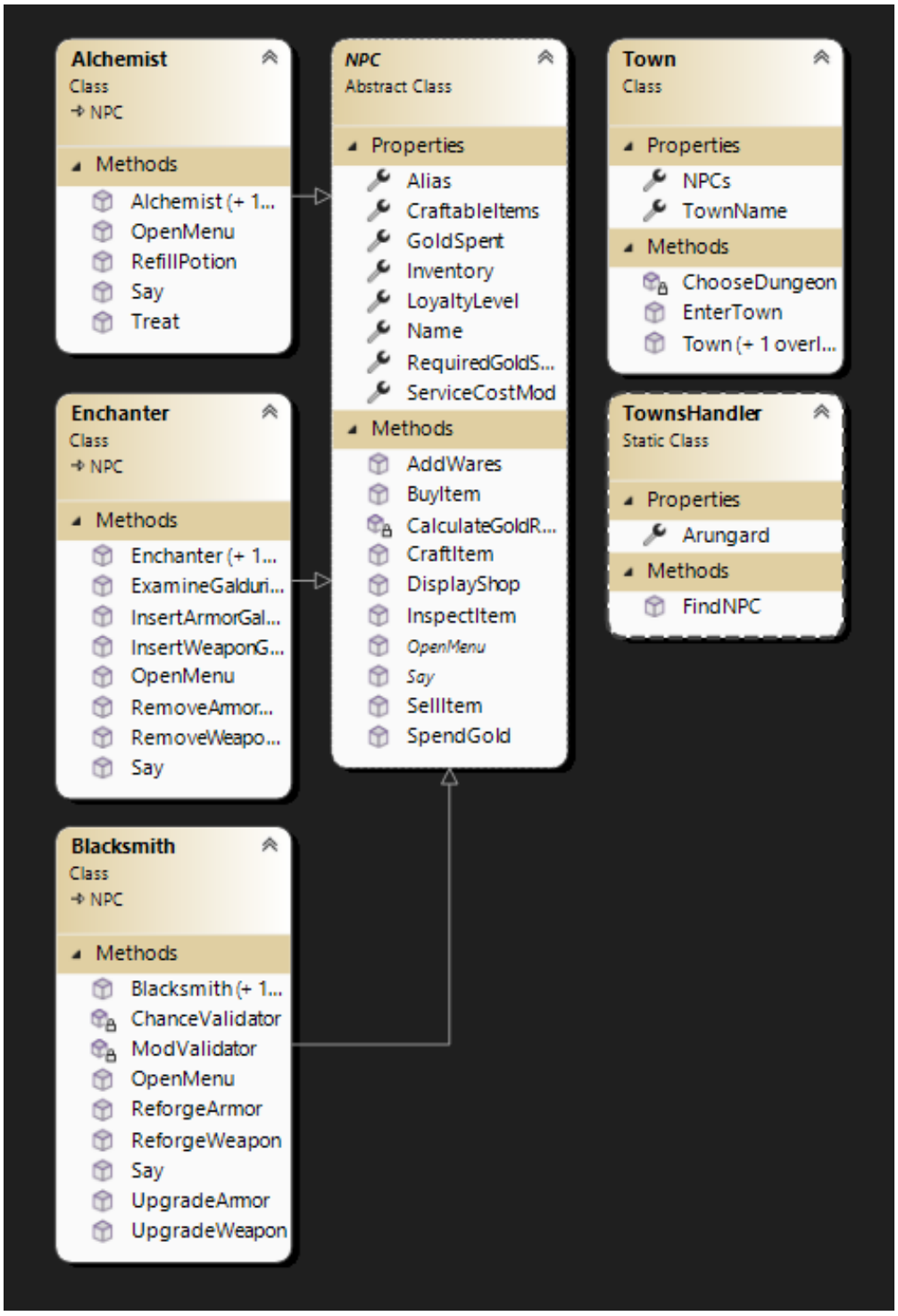
Rysunek 2.6: Diagram klas przedstawiający strukturę części ‘Quests’ projektu.

- Na rysunku 2.7, przedstawiono sekcję Text (Tekst), odpowiedzialną za klasy zarządzające tekstem, wyświetlaniem go na ekranie, oraz nawigacją po menu.



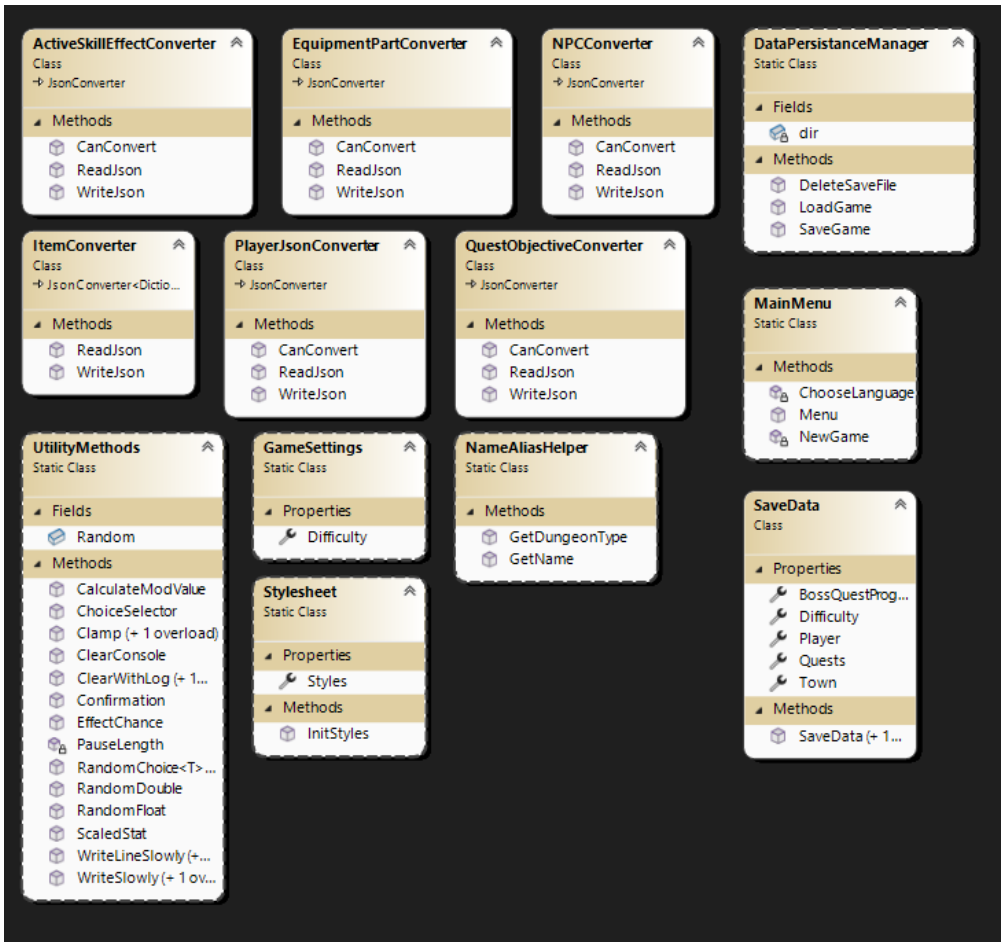
Rysunek 2.7: Diagram klas przedstawiający strukturę części ‘Text’ projektu.

- Na rysunku 2.8, przedstawiono sekcję Towns (Miasta), odpowiedzialną za klasy postaci niezależnych oraz miast.



Rysunek 2.8: Diagram klas przedstawiający strukturę części ‘Towns’ projektu.

- Na rysunku 2.9, przedstawiono sekcję Utilities (Narzędzia), odpowiedzialną za inne narzędzia pomocnicze i systemy, takie jak konwertery JSON, czy przydatne metody rozszerzające.



Rysunek 2.9: Diagram klas przedstawiający strukturę części ‘Utilities’ projektu.

2.5 Hierarchia i opisy klas

Projekt zawiera złożoną hierarchię klas, która umożliwia efektywne zarządzanie postaciami, przedmiotami oraz interakcjami w grze. Poniżej przedstawiono klasy oraz ich najważniejsze metody:

- **Character** - abstrakcyjna klasa bazowa dla wszystkich postaci w grze. Odpowiada za zarządzanie podstawowymi właściwościami postaci, takich jak zdrowie, atak czy obrona oraz na interakcje z otoczeniem. Najwazniejsze metody:
  - `TakeDamage(double damage, dynamic source)` - metoda do obliczania przyjętych obrażeń i odejmowaniu ich od zdrowia.
  - `Heal(double heal)` - metoda do leczenia postaci.
  - `UseResource(int amount)`: Używa określoną ilość zasobu (many, pędu lub wściekłości).
  - `RegenResource(int amount)`: Regeneruje określoną ilość zasobu.
  - `AddModifier(StatType stat, StatModifier modifier)`: Dodaje modyfikator do określonej statystyki postaci.
  - `HandleModifiers()` : Obsługuje modyfikatory postaci, aktualizując ich czas trwania.
- **PlayerCharacter** - klasa dziedzicząca po `Character`, reprezentująca postać gracza. Odpowiada za zarządzanie unikalnymi właściwościami i umiejętnościami gracza. Najważniejsze metody:

  - `SwitchWeapon(Weapon weapon)` - zmienia aktualnie używaną broń postaci na podaną. Jeśli postać ma już broń, dodaje ją do ekwipunku.
  - `SwitchArmor(Armor armor)` - zmienia aktualnie używaną zbroję postaci na podaną. Jeśli postać ma już zbroję, dodaje ją do ekwipunku i aktualizuje zdrowie postaci.
  - `GainGold(int gold)` - przyznaje postaci określoną ilość złota, uwzględniając modyfikatory pasywne dotyczące zdobywania złota.
  - `LoseGold(int gold)` - odejmuje określoną ilość złota od postaci.
  - `GainExperience(int experience)` - przyznaje postaci doświadczenie, co może prowadzić do awansu na wyższy poziom. Aktualizuje zdrowie postaci po awansie.
  - `GainHonor(int honor)` - przyznaje postaci honor, nie przekraczając maksymalnego limitu.
  - `LoseHonor(int honor)` - odejmuje honor od postaci, nie przekraczając minimalnego limitu.
  - `Say(string message)` - wyświetla wiadomość postaci w formacie dialogowym.

- **PlayerHandler** - statyczna klasa odpowiedzialna za zarządzanie stanem i interakcjami postaci gracza w grze. Przechowuje aktualny obiekt gracza w statycznym polu publicznym. Nie posiada żadnych metod.

- **Warrior, Rogue, Mage i Paladin** - klasy dziedziczące po `PlayerCharacter` służące do tworzenia obiektów konkretnej klasy postaci gracza. Posiadają konstruktor wypełniony wartościami bazowymi. Nie posiadają żadnych metod.

- **EnemyCharacter** - klasa dziedzicząca po `Character`, reprezentująca przeciwnika w grze. Odpowiada za zarządzanie unikalnymi właściwościami i zachowaniami przeciwników. Nie posiada żadnych metod.

- **EnemyFactory** - statyczna klasa odpowiedzialna za tworzenie instancji przeciwników w grze. Zarządza listą dostępnych przeciwników oraz logiką ich generowania. Najważniejsze metody:
  - `CreateEnemy(string alias, int level)` - metoda, która tworzy konkretnego przeciwnika na podstawie podanego aliasu oraz poziomu. Wyszukuje przeciwnika w liście i zwraca nową instancję z odpowiednim poziomem.
  - `CreateEnemy(DungeonType dungeonType, int level)` - metoda, która generuje losowego przeciwnika na podstawie typu lochu oraz poziomu. Sprawdza, czy można wygenerować bossa, a jeśli nie, wybiera losowego przeciwnika z dostępnych w danym lochu.
  - `InitializeEnemies()` - metoda inicjalizująca listę przeciwników poprzez wczytanie danych z pliku JSON. Sprawdza, czy plik istnieje, a następnie deserializuje dane do listy przeciwników.

- **BossEnemy** - klasa dziedzicząca po `EnemyCharacter`, reprezentująca bossa w grze. Odpowiada za zarządzanie unikalnymi właściwościami i zachowaniami bossów. Nie posiada żadnych metod.

- **Battle** - klasa odpowiedzialna za zarządzanie przebiegiem walki w grze. Odpowiada za interakcje między postaciami gracza a przeciwnikami, zarządzanie turami oraz wynikami walki. Najważniejsze metody:
  - `NewTurn()` - metoda, która rozpoczyna nową turę walki, resetując stany użytkowników i umożliwiając im wykonanie akcji.

  - `HandleEffects(BattleUser user)` - obsługuje efekty statusowe oraz regenerację zasobu dla podanego użytkownika.
  - `ChooseSkill(BattleUser player, BattleUser target)` - daje graczowi wybór umiejętności do użycia na wybranym celu.
  - `PlayerMove(BattleUser player, BattleUser target)` - obsługuje ruch gracza, pozwalając na wykonanie akcji, takich jak użycie umiejętności lub przedmiotu czy ucieczkę z walki.
  - `AIMove(BattleUser enemy, BattleUser target)` - obsługuje ruch przeciwnika, wybierając odpowiednie umiejętności do użycia.
  - `CheckForResult()` - sprawdza wynik walki, określając, czy gracz, czy przeciwnik wygrał, czy też może gracz uciekł.
  - `CheckForDead()` - sprawdza, które postacie są martwe, aktualizując stan walki i usuwając martwych użytkowników z walki.

- **BattleManager** - statyczna klasa odpowiedzialna za zarządzanie stanem aktualnie przebiegającej walki w grze. Zarządza aktualną walką, jej rozpoczęciem, zakończeniem oraz generowaniem nagród po zakończeniu. Najważniejsze metody:
  - `StartNewBattle(Battle battle)` - rozpoczyna nową walkę, resetując stan walki oraz regenerując zasoby użytkowników.
  - `GenerateReward(Dictionary<BattleUser, int> usersTeams)` - generuje nagrody dla gracza po zakończeniu walki, przyznając złoto, honor i doświadczenie i przedmiotów podstawie pokonanych przeciwników.

- **BattleUser** - klasa odpowiedzialna za reprezentowanie uczestnika walki walce. Zarządza stanem postaci, punktami akcji oraz interakcjami w trakcie tury. Najważniejsze metody:
  - `ResetAction()` - resetuje punkty akcji oraz wskaźnik akcji uczestnika, przygotowując go do nowej tury.
  - `TryMove()` - próbuje wykonać ruch, zmniejszając wartość akcji i sprawdzając, czy użytkownik może się poruszyć.
  - `AdvanceMove(int amount)` - zmniejsza wartość akcji o podaną ilość, co pozwala na wcześniejszy ruch w turze.
  - `UseActionPoints(double amount)` - zużywa określoną ilość punktów akcji, aktualizując ich bieżącą wartość.
  - `StartNewTurn()` - resetuje stan uczestnika na początku nowej tury, przygotowując go do działania.

- **Stat** - klasa odpowiedzialna za zarządzanie statystykami postaci w grze. Zarządza wartościami bazowymi, współczynnikami skalowania oraz modyfikatorami. Najważniejsze metody:
  - `Value(Character owner, string statName)` - metoda, która oblicza i zwraca wartość statystyki, uwzględniając modyfikatory oraz poziom postaci.
  - `Tick()` - metoda, która aktualizuje modyfikatory, usuwając te, które wygasły.
  - `AddModifier(StatModifier modifier)` - metoda, która dodaje nowy modyfikator do listy modyfikatorów statystyki.

- **StatModifier** - klasa reprezentująca modyfikator statystyk w grze. Odpowiada za przechowywanie informacji o typie modyfikatora, jego wartości, źródle oraz czasie trwania. Najważniejsze metody:
  - `Tick()` - metoda, która zmniejsza czas trwania modyfikatora o jeden i usuwa go, jeśli czas trwania osiągnie zero.

- **PassiveEffect** - klasa abstrakcyjna, która reprezentuje pasywne efekty przypisane do postaci w grze. Odpowiada za zarządzanie efektami, które wpływają na statystyki i zachowanie postaci. Nie posiada żadnych metod.

- **BattleEventData** - klasa reprezentująca dane zdarzenia w trakcie walki. Zawiera informacje o typie zdarzenia, źródle oraz celu zdarzenia. Nie posiada żadnych metod.

- **InnatePassiveEffect** - klasa dziedzicząca po `PassiveEffect`, reprezentująca pasywne efekty, które są przypisane do postaci. Efekty te są przypisane oraz działają cały czas, dopóki ich źródło nie zostanie usunięte (broń, umiejętność pasywna). Nie posiada żadnych metod.

- **ListenerPassiveEffect** - klasa reprezentująca pasywny efekt, który reaguje na zdarzenia w grze. Efekty te są przypisane cały czas, dopóki ich źródło nie zostanie usunięte, ale efekt aktywuje się dopiero po spełnieniu warunku przez zdarzenie w walce. Najważniejsze metody:

  - `OnTrigger(BattleEventData eventData)` - metoda, która sprawdza, czy warunek wyzwalający został spełniony, a następnie wykonuje zdefiniowaną akcję.

- **TimedPassiveEffect** - klasa reprezentująca pasywny efekt, który ma ograniczony czas trwania. Odpowiada za zarządzanie czasem trwania efektu oraz jego działaniem w trakcie trwania. Najważniejsze metody:

  - `Tick()` - metoda, która zmniejsza czas trwania efektu oraz wykonuje akcję, jeśli została zdefiniowana. Usuwa efekt, jeśli czas trwania osiągnie zero.
  - `Extend(int turns)` - metoda, która wydłuża czas trwania efektu o podaną liczbę tur.

- **PassiveEffectList** - klasa odpowiedzialna za zarządzanie listą pasywnych efektów przypisanych do postaci. Zarządza dodawaniem, usuwaniem oraz obsługą efektów w trakcie walki. Najważniejsze metody:

  - `Add(PassiveEffect effect)` - metoda, która dodaje nowy efekt do odpowiedniej listy (Innate, Listener, Timed) w zależności od jego typu.
  - `Remove(PassiveEffect effect)` - metoda, która usuwa efekt z odpowiedniej listy na podstawie jego typu.
  - `HandleBattleEvent(BattleEventData eventData)` - metoda, która obsługuje zdarzenia walki, wywołując odpowiednie efekty, które są aktywne.
  - `TickEffects()` - metoda, która aktualizuje efekty czasowe, zmniejszając ich czas trwania.
  - `GetModifiers(string ofType)` - metoda, która zwraca listę modyfikatorów statystyk na podstawie podanego typu efektu.

- **StatusEffect** - klasa reprezentująca efekt statusowy, który może być nałożony na postać w grze. Odpowiada za zarządzanie czasem trwania efektu oraz jego wpływem na postać. Najważniejsze metody:

  - `Tick(Character target)` - zmniejsza czas trwania efektu o jeden i usuwa go, jeśli czas trwania osiągnie zero.
  - `Extend(int turns)` - wydłuża czas trwania efektu o podaną liczbę tur.

- **StatusEffectHandler** - klasa statyczna odpowiedzialna za zarządzanie efektami statusowymi w grze. Obsługuje różne efekty, takie jak obrażenia w czasie, regeneracje oraz inne statusy, które wpływają na postacie. Najważniejsze metody:

  - `HandleEffects(List<StatusEffect> effects, Character target)` - metoda, która obsługuje efekty statusowe dla podanego celu, w tym obrażenia w czasie (DoTs), sen, regeneracje oraz wykonuje tick dla wszystkich efektów.
  - `AddStatusEffect(StatusEffect statusEffect, Character target)` - metoda, która dodaje nowy efekt statusowy do celu. Sprawdza, czy efekt już istnieje i w razie potrzeby przedłuża jego czas trwania.
  - `TakeShieldsDamage(List<Shield> shields, Character target, double damage)` - metoda, która obsługuje obrażenia zadawane tarczom, zmniejszając obrażenia, które docierają do celu o sumę tarcz.
  - `HandleDoTs(List<DoTStatusEffect> dots, Character target)` - metoda, która obsługuje efekty obrażeń w czasie, zadając obrażenia celowi na podstawie zgromadzonych efektów.
  - `HandleSleep(List<Sleep> sleeps, Character target)` - metoda, która obsługuje efekty snu, regenerując zdrowie celu, jeśli regeneracja jest dodatnia.
  - `HandleRegens(List<Regeneration> regens, Character target)` - metoda, która obsługuje efekty regeneracji, przywracając zdrowie lub zasób celowi na podstawie zgromadzonych efektów regeneracji.

- **DoTStatusEffect** - klasa reprezentująca efekty obrażeń w czasie (DoT) w grze. Nie posiada żadnych metod.

- **Freeze** - klasa reprezentująca efekt zamrożenia w grze. Najważniejsze metody:
  - `Handle(Character target)` - metoda, która obsługuje efekt zamrożenia na celu, zmniejszając jego prędkość.

- **Frostbite** - klasa reprezentująca efekt odmrożenia w grze. Najważniejsze metody:
  - `Handle(Character target)` - metoda, która obsługuje efekt odmrożenia na celu, dodając efekt odmrożenia.

- **Regeneration** - klasa reprezentująca efekt regeneracji w grze. Odpowiada za przywracanie zdrowia lub zasobów w czasie. Najważniejsze metody:
  - `Handle(Character target)` - metoda, która obsługuje efekt regeneracji na celu, przywracając zdrowie lub zasoby.

- **Shield** - klasa reprezentująca efekt tarczy w grze. Odpowiada za absorbowanie obrażeń przez postać zamiast zdrowia. Najważniejsze metody:
  - `TakeDamage(double amount)` - metoda, która obsługuje przyjmowanie obrażeń przez tarczę, zmniejszając jej siłę.

- **Sleep** - klasa reprezentująca efekt uśpienia w grze. Najważniejsze metody:
  - `Handle(Character target)` - metoda, która obsługuje efekt uśpienia na celu.

- **ActiveSkill** - klasa reprezentująca aktywną umiejętność, którą postać może wykorzystać w trakcie walki. Odpowiada za zarządzanie właściwościami umiejętności oraz jej wykonaniem. Najważniejsze metody:
  - `Use(BattleUser caster, BattleUser enemy)` - metoda, która wykonuje umiejętność na przeciwniku, sprawdzając koszty zasobów i punkty akcji, a następnie stosując efekty umiejętności.

  - `CheckHit(Character caster, Character target)` - metoda, która oblicza szansę trafienia umiejętności w oparciu o dokładność i unik celu, zwracając informację czy cel został trafiony oraz szansę na trafienie.

- **IActiveSkillEffect** - interfejs definiujący efekty aktywnych umiejętności w grze. Odpowiada za wykonanie efektów umiejętności na postaciach. Najważniejsze metody:
  - `void Execute(Character caster, Character enemy, string source)` - metoda, która wykonuje efekt umiejętności na podanych postaciach. Przyjmuje jako argumenty postać rzucającą umiejętność, postać będącą celem oraz źródło umiejętności.

- **AdvanceMove** - klasa reprezentująca efekt umiejętności, który pozwala na przyspieszenie ruchu postaci.

- **BuffStat** - klasa reprezentująca efekt umiejętności, który zwiększa określony statystyki postaci.

- **ClearStatusEffect** - klasa reprezentująca efekt umiejętności, który usuwa określony typ efektów statusu z postaci.

- **DealDamage** - klasa reprezentująca efekt umiejętności, który zadaje obrażenia postaci.

- **DebuffResistance** - klasa reprezentująca efekt umiejętności, który zmniejsza określoną odporność postaci.

- **DebuffStat** - klasa reprezentująca efekt umiejętności, który zmniejsza określone statystyki postaci.

- **ExtendDoT** - klasa reprezentująca efekt umiejętności, który wydłuża czas trwania efektu DoT.

- **GainShield** - klasa reprezentująca efekt umiejętności, który przyznaje tarczę postaci.

- **HealTarget** - klasa reprezentująca efekt umiejętności, który leczy postać.

- **InflictDoTStatusEffect** - klasa reprezentująca efekt umiejętności, który nakłada efekt statusu zadający obrażenia w czasie (DoT).

- **InflictGenericStatusEffect** - klasa reprezentująca efekt umiejętności, który nakłada ogólny efekt statusu.

- **InflictTimedPassiveEffect** - klasa reprezentująca efekt umiejętności, który nakłada pasywny efekt czasowy.

- **RegenResource** - klasa reprezentująca efekt umiejętności, który regeneruje zasoby postaci.

- **TradeHealthForResource** - klasa reprezentująca efekt umiejętności, który wymienia zdrowie na zasoby.

- **PlantDropManager** - statyczna klasa odpowiedzialna za zarządzanie losowymi wystąpieniami roślin w grze. Zarządza bazą wystąpieniami łupów roślin w zależności od typu lochu. Najważniejsze metody:
  - `DropDatabase` - właściwość, która przechowuje słownik z bazą danych wystąpień, gdzie kluczem jest typ lochu, a wartością jest pula wystąpień.
  - `InitPlantDrops()` - metoda inicjalizująca bazę danych wystąpień roślin poprzez wczytanie danych z pliku JSON. Sprawdza, czy plik istnieje, a następnie deserializuje dane do słownika.

- **Stash** - klasa reprezentująca skrzynie w grze, które można znaleźć losowo w lochach. Najważniejsze metody:

  - `GetDrops()` - metoda, która zwraca losowo wygenerowany słownik z przedmiotami znajdującymi się w skrzynii.

- **Trap** - klasa reprezentująca pułapkę w grze, odpowiedzialna za aktywację i zarządzanie efektami pułapek. Najważniejsze metody:
  - `Activate()` - metoda, która aktywuje pułapkę, uruchamiając odpowiednią mini-grę w zależności od poziomu trudności i typu pułapki.
  - `Disarm()` - metoda, która dezaktywuje pułapkę, czyszcząc jej lokalizację.
  - `Trigger()` - metoda, która wywołuje efekty pułapki, zadając obrażenia graczowi i wywołując efekty w zależności od typu lochu.

- **TrapMinigameManager** - statyczna klasa odpowiedzialna za zarządzanie mini-grami związanymi z pułapkami w grze. Najważniejsze metody:
  - `StartMinigame(Difficulty difficulty, int TrapType)` - metoda, która rozpoczyna odpowiednią mini-grę w zależności od poziomu trudności i typu pułapki.
  - `CodeChallenge(Difficulty difficulty)` - metoda, która obsługuje mini-grę polegającą na wprowadzeniu poprawnego kodu w określonej ilości prób.
  - `MemoryChallenge(Difficulty difficulty)` - metoda, która obsługuje mini-grę polegającą na zapamiętaniu sekwencji znaków.
  - `ReactionChallenge(Difficulty difficulty)` - metoda, która obsługuje mini-grę polegającą na szybkim naciskaniu odpowiednich klawiszy w odpowiednim czasie.
  - `ColorWordleChallenge(Difficulty difficulty)` - metoda, która obsługuje mini-grę polegającą na odgadnięciu zakodowanego kolorami słowa.
  - `GambaGridChallenge(Difficulty difficulty)` - metoda, która obsługuje mini-grę polegającą na unikaniu ukrytych bomb w siatce.

- **Dungeon** - klasa odpowiedzialna za zarządzanie lochami w grze. Zarządza poziomami lochu oraz generowaniem nowych pokoi i korytarzy. Najważniejsze metody:
  - `AddNewFloor()` - metoda, która dodaje nowy poziom do lochu, generując odpowiednią długość korytarzy i pokoi.
  - `Ascend()` - metoda, która przenosi gracza na poprzedni poziom lochu, jeśli jest dostępny.
  - `Descend()` - metoda, która przenosi gracza na następny poziom lochu, dodając nowy poziom, jeśli jest to konieczne.
  - `ScoutFloor(DungeonFloor floor)` - metoda, która odkrywa wszystkie pokoje i korytarze na podanym poziomie lochu.

- **DungeonField** - klasa abstrakcyjna reprezentująca pole w lochu. Zarządza typem pola oraz jego stanem. Najważniejsze metody:
  - `Reveal()` - metoda, która odkrywa pole, zmieniając jego stan na ujawnione.
  - `Clear()` - metoda, która czyści pole, ustawiając jego typ na pusty.

- **DungeonRoom** - klasa reprezentująca pokoje w lochu. Dziedziczy po klasie `DungeonField`. Nie posiada żandych metod.

- **DungeonCorridor** - klasa reprezentująca korytarze w lochu. Dziedziczy po klasie `DungeonRoom`. Nie posiada żadnych metod.

- **DungeonFloor** - klasa reprezentująca poziom lochu. Zarządza pokojami, korytarzami oraz pułapkami na danym poziomie. Nie posiada żadnych metod.

- **DungeonMovementManager** - statyczna klasa odpowiedzialna za zarządzanie ruchem gracza w lochu. Zarządza aktualną lokalizacją gracza oraz interakcjami w lochu. Najważniejsze metody:
  - `EnterDungeon(Dungeon dungeon)` - metoda, która wprowadza gracza do lochu, ustawiając jego początkową lokalizację.
  - `TraverseDungeon()` - metoda, która obsługuje ruch gracza w lochu, umożliwiając różne akcje.
  - `ExitDungeon()` - metoda, która pozwala graczowi opuścić loch.
  - `MoveForward()` - metoda, która przesuwa gracza do przodu w lochu.
  - `MoveBackwards()` - metoda, która przesuwa gracza do tyłu w lochu.
- **SkillTarget** - typ wyliczający definiujący cele umiejętności w grze. Określa, na kogo umiejętność może być użyta. Możliwe wartości:
  - `Self` - umiejętność skierowana na siebie.
  - `Enemy` - umiejętność skierowana na przeciwnika.
- **CharacterClass** - typ wyliczający definiujący klasy postaci w grze. Określa, jaką rolę pełni postać. Możliwe wartości:
  - `Warrior` - wojownik, specjalizujący się w walce wręcz.
  - `Scout` - zwiadowca, specjalizujący się w szybkości i zwinności.
  - `Sorcerer` - czarodziej, specjalizujący się w magii.
  - `Paladin` - paladyn, łączący umiejętności walki i magii.
- **DamageBase** - typ wyliczający definiujący różne podstawy obrażeń w grze. Określa, jak obrażenia są obliczane. Możliwe wartości:
  - `Flat` - stałe obrażenia.
  - `Minimal` - minimalne obrażenia.
  - `Random` - losowe obrażenia (pomiędzy minimalnymi a maksymalnymi).
  - `Maximal` - maksymalne obrażenia.
  - `CasterMaxHealth` - obrażenia oparte na maksymalnym zdrowiu rzucającego.
  - `TargetMaxHealth` - obrażenia oparte na maksymalnym zdrowiu celu.
  - `CasterCurrentHealth` - obrażenia oparte na bieżącym zdrowiu rzucającego.
  - `TargetCurrentHealth` - obrażenia oparte na bieżącym zdrowiu celu.
  - `CasterMissingHealth` - obrażenia oparte na brakującym zdrowiu rzucającego.
  - `TargetMissingHealth` - obrażenia oparte na brakującym zdrowiu celu.
- **DamageType** - typ wyliczający definiujący różne typy obrażeń w grze. Określa, jakiego rodzaju obrażenia są zadawane. Możliwe wartości:
  - `Physical` - obrażenia fizyczne.
  - `Magic` - obrażenia magiczne.
  - `True` - obrażenia nieuchronne, ignorujące odporności i obronę.
  - `Bleed` - obrażenia od krwawienia.
  - `Poison` - obrażenia od trucizny.
  - `Burn` - obrażenia od ognia.

- **Difficulty** - typ wyliczający definiujący poziomy trudności w grze. Określa trudności mechanik, takich jak walka, czy zagadki w pułapkach. Możliwe wartości:
  - `Easy` - łatwy poziom trudności.
  - `Normal` - normalny poziom trudności.
  - `Hard` - trudny poziom trudności.
  - `Nightmare` - bardzo trudny poziom trudności.
- **EnemyType** - typ wyliczający definiujący typy przeciwników w grze. Określa, z jakim rodzajem przeciwnika ma się do czynienia. Możliwe wartości:
  - `Undead` - nieumarły.
  - `Beast` - bestia.
  - `Human` - człowiek.
  - `Demon` - demon.
  - `Boss` - boss.
- **HonorLevel** - typ wyliczający definiujący poziomy honoru w grze. Możliwe wartości:
  - `Exile` - wygnaniec - poziom najniższy.
  - `Useless` - bezużyteczny.
  - `Shameful` - haniebny.
  - `Uncertain` - niepewny.
  - `Recruit` - rekrut - poziom bazowy.
  - `Mercenary` - najemnik.
  - `Fighter` - wojownik.
  - `Knight` - rycerz.
  - `Leader` - przywódca - poziom najwyższy.
- **QuestState** - typ wyliczający definiujący stany zadań w grze. Określa, na jakim etapie znajduje się zadanie. Możliwe wartości:
  - `Available` - zadanie dostępne do akceptacji.
  - `Accepted` - zadanie przyjęte.
  - `Completed` - zadanie ukończone.
  - `HandedIn` - zadanie oddane.
- **ResourceType** - typ wyliczający definiujący typy zasobów w grze. Określa jakiego zasobu używa dana postać w grze. Możliwe wartości:
  - `Mana` - mana.
  - `Fury` - wściekłość.
  - `Momentum` - pęd.
- **StatusEffectType** - typ wyliczający definiujący różne typy efektów statuu, które mogą wpływać na postacie w grze. Możliwe wartości:
  - `Buff` - daje niestandardowy pozytywny efekt postaci.
  - `Debuff` - daje niestandardowy negatywny efekt postaci.

  - `Bleed` - powoduje, że postać otrzymuje obrażenia w czasie (DoT) od krwawienia.
  - `Poison` - powoduje, że postać otrzymuje obrażenia w czasie (DoT) od zatrucia.
  - `Burn` - powoduje, że postać otrzymuje obrażenia w czasie (DoT) od ognia.
  - `Stun` - postać nie może podejmować akcji.
  - `Freeze` - postać nie może podejmować akcji i jest spowolniony po zakończeniu efektu.
  - `Frostbite` - postać może wykonywać tylko jedną akcję na turę.
  - `Sleep` - postać jest ogłuszony, ale regeneruje zdrowie w czasie.
  - `Invisible` - postać nie może być celem umiejętności.
  - `Paralysis` - postać nie może działać i nie może być celem umiejętności.
  - `Provocation` - cel umiejętności postaci musi być osobą, która spowodowała prowokację.
  - `Shield` - blokuje określoną ilość obrażeń zamiast zdrowia.
  - `Regeneration` - regeneruje określoną ilość zdrowia/zasobu na turę.
- **ModifierType** - typ wyliczający definiujący różne typy modyfikatorów, które mogą być stosowane do statystyk w grze. Możliwe wartości:
  - `Relative` - modyfikator aplikowany przez mnożenie, w pierwszej kolejności.
  - `Additive` - modyfikator apllikowany przez dodawania, w drugiej kolejności.
  - `Multiplicative` - modyfikator aplikowany przez mnożenie, w trzeciej kolejności.
  - `Absolute` - modyfikator aplikowany przez dodawanie, w czwartej kolejności.
- **StatType** - typ wyliczający definiujący różne typy statystyk, które występują w grze. Możliwe wartości:
  - `MaximalHealth` - maksymalne zdrowie postaci.
  - `MinimalAttack` - minimalne obrażenia zadawane przez postać.
  - `MaximalAttack` - maksymalne obrażenia zadawane przez postać.
  - `Dodge` - determinuje szansę na unikanie ataków.
  - `PhysicalDefense` - redukcja obrażeń fizycznych przyjmowanych przez postać.
  - `MagicDefense` - redukcja obrażeń magicznych przyjmowanych przez postać.
  - `CritChance` - szansa na krytyczne trafienie.
  - `Speed` - szybkość postaci, wyznacza kolejność i częstość ruchów.
  - `Accuracy` - celność ataków postaci.
  - `MaximalResource` - maksymalna ilość zasobu postaci.
  - `DamageDealt` - obrażenia zadawane przez postać.
  - `DamageTaken` - obrażenia otrzymywane przez postać.
- **SortType** - typ wyliczający definiujący różne typy sortowania przedmiotów w grze. Możliwe wartości:
  - `ItemType` - sortowanie według typu przedmiotu.
  - `Rarity` - sortowanie według unikalności przedmiotu.
  - `Cost` - sortowanie według kosztu przedmiotu, malejąco.
  - `Name` - sortowanie według nazwy przedmiotu, alfabetycznie.

- **ItemRarity** - typ wyliczający definiujący różne poziomy unikalności przedmiotów w grze. Możliwe wartości:
  - `Junk` - przedmioty określane w grze jako ‘śmiecie’.
  - `Destroyed` - przedmioty zniszczone.
  - `Damaged` - przedmioty uszkodzone.
  - `Common` - przedmioty pospolite.
  - `Uncommon` - przedmioty niepospolite.
  - `Rare` - przedmioty rzadkie.
  - `Ancient` - przedmioty starożytne.
  - `Legendary` - przedmioty legendarne.
  - `Mythical` - przedmioty mityczne.
  - `Godly` - przedmioty boskie.
- **ItemType** - typ wyliczający definiujący różne typy przedmiotów w grze. Możliwe wartości:
  - `Weapon` - broń.
  - `Armor` - zbroja.
  - `Smithing` - przedmioty do kowalstwa.
  - `Alchemy` - przedmioty do alchemii.
  - `Runeforging` - przedmioty do runotwórstwa.
  - `Potion` - mikstury.
  - `WeaponGaldurite` - galduryty do broni.
  - `ArmorGaldurite` - galduryty do zbroi.
  - `LootBag` - torby z łupem.
- **PotionCatalystEffect** - typ wyliczający definiujący różne efekty katalizatorów mikstur w grze. Możliwe wartości:
  - `Duration` - efekt wpływający na czas trwania mikstury.
  - `Strength` - efekt zwiększający moc mikstury.
  - `Condensation` - efekt kondensacji efektu regeneracji mikstury (jeśli go posiada).
  - `Capacity` - efekt zwiększający maksymalną ilość użyć mikstury.
- **PotionEffect** - typ wyliczający definiujący różne efekty mikstur w grze. Możliwe wartości:
  - `HealthRegain` - przywracanie zdrowia.
  - `HealthRegen` - regeneracja zdrowia.
  - `ResourceRegain` - przywracanie zasobu.
  - `ResourceRegen` - regeneracja zasobu.
  - `MaxResourceIncrease` - zwiększenie maksymalnej ilości zasobu
  - `DamageDealtIncrease` - zwiększenie zadawanych obrażeń.
  - `DamageTakenDecrease` - zmniejszenie otrzymywanych obrażeń.
  - `ResistanceIncrease` - zwiększenie odporności.
  - `SpeedIncrease` - zwiększenie szybkości.

  - `CritChanceIncrease` - zwiększenie szansy na krytyczne trafienie.
  - `DodgeIncrease` - zwiększenie uniku.
  - `AccuracyIncrease` - zwiększenie celności.

- **Quality** - typ wyliczający definiujący różne poziomy jakości wyposażenia w grze. Możliwe wartości:
  - `Weak` - słaba jakość (wymagany poziom i statystyki niższe o 3 poziomy).
  - `Normal` - normalna jakość (standardowy wymagany poziom i statystyki)
  - `Excellent` - doskonała jakość(wymagany poziom i statystyki wyższe o 3 poziomy)
  - `Masterpiece` - arcydzieło (wymagany poziom i statystyki wyższe o 5 poziomów).

- **DungeonFieldType** - typ wyliczający reprezentujący różne typy pól w lochach. Umożliwia określenie, jakie interakcje mogą mieć miejsce na danym polu. Możliwe wartości:
  - `Empty` - puste pole, bez interakcji.
  - `Battle` - pole, na którym odbywa się walka.
  - `Stash` - pole, które zawiera skrzynię z przedmiotami.
  - `Trap` - pole z pułapką, które może zadać obrażenia lub wywołać negatywne efekty jeśli nie zostanie rozbrojona.
  - `Bonfire` - pole, które umożliwia regenerację zdrowia, z ryzykiem zasadzki bez możliwości ucieczki.
  - `Plant` - pole, które zawiera rośliny, które można zebrać.

- **DungeonType** - enum reprezentujący różne typy lochów w grze. Określa, jakie środowisko i przeciwnicy mogą występować w danym lochu. Możliwe wartości:
  - `Catacombs` - lochy o tematyce katakumb z nieumarłymi.
  - `Forest` - lochy w leśnym otoczeniu z bandytami, elfami i bestiami leśnymi.
  - `ElvishRuins` - lochy w ruinach elfickich, zmienionych pod wpływem demonów.
  - `Cove` - lochy w nadmorskim otoczeniu z wodnymi przeciwnikami i piratami.
  - `Desert` - lochy w pustynnym otoczeniu z pustynnymi bandytami i potworami.
  - `Temple` - lochy w świątyniach, z wyznawcami boga Mista.
  - `Mountains` - lochy w górzystym terenie, z górskimi przeciwnikami.
  - `Swamp` - lochy w bagnistym otoczeniu, z żyjącymi roślinami i niebezpiecznymi bestiami.

- **IItem** - interfejs definiujący podstawowe właściwości przedmiotu w grze. Zawiera metody i właściwości, które muszą być zaimplementowane przez wszystkie przedmioty. Najważniejsze metody:
  - `Inspect(int amount = 1)` - metoda, która wyświetla szczegóły przedmiotu.
  - `WriteName()` - metoda, która wyświetla nazwę przedmiotu w odpowiednim stylu.

- **BaseItem** - klasa abstrakcyjna reprezentująca podstawowy przedmiot w grze. Definiuje wspólne właściwości i metody dla wszystkich przedmiotów. Najważniejsze metody:
  - `Inspect(int amount = 1)` - wyświetla szczegóły przedmiotu, w tym jego nazwę, rzadkość, typ i koszt.
  - `WriteName()` - wyświetla nazwę przedmiotu w odpowiednim stylu.

  - `NameStyle()` - zwraca styl wyświetlania nazwy przedmiotu w zależności od jego rzadkości.
  - `WriteItemType()` - wyświetla typ przedmiotu.
  - `WriteRarity()` - wyświetla rzadkość przedmiotu.

- **BaseIngredient** - klasa reprezentująca podstawowy składnik w grze. Dziedziczy po klasie `BaseItem` i definiuje właściwości składników. Nie posiada żadnych metod.

- **ICraftable** - interfejs definiujący właściwości przedmiotów, które można wykonać. Zawiera właściwości związane z tworzeniem przedmiotów. Najważniejsze metody:
  - `CraftingRecipe` - właściwość, która zwraca recepturę na wykonanie przedmiotu.
  - `CraftedAmount` - właściwość, która zwraca ilość przedmiotów, które otrzymuje się po jednokrotnym wytworzeniu.

- **CraftableIngredient** - klasa reprezentująca składnik, który można wykonać w grze. Dziedziczy po klasie `BaseItem` i implementuje interfejs `ICraftable`. Nie posiada żadnych metod.

- **Inventory** - klasa odpowiedzialna za zarządzanie ekwipunkiem gracza, przechowującym przedmioty. Zarządza przedmiotami, ich ilościami oraz operacjami na ekwipunku. Najważniejsze metody:
  - `AddItem(IItem item, int quantity = 1)` - dodaje przedmiot do ekwipunku, sprawdzając maksymalną wagę.
  - `TryRemoveItem(IItem item, int amount = 1)` - próbuje usunąć przedmiot z ekwipunku, zwracając informację o sukcesie.
  - `SortInventory(SortType sortType)` - sortuje ekwipunek według podanego kryterium.
  - `UseItem(IItem item)` - używa przedmiotu z ekwipunku, jeśli to możliwe.

- **NPCInventory** - klasa odpowiedzialna za zarządzanie ekwipunkiem NPC. Zarządza przedmiotami, które NPC mogą sprzedawać oraz tymi, które kupili od gracza. Najważniejsze metody:
  - `UpdateWares(int loyaltyLevel)` - aktualizuje dostępne przedmioty w sklepie NPC.
  - `AddItem(IItem item, int quantity = 1)` - dodaje przedmiot do ekwipunku NPC, zwiększając jego ilość, jeśli jest to przedmiot stakowalny.
  - `RemoveAt(int index, int amount = 1)` - usuwa przedmiot z ekwipunku NPC na podstawie indeksu.
  - `ElementAt(int index)` - zwraca przedmiot znajdujący się na podanym indeksie w ekwipunku NPC.

- **ItemManager** - statyczna klasa odpowiedzialna za zarządzanie przedmiotami w grze. Zarządza inicjalizacją i dostępem do przedmiotów. Najważniejsze metody:
  - `InitItems()` - inicjalizuje przedmioty, wczytując je z plików JSON.
  - `GetItem(string alias)` - zwraca przedmiot na podstawie podanego aliasu.

- **CraftingManager** - statyczna klasa odpowiedzialna za zarządzanie procesem tworzenia przedmiiotów w grze. Zarządza interfejsem użytkownika do tworzenia przedmiotów. Najważniejsze metody:
  - `OpenCraftingMenu(List<ICraftable> possibleItems)` - otwiera menu tworzenia, pokazując listę przedmiotów możliwych do stworzenia.

  - `ChooseItem(List<ICraftable> possibleItems)` - pozwala graczowi wybrać przedmiot do tworzenia z listy dostępnych przedmiotów.
  - `CraftItem(ICraftable item)` - wykonuje przedmiot na podstawie wybranej receptury.

- **IUsable** - interfejs definiujący właściwości przedmiotów, które można używać. Najważniejsze metody:
  - `Use()` - metoda, która wykonuje akcję użycia przedmiotu, zwracając czy podany przedmiot został zużyty.

- **Potion** - klasa reprezentująca miksturę w grze. Zarządza jej właściwościami oraz efektami. Najważniejsze metody:
  - `Use()` - używa miksturę, zmniejszając liczbę dostępnych użyć i przetwarzając jej efekty.
  - `Refill(int amount)` - uzupełnia użycia mikstury o określoną wartość.
  - `Inspect(int amount = 1)` - wyświetla szczegóły mikstury, w tym liczbę użyć oraz efekty jej efekty.

- **PotionComponent** - klasa reprezentująca komponent mikstury. Zawiera jeden efekt i jego właściwości. Najważniejsze metody:
  - `EffectDescription(PotionCatalyst? catalyst, bool showMaterial, int amount = 1)` - zwraca opis efektu komponentu, uwzględniając katalizator, jeśli istnieje, oraz pokazując użyty materiał, jeśli to wymagane.

- **PotionCatalyst** - klasa reprezentująca katalizator mikstury. Zawiera jego efekt i właściwości tego efektu. Najważniejsze metody:
  - `DescriptionText()` - zwraca opis katalizatora, w tym jego efekt oraz materiał.

- **PotionManager** - statyczna klasa odpowiedzialna za przechowywanie i zarządzanie komponentami mikstur oraz ich tworzeniem i wyborem. Zarządza inicjalizacją komponentów oraz przetwarzaniem efektów mikstur. Najważniejsze metody:
  - `InitComponents()` - inicjalizuje możliwe komponenty mikstur, wczytując dane z pliku JSON. W przypadku braku pliku zgłasza wyjątek.
  - `ProcessComponent(PotionComponent component, ...)` - przetwarza dany komponent mikstury, wykonując jego efekt.
  - `GetRandomPotion(int tier)` - generuje losową miksturę na podstawie podanego poziomu, wybierając trzy losowe komponenty.
  - `GetCatalystMaterial(PotionCatalystEffect effect, int tier)` - zwraca materiał katalizatora na podstawie podanego efektu i poziomu.
  - `GetCatalystStrength(PotionCatalystEffect effect, int tier)` - zwraca siłę katalizatora na podstawie podanego efektu i poziomu.
  - `ChoosePotion(List<Potion> potions, bool listMaterials)` - umożliwia wybór mikstury z listy, z opcjonalnym wyświetleniem materiałów.

- **WetTowel** - klasa reprezentująca mokrą szmatkę w grze. Odpowiada za zarządzanie właściwościami i działaniem. Najważniejsze metody:
  - `Use()` - metoda, która usuwa efekt podpalenia z postaci gracza, gdy szmatka jest użyta.

- **Antidote** - klasa reprezentująca antidotum w grze. Odpowiada za zarządzanie właściwościami i działaniem antidotum. Najważniejsze metody:
  - `Use()` - metoda, która usuwa efekt zatrucia z postaci gracza, gdy antidotum jest użyte.
- **Bandage** - klasa reprezentująca bandaż w grze. Odpowiada za zarządzanie właściwościami i działaniem bandaża. Najważniejsze metody:
  - `Use()` - metoda, która usuwa efekt krwawienia z postaci gracza, gdy bandaż jest użyty.
- **TownPortalScroll** - klasa reprezentująca zwoje portalu do miasta w grze. Odpowiada za zarządzanie właściwościami i działaniem zwoju. Najważniejsze metody:
  - `Use()` - metoda, która teleportuje gracza do ostatniego miasta, gdy zwój jest użyty (opuszcza aktualny loch).
- **Lootbag** - klasa reprezentująca torbę z łupem w grze. Odpowiada za zarządzanie unikalnymi właściwościami torby oraz jej użyciem. Najważniejsze metody:
  - `Use()` - metoda, która otwiera torbę z łupem, dodając przedmioty z tabeli łupów do ekwipunku gracza. Umożliwia wybór liczby torb do otwarcia oraz walidację tej liczby.
- **LootbagManager** - statyczna klasa odpowiedzialna za zarządzanie przedmiotami wypadający z torba z łupem w grze. Posiada bazę danych wczytaną z pliku JSON w postaci słownika. Najważniejsze metody:
  - `InitItems()` - metoda inicjalizująca słownik z torbami z łupem poprzez wczytanie danych z pliku JSON. Sprawdza, czy plik istnieje, a następnie deserializuje dane do słownika.
  - `GetLootbag(string alias, int level)` - metoda, która tworzy nową instancję torby z łupem na podstawie podanego aliasu oraz poziomu. Wyszukuje odpowiednią tabelę łupów i zwraca nową instancję torby.
  - `GetSupplyBag(DungeonType dungeonType, int level)` - metoda, która generuje torbę z zasobami na podstawie typu lochu oraz poziomu. Zwraca nową instancję torby z zasobami z odpowiednią tabelą łupów.
- **Galdurite** - klasa reprezentująca galduryt, który może być używany jako wzmocnienie wyposażenia w grze. Zarządza właściwościami galdurytu oraz jego komponentami. Najważniejsze metody:
  - `Reveal()` - metoda, która ujawnia właściwości galdurytu, zmieniając jego stan na ujawniony. Umożliwia to graczowi zobaczenie efektów galdurytu.
  - `Inspect(int amount = 1)` - metoda, która wyświetla szczegóły galdurytu, w tym jego efekty oraz właściwości. Umożliwia to graczowi zapoznanie się z galduritem przed włożeniem go do wyposażenia.
  - `ShowEffects()` - metoda, która wyświetla efekty galdurytu, jeśli jest on ujawniony. Umożliwia to graczowi zrozumienie, jakie korzyści przynosi galduryt.
- **GalduriteComponent** - klasa reprezentująca komponent galdurytu, który definiuje jeden efekt i jego własciwości. Zarządza informacjami o kolorze, poziomie efektu oraz typie efektu. Nie posiada żadnych metod.
- **GalduriteManager** - statyczna klasa odpowiedzialna za zarządzanie galdurytami. Zawiera bazę danych z możliwymi efektami wczytaną z pliku JSON. Najważniejsze metody:
  - `InitComponents()` - metoda inicjalizująca listę możliwych komponentów poprzez wczytanie danych z pliku JSON. Sprawdza, czy plik istnieje, a następnie deserializuje dane do listy.

  - `GetGalduriteComponent(string tier, string color, bool type, ...)` - metoda, która zwraca losowy komponent galdurytu na podstawie podanego poziomu, koloru oraz typu. Umożliwia to dynamiczne generowanie komponentów w zależności od wymagań.
  - `GetColorMaterial(int tier, string color)` - metoda, która zwraca materiał galdurytu na podstawie poziomu i koloru. Umożliwia to uzyskanie odpowiednich materiałów do tworzenia galdurytu.
  - `ChooseGaldurite(List<Galdurite> galdurites)` - metoda, która umożliwia wybór galdurytu z listy dostępnych galdurytów. Zwraca wybrany galduryt lub null, jeśli lista jest pusta.

- **ItemDrop** - klasa zawierająca informacje o łupie w postaci przedmiotu - możliwej ilości, wadze (szansie na wypadnięcie) i wymaganym poziomie: Nie posiada żadnych metod.

- **DropPool** - klasa reprezentująca pulę przedmiotów, które mogą być zdobyte z danego źródła. Zarządza listą przedmiotów oraz ich szansami na zdobycie. Najważniejsze metody:

  - `Choice(int level)` - metoda, która losowo wybiera przedmiot z puli na podstawie poziomu źródła oraz szansy na zdobycie.
  - `GetDrop(int level)` - metoda, która zwraca losowo wybrany przedmiot oraz jego ilość na podstawie poziomu źródła.

- **DropTable** - klasa reprezentująca tabelę łupów, która zawiera różne pule przedmiotów, które mogą być zdobyte z danego źródła. Odpowiada za zarządzanie tymi pulami oraz obliczanie zdobytych przedmiotów. Najważniejsze metody:

  - `GetDrops(int level)` - metoda, która zwraca słownik przedmiotów oraz ich ilości, które zostały losowo wybrane z każdej puli.

- **IEquippable** - interfejs definiujący własciwości przedmiotów, które można wyposażyć. Zarządza właściwościami związanymi z wyposażeniem. Nie posiada żadnych metod.

- **IEquipmentPart** - interfejs definiujący własciwości części wyposażenia, takich jak głowica broni lub płyta zbroi. Zarządza właściwościami związanymi z częściami wyposażenia. Nie posiada żadnych metod.

- **EquipmentPartManager** - statyczna klasa odpowiedzialna za zarządzanie częściami wyposażenia. Zarządza inicjalizacją i dostępem do różnych części ekwipunku. Najważniejsze metody:

  - `InitItems()` - metoda inicjalizująca listę możliwych części ekwipunku poprzez wczytanie danych z pliku JSON.
  - `GetPart<T>(string alias, CharacterClass intendedClass)` - metoda, która zwraca część ekwipunku na podstawie aliasu i klasy postaci.
  - `GetPart<T>(int id)` - metoda, która zwraca część ekwipunku na podstawie identyfikatora.
  - `GetRandomPart<T>(int tier, CharacterClass intendedClass)` - metoda, która zwraca losową część ekwipunku na podstawie poziomu i klasy postaci.

- **EquippableItemService** - statyczna klasa odpowiedzialna za zarządzanie wyposażenia, w tym generowaniem losowego wyposażenia oraz obliczaniem modyfikatorów tego typu przedmiotów. Najważniejsze metody:

  - `RarityPriceModifier(ItemRarity rarity)` - metoda, która zwraca modyfikator ceny na podstawie rzadkości przedmiotu.

  - `GetRandomWeapon(int tier, CharacterClass requiredClass)` - metoda, która zwraca losową broń na podstawie poziomu i wymaganej klasy.
  - `GetRandomArmor(int tier, CharacterClass requiredClass)` - metoda, która zwraca losową zbroję na podstawie poziomu i wymaganej klasy.
  - `GetBossDrop(int tier, string bossAlias)` - metoda, która zwraca losowy przedmiot od bossa na podstawie poziomu i aliasu bossa.

- **WeaponHead, WeaponBinder, WeaponHandle** - klasy reprezentujące części broni. Odpowiedzialne za zarządzanie statystykami broni. Nie posiadają żadnych metod.

- **ArmorPlate, ArmorBinder, ArmorBase** - klasa reprezentująca częsci zbroi. Odpowiada za zarządzanie statystykami zbroi. Nie posiadają żadnych metod.

- **Weapon** - klasa reprezentująca broń w grze. Odpowiada za zarządzanie właściwościami broni oraz jej użyciem. Najważniejsze metody:
  - `Use()` - metoda, która umożliwia wyposażenie broni przez gracza, sprawdzając wymagania klasy i poziomu.
  - `Inspect(int amount = 1)` - metoda, która wyświetla szczegóły broni, porównując ją z aktualnie używaną bronią przez gracza.
  - `UpdatePassives(PlayerCharacter player)` - aktualizuje pasywne efekty gracza w oparciu o galduryty broni.
  - `AddGaldurite(Galdurite galdurite)` - dodaje galduryt do broni, aktualizując pasywne efekty.
  - `RemoveGaldurite(Galdurite galdurite)` - usuwa galduryt z broni, aktualizując pasywne efekty.

- **Armor** - klasa reprezentująca zbroję w grze. Odpowiada za zarządzanie właściwościami zbroi oraz jej użyciem. Najważniejsze metody:
  - `Use()` - metoda, która umożliwia wyposażenie zbroi przez gracza, sprawdzając wymagania klasy i poziomu.
  - `Inspect(int amount = 1)` - metoda, która wyświetla szczegóły zbroi, porównując ją z aktualnie używaną zbroją przez gracza.
  - `UpdatePassives(PlayerCharacter player)` - aktualizuje pasywne efekty gracza w oparciu o galduryty zbroi.
  - `AddGaldurite(Galdurite galdurite)` - dodaje galduryt do zbroi, aktualizując pasywne efekty.
  - `RemoveGaldurite(Galdurite galdurite)` - usuwa galduryt ze zbroi, aktualizując pasywne efekty.

- **Quest** - klasa reprezentująca zadanie w grze. Odpowiada za zarządzanie stanem zadania oraz jego celami. Najważniejsze metody:
  - `AcceptQuest()` - akceptuje zadanie, zmieniając jego stan na przyjęte i wyświetlając dialog z NPC.
  - `TryProgress(QuestObjectiveContext context)` - próbuje zaktualizować postęp zadania na podstawie podanego kontekstu.
  - `HandInQuest()` - oddaje zadanie, przyznając nagrody i zmieniając stan na oddane.
  - `GetCurrentStage()` - zwraca aktualny etap zadania.

- **QuestManager** - statyczna klasa odpowiedzialna za zarządzanie zadaniami w grze. Odpowiada za inicjalizację i aktualizację zadań. Najważniejsze metody:
  - `InitMainQuests()` - inicjalizuje główne zadania, wczytując je z pliku JSON.
  - `CheckForProgress(QuestObjectiveContext context)` - sprawdza postęp zadań i aktualizuje ich stan.
  - `RerollSideQuests()` - losowo generuje nowe zadania poboczne, bazując na aktualnym stanie gry.
- **QuestLog** - statyczna klasa odpowiedzialna za zarządzanie dziennikiem zadań gracza. Umożliwia przeglądanie i inspekcję zadań. Najważniejsze metody:
  - `OpenLog()` - otwiera dziennik zadań, umożliwiając aktualny i poprzedni stan wszystkich zadań.
  - `InspectQuest(Quest quest)` - wyświetla szczegóły wybranego zadania, w tym jego cele i stan.
- **QuestNPCHandler** - statyczna klasa odpowiedzialna za interakcje z NPC w kontekście zadań. Umożliwia akceptację i oddawanie zadań. Najważniejsze metody:
  - `GetAvailableQuests(string questGiver)` - zwraca listę dostępnych zadań od danego NPC.
  - `SelectQuestToAccept(string questGiver)` - umożliwia graczowi wybór i akceptację zadania od NPC.
  - `SelectQuestToReturn(string questEnder)` - umożliwia graczowi oddanie ukończonego zadania NPC.
- **QuestReward** - klasa reprezentująca nagrody za ukończenie zadania. Odpowiada za przechowywanie informacji o nagrodach. Nie posiada żadnych metod
- **QuestStage** - klasa reprezentująca etap zadania. Odpowiada za zarządzanie celami w ramach etapu. Nie posiada żadnych metod.
- **IQuestObjective** - interfejs definiujący podstawowe właściwości i metody dla celów zadań w grze. Najważniejsze metody:
  - `void Progress(QuestObjectiveContext context)` - metoda do aktualizacji postępu, jeśli podany kontekst zgadza się z celem.
- **TalkQuestObjective** - klasa implementująca interfejs `IQuestObjective`, reprezentująca cel zadania polegający na rozmowie z NPC.
- **DescendQuestObjective** - klasa implementująca interfejs `IQuestObjective`, reprezentująca cel zadania polegający na zejściu na określone piętro w lochu.
- **KillInDungeonQuestObjective** - klasa implementująca interfejs `IQuestObjective`, reprezentująca cel zadania polegający na zabiciu określonej liczby przeciwników w lochu.
- **KillQuestObjective** - klasa implementująca interfejs `IQuestObjective`, reprezentująca cel zadania polegający na zabiciu konkretnego typu przeciwnika.
- **QuestObjectiveContext** - klasa przechowująca kontekst dla celów zadań, zawierająca informacje o stanie zadania. Nie posiada żadnych metod.
- **ActiveSkillTextService** - klasa odpowiedzialna za wyświetlanie tekstów związanych z używaniem aktywnych umiejętności w grze. Najważniejsze metody:

  - `DisplayUseSkillText(Character caster, Character target, ActiveSkill skill, double hitChance)` - wyświetla tekst informujący o użyciu umiejętności przez postać na celu, z uwzględnieniem szansy na trafienie.
  - `DisplayCritText(Character caster)` - wyświetla tekst informujący o krytycznym trafieniu przez postać.
  - `DisplayMissText(Character caster)` - wyświetla tekst informujący o nietrafieniu przez postać.

- **BattleTextService** - klasa odpowiedzialna za wyświetlanie tekstów związanych z walką w grze. Najważniejsze metody:
  - `DisplayStatusText(BattleUser player, BattleUser enemy)` - wyświetla status (zdrowie, zasób, statystyki i akcje) gracza oraz przeciwnika.
  - `DisplayMovementText(Character moving)` - wyświetla tekst informujący o ruchu postaci.
  - `DisplayEndBattleText(bool won)` - wyświetla tekst informujący o zakończeniu walki, wskazując, czy gracz wygrał, czy przegrał.
  - `DisplayDeathText(Character dead)` - wyświetla tekst informujący o śmierci postaci.
  - `DisplayBattleStartText(EnemyCharacter enemy)` - wyświetla tekst informujący o rozpoczęciu walki z przeciwnikiem.
  - `DisplayCannotMoveText(Character target)` - wyświetla tekst informujący, że postać nie może się poruszyć.

- **CharacterEventTextService** - klasa odpowiedzialna za wyświetlanie tekstów związanych z wydarzeniami postaci w grze. Najważniejsze metody:
  - `DisplayTakeDamageText(Character character, ...)` - wyświetla tekst informujący o obrażeniach zadanych postaci.
  - `DisplayHealText(Character character, int heal)` - wyświetla tekst informujący o wyleczeniu postaci.
  - `DisplayResourceRegenText(Character character, int regen)` - wyświetla tekst informujący o regeneracji zasobu postaci.
  - `DisplayGoldGainText(PlayerCharacter character, int gold)` - wyświetla tekst informujący o zdobyciu złota przez postać.
  - `DisplayGoldLossText(PlayerCharacter character, int gold)` - wyświetla tekst informujący o utracie złota przez postać.

- **DungeonTextService** - klasa odpowiedzialna za wyświetlanie tekstów związanych z lochami w grze. Najważniejsze metody:
  - `DisplayDungeonHeader(Dungeon dungeon)` - wyświetla nagłówek lochu, zawierający informacje o typie lochu i poziomie.
  - `DisplayDungeonMap(Dungeon dungeon, int locationIndex, DungeonRoom currentLocation)` - wyświetla mapę lochu z aktualną lokalizacją postaci.
  - `DisplayCollectPlantText(IItem plant)` - wyświetla tekst informujący o zebraniu rośliny.
  - `DisplayTrapDisarmText(bool success)` - wyświetla tekst informujący o udanym lub nieudanym rozbrojeniu pułapki.

  - `DisplayRestAtBonfire(bool ambushed)` - wyświetla tekst informujący o odpoczynku przy ognisku, z informacją o ewentualnej zasadzce.

- **InventoryMenuHandler** - klasa odpowiedzialna za zarządzanie menu ekwipunku w grze. Najważniejsze metody:
  - `OpenInventoryMenu()` - otwiera menu ekwipunku, umożliwiając przeglądanie i zarządzanie przedmiotami.
  - `ChooseItem(bool showPrices)` - umożliwia wybór przedmiotu z ekwipunku, z opcjonalnym wyświetlaniem cen.
  - `DeleteItem()` - usuwa wybrany przedmiot z ekwipunku.
  - `SortInventory()` - sortuje przedmioty w ekwipunku według wybranej metody.
  - `InspectItem()` - wyświetla szczegóły wybranego przedmiotu.
- **NPCMenuHandler** - klasa odpowiedzialna za zarządzanie menu interakcji z ekwpiunkiem NPC w grze. Najważniejsze metody:
  - `OpenInventoryMenu(NPCInventory inventory)` - otwiera menu ekwipunku NPC, umożliwiając przeglądanie i interakcję z przedmiotami.
  - `ChooseItem(NPCInventory inventory)` - umożliwia wybór przedmiotu z ekwipunku NPC.
- **Town** - klasa reprezentująca miasto w grze. Odpowiada za zarządzanie NPC w mieście oraz interakcjami gracza z miastem. Najważniejsze metody:
  - `EnterTown()` - metoda, która umożliwia graczowi wejście do miasta i interakcję z NPC oraz innymi elementami.
  - `ChooseDungeon()` - metoda, która pozwala graczowi wybrać loch, do którego chce wejść.
- **TownsHandler** - statyczna klasa odpowiedzialna za przechowywanie i zarządzanie miastami w grze (aktualnie znajduje się tylko jedno). Najważniejsze metody:
  - `FindNPC(string alias)` - metoda, która wyszukuje NPC w mieście na podstawie podanego aliasu.
- **NPC** - klasa abstrakcyjna reprezentująca postacie niezależne w grze. Odpowiada za zarządzanie interakcjami z NPC. Najważniejsze metody:
  - `OpenMenu()` - metoda abstrakcyjna, która otwiera menu dla konkretnego NPC.
  - `Say(string message)` - metoda abstrakcyjna, która wyświetla wiadomość dialogową NPC.
  - `SpendGold(int gold)` - metoda, która aktualizuje wydane złoto i poziom lojalności NPC.
- **Alchemist** - klasa dziedzicząca po `NPC`, reprezentująca alchemika w grze. Odpowiada za interakcje związane z alchemią. Najważniejsze metody:
  - `OpenMenu()` - metoda, która otwiera menu alchemika, umożliwiając graczowi wybór akcji.
  - `Treat()` - metoda, która leczy gracza za określoną ilość złota.
  - `RefillPotion()` - metoda, która napełnia mikstury gracza, pobierając odpowiednie materiały.
- **Blacksmith** - klasa dziedzicząca po `NPC`, reprezentująca kowala w grze. Odpowiada za interakcje związane z rzemiosłem. Najważniejsze metody:

  - `OpenMenu()` - metoda, która otwiera menu kowala, umożliwiając graczowi wybór akcji.
  - `UpgradeWeapon()` - metoda, która umożliwia graczowi ulepszanie broni.
  - `ReforgeWeapon()` - metoda, która przekształca broń gracza, zmieniając jej rzadkość.

- **Enchanter** - klasa dziedzicząca po `NPC`, reprezentująca zaklinacza w grze. Odpowiada za interakcje związane z zaklęciami. Najważniejsze metody:
  - `OpenMenu()` - metoda, która otwiera menu zaklinacza, umożliwiając graczowi wybór akcji.
  - `ExamineGaldurite()` - metoda, która pozwala graczowi na ujawnienie efektów galdurytu.
  - `InsertWeaponGaldurite()` - metoda, która umożliwia dodanie galdurytu do broni gracza.

- **SaveData** - klasa odpowiedzialna za przechowywanie danych dotyczących stanu gry. Zawiera informacje o postaci gracza, poziomie trudności, zadaniach oraz postępach w grze. Nie posiada żadnych metod.

- **DataPersistanceManager** - statyczna klasa odpowiedzialna za zapisywanie i ładowanie stanu gry. Zarządza operacjami związanymi z plikami zapisu. Najważniejsze metody:
  - `SaveGame(SaveData data)` - metoda, która zapisuje dane gry do pliku JSON. Umożliwia użytkownikowi podanie nazwy pliku zapisu oraz tworzy katalog, jeśli nie istnieje.
  - `LoadGame()` - metoda, która ładuje dane gry z pliku JSON. Umożliwia wybór zapisu z listy dostępnych plików i aktualizuje stan gry na podstawie załadowanych danych.
  - `DeleteSaveFile()` - metoda, która usuwa wybrany plik zapisu z dysku. Umożliwia użytkownikowi wybór zapisu do usunięcia z listy dostępnych plików.

- **UtilityMethods** - statyczna klasa zawierająca różne metody pomocnicze używane w grze. Pozwala między innymi na proste generowanie losowych wartości, potwierdzenia oraz manipulację tekstem. Najważniejsze metody:
  - `RandomDouble(double minValue, double maxValue)` - generuje losową wartość typu double w podanym zakresie.
  - `RandomFloat(float minValue, float maxValue, int round)` - generuje losową wartość typu float w podanym zakresie, zaokrągloną do określonej liczby miejsc po przecinku.
  - `EffectChance(double resistance, double baseChance)` - oblicza szansę na efekt na podstawie odporności i podstawowej szansy.
  - `Confirmation(string message, bool defaultValue = false)` - wyświetla komunikat potwierdzenia i zwraca wartość boolowską w zależności od wyboru użytkownika.
  - `ClearConsole(int lines = 1)` - czyści określoną liczbę linii w konsoli.
  - `CalculateModValue(double initial, List<StatModifier> modifiers)` - oblicza wartość liczby na podstawie listy modyfikatorów.

- **GameSettings** - statyczna klasa odpowiedzialna za przechowywanie ustawień gry, takich jak poziom trudności. Nie posiada żadnych metod.

- **MainMenu** - statyczna klasa odpowiedzialna za zarządzanie głównym menu gry. Oferuje opcje takie jak rozpoczęcie nowej gry, wczytanie zapisu oraz wybór języka. Najważniejsze metody:

  - `Menu()` - wyświetla główne menu gry i obsługuje wybór opcji przez gracza.
  - `ChooseLanguage()` - umożliwia graczowi wybór języka interfejsu.
  - `NewGame()` - inicjalizuje nową grę, ustawiając odpowiednie parametry i tworząc nową instancję miasta.

- **NameAliasHelper** - klasa pomocnicza do zarządzania aliasami i nazwami w grze. Odpowiada za tłumaczenie aliasów na odpowiednie zlokalizowane językowo nazwy. Najważniejsze metody:
  - `GetName(string alias)` - zwraca nazwę odpowiadającą podanemu aliasowi, korzystając z lokalizacji zasobów.
  - `GetDungeonType(DungeonType type, string grammarCase)` - zwraca typ lochu w odpowiednim przypadku gramatycznym.

- **Stylesheet** - statyczna klasa odpowiedzialna za zarządzanie stylami wyświetlania w grze. Umożliwia definiowanie i inicjalizowanie różnych stylów. Najważniejsze metody:
  - `InitStyles()` - inicjalizuje słownik stylów, definiując różne style używane w grze.

- **ActiveSkillEffectConverter** - klasa dziedzicząca po `JsonConverter` odpowiedzialna za konwersję obiektów typu `IActiveSkillEffect` do formatu JSON i odwrotnie.

- **EquipmentPartConverter** - klasa dziedzicząca po `JsonConverter` odpowiedzialna za konwersję obiektów typu `IEquipmentPart` do formatu JSON i odwrotnie.

- **ItemConverter** - klasa dziedzicząca po `JsonConverter` odpowiedzialna za konwersję słowników przechowujących typ `<IItem, int>` do formatu JSON i odwrotnie.

- **NPCConverter** - klasa dziedzicząca po `JsonConverter` odpowiedzialna za konwersję obiektów typu `NPC` do formatu JSON i odwrotnie.

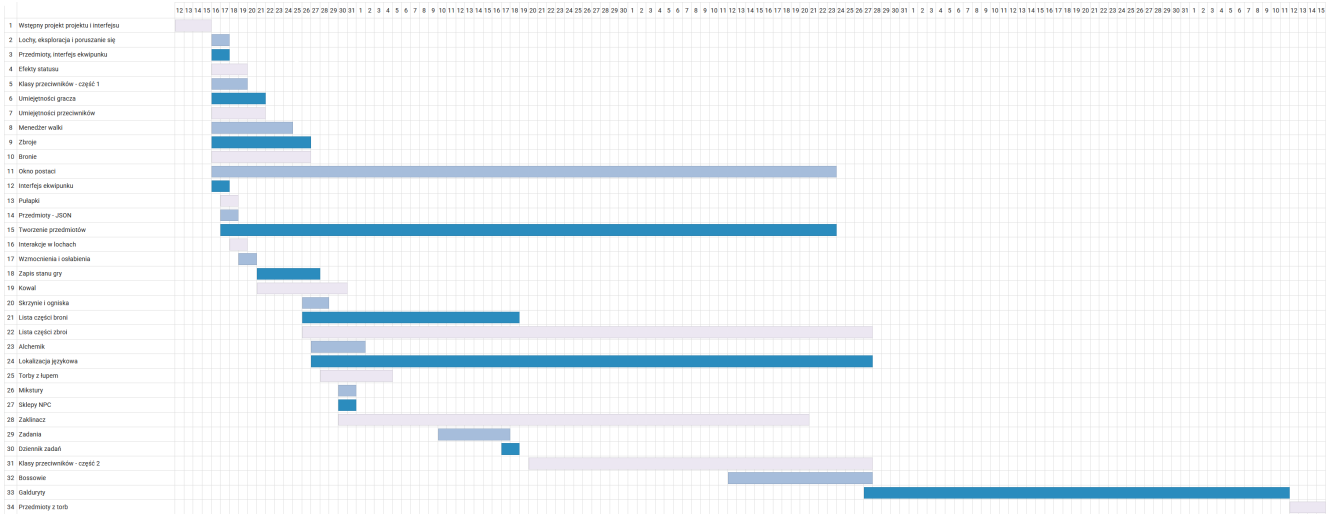
- **PlayerJsonConverter** - klasa dziedzicząca po `JsonConverter` odpowiedzialna za konwersję obiektów typu `PlayerCharacter` do formatu JSON i odwrotnie.

- **QuestObjectiveConverter** - klasa dziedzicząca po `JsonConverter` odpowiedzialna za konwersję obiektów typu `IQuestObjective` do formatu JSON i odwrotnie.

Rozdział 3

Harmonogram realizacji projektu

W niniejszym rozdziale przedstawiono harmonogram realizacji projektu w formie diagramu Gantta. Diagram ten ilustruje poszczególne etapy projektu oraz ich czas trwania.



Rysunek 3.1: Diagram Gantta przedstawiający harmonogram realizacji projektu.

3.1 Problemy i trudności

Podczas realizacji projektu napotkano kilka trudności, w tym:

- **Niedoszacowanie czasu** - Wstępne oszacowania czasowe okazały się niewystarczające, co prowadziło do opóźnień.
- **Konieczność refaktoryzacji** - Niedopracowanie niektórych struktur projektu wymagało przeplanowania i dokonania refaktoryzacji, co również opóźniło ukończenie projektu
- **Problemy z integracją** - Integracja różnych komponentów systemu (np. JSON) napotkała na nieprzewidziane trudności techniczne.

3.2 Repozytorium i system kontroli wersji

Projekt jest zarządzany w systemie kontroli wersji Git, a wszystkie zmiany są dokumentowane w repozytorium dostępnym pod adresem: `https://github.com/xzantem/ConsoleGodmist`. Użycie systemu kontroli wersji umożliwiło śledzenie postępów, kontrolę wprowadzanych zmian i łatwe cofanie do poprzednich wersji.

Rozdział 4

Prezentacja warstwy użytkowej projektu

Aplikacja jest interaktywną grą RPG, w której gracze mogą eksplorować różne lokacje, walczyć z przeciwnikami oraz rozwijać swoje postacie. Gra oferuje różnorodne mechaniki, takie jak system walki, tworzenia przedmiotów, wykonywania misji oraz interakcje z NPC. Użytkownicy mogą również zarządzać swoim ekwipunkiem oraz brać udział w zadaniach.

4.1 Prezentacja

W tej sekcji przedstawiono zrzuty ekranu i opisy ilustrujących kluczowe interfejsy i funkcjonalności aplikacji.

4.1.1 Menu główne

- Na rysunku 4.1 przedstawiono głowne menu aplikacji, które ukazuje się po włączeniu jej. Aplikacja pozwala na rozpoczęcie nowej gry, załadowanie lub usunięcie istniejącego zapisu gry, zmianę języka, lub wyjście z gry.

Rysunek 4.1: Menu główne, opcje początkowe

- Na rysunku 4.2 przedstawiono wybór języka dostępny z głównego menu. Po wybraniu języka aplikacja wraca do początkowego ekranu. (język zmieniono na polski)

Rysunek 4.2: Menu główne, wybór języka

- Na rysunku 4.3 przedstawiono wybór poziomu trudności, dostępny po rozpoczęciu nowej gry.

Rysunek 4.3: Menu główne, wybór poziomu trudności

- Po wybraniu poziomu trudności, gra przedstawia wprowadzenie oraz pozwala na wybranie klasy postaci. Na rysunku 4.4 znajduje się wybór klasy postaci.

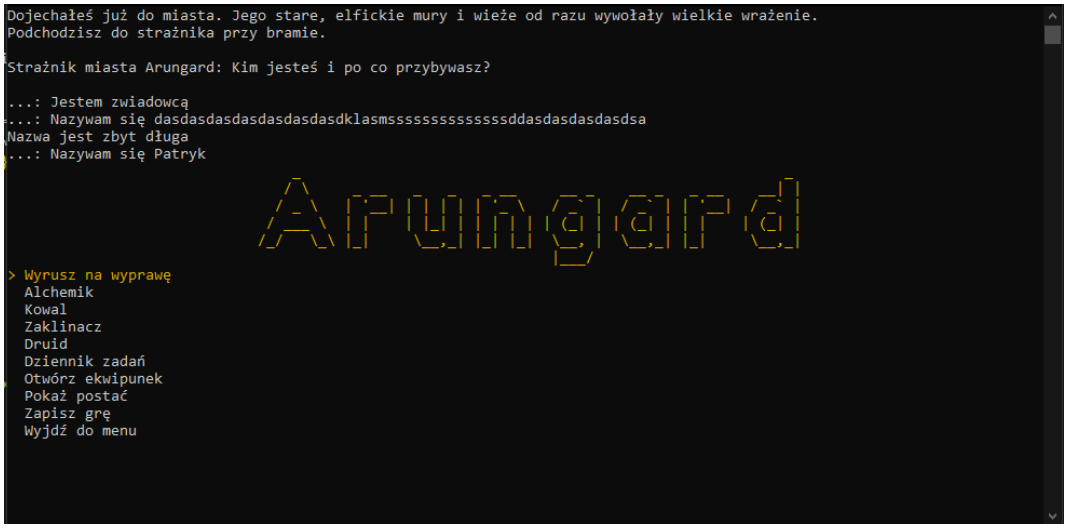
Rysunek 4.4: Menu główne, wybór klasy postaci

- Po wybraniu klasy postaci, gra pozwala na wybranie nazwy postaci. Aplikacja zabezpieczona jest przed wpisaniem pustej nazwy (wciśniecie klawisza Enter nie przenosi dalej), lub zbyt długiej (otrzymujemy informację zwrotną). Na rysunku 4.5 przedstawiono wybór nazwy postaci.

Rysunek 4.5: Menu główne, wybór nazwy

4.1.2 Miasto

- Po wybraniu nazwy, gracz przenoszony jest do miasta. Otrzymuje wiele opcji takich jak wyruszenie na wyprawę do lochu, rozmawianie z NPC, otwarcie dziennika zadań, ekwipunku, włączenie okna postaci, zapis gry, lub powrót do menu. Na rysunku 4.6 przedstawiono ekran główny miasta.

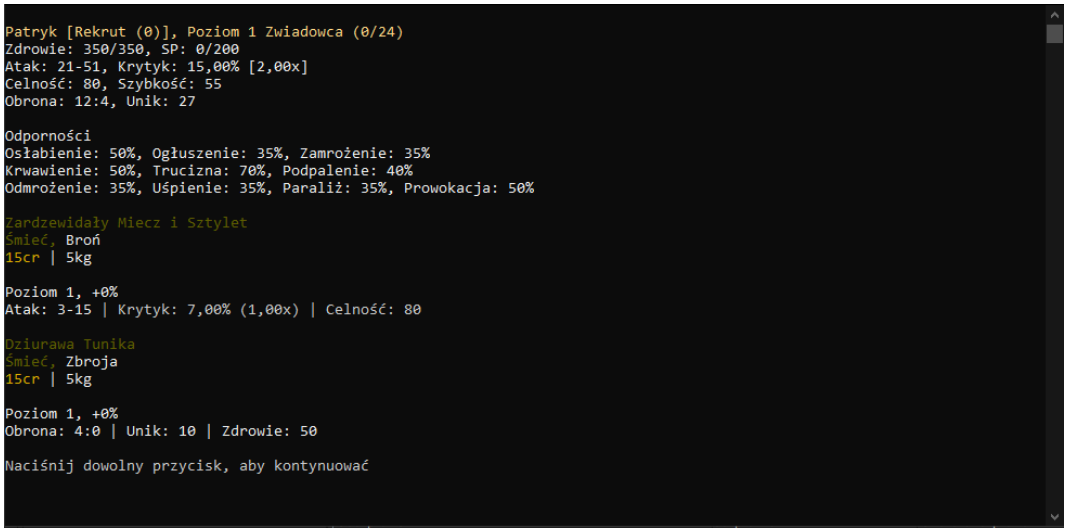


Rysunek 4.6: Miasto, ekran główny

- Jeśli użytkownik wybierze opcję ‘Zapisz grę’, zostaje zapytany o podanie nazwy pliku zapisu, który chce utworzyć. Tutaj również występuje zabezpieczenie przed pustą i zbyt długą nazwą. Po wpisaniu nazwy plik zapisu tworzy się, a gracz ponownie otrzymuje ekran głowny miasta. Na rysunku 4.7 przedstawiono proces zapisu gry.

Rysunek 4.7: Miasto, zapis gry

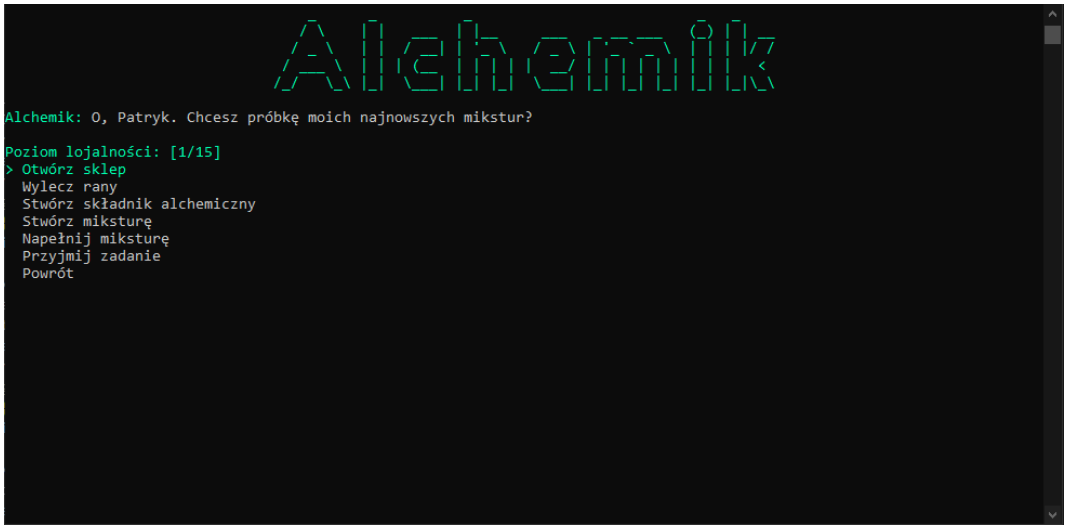
- Jeśli użytkownik wybierze opcję ‘Pokaż postać’, wyświetlone zostają informacje o statystykach i wyposażeniu postaci. Po naciśnięciu dowolnego przycisku na klawiaturze, gracz ponownie otrzymuje ekran główny miasta. Na rysunku 4.8 przedstawiono ekran postaci.



Rysunek 4.8: Ekran postaci

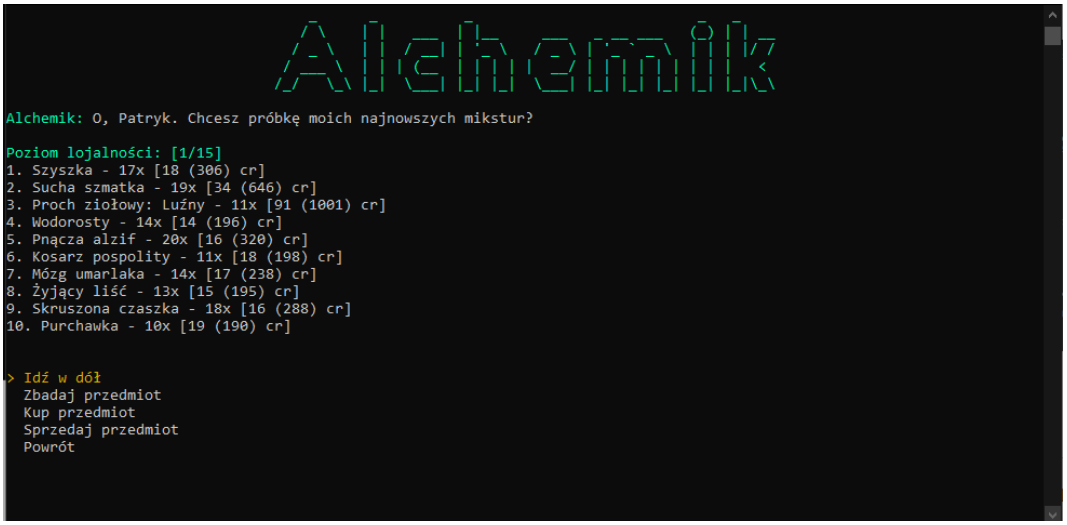
4.1.3 Postacie niezależne

- Po wybraniu opcji ‘Alchemik’, ‘Kowal’ lub ‘Zaklinacz’, gracz zostanie przeniesiony do rozmowy z odpowiednim NPC. Stąd może wykonać jedną z dostępnych akcji, takich jak wejście do sklepu, czy wyleczenie ran. Na rysunku 4.9 przedstawiono okno alchemika.

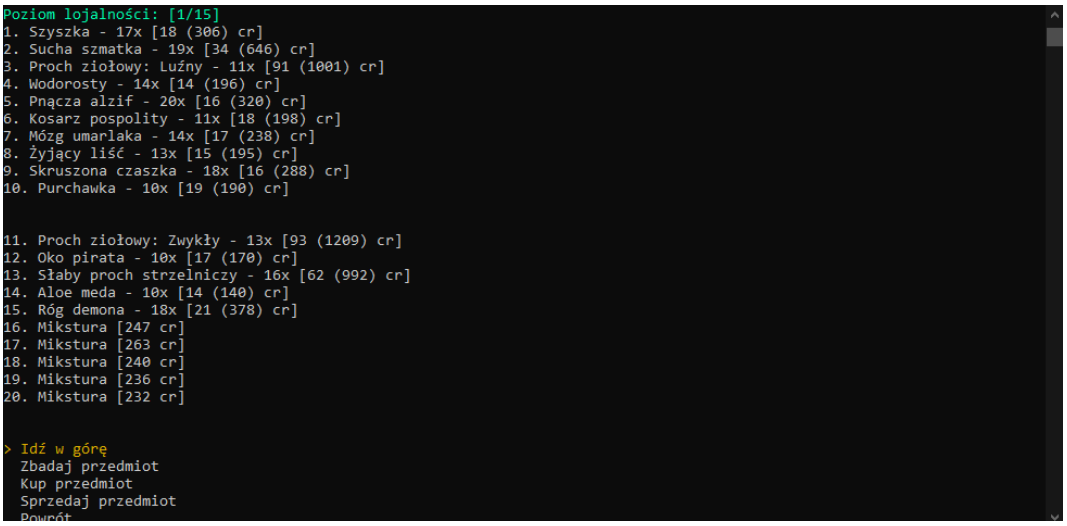


Rysunek 4.9: Ekran NPC

- W przypadku, gdy użytkownik wybierze ‘Otwórz sklep’, otworzy się okno przedstawiające listę przedmiotów posiadanych przez postać niezależną. Tutaj, ma tak jak w ekwipunku opcje zbadania wybranego przedmiotu. Może również kupować przedmioty od NPC, lub sprzedawać przedmioty ze swojego ekwipunku. Na rysunkach 4.10 i 4.11 przedstawiono menu sklepu.

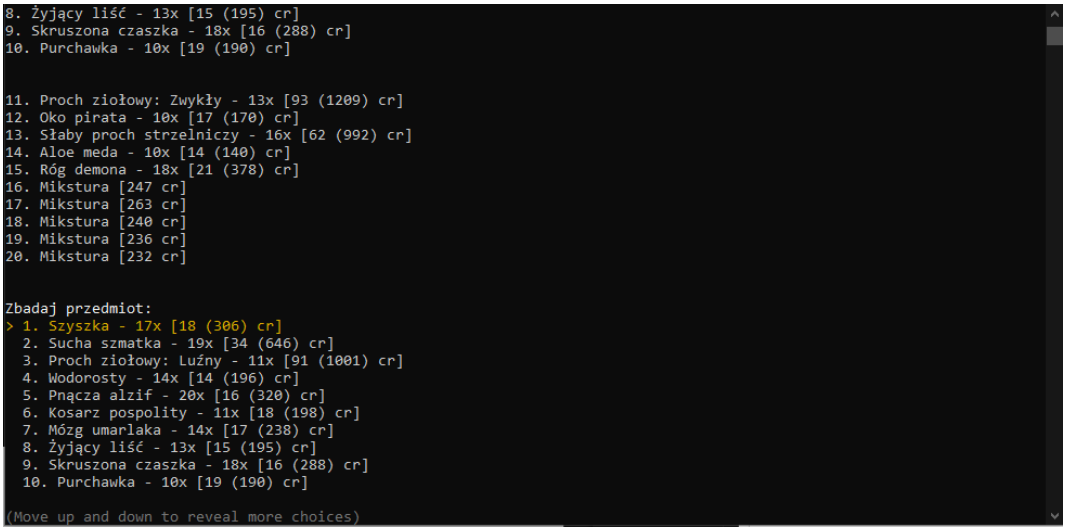


Rysunek 4.10: Menu sklepu NPC, widok początkowy

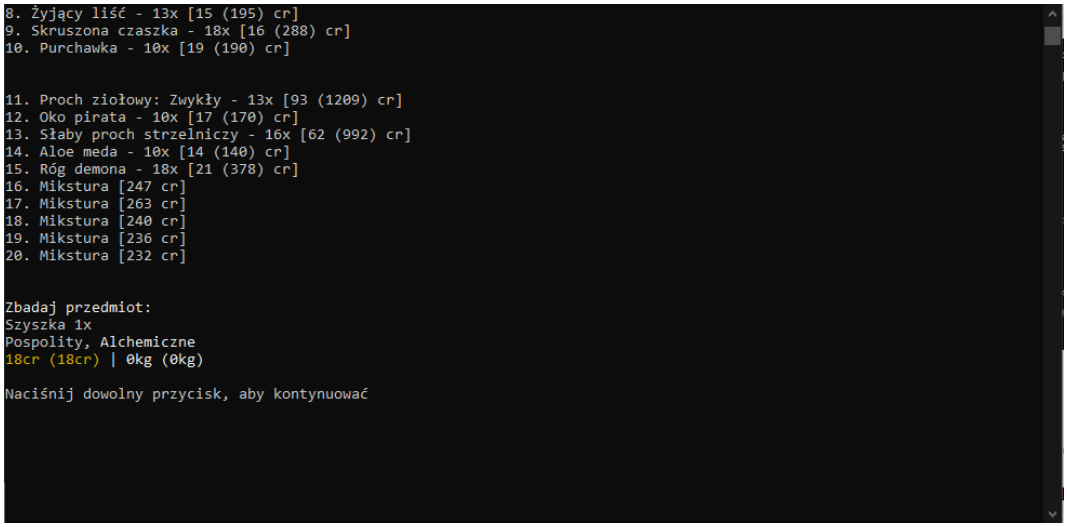


Rysunek 4.11: Menu sklepu NPC, po wybraniu ‘Idź w dół’

- Po wybraniu opcji ‘Zbadaj przedmiot’, ukazuje się się pole wyboru przedmiotu z ekwipunku alchemika. Po wybraniu z listy, przedstawione zostają szczegóły badanego przedmiotu. Na rysunkach 4.12 i 4.13 przedstawiono proces badania przedmiotu.

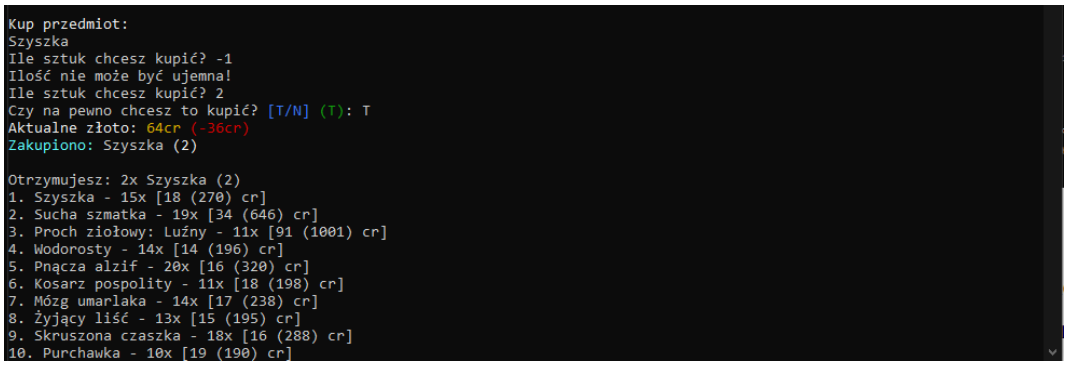


Rysunek 4.12: Menu sklepu NPC, badanie przedmiotu, wybór



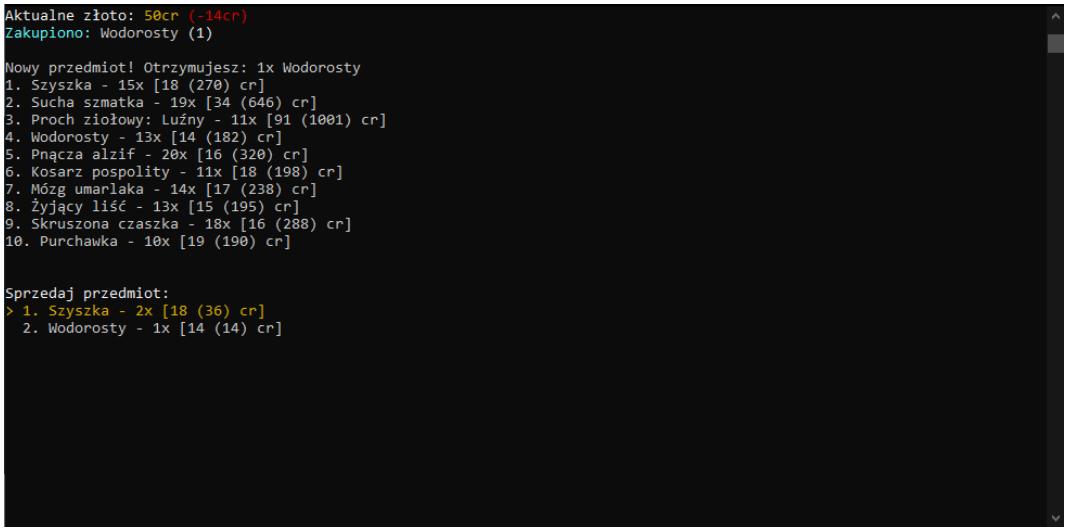
Rysunek 4.13: Menu sklepu NPC, badanie przedmiotu, rezultat

- Po wybraniu opcji ‘Kup przedmiot‘, użytkownik otrzymuje identyczne okno wyboru. Po jego dokonaniu i potwierdzeniu, że chce kupić przedmiot, otrzymuje go i traci złoto. Kupowanie przedmiotu przedstawiono na rysunku 4.14

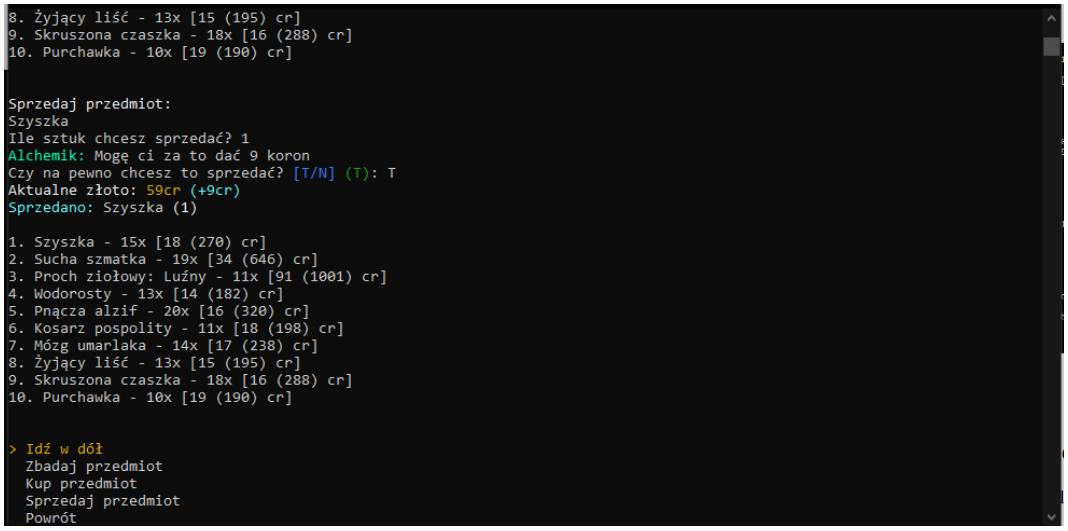


Rysunek 4.14: Menu sklepu NPC, kupno przedmiotu

- Po wybraniu opcji ‘Sprzedaj przedmiot‘, użytkownik może wybrać przedmiot ze swojego ekwipunku Po wybraniu przedmiotu i potwierdzeniu, że chce sprzedać przedmiot, traci go i zyskuje złoto. Na rysunkach 4.15 i 4.16 przedstawiono proces sprzedaży przedmiotu

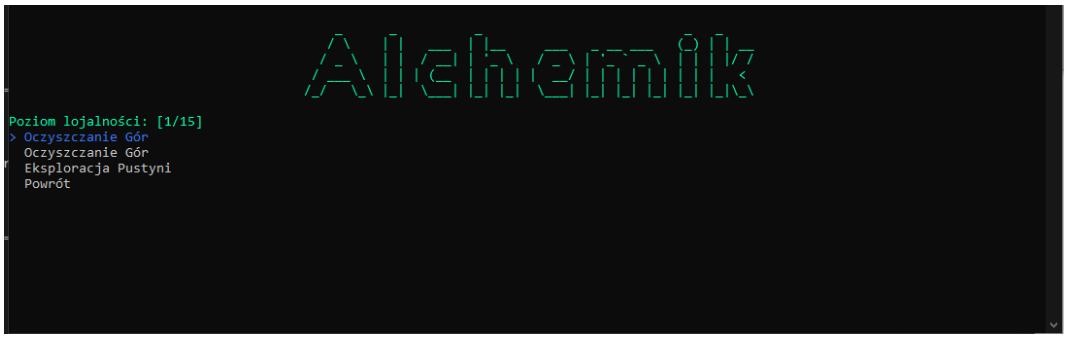


Rysunek 4.15: Menu sklepu NPC, sprzedaż przedmiotu, wybór



Rysunek 4.16: Menu sklepu NPC, sprzedaż przedmiotu, rezultat

- Jeśli NPC posiada zadanie do zaoferowania, użytkownik może wybrać ‘Przyjmij zadanie’. Otrzymuje następnie wybór z dostępnych zadań u NPC. Po dokonaniu wyboru i potwierdzeniu go, zadanie zostaje zaakceptowane. Na rysunkach 4.17 i 4.18 przedstawiono akceptowanie zadania od NPC.



Rysunek 4.17: Ekran NPC, wybór zadania

Rysunek 4.18: Ekran NPC, zaakceptowane zadanie

4.1.4 Ekwipunek

- Jeśli użytkownik wybierze opcję "Otwórz ekwipunek"w mieście lub w lochu, wyświetlone zostają informacje o ekwipunku i opcje zarządzania nim. Na rysunku 4.19 przedstawiono okno ekwipunku.

Rysunek 4.19: Ekran ekwipunku

- Gracz może zbadać dowolny przedmiot z ekwipunku, wyświetlone zostaną wtedy szczegóły tego konkretnego przedmiotu, lub jeśli przedmiot spełnia własność ‘Stackable’, to stosu przedmiotu. Na rysunkach 4.20 i 4.21 przedstawiono kolejne kroki badania przedmiotu.

Rysunek 4.20: Ekran ekwipunku. Badanie przedmiotu, wybór.

Rysunek 4.21: Ekran ekwipunku. Badanie przedmiotu, rezultat.

- Gracz może wyrzucić dowolny przedmiot z ekwipunku, zostanie on wtedy po potwierdzeniu usunięty z ekwipunku. Na rysunku 4.22 przedstawiono wyrzucanie przedmiotu

Rysunek 4.22: Ekran ekwipunku, wyrzucanie przedmiotu.

- Po kliknięciu ‘Wyrzuć wyposażenie na inne klasy’, po potwierdzeniu zostaną usunięte wszystkie elementy wyposażenia (bronie i pancerze), które nie mogą zostać użyte przez klasę postaci gracza.
- Po kliknieciu ‘Sortuj ekwipunek’, ukazane zostaną możliwości sortowania ekwipunku na różne sposoby. Po wybraniu opcji, ekwipunek zostanie posortowany według wybranego kryterium. Na rysunkach 4.23 i 4.24 przedstawiono sortowanie ekwipunku alfabetycznie.

Rysunek 4.23: Ekran ekwipunku, sortowanie ekwipunku, wybór metody.

Rysunek 4.24: Ekran ekwipunku, sortowanie ekwipunku, rezultat.

- W przypadku wybrania opcji ‘Użyj przedmiot’, użytkownik może wybrać przedmiot do użycia. Jeśli będzie to przedmiot, który dziedziczy po `IUsable`, to wykona się akcja odpowiednia dla tego przedmiotu (użyta mikstura, wyposażona broń, itp.).

4.1.5 Lochy

- Poprzez opcję ‘Wyrusz na wyprawę’ w mieście, istnieje możliwość wygenerowania i wejścia do instancji lochu. Na rysunku 4.25 przedstawiono proces wejścia do lochu Katakumb na poziomie 1. Na rysunku 4.26 przedstawiono widok startowy lochu.

Rysunek 4.25: Lochy, wybór typu lochu

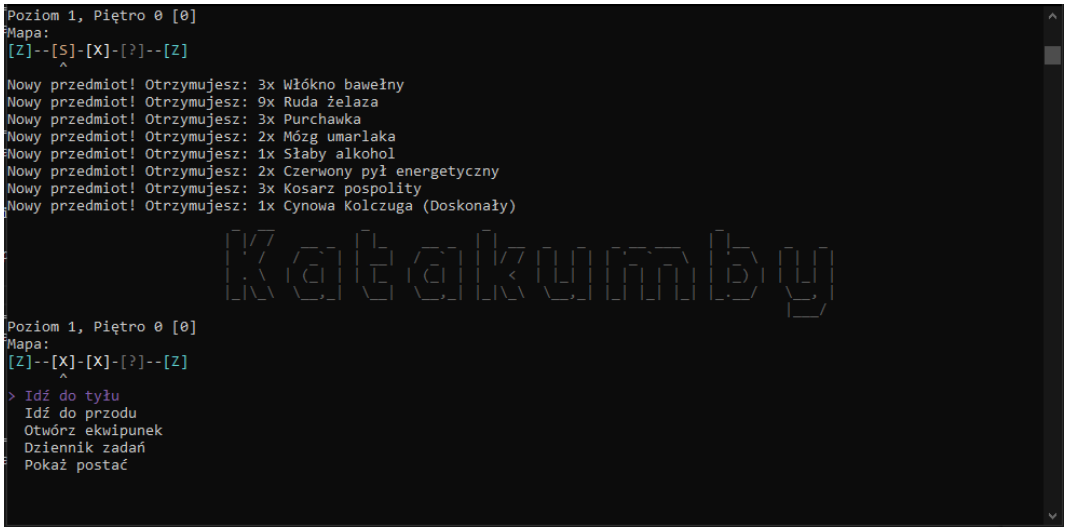
Rysunek 4.26: Lochy, widok mapy

- Po wejściu do lochu, pokazują się opcje poruszania się po lochu (przód i tył - jeśli gracz znajduje się na granicy pięter, mogą się zmieniać w góra, dół, lub całkowite wyjście). Gracz może z tego poziomu także otworzyć ekwipunek lub dziennik zadań. Te opcje działają dokładnie tak samo jak w mieście. Na rysunkach 4.27 i 4.28 przedstawiono poruszanie się po lochu.

Rysunek 4.27: Lochy, poruszanie się w przód

Rysunek 4.28: Lochy, poruszanie się w tył

- W lochach znajduje się wiele przedmiotów interaktowalnych: pułapki, walki, rośliny, skrzynki i ogniska. W przypadku, gdy gracz napotka jeden z nich, może z nim interaktować poprzez odpowiednią opcję (np. ‘Zbierz skarb’ widoczne na rysunku 4.28), lub interaktuje automatycznie po nadepnięciu na nie w przypadku walk i pułapek. Na rysunkach 4.29-4.31 pokazano interakcje z skrzynią, rośliną i ogniskiem.



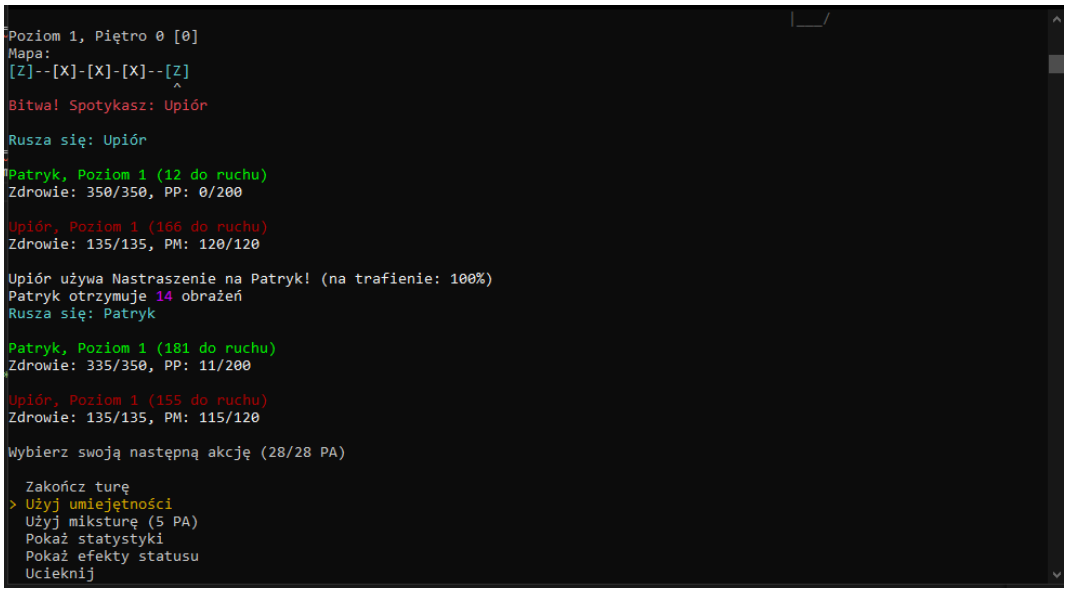
Rysunek 4.29: Lochy, otwieranie skrzyni

Rysunek 4.30: Lochy, zbieranie roślin

Rysunek 4.31: Lochy, odpoczywanie przy ognisku

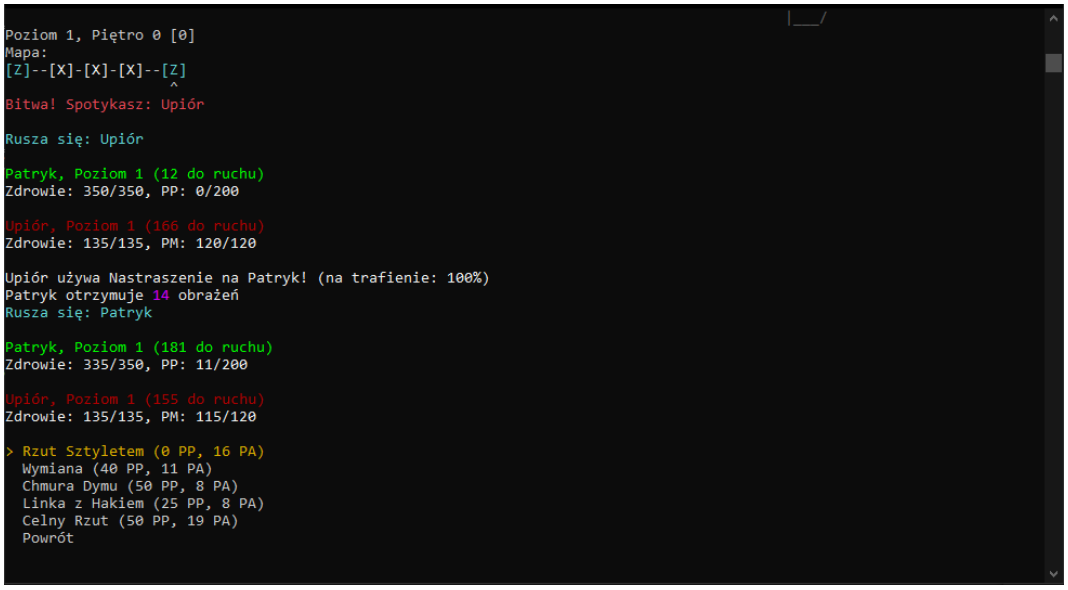
4.1.6 System walki

- Po napotkaniu walki w lochu, lub zasadzki podczas odpoczywania przy ognisku, rozpoczyna się ona automatycznie. Użytkownik otrzymuje informację o przeciwniku i walka rozpoczyna się. Kolejność i częstość ruchów jest określana poprzez statystykę szybkości. W przykładzie gracz napotkuje przeciwnika ‘Upiór’ i jest on szybszy, więc rusza się pierwszy. Gracz otrzymuje możliwość wykonania następnego ruchu. Gracz może dowolnie używać umiejętności, mikstur i wyświetlać informacje. Tura kończy się dopiero przy manualnym zakończeniu lub ucieczce. Na rysunku 4.32 przedstawiono rozpoczęcie walki

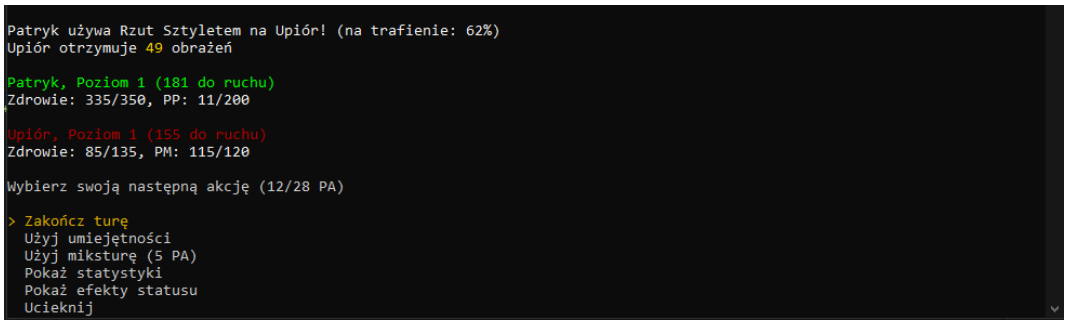


Rysunek 4.32: System walki, rozpoczęcie walki

- Na rysunkach 4.33 i 4.34 przedstawiono użycie przez gracza umiejętności ‘Rzut Sztyletem’.

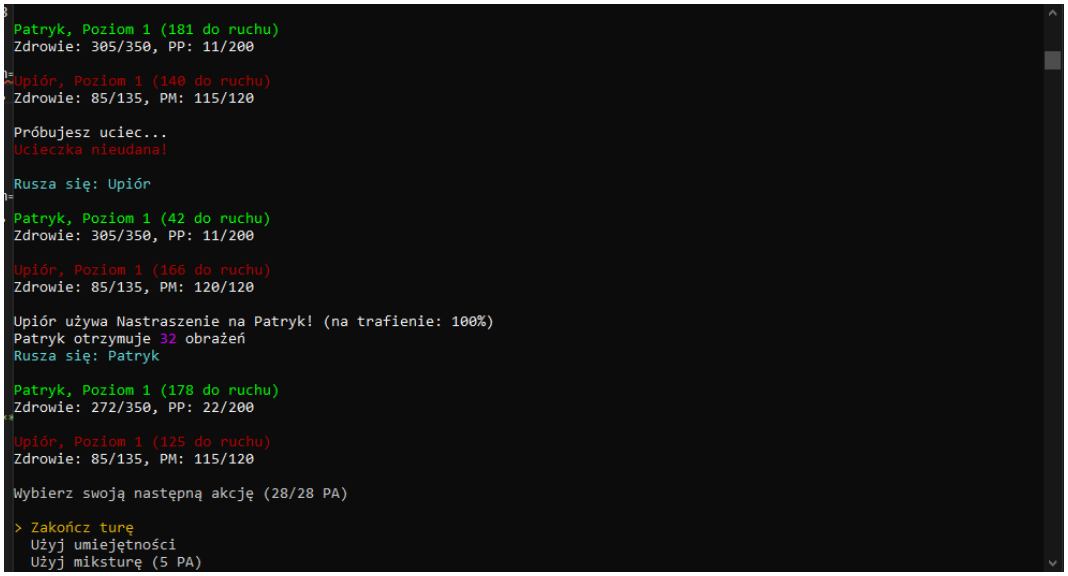


Rysunek 4.33: System walki, wybór umiejętności

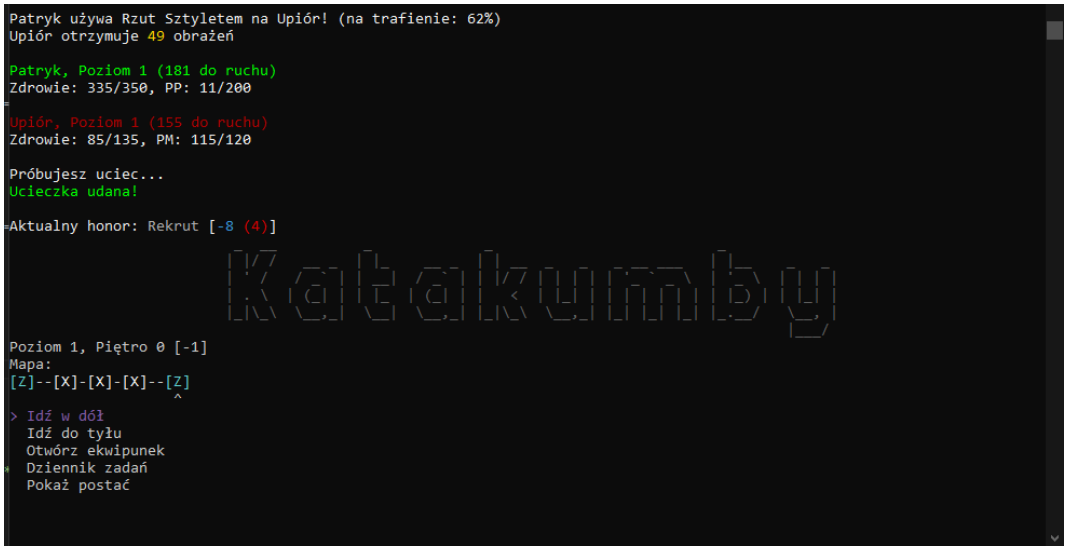


Rysunek 4.34: System walki, użycie umiejętności

- Na rysunkach 4.35 i 4.36 przedstawiono nieudaną i udaną próbę ucieczki.

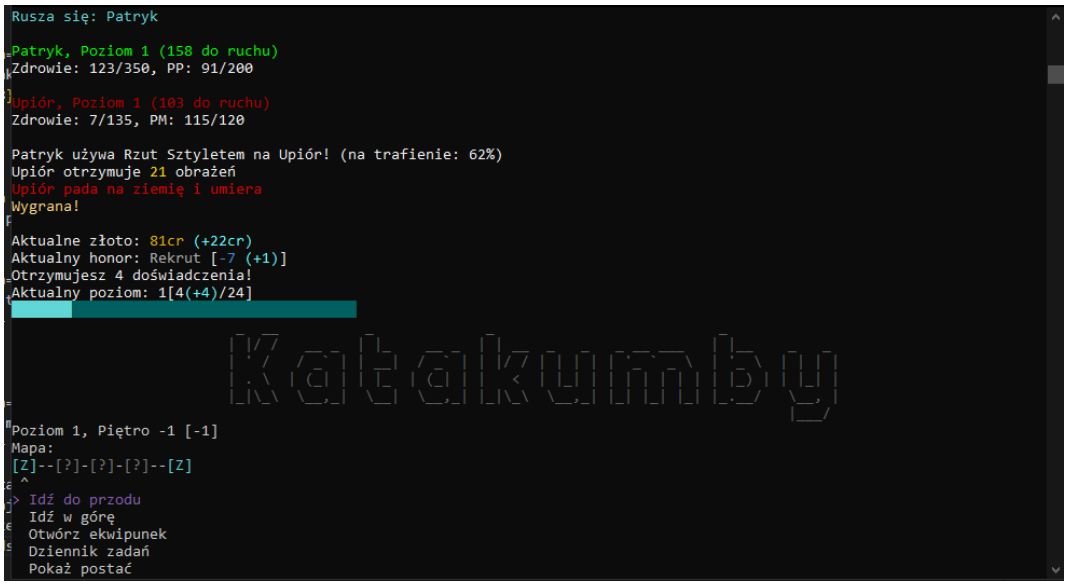


Rysunek 4.35: System walki, nieudana ucieczka



Rysunek 4.36: System walki, udana ucieczka

- Walka kończy się, gdy zdrowie jednego z jej uczestników spadnie do 0. W przykładzie ukazanym na rysunku 4.37, gracz wygrywa i otrzymuje nagrody.



Rysunek 4.37: System walki, nagrody za wygraną

Rozdział 5

Podsumowanie

Wynikiem projektu jest wysoce rozwinięta, ale prosta do obsługi gra konsolowa RPG. Prostość w obsłudze i złożoność systemów, wraz z proceduralnością wbudowaną w mechaniki gry zapewni godziny rozgrywki i zwiększy popularność gry. System zapisów gry pozwoli na tworzenie ciągłych rozgrywek, gdzie gracze angażują się w wiele poziomów i rowzijają jedną postać bardzo długo, co skutkuje satysfakcją i chęcią do dalszej gry.

5.1 Zrealizowane prace

W ramach projektu zrealizowano następujące prace:

- **Opracowanie i implementacja głównych klas:** Stworzono klasy, takie jak `PlayerCharacter`, `EnemyCharacter`, `Battle`, oraz `BattleManager`, które stanowią fundament mechaniki gry.
- **Implementacja systemu NPC:** Stworzono klasy reprezentujące postacie niezależne (NPC), takie jak `Enchanter`, `Blacksmith` oraz `Alchemist`. Każda z tych klas posiada unikalne metody interakcji z graczem oraz system lojalności.
- **Zarządzanie ekwipunkiem:** Opracowano system zarządzania ekwipunkiem gracza oraz NPC, umożliwiający kupno, sprzedaż i tworzenie przedmiotów.
- **System walki:** Zaimplementowano mechanikę walki, w tym klasy `Battle`, `BattleManager` oraz `BattleUser`, które zarządzają przebiegiem walki oraz interakcjami między postaciami.
- **Zarządzanie danymi:** Opracowano system zapisywania i ładowania stanu gry, co pozwala graczom na kontynuowanie rozgrywki w późniejszym czasie.
- **Wprowadzenie mechaniki rozwoju postaci:** Wprowadzono system rozwoju postaci, w tym zdobywania doświadczenia, honoru oraz modyfikatorów, które wpływają na interakcje w grze.
- **Implementacja systemu zadań:** Opracowano system zadań, które dodają dodatkowy cel rozgrywce i dodatkowe nagrody dla graczy.
- **Eksploracja lochów:** Wprowadzono system eksploracji proceduralnie generowanych lochów, które zmniejszają powtarzalność rozgrywki.

5.2 Planowane prace rozwojowe

W przyszłości planowane są następujące prace rozwojowe:

- **Rozbudowa systemu questów:** Dodanie nowych zadań oraz mechanik związanych z ich realizacją, co zwiększy interaktywność i zaangażowanie graczy. Wprowadzenie angażującej i oryginalnej fabuły głównej.
- **Optymalizacja algorytmów walki:** Udoskonalenie sztucznej inteligencji przeciwników oraz wprowadzenie nowych umiejętności i strategii walki, takich jak: umiejętności pasywne, fazy bossów, walka drużynowa.
- **Optymalizacja UX/UI:** Dodanie większej ilości informacji zwrotnej dla gracza przy interakcjach z interfejsem, Wprowadzenie bardziej czytelnego interfejsu.
- **System osiągnięć:** Implementacja systemu osiągnięć, który nagradza graczy za różne osiągnięcia w grze, co zwiększy motywację do eksploracji i interakcji z grą.
- **Drzewka umiejętności:** Implementacja drzewek umiejętności, które pozwolą graczom na wybranie ścieżki rozwoju i zwiększą kompleksowość rozwoju postaci.

Bibliografia

[1] https://kanbanflow.com/ z dnia 16.02.2025

Spis rysunków

Spis tabel