Steven Braun

Matrikelnummer: 6850277

1. Konzeptbeschreibung

Das Projekt ist ein einfaches textbasiertes Dungeon-Spiel in Python. Der Spieler durchläuft nacheinander beliebig viele Räume, in denen jeweils ein Monster wartet.

1.1 Initialisierung

Beim Spielstart wird der Spieler nach folgenden Eingaben gefragt:

- Name des Spielers
- Anzahl der Räume (mindestens 1)
- Für jeden Raum: Name des Monsters

Der Spieler erhält 100 Lebenspunkte und eine zufällige, aber für das gesamte Spiel feste Stärke zwischen 40 und 80 Punkten. Jedes Monster erhält zufällige Lebenspunkte (40-80) und Stärke (20-40).

1.2 Spielablauf

In jedem Raum kann der Spieler kämpfen oder fliehen:

- **Kampf**: Spieler und Monster tauschen Schaden aus (Höhe = jeweilige Stärke). Der Kampf wiederholt sich, bis einer von beiden keine Lebenspunkte mehr hat.
- **Flucht**: Spieler erhält 10 Lebenspunkte Schaden und wechselt zum nächsten Raum.

1.3 Spielmechanik: Lebenspunkte-Regeneration

Eine zentrale Spielmechanik ist die Regeneration nach erfolgreichen Kämpfen: Wenn der Spieler ein Monster besiegt, erhält er Lebenspunkte in Höhe der Hälfte des dem Monster zugefügten Schadens zurück. Beispiel: Monster erleidet 60 Schadenspunkte, dann erhält der Spieler 30 Lebenspunkte. Dies ermöglicht es, längere Dungeon-Durchläufe zu überleben.

1.4 Spielende

- **Verloren**: Lebenspunkte des Spielers fallen auf 0 oder darunter
- **Gewonnen**: Alle Räume erfolgreich durchlaufen (Monster besiegt oder erfolgreich geflohen)

2. Struktur / Designskizze

2.1 Klassenübersicht

- **Game**
  - Aufgabe: Steuert den kompletten Spielablauf (Spielschleife), fragt Eingaben ab und löst Aktionen aus.
  - Verwendet: Player, RoomManager

- **RoomManager**
    - Aufgabe: Verwaltet die Raum-Reihenfolge, liefert den aktuellen Raum und entscheidet, wann das Spiel endet.
    - Verwaltet: Liste von Room
- **Room**
    - Aufgabe: Enthält genau ein Monster und kapselt die Logik für Kampf und Flucht.
- **Player**
    - Aufgabe: Repräsentiert den Spieler (Name, Lebenspunkte, feste Stärke) und führt Angriffe aus.
- **Monster**
    - Aufgabe: Gegner im Raum (Lebenspunkte, Stärke), kann angreifen und besiegt werden.

2.2 Beziehungen

- **Game** → nutzt **RoomManager**, um den aktuellen Raum zu bekommen und zum nächsten Raum zu wechseln.
- **Game** → nutzt **Player**, um Status und Lebenspunkte zu verwalten.
- **RoomManager** → verwaltet **Rooms** und liefert den aktuellen Room.
- **Room** enthält genau ein **Monster**.
- **Room** interagiert mit **Player**:
    - bei **fight** tauschen Player und Monster Schaden aus
    - bei **escape** erhält der Player eine feste Fluchtstrafe (10 HP)

2.3 Kurzablauf

1. **Game** initialisiert **Player** (mit Nutzereingabe für Namen) und **RoomManager** (mit Nutzereingaben für Räume und Monster).
2. Solange das Spiel nicht vorbei ist (game_over = False und player.health > 0), holt **Game** den aktuellen **Room** über den **RoomManager**.
3. Der Spieler wählt Kampf (f) oder Flucht (e).
4. **Room** führt die Logik aus (Kampf/Flucht) und passt den **Player** und das **Monster** an.
5. **RoomManager** wechselt zum nächsten Raum oder setzt das Spielende (game_over = True).
6. Nach Spielende wird eine Sieg- oder Niederlagen-Meldung ausgegeben.

3. Zentrale Klassen (Zweck, Attribute, Methoden)

Game

- **Zweck:** Steuert den Spielablauf (Start, Raumerzeugung, Aktionen, Spielende)
- **Attribute (privat):**

    - _player (Player): Der Spieler
    - _room_manager (RoomManager): Verwaltet die Räume
    - _num_rooms (int): Anzahl der Räume
- **Konstante:** MIN_ROOMS = 1
- **Methoden:**
    - __init__(): Initialisiert Game-Objekt
    - start(): Fragt Spielername und Raumanzahl ab, erstellt Player
    - run_game(): Hauptspielschleife mit Raum-Interaktion

RoomManager

- **Zweck:** Verwaltet die Räume und den Spielfortschritt.
- **Attribute (privat):**
    - _rooms (list): Liste aller Räume
    - _current_room_index (int): Index des aktuellen Raums
    - _game_over (bool): Flag für Spielende
- **Properties:**
    - current_room_index: Gibt Index zurück
    - game_over: Gibt game_over Flag zurück
- **Methoden:**
    - __init__(rooms): Initialisiert mit Raumliste
    - get_current_room(): Gibt aktuellen Raum zurück (oder None)
    - move_to_next_room(): Wechselt zum nächsten Raum oder setzt game_over
    - is_game_over(): Prüft, ob Spiel vorbei ist

Room

- **Zweck:** Repräsentiert einen Raum mit Monster und Kampf/Flucht-Logik.
- **Attribute (privat):**
    - _name (str): Name des Raums
    - _monster (Monster): Das Monster in diesem Raum
- **Properties:**
    - name: Gibt Raumnamen zurück
    - monster: Gibt Monster zurück
- **Methoden:**
    - __init__(name, monster): Initialisiert Raum mit Name und Monster
    - get_status(): Gibt Raum- und Monster-Status zurück
    - fight_monster(player): Führt Kampfrunde durch, bei Sieg: HP-Regeneration
    - escape_room(player): Spieler flieht, erhält 10 HP Schaden

Player

- **Zweck:** Repräsentiert den Spieler mit allen spielrelevanten Attributen und Aktionen.
- **Attribute (privat):**
  - `_name` (str): Name des Spielers
  - `_health` (int): Lebenspunkte (Standard: 100, Maximum: 100)
  - `_strength` (int): Angriffsstärke (zufällig 40-80, wird bei Initialisierung gesetzt und bleibt konstant)
- **Konstanten:**
  - `MAX_HEALTH = 100`
  - `MIN_STRENGTH = 40`
  - `MAX_STRENGTH = 80`
- **Properties:**
  - `name`, `health`, `strength`: Ermöglichen lesenden Zugriff auf private Attribute
- **Methoden:**
  - `__init__(name, health=MAX_HEALTH)`: Initialisiert Spieler mit Name, Standard-HP und zufälliger fester Stärke
  - `__str__()`: Gibt zurück: "Player has ... strength and ... health."
  - `attack()`: Gibt die feste Stärke des Spielers zurück
  - `take_damage(damage)`: Reduziert Health (Minimum: 0)
  - `regain_health(health_regain)`: Erhöht Health (Maximum: MAX_HEALTH)
  - `get_status()`: Gibt formatierten Status-String zurück

Monster

- **Zweck:** Repräsentiert das Monster im Raum mit zufälligen Startwerten.
- **Attribute (privat):**
  - `_name` (str): Name des Monsters
  - `_health` (int): Lebenspunkte (zufällig 40-80 inkl.)
  - `_strength` (int): Angriffsstärke (zufällig 20-40 inkl.)
- **Konstanten:**
  - `MIN_MONSTER_HEALTH = 40`
  - `MAX_MONSTER_HEALTH = 80`
  - `MIN_MONSTER_STRENGTH = 20`
  - `MAX_MONSTER_STRENGTH = 40`
- **Properties:**
  - `name`, `health`, `strength`: Ermöglichen lesenden Zugriff
- **Methoden:**

- `__init__(name)`: Initialisiert Monster mit zufälligen HP und Stärke
- `attack()`: Gibt Stärke zurück
- `take_damage(amount)`: Reduziert Health (Minimum: 0)
- `is_alive()`: Prüft, ob Monster noch lebt (health > 0)
- `get_status()`: Gibt formatierten Status-String zurück

4. Testkonzept

Getestete Klassen

- **Player** (tests/TestPlayer.py): 10 Testfälle
- **RoomManager** (tests/TestRoomManager.py): 6 Testfälle
- **Room** (tests/TestRoom.py): 3 Testfälle

Hilfsklassen für Tests

- **MockRoom** (in TestRoomManager.py): Simuliert Räume für RoomManager-Tests
- **MockMonster und MockPlayer** (in TestRoom.py): Simulieren Spieler und Monster mit festen Werten für deterministische Tests

Abgedeckte Fälle

Player

1. `test_initial_health` - Standardwert 100
2. `test_custom_health` - Benutzerdefinierte HP
3. `test_initial_strength_in_range` - Stärke liegt zwischen 40-80
4. `test_take_damage` - Normale Schadensberechnung
5. `test_take_damage_not_negative` - Untergrenze 0
6. `test_regain_health` - Normale Heilung
7. `test_regain_health_not_above_max` - Obergrenze 100
8. `test_attack_returns_strength` - attack() gibt feste Stärke zurück
9. `test_str_representation` - **str**() Format korrekt
10. `test_get_status` - Statusausgabe enthält alle Infos

RoomManager

1. `test_initial_room_index_is_zero` - Startindex = 0
2. `test_get_current_room_returns_first_room` - Erster Raum korrekt
3. `test_move_to_next_room_increments_index` - Index erhöht sich
4. `test_get_current_room_after_move` - Richtiger Raum nach Wechsel
5. `test_game_over_after_last_room` - game_over Flag nach letztem Raum
6. `test_get_current_room_with_empty_list` - Leere Liste liefert None

Room

1. `test_fight_monster_player_wins` - Kampf mit Spieler-Sieg

2. `test_escape_room_deals_10_damage` - Flucht verursacht 10 HP Schaden
3. `test_fight_monster_hp_regeneration` - HP-Regeneration nach Sieg (Spieler erhält halben Schaden zurück)

Testausführung

Alle Tests gleichzeitig:
python -m unittest discover -s tests -p "Test*.py" -v

5. Reflexion

Größte Herausforderungen

Meine größte Herausforderung war es, die Interaktion der Klassen sauber aufeinander abzustimmen, vor allem bei der Integration in der Game-Klasse. Zwar war es sehr hilfreich, vorab eine Konzeptbeschreibung zu erstellen und eine Struktur zu skizzieren, sowie für jede Klasse grob Zweck, Attribute und Methoden festzuhalten. In der Umsetzung war es dann aber doch deutlich schwieriger, weil sich erst beim Zusammensetzen gezeigt hat, wie viele Abhängigkeiten tatsächlich entstehen.

Besonders herausfordernd war es, den Überblick darüber zu behalten, welche Klasse wofür zuständig ist und wann welche Methode aufgerufen werden muss. Gerade in der Spielschleife musste ich mehrere Bedingungen gleichzeitig berücksichtigen (zum Beispiel, ob das Spiel schon vorbei ist und ob der Spieler noch Lebenspunkte hat). Außerdem hat mich die objektorientierte Denkweise anfangs verwirrt, weil ich mich oft gefragt habe, ob ich gerade auf Attribute des Spielers, des Monsters, des Raums oder des RoomManagers zugreife. Die vielen self.-Zugriffe haben dabei zusätzlich für Verwirrung gesorgt.

Am Ende hat mir meine vorher erstellte Struktur aber sehr geholfen: Dadurch konnte ich die Verantwortlichkeiten der Klassen wieder nachvollziehen, Fehler schneller finden und die einzelnen Teile Schritt für Schritt zu einem funktionierenden Ablauf zusammenbauen.

Lernprozess und Überarbeitung

Nach Fertigstellung meiner ersten Version habe ich durch intensivere Auseinandersetzung mit den Prüfungsanforderungen und OOP-Prinzipien mehrere wichtige Erkenntnisse gewonnen, die mich zu einer grundlegenden Überarbeitung des Codes geführt haben:

Erkenntnisse aus der ersten Version

In meiner ersten Implementierung hatte ich einige konzeptionelle Fehler:

- Player.strength fehlte als Attribut: Ich hatte attack() so implementiert, dass bei jedem Aufruf ein neuer Zufallswert generiert wurde. Erst beim erneuten Lesen der Anforderungen wurde mir klar, dass die Stärke bei Initialisierung gesetzt und dann konstant bleiben soll.

- Öffentliche statt private Attribute: Ich verwendete self.name statt self._name. Mir war zunächst nicht bewusst, dass die Aufgabenstellung explizit private Attribute fordert und dies eine wichtige OOP-Konvention in Python ist.

Durchgeführte Überarbeitungen

Nach diesen Erkenntnissen habe ich folgende Änderungen vorgenommen:

1. Player-Klasse komplett überarbeitet:

- self._strength als Attribut im Konstruktor hinzugefügt (random.randint(40, 80))
- attack() gibt nun die feste Stärke zurück, nicht mehr einen Zufallswert
- **str**() angepasst, um das exakte Format der Aufgabenstellung zu erfüllen: "Player <name> has <strength> strength and <health> health."
- Alle Attribute auf privat umgestellt (_name, _health, _strength)
- Properties (@property) hinzugefügt, um kontrollierten Lesezugriff zu ermöglichen

2. Konsequente Kapselung in allen Klassen:

- In Monster, Room, RoomManager und Game alle Attribute auf privat umgestellt
- Properties für notwendige externe Zugriffe implementiert
- Dadurch bessere Einhaltung von OOP-Prinzipien (Kapselung)

3. TestRoomManager.py komplett neu geschrieben:

- MockRoom-Klasse erstellt, um RoomManager isoliert zu testen
- 6 sinnvolle Tests implementiert (Initialisierung, Raumwechsel, Spielende, Edge Cases)
- Tests decken jetzt die RoomManager-Funktionalität ab

4. TestRoom.py neu erstellt:

- Test-Datei für die kritische Kampflogik
- MockPlayer und MockMonster für deterministische Tests
- Tests für Kampf, Flucht und HP-Regeneration

5. Code-Qualität verbessert:

- PEP 8 Indentation (4 Spaces statt 2)
- Docstrings für alle Klassen und Methoden hinzugefügt
- Einheitliche Formatierung

Was ich dabei gelernt habe

Diese Überarbeitung war eine wertvolle Lernerfahrung:

- Genauigkeit ist wichtig: Beim ersten Lesen der Anforderungen hatte ich Details übersehen. Erst das systematische Abarbeiten hat die Lücken aufgezeigt.
- OOP-Prinzipien haben Gründe: Private Attribute und Properties erschienen mir anfangs überflüssig, aber sie verbessern tatsächlich die Wartbarkeit und Erweiterbarkeit des Codes.
- Tests brauchen Aufmerksamkeit: Copy-Paste bei Tests ist gefährlich, weil man schnell vergisst, sie anzupassen. Echte Tests zu schreiben dauert länger, aber man lernt dabei viel über die getestete Klasse.
- Refactoring lohnt sich: Obwohl die erste Version funktioniert hat, ist die überarbeitete Version deutlich sauberer, besser strukturiert und entspricht den professionellen Standards.

Aufgetretene Fehler/Bugs und Lösung

Bugs aus der ersten Implementierung

Bug 1: "AttributeError: 'Room' object has no attribute 'health'"

In fight_monster() in Room.py habe ich vergessen, dass ich auf self.monster.health zugreifen muss, nicht auf self.health. Ich versuchte auf die Health des "Rooms" zuzugreifen, statt auf die Health des Monsters.

Behebung: Ich habe alle Zugriffe auf Health in Room.py korrigiert - jetzt ist überall self.monster.health oder player.health, nicht irgendwelche falsche Attribute.

Bug 2: Logischer Fehler: Health wurde negativ

Am Anfang konnte die Health ins Minus gehen (z.B. -20). Das machte keinen Sinn.

Behebung: In Monster.py und Player.py habe ich if self.health < 0: self.health = 0 hinzugefügt, damit Health nie negativ wird.

Bug 3: "AttributeError: 'RoomManager' object has no attribute 'game_over'"

In is_game_over() in RoomManager.py versuche ich auf self.game_over zuzugreifen, aber ich habe vergessen, dieses Attribut im init() zu initialisieren.

Behebung: Ich habe self.game_over = False im init() hinzugefügt, damit is_game_over() korrekt funktioniert.

Bug 4: Fehlende Integration der Flucht-Funktion

In run_game() in Game.py habe ich vergessen, die Escape-Möglichkeit in die Methode zu integrieren. Obwohl ich die escape_room() Methode in Room.py implementiert habe, habe ich es versäumt, diese in der Spielschleife aufzurufen. Wenn der Spieler "e" eingibt, passiert nichts außer einer Fehlermeldung.

Behebung: Ich habe den else-Block in run_game() angepasst, sodass bei "e" die escape_room() Methode korrekt aufgerufen wird.

Bug 5: Spiel endet nicht nach Sieg: game_over Flag fehlt

In move_to_next_room() in RoomManager.py habe ich vergessen, self.game_over = True zu setzen, wenn der Spieler den letzten Raum besiegt. Die Schleife lief weiter, obwohl alle Räume durchlaufen waren.

Behebung: Ich habe self.game_over = True in den else-Block von move_to_next_room() hinzugefügt.

Bug 6: Keine Spielwiederholungs-Schleife

Die main.py startete das Spiel nur einmal und beendete sich dann. Der Nutzer musste das Python-Skript manuell neu ausführen.

Behebung: Eine while-Schleife in main.py hinzugefügt, die den Spieler fragt, ob er nochmal spielen möchte.

Bug 7: Keine Raumwechsel bei Flucht

Beim Drücken der Taste "e" (Escape) wurde zwar korrekt Schaden am Spieler berechnet, der Raum wurde jedoch nicht verlassen.

Behebung: Nach der Escape-Aktion habe ich eine Prüfung ergänzt, ob der Spieler noch Lebenspunkte größer 0 besitzt. Ist das der Fall, wird der Wechsel in den nächsten Raum ausgelöst.

Bugs aus der Überarbeitung

Bug 8: Player.strength fehlte als festes Attribut

Fehler: Player.attack() würfelte bei jedem Aufruf einen neuen Wert. Die Aufgabenstellung verlangte aber, dass die Stärke bei Initialisierung gesetzt und dann konstant bleibt.

Ursache: Missverständnis der Anforderung. Ich dachte, variable Angriffswerte machen das Spiel spannender.

Behebung:

- self._strength im **init**() gesetzt: self._strength = random.randint(MIN_STRENGTH, MAX_STRENGTH)
- attack() gibt jetzt nur noch self._strength zurück (nicht mehr zufällig)
- **str**() angepasst, um Stärke korrekt auszugeben

Bug 9: TestRoomManager.py war Copy-Paste von TestPlayer.py

Fehler: Beim Erstellen der zweiten Test-Datei habe ich TestPlayer.py kopiert, aber vergessen, den Inhalt anzupassen. RoomManager wurde dadurch nicht getestet.

Ursache: Zeitdruck und fehlende Kontrolle vor Abgabe.

Behebung: TestRoomManager.py komplett neu geschrieben mit MockRoom-Hilfsklasse und 6 echten RoomManager-Tests (test_initial_room_index_is_zero, test_get_current_room_returns_first_room, test_move_to_next_room_increments_index, test_get_current_room_after_move, test_game_over_after_last_room, test_get_current_room_with_empty_list).

Bug 10: Attribute waren nicht privat

Fehler: Alle Attribute waren public (self.name statt self._name), obwohl die Aufgabenstellung "private Instanzvariablen" verlangte.

Ursache: Python-Convention für private Attribute nicht beachtet.

Behebung: Alle Attribute mit Unterstrich-Präfix versehen (_name, _health, _strength, etc.) und Properties (@property) für kontrollierten Lesezugriff hinzugefügt. Alle externen Zugriffe in anderen Klassen über Properties angepasst.

Hilfsmittel

IDE-Hilfestellung (VS Code)

Die IDE wurde für Fehlermarkierungen, schnelle Navigation und automatische Formatierung genutzt. Besonders hilfreich waren die Pylint-Warnungen, die mich auf fehlende Docstrings und PEP 8 Violations hingewiesen haben.

Gelernt: Schnellere Fehlererkennung, Code-Formatierung und Strukturierung.

Vorlesungsskript

Das Vorlesungsskript diente mir als Grundlage und Orientierung für die Umsetzung der Hausarbeit. Es hat mir geholfen, neue Konzepte gezielt wahrzunehmen und gezielter weiter zu recherchieren.

Gelernt: Vor allem ein systematisches Vorgehen beim Programmieren sowie ein besseres Verständnis für grundlegende Python-Konzepte wie Klassen, Kapselung und Datenstrukturen.

Offizielle Python Dokumentation (The Python Tutorial --- Python 3.14.3 documentation)

Die offizielle Dokumentation habe ich genutzt, um Syntax und Standardfunktionen nachzuschlagen. Übernommen wurden Beispiele für Klassen, Methoden, Properties und unittest-Struktur.

Gelernt: Saubere Teststruktur, korrekte Nutzung von unittest und Properties. Die Dokumentation zu @property Decorators war besonders hilfreich bei der Überarbeitung.

Online-Foren/Stack Overflow

Hilfreich für konkrete Fragen, besonders zu Mocking von Zufallswerten, unittest-Patterns und PEP 8 Conventions. Übernommen wurden kurze Code-Ideen und Best Practices für private Attribute. Verworfen wurden komplexe Lösungen, die für das Projekt zu groß waren.

Gelernt: Tests unabhängig von Zufall machen (durch Mocking), Python-Conventions für private Attribute (_name statt name), und dass Properties eine elegante Lösung für kontrollierten Attributzugriff sind.

W3Schools -- "Python Classes and Objects"

Die Seite habe ich nach dem Lesen des Vorlesungsskripts aufgesucht, um den Aufbau von Klassen und die Verwendung des Konstruktors (**init**) nochmals an einfachen Beispielen nachzuvollziehen.

Gelernt: Den grundlegenden Aufbau von Klassen sowie den Unterschied zwischen Klassen- und Instanzattributen.

6. Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe und diese Arbeit bei keiner anderen Prüfung mit gleichem oder vergleichbarem Inhalt vorgelegt habe und diese bislang nicht veröffentlicht wurde.