

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. In jedem Raum kann man kämpfen oder fliehen. Beim Kampf tauschen Spieler und Monster Schaden aus, beim Fliehen erhält der Spieler einen festen Schaden von 10 Lebenspunkten. Ziel des Spiels ist es, alle Monster zu besiegen, bevor die Lebenspunkte des Spielers auf 0 fallen.

2. Struktur / Designskizze

Ziel: Überblick über Klassen und deren Beziehungen (für Word gut kopierbar).

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, Stärke) und führt Angriffe aus.
- **Monster**
 - Aufgabe: Gegner im Raum (Lebenspunkte, Stärke), kann angreifen und besiegt werden.

2.2 Beziehungen (Designskizze als Text)

- **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

2.3 Kurzablauf

1. **Game** initialisiert **Player** und **RoomManager**.
2. Solange das Spiel nicht vorbei ist, holt **Game** den aktuellen **Room** über den **RoomManager**.
3. Der Spieler wählt Kampf oder Flucht.
4. **Room** führt die Logik aus (Kampf/Flucht) und passt den **Player** (und ggf. das **Monster**) an.
5. **RoomManager** wechselt zum nächsten Raum oder setzt das Spielende.

3. Zentrale Klassen (Zweck, Attribute, Methoden)

Game

- **Zweck:** Steuert den Spielablauf (Start, Raumerzeugung, Aktionen, Spielende)
- **Attribute:** player, room_manager, num_rooms
- **Methoden:** init(), start(), run_game()

RoomManager

- **Zweck:** Verwaltet die Räume und den Spielfortschritt.
- **Attribute:** rooms, current_room_index, game_over
- **Methoden:** init(rooms), current_room_index(), move_to_next_room(), is_game_over()

Room

- **Zweck:** Repräsentiert einen Raum mit Monster und Kampf/Flucht-Logik.
- **Attribute:** name, monster
- **Methoden:** `init(name, monster)`, `get_status()`, `fight_monster(player)`, `escape_room(player)`

Player

- **Zweck:** Repräsentiert den Spieler und alle Spielerseitigen Kampf/-Heilaktionen.
- **Attribute:** name, health
- **Methoden:** `init(name, health=MAX_HEALTH)`, `str()`, `attack()`, `take_damage(damage)`, `regain_health(health_regain)`, `get_status()`

Monster

- **Zweck:** Repräsentiert das Monster im Raum und modelliert zufällige Startwerte für einfaches Kampfverhalten.
- **Attribute:** name, health, strength
- **Methoden:** `init(name)`, `attack()`, `take_damage(amount)`, `is_alive()`, `get_status()`

4. Testkonzept

Getestete Klassen

- **Klassen:** Player und RoomManager
 - Tests über: `TestPlayer.py` und `TestRoomManager.py`
- **Hilfsklasse im Test:** `MockRoom` (in `TestRoomManager.py`)
 - Zweck: Simuliert Räume für RoomManager-Tests

Abgedeckte Fälle

- **Player**
 - Initialwerte
 - benutzerdefinierte Health
 - Schaden nehmen
 - Untergrenze bei 0
 - Heilung
 - Obergrenze bei `MAX_HEALTH`

- Status-String
- Attacke im gültigen Zufallsbereich
- **RoomManager**
 - Startindex
 - aktueller Raum
 - Raumwechsel
 - korrekter Raum nach Wechsel
 - kein Wechsel über den letzten Raum hinaus
 - Verhalten bei leerer Raumliste (None)

Testausführung

- Direkt pro Datei (funktioniert sicher, weil beide Dateien unittest.main() enthalten):
 - python tests/TestPlayer.py
 - python tests/TestRoomManager.py

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.

Aufgetretene Fehler/Bugs und Lösung

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` Zeile 20 und `Player.py`: Zeile 18 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. Wenn der Spieler das Monster im letzten Raum besiegt:

1. `move_to_next_room()` wird aufgerufen
2. Da `current_room_index >= len(rooms) - 1`, wird nur "Congratulations!" ausgegeben
3. Der Index wird NICHT inkrementiert und `game_over` bleibt False
4. Die Schleife in `run_game()` läuft weiter, weil `is_game_over()` noch False zurückgibt
5. `get_current_room()` gibt immer noch den letzten Raum zurück
6. Das Spiel fragt weiter nach Eingabe

Behebung: Ich habe `self.game_over = True` in den `else`-Block von `move_to_next_room()` hinzugefügt. In der `Game`-Klasse

Bug 6: Keine Spielwiederholungs-Schleife

Die `main.py` startet das Spiel nur einmal und beendet sich dann. Nachdem `game.run_game()` beendet ist, hat der Spieler keine Möglichkeit, das Spiel direkt erneut zu starten. Er muss das Python-Skript manuell neu ausführen.

Behebung: Eine `while`-Schleife wird hinzugefügt, um den Spielaufruf implementieren, die den Spieler fragt, ob er nochmal spielen möchte.

Vorgehen beim Entwurf der Klassen/Struktur

Zuerst habe ich die Kernobjekte (Player, Monster, Room) festgelegt. Darauf aufbauend dann `RoomManager` als Raum-Verwaltung erstellt und `Game` als Steuerung der Abläufe. So konnte ich die Logik in kleinere, übersichtliche Klassen aufteilen.

Hilfsmittel

IDE-Hilfestellung (VS Code):

Die IDE wurde für Fehlermarkierungen und schnelle Navigation genutzt.

Gelernt: schnellere Fehlererkennung und Strukturierung.

Vorlesungsskript:

Die Vorlesungsunterlagen wurden als Grundlage für die OOP-Umsetzung genutzt. Außerdem diente das Skript als Checkliste für typische Aufgabenstellung (z.B saubere Klassenstruktur, Methodenaufteilung). Gelernt: Systematisches Vorgehen beim Programmieren und Python Grundlagenverständnis.

Offizielle Python Dokumentation([The Python Tutorial — Python 3.14.3 documentation](#)):

Die offizielle Dokumentation wurde genutzt, um Syntax und Standardfunktionen nachzuschlagen. Übernommen wurden Beispiele für Klassen, Methoden und unittest-Struktur. Gelernt: saubere Teststruktur und korrekte Nutzung von unittest.

Online-Foren/Stack Overflow:

Hilfreich für konkrete Fragen besonders zu Mocking von Zufallswerten und zu unittest-Patterns. Übernommen wurden kurze Code-Ideen. Verworfen wurden komplexe Lösungen, die für das Projekt zu groß waren. Gelernt: Tests unabhängig von Zufall machen.