Die Aufgaben könnt ihr nur machen, wenn ihr Aufgabe 1 und Aufgabe 2 fertig gestellt habt!

Aufgabe 3 – Implementierung der Gegner- und Rätselwahl

Schwierigkeit *: In eurem Klassendiagramm gibt es in der Methode *DungeonCrawl* zwei Methoden: chooseEnemy(), die den Gegner bestimmt, und die Methode chooseMystery(), die das Rätsel bestimmt.

In der Geschichte steht eine feste Reihenfolge deiner Gegner. Für die Implementierung ist es zunächst einfacher einen beliebigen Gegner zu wählen. Das gilt auch für die Rätsel. Später kann auch eine feste Auswahl stattfinden.

chooseEnemy():

- Die Methode soll zunächst eine Zufallszahl bestimmen zwischen 0 und der Anzahl deiner Gegner – 1. (Wir müssen hier beachten, dass wir bei 0 beginnen zu zählen und nicht bei 1. Wenn wir bei der Anzahl keinen abziehen, haben wir mehr Gegner zur Auswahl als tatsächlich vorhanden.)
- Erstelle eine switch-case-Anweisung, die jeweils für jeden Fall ein neues Objekt deines ausgewählten Gegners erstellt und zurückgibt.
- Bedenke auch den default-case

chooseMystery():

- Die Methode soll zunächst eine Zufallszahl bestimmen zwischen 0 und der Anzahl deiner Rätsel – 1. (Wir müssen hier beachten, dass wir bei 0 beginnen zu zählen und nicht bei 1. Wenn wir bei der Anzahl keinen abziehen, haben wir mehr Rätsel zur Auswahl als tatsächlich vorhanden.)
- Erstelle eine switch-case-Anweisung, die jeweils für jeden Fall ein neues Objekt deines ausgewählten Rätsels erstellt und zurückgibt.
- Bedenke auch den default-case

Aufgabe 4 – Implementierung der Rätsel

Unser Dungeon Crawl hat unterschiedliche Quest. Diese müssen für unser Spiel entsprechend implementiert werden.

Schwierigkeit *:

1. Wir verwenden das Konzept der abstrakten Klasse, um uns unsere Klasse *Mystery* als Vorlage für unsere tatsächlichen Rätsel zu gestalten.
    a. Wir benötigen die Attribute question und rightAnswer. Welche Datentypen müssen wir verwenden? Welche Zugriffsmodifier sind angebracht?
    b. Wir benötigen einen Konstruktor, der die Frage und die Antwort übergeben bekommt.
    c. Wir benötigen eine Methode solving(), die eine Antwort als String übergeben bekommt und einen boolean zurückgibt, ob die Antwort mit der rightAnswer übereinstimmt. Ihr habt den Vergleichsoperator

equals() kennengelernt. Hiermit habt ihr die Möglichkeiten zwei Objekte zu vergleichen. Wenn euch zusätzlich die Groß- und Kleinschreibung bei String im Vergleich egal ist, dann könnt ihr auch equalsIgnoreCase() verwenden.

   d. Zum Schluss benötigen wir eine abstrakte Methode outputQuestion().
2. Jetzt möchten wir die verschiedenen Rätsel anlegen. Im Folgenden wird beschrieben, wie ein Rätsel angelegt wird. Die anderen Rätsel sind analog anzulegen.
   a. Alles Rätsel müssen mit extends von *Mystery* erben.
   b. Du benötigst für jedes Rätsel einen Konstruktor, in dem der super-Klasse deine Frage und deine Antwort übergeben werden müssen als String.
   c. Jetzt müssen wir noch die abstrakte Methode aus *Mystery* überschreiben. Schreibe ein @Override über deine Methode outputQuestion().
   d. **Hinweis**: Du kannst die Rätsel beliebig gestalten. Das Einfachste ist einfach eine Frage und eine Antwort in einem String zu formulieren. Darüber hinaus sind dir allerdings alle Möglichkeiten offen.

Aufgabe 4 – Implementierung der Spieler und Gegner

Damit wir auch verschiedene Gegner haben und selbst auch als Spieler fungieren können, benötigen wir hierfür jeweils verschieden Objekte der Gegner und Spieler. Hierfür müssen wir die verschiedenen Klassen konstruieren.

Schwierigkeit *:

1. Wir implementieren zunächst die abstrakte Klasse *Figure*, die uns eine Struktur vorgibt, wie ein Spieler, sowie ein Gegner grundsätzlich aufgebaut werden muss. Bis hierhin gibt es kaum unterschiede zwischen den beiden Kategorien.
   a. Wir benötigen die Attribute name, health und attack. Denke an passende Zugriffsmodifier und Datentypen.
   b. Wir benötigen einen Konstruktor, der einen Namen, die Gesundheit und den Angriff übergeben bekommt.
   c. Da der Angriff eine sehr spezifische Möglichkeit der Modellierung darstellt, soll diese Methode abstrakt implementiert werden.
   d. Wir definieren uns, dass jede Figur, bei einer Gesundheit <= 0 nicht mehr lebt. Entsprechend implementieren wir uns eine Methode isDead(), die überprüft, ob eine Figur tot ist und einen Boolean zurück gibt.
   e. **Hinweis**: Falls du private Modifier gewählt hast, dann benötigst du getter und setter. Andernfalls geht’s auch ohne.
2. Als nächstes benötigen wir einen Spieler. Die Klasse *Player* muss mit extends von *Figure* erben.
   a. Wir benötigen einen Konstruktor, der einen Namen, die Gesundheit und den Angriff an die Super-Klasse übergibt.
   b. Und wir müssen die abstrakte Methode attack(Figure character) der abstrakten Klasse *Figure* implementieren. Wir definieren und hierfür zunächst eine Variable, die den Schaden des Angriffs abspeichert. Dann müssen wir der Gesundheit der übergeben Figur um den

Schaden reduzieren und eine entsprechende Konsolenausgabe für die Erzählung der Geschichte vorgeben. Ein möglicher Beispieltext, der dem Spieler in der Konsole angezeigt werden kann wäre: „Der Spieler fügt dem Feuerdrachen 5 Schäden zu!“.

3. Für den Gegner (*Enemy*) verwenden wir zunächst wieder das Konzept der abstrakten Klasse, um die Struktur der *Figure* noch etwas zu konkretisieren für die unterschiedlichen Gegner-Typen.
    a. Wir benötigen einen Konstruktor, der einen Namen, die Gesundheit und den Angriff an die Super-Klasse übergibt.
    b. Und wir müssen die abstrakte Methode attack(Figure spieler) der abstrakten Klasse *Figure* implementieren. Wir definieren und hierfür zunächst eine Variable, die den Schaden des Angriffs abspeichert. Dann müssen wir der Gesundheit der übergeben Figur um den Schaden reduzieren und eine entsprechende Konsolenausgabe für die Erzählung der Geschichte vorgeben. Ein möglicher Beispieltext, der dem Spieler in der Konsole angezeigt werden kann wäre: „Der Feuerdrache fügt dem Spieler 5 Schäden zu!“.
    c. Dann benötigen wir noch spezielle Fähigkeiten der unterschiedlichen Charaktere. Hierfür stellen wir eine abstrakte Methode specialSkill(Player player) zur Verfügung.
4. Jetzt fehlen uns noch die konkreten Gegner. Jeder der Gegner erbt von *Enemy*.
   Hinweis: Das Folgende sind nur Vorschläge zur Implementierung. Ihr könnt und dürft das gerne anpassen! 😊
    a. **VenomCreature**:
        i. Wir benötigen einen Konstruktor, der konkret den „Gegner Giftkreatur“, einen Wert für die Gesundheit bspw. 60 und einen Wert für den Angriff bspw. 15 übergibt.
        ii. Dann muss die Methode specialSkill(Player player) mit @Override überschrieben werden. Hierfür können wir dem Spieler zuerst einmal mitteilen, dass „Die Giftkreatur vergiftet dich!“. Jetzt können wir einen Vergiftungsschaden definieren von bspw. 5, der von der Gesundheit des Spielers abgezogen wird und dem Spieler mitteilen, dass „Du hast 5 Vergiftungsschaden erlitten."
    b. **FireDragon**:
        i. Wir benötigen einen Konstruktor, der konkret den „Gegner Feuerdrache“, einen Wert für die Gesundheit bspw. 70 und einen Wert für den Angriff bspw. 20 übergibt.
        ii. Dann muss die Methode specialSkill(Player player) mit @Override überschrieben werden. Hierfür können wir dem Spieler zuerst einmal mitteilen, dass „Der Feuerdrache entfacht einen Flammensturm!“. Jetzt können wir einen Feuerschaden definieren von bspw. 10, der von der Gesundheit des Drachens abgezogen wird und dem Spieler mitteilen, dass „Der Drache nimmt 10 Schaden durch den Flammensturm."
    c. **GhostWarrior**:
        i. Wir benötigen einen Konstruktor, der konkret den „Gegner Geisterkrieger“, einen Wert für die Gesundheit bspw. 50 und einen Wert für den Angriff bspw. 10 übergibt.

ii. Dann muss die Methode specialSkill(Player player) mit @Override überschrieben werden. Hierfür können wir dem Spieler zuerst einmal mitteilen, dass „Der Geisterkrieger wird unsichtbar und greift aus dem Schatten an!“. Jetzt können wir einen Schaden definieren von bspw. 20, der von der Gesundheit des Kriegers abgezogen wird und dem Spieler mitteilen, dass „Du hast den unsichtbaren Geisterkrieger getroffen!"

d. **ShadowGolem**:
   i. Wir benötigen einen Konstruktor, der konkret den „Gegner Schattengolem“, einen Wert für die Gesundheit bspw. 50 und einen Wert für den Angriff bspw. 15 übergibt.
   ii. Dann muss die Methode specialSkill(Player player) mit @Override überschrieben werden. Hierfür können wir dem Spieler zuerst einmal mitteilen, dass „Der Schattengolem nutzt seine Schattenkraft, um sich zu heilen!“. Die Gesundheit des Golems um bspw. 10 erhöhen.

e. **Titan**:
   i. Wir benötigen einen Konstruktor, der konkret den „Gegner Titan
   ii. “, einen Wert für die Gesundheit bspw. 150 und einen Wert für den Angriff bspw. 40 übergibt.
   iii. Dann muss die Methode specialSkill(Player player) mit @Override überschrieben werden. Hierfür können wir dem Spieler zuerst einmal mitteilen, dass „Der Titan schlägt mit seiner riesigen Faust!“. Jetzt können wir einen Schaden definieren von bspw. 30, der von der Gesundheit des Spielers abgezogen wird und dem Spieler mitteilen, dass „Du hast 30 Schaden durch den Titanenangriff erlitten."

Aufgabe 4 – Implementierung des Spielablaufs

Schwierigkeit *: Der Spielablauf findet nach unserem geplanten Vorgehen aus Aufgabe 1 (dem Entscheidungsbaums) statt. Implementiert wird das Vorgehen in der Klasse *DungeonCrawl* und dessen main-Methode.

Hinweis: Du kannst den Ablauf auch individuell verändern. Hier stehen dir alle Möglichkeiten offen!

- Wir benötigen zum Einlesen für den Spieler zunächst einen Scanner.
- Dann muss ein Player-Objekt erstellt werden, welches einen Namen, eine initiale Gesundheit besitzt, sowie einen Wert für den Angriff.
- Jetzt muss der erste Block des Storytellings kommen, indem du dem Spieler in den Dungeon einführst. Ein möglicher Beispiel-Text wäre:

```
System.out.println("Du stehst vor dem Eingang des Dungeons.");
System.out.println("Eine massive Eisentür und eine hölzerne Tür
stehen dir offen.");
System.out.println("1. Gehe durch die Eisentür.");
System.out.println("2. Versuche, die hölzerne Tür zu öffnen.");
```

- Lese die Entscheidung des Spielers aus und speichere sie in einer Variablen.
- Falls die Entscheidung 1 lautet:

    - Dann soll ein Gegner mit Hilfe der chooseEnemy()-Methode gewählt werden.
    - Der Spieler soll informiert werden gegen welchen Gegner er spielt.
    - Es soll der Spieler gefragt werden, ob er Kämpfen möchte oder nicht. Die Entscheidung des Spielers soll ausgelesen und in einer Variablen gespeichert werden.
    - Möchte der Spieler kämpfen, dann…
        - Solange der Player oder Gegner nicht tot ist, soll der Player den Gegner angreifen.
            - Wenn der Gegner dann nicht tot ist, dann soll der Gegner den Player angreifen und seine speziellen Fähigkeiten anwenden.
        - Wenn der Player gestorben ist, dann soll der Spieler informiert werde, dass er gestorben und das Spiel vorbei ist. Das Spiel soll hier dann auch enden.
        - Andernfalls soll der Spieler informiert werden, dass er gegen seinen Gegner gewonnen hat.
    - Möchte der Spieler nicht kämpfen, so soll der Spieler informiert werden, dass er geflohen ist. Damit soll das Spiel enden.
    - Falls es eine andere Eingabe gab, so soll dem Spieler mitgeteilt werden, dass es eine ungültige Eingabe gab und das Spiel (vorerst) enden. **Alternativ**: Erneute Frage.
- Fiel die Entscheidung auf 2 (die Holztür), dann soll die Warnung an den Spieler kommen, dass er in einen Raum voller Fallen gelangt.
    - Mit Hilfe der Methode chooseMystery() soll ein Rätsel gewählt werden.
    - Der Spieler soll informiert werden, dass er einen geheimen Raum mit einem Rätsel erreicht hat.
    - Die Rätselfrage soll dem Spieler gestellt werden.
    - Leere den Zwischenspeicher des Scanners mit einer nextLine()
    - Speichere die Lösungs-Eingabe des Spielers in einer Variablen.
    - Falls die Antwort richtig ist dann soll der Spieler darüber informiert werden.
    - Falls die Antwort falsch war…
        - Muss der Spieler über die falsche Antwort informiert werden
        - Die Gesundheit auf 0 gesetzt werden
        - Das Spiel beendet werden
        - **Hinweis**: Alternativ kann hier auch bspw. eingebaut werden, dass der Spieler schwerverletzt überlebt o.Ä.
- Falls es eine andere Eingabe gab, so soll dem Spieler mitgeteilt werden, dass es eine ungültige Eingabe gab und das Spiel (vorerst) enden. **Alternativ**: Erneute Frage.
- Jetzt triffst du erneut auf einen Gegner. Implementiere diesen Schritt analog zu der ersten Begegnung mit einem Gegner. (**Hinweis**: Ggf. macht es Sinn doppelten Code in einer externen Methode auszulagern.)
- Leider trifft der Spieler noch ein drittes Mal auf einen anderen Gegner. Implementiere diesen Schritt analog zu der ersten Begegnung mit einem Gegner. (**Hinweis**: Ggf. macht es Sinn doppelten Code in einer externen Methode auszulagern.)

- Jetzt kommt der Spieler in die Dunkelheit und sieht das Artefakt. Hier ist weniger Spiellogik verankert, sondern mehr Storytelling. Eine mögliche Implementierung könnte wie folgt aussehen:

```
- // Kapitel 4: Die Entscheidung über das Artefakt
  System.out.println("Du erreichst das Erbe der Dunkelheit, ein
  mächtiges Artefakt in einem glänzenden Kristall.");
  System.out.println("Der Dunkle Wächter tritt vor und stellt dir eine
  Frage:");
  System.out.println("\"Bist du bereit, den Preis der Macht zu zahlen?
  Der Kristall verleiht dir ungeahnte Kräfte, aber er nimmt auch etwas
  von dir.\"");
  System.out.println("1. Nimm das Artefakt.");
  System.out.println("2. Lasse das Artefakt zurück.");

  entscheidung = scanner.nextInt();
  if (entscheidung == 1) {
      System.out.println("Du hast das Artefakt genommen und fühlst
  plötzlich die Macht in dir aufsteigen!");
      System.out.println("Doch du merkst, dass du etwas Wichtiges
  verloren hast... Deine Freiheit.");
      System.out.println("Du bist nun ein mächtiger, aber gefangener
  Wächter des Dungeons.");
  } else if (entscheidung == 2) {
      System.out.println("Du hast das Artefakt zurückgelassen und den
  Dungeon verlassen.");
      System.out.println("Du hast dich entschieden, den Preis der Macht
  nicht zu zahlen.");
  } else {
      System.out.println("Ungültige Eingabe.");
  }

  System.out.println("Das Spiel ist zu Ende.");
```

Aufgabe 5 – Spiele dein Spiel

Aufgabe 6 – Implementierung von Variationen im Spielablauf

Du kannst nun deine Implementierung des Spiels so anpassen, dass der Spielablauf nach deinen Wünschen geregelt wird. Eine Möglichkeit wäre z.B. noch weitere Rätsel mit einzubeziehen.