**Thema 8: Objektorientertes Modellieren und Programmieren**

**8.0 Arbeiten mit einem Framework**

Ein Bild, das Bild, Fiktive Gestalt, Cartoon, Zeichnung enthält.

KI-generierte Inhalte können fehlerhaft sein.

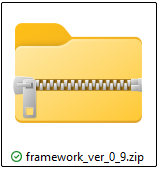
In diesem Thema werden wir mit **virtuellen Figuren** arbeiten, die sich einer **virtuellen Welt** bewegen, wie es auch bei den meisten **Computer- und Videospielen** der Fall ist. Mittelpunkt des Spiels ist der von uns kontrollierte **namenlose Held**, der in einem Level alle Herzen einsammeln muss und dabei verschiedene Hindernisse überwinden muss.

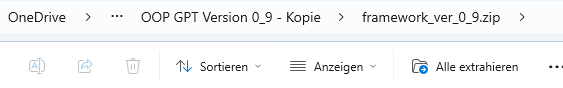
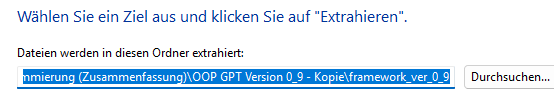
Ein Bild, das Bild, Zeichnung, Menschliches Gesicht, Kunst enthält.

KI-generierte Inhalte können fehlerhaft sein.Um uns auf die **wesentlichen Aspekte** der **objektorientierten** **Programmierung** zu konzentrieren, werden Sie nur Teile eines bereits vorhandenen Programms ergänzen. Dies nennen wir auch **Framework** (*zu* *dt. Rahmen).* Dies ist aus zwei Gründen ähnlich zu realen Programmierarbeiten:

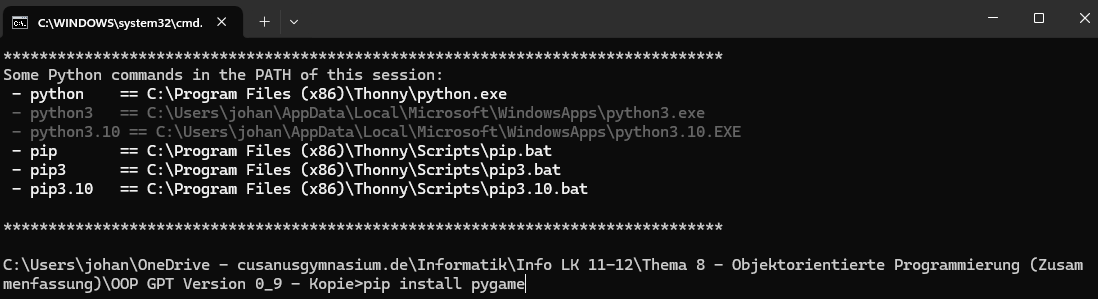
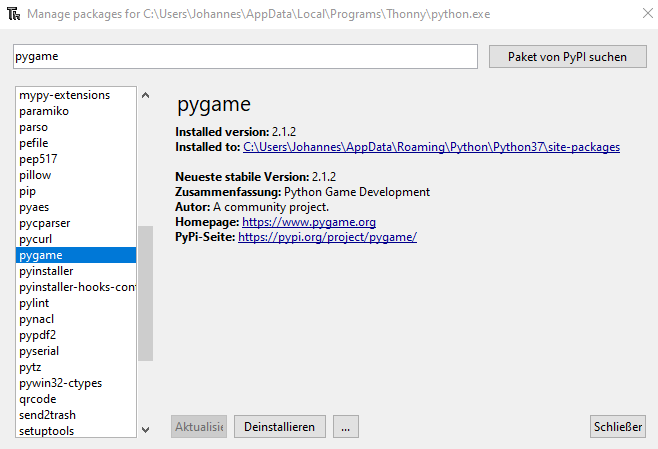
1. Sie arbeiten oft **im Team** und arbeiten nur an einem **Teil** des ganzen Programms
2. **Spiele-Engines** wie Unity, Unreal oder Godot erleichtern das Erstellen von Spielen, indem z.B. das Rendering (*dt. Darstellen*) der Welten und Figuren bereits implementiert ist

**Aufgabe 0** – *Herunterladen des Frameworks*

1. **Laden** Sie den Ordner framework.zip von der OSS **herunter**. Sie können alle Aufgaben mit dem gleichen Framework bearbeiten und müssen dies nur bei Updates neu herunterladen.
2. **Finden** Sie die heruntergeladene Datei im Datei-**Explorer**. Sie sollte so aussehen: 
3. **Entpacken** Sie die Datei, indem Sie einen Doppelklick auf die Datei machen und dann Alle Extrahieren wählen. Wählen Sie Durchsuchen und wählen Sie dann einen Ordner auf Ihrem Laufwerk aus. Die folgende Bilderstrecke hilft Ihnen.



Öffnen Sie die aufgaben.py mit Thonny. **Zu Beginn jeder Stunde** müssen Sie nun pygame neu installieren. Klicken Sie dazu auf Werkzeuge > Verwalte Pakete und suchen Sie nach pygame. Installieren Sie dieses Modul. Alternativ können Sie auf Werkzeuge > Öffne System Terminal klicken und dort den Befehl pip install pygame eingeben und **mit** **Enter bestätigen**. Danach können Sie das Terminal oder den Paketmanager schließen.



1. **Führen** Sie jetzt die Datei mit **F5** oder dem **Play-Symbol ** aus. Es sollte das folgende Fenster erscheinen:

Ein Bild, das Screenshot, Text, Grafiksoftware enthält.

KI-generierte Inhalte können fehlerhaft sein.

**Aufgabe 1** – *Die Umgebung kennenlernen*

Es ist stets das Ziel des Spiels, mit dem Helden alle Herzen im Level einzusammeln. Um den Weg in einem Level zu planen, dürfen Sie dies zunächst mit der Tastatur umsetzen.

1. Bewegen Sie sich mit den **Pfeiltasten ⬆️ ⬇️ ⬅️ ➡️** durch das Level. Testen Sie zusätzlich die Tasten Enter, Leeraste, C, F und V an verschiedenen Orten im Level. Achten Sie auf die Konsole und prüfen Sie insbesondere den Unterschied zwischen der Tür und dem Tor. Ergänzen Sie:

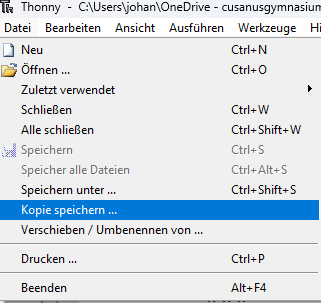
* Enter bewirkt:
* Leertaste bewirkt:
* C bewirkt:
* V bewirkt:
* F bewirkt:

1. Sammeln Sie alle Herzen im Level ein.

**Aufgabe 2** – *Den Helden programmieren*

Wir beginnen mit einem einfachen Level, das nun durch **Programmieren des Helden** gelöst werden soll.

1. Der Befehl level.lade(0) lädt das Demo-Level mit der Nummer **0**. Ändern Sie die **0** zu einer **1**, um das erste Level zu laden.
2. Führen Sie das Programm einmalig aus, damit Sie das Level sehen können.
3. Suchen Sie im Programm den Befehl held.geh(). Ab dieser Stelle können Sie Ihre eigenen Programme implementieren, um den Helden zu steuern. Als letzter Befehl muss immer framework.starten() ausgeführt werden, da sonst das Fenster einfriert.
4. Unser Held kennt die Befehle geh(), links(), rechts() und nimm\_herz(). Schreiben Sie ein Programm, um das Level zu lösen.



1. Speichern Sie die Lösung für Level 1 unter dem Dateinamen **level1.py** ab, indem Sie in Thonny **Datei > Kopie speichern** wählen.
2. Da Sie eine Kopie gespeichert haben, ist weiterhin aufgabe.py geöffnet. Stellen Sie nun die ursprüngliche Datei wieder her, indem Sie Ihren Teil des Programms löschen.
3. Lösen Sie jetzt Level 2 als Übung.

**Aufgabe 3** – *Andere Objekte*

1. Laden Sie jetzt **Level 3**.

Dort finden Sie neben dem Helden auch noch eine weitere Figur, nämlich den Knappen. Dieser lässt sich wie der Held bewegen, wird jedoch über die Variable knappe angesprochen.

1. Sammeln Sie mithilfe des Knappen und des Helden alle Herzen ein.
2. Laden Sie jetzt **Level 4**. Leider scheint das level unlösbar zu sein, deswegen bedienen wir uns einer kleiner Mogelei. Vielleicht lässt sich der Orc ja über die Variable monster ansprechen…

**Aufgabe 4** – *Reflexion*

Welche Rückschlüsse können Sie über die Funktionsweise der Methoden geh(), links() und rechts() des Knappen ziehen? Überlegen Sie dazu noch einmal, wie die Bewegung ohne den objektorientierten Ansatz umgesetzt wurde. Welche Voraussetzungen müssen erfüllt sein, damit die Methoden genau das gewünschte Verhalten zeigen und sich z.B. nur der Held bewegt, nicht aber der Knappe.

**Aufgabe 5** – *Schleifen verwenden*

Je größer das Level ist, desto aufwendiger ist auch die Lösung des Problems. Daher möchten wir die Lösung (falls möglich) effizienter gestalten. Dies erreichen wir durch Nutzen von Schleifen, z.B. der for-Schleife.

*Erinnerung: Die Schleife for i in range(5): führt eine Wiederholung 5-mal aus.*

1. Nutzen Sie Schleifen, um **Level 5** und **Level 6** effizient zu lösen.
2. Sie können auch Unterprogramme verwenden, um vorkommende Muster wiederholt abrufen zu können. Implementieren Sie eine Methode lauf(n), welche den Helden n Schritte weit gehen lässt.

**Expertenaufgabe**: Implementieren Sie eine Methode lauf(n, char), welche den übergebenen Charakter (held, knappe oder monster) die n Schritte laufen lässt.

**Aufgabe 6** – *Weitere Methoden*

Held und Knappe besitzen noch die Fähigkeiten lese\_spruch(), bediene\_tor() und spruch\_sagen(). Nutzen Sie diese zusätzlich zu Schleifen und Unterprogrammen, um **Level 7** und **Level 8** zu lösen.

**8.1 Das Prinzip der Objektorientierung**

Grundsätzlich lassen sich Programmsysteme auf zweierlei Weise strukturieren:

**8.1.1 Funktionale Abstraktion**

Dies ist die bisher von uns verwendete Art zu programmieren. Das **gesamte Programm** wird als ein großer Algorithmus aufgefasst, der mithilfe von schrittweiser Verfeinerung der Algorithmen bis auf elementare Anweisungen zerlegt wird. Prozeduren werden in Unterprozeduren zerlegt.

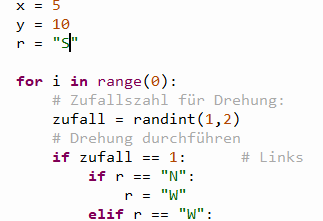
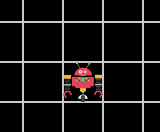


Funktion

Dabei müssen dann die Daten immer wieder durchgereicht werden. Daten und Algorithmen werden bei der Strukturierten Programmierung getrennt behandelt.

**Aufgabe 1** – *Rückblick*

Erinnern Sie sich zurück an unseren Exkurs, in welcher ein Roboter einen „Random-Walk“ durchgeführt hat. Dies war grundsätzlich machbar, führte jedoch an dem Zeitpunkt zu einem Problem, wenn mehrere Roboter diesen Algorithmus durchführen sollten.



Beschreiben Sie diese Problematik kurz:

**8.1.2 Prinzip der Datenabstraktion**

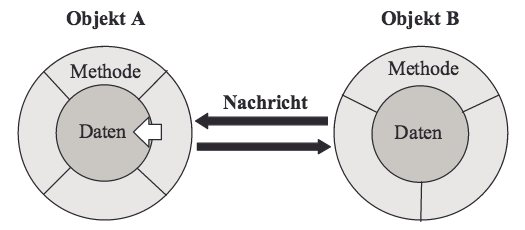
Bei diesem Programmierstil stehen die Datenstrukturen im Mittelpunkt der Betrachtung und sie haben eigene Algorithmen zur Verfügung, die sie zur Erledigung ihrer Aufgaben und Änderung ihres Zustands brauchen. Diese Sichtweise ist Grundlage der **Objektorientierten Programmierung** (kurz **OOP**).

Ein Bild, das Text enthält.

Automatisch generierte Beschreibung

Der entscheidende Unterschied liegt darin, dass bei OOP die Objekte „ein Gedächtnis“ haben, also immer den Zustand ihrer Daten kennen.

Objekte können mit dem Programm und untereinander kommunizieren und sich „Nachrichten“ schicken (**Message-Passing-Prinzip**), die dann Einfluss auf Eigenschaften und Methoden haben kann:



Im Framework haben Sie dieses Prinzip bereits in Aktion erlebt. So kann beispielsweise der Held oder der Knappe Tore öffnen und schließen, allerdings auch nur dann, wenn sich dieser direkt vor dem Tor befindet. Die beiden Objekte tauschen also ihre Koordinaten aus und können dann Änderungen, in diesem Fall am Zustand des Tores, durchführen.

**Aufgabe 2**

1. Nennen Sie weitere Beispiele im Framework, bei denen Objekte miteinander kommunizieren müssen. Welche Informationen müssen hierbei ausgetauscht werden?
2. Über welches „Bindeglied“ erfolgt die Kommunikation?

**8.2 Objekte**

**8.2.1 Das Framework**

Unser Helden-**Objekt** befindet sich in einer rechteckigen Welt (ebenfalls eine andere Art von **Objekt**), die in einzelne quadratische Felder aufgeteilt ist. Abstrakt betrachtet interessiert uns also nur, an welcher x- und y-Koordinate sich der Held befindet und in welche Richtung er schaut.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| y/x | 0 | 1 | 2 | 3 | 4 |
| 0 |  |  |  |  |  |
| 1 |  |  |  |  |  |
| 2 |  |  |  |  |  |
| 3 |  |  |  |  |  |
| 4 |  |  |  |  |  |

x-Koordinate

y-Koordinate



**8.2.2 Objektdiagramme**

In OOP besitzen **Objekte** (Held, Knappe, Tür, Tor, Code, Monster …) einen **eigenen Zustand** und **eigene Methoden**. Aktionen werden als **Nachrichten** an das jeweilige Objekt formuliert (held.geh(), tuer.spruch\_sagen(...), tor.oeffnen() …). Das erspart ständiges „Daten Herumreichen“ und macht Verhalten an **die zuständigen Objekte** gebunden.

**Im Framework wird das sichtbar:**

* **Zustand** steckt im Objekt: Jedes Objekt verwaltet seine „Variablen“ selbst
* **Verhalten** ist als Methode **am Objekt** implementiert (geh(), links(), …)
* **Message Passing:** Das **aufrufende Objekt** sendet eine Nachricht an ein **Zielobjekt**; das Ziel ändert **seinen eigenen Zustand**.

Der Zustand des Helden-Objektes im Bild oben kann wie folgt genauer beschrieben werden: Er befindet sich auf der **x**-Koordinate **1** und der **y**-Koordinate **0** und schaut in **Richtung** Süden (**S**).

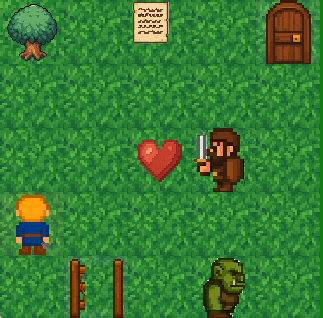
Wenn man dem Helden-Objekt den Namen **held** gibt, dann lässt sich der Zustand des Helden-Objektes im Bild oben mit folgendem **UML-Objektdiagramm** (hierzu später mehr) beschreiben:

|  |
| --- |
| **held** |
| x = 1  y = 0  r = "S" |

Hier werden mit x und y die Koordinaten des Feldes, in dem sich der Held befindet, abgekürzt und mit r die Richtung, in die der Held gerade schaut.

**Aufgabe 1**

Zeichnen Sie Objektdiagramme zu allen Objekten im folgenden Screenshot:



**Aufgabe 2**

Objekte unserer Welt sind fassbare Gegenstände (z.B. Schuhe) oder auch Konstrukte unseres Denkens (z.B. Bankkonto). Betrachten wir Schuhe als Beispiel. Wenn man seine neuen Schuhe charakterisieren will, dann fallen einem sofort eine Reihe von Eigenschaften ein:

* Modell: Sneaker
* Größe: 40; Farbe: rot
* Verschluss: Schnürsenkel,
* ...

1. Zeichnen Sie ein passendes UML-Objektdiagramm zu einem beliebigen Paar Schuhe, das Sie besitzen.
2. Handelt es sich bei allen Eigenschaften der Schuhe um Text, also um Daten vom Typ String?

Mit Objekten kann man auch bestimmte „Operationen“ ausführen: Zum An- und Ausziehen kann man z.B. die Schuhe in einem gewissen Sinn öffnen und schließen.

1. Listen Sie weitere Operationen auf, die mit Schuhen durchgeführt werden können. Ändern diese Operationen auch den Zustand der Schuhe?
2. Modellieren Sie ein Autoobjekt, z.B. Ihr eigenes, das Ihrer Eltern, …

Ein Bild, das Pixel enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.2.3 Zustände von Objekten nutzen**

Da Objekte ihren aktuellen Zustand kennen und auch mit anderen Objekten kommunizieren können, nutzen wir dies für unser Framework aus.

Unser held kann mit der Methode ist\_auf\_herz() prüfen, ob er gerade auf einem Herzen steht. Dies können wir dann in der Form **if** held.ist\_auf\_herz(): nutzen, um Aktionen nur dann durchzuführen, wenn wir es auch wirklich möchten.

**Aufgabe 1**

1. Laden Sie Level 9 und testen Sie die folgende vorgeschlagene Lösung:

held.links()

**for** i **in** range(7):

held.geh()

held.nimm\_herz()

1. Ergänzen Sie eine Abfrage so, dass nur dann das Herz aufgesammelt wird, wenn der Held auf einem Herzen steht.
2. Lösen Sie selbstständig **Level 10**. Hier müssen Sie auch den Knappen verwenden, welcher die gleiche Fähigkeit besitzt.

Verwenden Sie eine for-Schleife und zählen Sie, wie viele Schritte der Held insgesamt machen muss. Welche Aktionen muss der Held auf den Herzen ausführen?

*Tipp:*

**Aufgabe 2**

Betrachten Sie **Level 11** und **12**. Ihr Ziel soll es nun sein, beide Level mit dem gleichen Code zu lösen, d.h. Ihr Programm aus Level 11 soll auch Level 12 lösen und umgekehrt. Dies können wir durch das Einbinden einer **while**-Schleife erreichen.

Bewegen Sie den Helden zunächst auf das erste Herz.

*Tipps:*

Prüfen Sie dann in der Bedingung der while-Schleife, ob sich der Held noch auf einem Herz befindet. Falls ja, muss der Held noch weiter Herzen einsammeln.

**Aufgabe 3**

Wir ergänzen nun die Fähigkeit verbleibende\_herzen(). Damit zählt der Held, wie viele Herzen er noch einsammeln muss. Lösen Sie mit einer while-Schleife und dieser Fähigkeit sowohl Level 13 als auch Level 14 mit dem gleichen Code.

Verwenden Sie eine while Schleife und prüfen Sie jeweils, ob noch Herzen vorhanden sind. Wenn ja, muss der Held über die Herzen laufen und darf natürlich nur sammeln, wenn er auf einem Herzen steht.

*Tipp:*

**Aufgabe 4**

1. Beschreiben Sie, inwiefern sich die Methoden geh(), ist\_auf\_herz() und verbleibende\_herzen() unterscheiden.
2. Öffnen Sie erneut Level 4 und versuchen Sie, das Herz mit dem Monster zu nehmen und dabei vorher zu prüfen, ob das Monster auf dem Herz steht.
3. Beschreiben Sie die Unterschiede zwischen den beiden Objekten **held** und **monster** bezüglich ihrer Fähigkeiten.

**Aufgabe 5**

Vervollständigen Sie die beiden Merksätze bezüglich der Fähigkeiten von Objekten.

*Hinweis: Unten finden Sie Formulierungshilfen.*

**Merke:**

Objekte können verschiedene Fähigkeiten besitzen, welche wir **Methoden** nennen. Nicht jedes Objekt hat die gleichen Methoden, wobei manche Methoden mehreren Objekten zur Verfügung stehen.

**Beispiele**:

Die Methode geh() sowohl dem Helden, als auch dem Monster zur Verfügung.

Die Methode nimm\_herz() steht dem Helden, nicht aber dem Monster zu Verfügung.

**Merke**:

Methoden können auf den Zustand eines Objektes zugreifen und diesen Verändern. Gleichzeitig können Sie auf dessen Grundlage auch Antworten liefern, welche im Programm weiter verwendet werden können.

**Beispiele:**

Die Methode geh() verändert nur den Zustand des Objektes, indem anhand der Richtung die korrekte Koordinate geändert wird. Die Methode liefert keinen Wert zurück. In diesem Fall sprechen wir von einem **Auftrag**.

Die Methode ist\_auf\_herz() prüft anhand der Koordinaten des Objektes, ob sich dieses auf einem Herzen befindet und gibt entsprechend True oder False zurück. Solche Methoden nennen wir Anfrage.

Ein Bild, das Pixel enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.2.4 Auftrag und Anfrage**

Unser Held hat gelernt, in die Ferne zu blicken und kann nun mit herzen\_vor\_mir() alle Herzen zählen, welche vor ihm liegen und für ihn erreichbar sind.

**Aufgabe 1**

1. Lösen Sie mithilfe des neuen Befehls sowie mit ist\_auf\_herz() und verbleibende\_herzen() Level 15.

Der Held muss solange weitermachen bis alle Herzen gesammelt sind. Sie benötigen also eine while-Schleife, in welcher verbleibende\_herzen verwendet wird. Dann müssen Sie unterscheiden: Gibt es noch Herzen vor dem Helden? Dann läuft er und sammelt sie ein, wenn er auf ihnen steht. Sind keine Herzen mehr vor ihm, dann kann er sich nach links wenden und dort weitermachen.

*Tipp:*

1. Beschreiben Sie, inwiefern sich die Methoden geh(), ist\_auf\_herz(), herzen\_vor\_mir() und verbleibende\_herzen() unterscheiden.
2. Öffnen Sie erneut Level 4 und versuchen Sie, das Herz mit dem Monster zu nehmen und dabei vorher zu prüfen, ob das Monster auf dem Herz steht.
3. Beschreiben Sie die Unterschiede zwischen den Objekten **held** und **monster** bezüglich ihrer Fähigkeiten.

**Aufgabe 2**

Vervollständigen Sie die beiden Merksätze bezüglich der Fähigkeiten von Objekten.

*Hinweis: Unten finden Sie Formulierungshilfen.*

**Merke:**

Objekte können verschiedene Fähigkeiten besitzen, welche wir **Methoden** nennen. Nicht jedes Objekt hat die gleichen Methoden, wobei manche Methoden mehreren Objekten zur Verfügung stehen.

**Beispiele**:

Die Methode geh() steht sowohl dem Helden, als auch dem Monster zur Verfügung.

Die Methode nimm\_herz() steht dem Helden, nicht aber dem Monster zu Verfügung.

**Merke**:

Methoden können auf den Zustand eines Objektes zugreifen und diesen Verändern. Gleichzeitig können Sie auf dessen Grundlage auch Antworten liefern, welche im Programm weiter verwendet werden können.

**Beispiele:**

Die Methode geh() verändert nur den Zustand des Objektes, indem anhand der Richtung die korrekte Koordinate geändert wird. Die Methode liefert keinen Wert zurück. In diesem Fall sprechen wir von einem **Auftrag**.

Die Methode ist\_auf\_herz() prüft anhand der Koordinaten des Objektes, ob sich dieses auf einem Herzen befindet und gibt entsprechend True oder False zurück. Solche Methoden nennen wir **Anfrage**.

Ein Bild, das Cartoon, Pixel enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.2.5 Methoden im Objektdiagramm**

Die Methoden eines Objektes können wir im Objektdiagramm darstellen. **Anfragen**, also Methoden, die einen **Wert** zurückliefern, werden um den **Datentyp** dieses Werts angeben:

|  |
| --- |
| held |
| x = 5  y = 4  r = "N" |
| geh()  links()  rechts()  nimm\_herz()  verbleibende\_herzen(): Integer |

**Aufgabe 1**

Ergänzen Sie die Methoden herzen\_vor\_mir() und ist\_auf\_herz() im Diagramm.

**Aufgabe 2**

Weitere Fähigkeiten, die unser Held und unser Knappe besitzen sind die was\_ist\_vorn(), was\_ist\_links() und was\_ist\_rechts() Methoden.

1. Öffnen Sie **Level 16**. Es soll nicht gelöst werden, sondern dient zum Testen der vorgestellten Fähigkeiten.
2. Führen Sie den Befehl print(knappe.was\_ist\_vorn()) aus und ergänzen Sie mehr dieser Befehle, um herauszufinden, wie Sie die unterschiedlichen Objekte und Geländearten erkennen können. Den Knappen müssen Sie um einen Schritt bewegen, um alles erfassen zu können.
3. Ergänzen Sie mindestens die was\_ist\_vorn() Methode im Objektdiagramm oben.
4. **Lösen** Sie mithilfe dieser Fähigkeit **Level 17**.

*Tipp:*

Sie benötigen erneut eine while-Schleife, da der Herz solange weitermachen muss, bis alle Herzen gesammelt sind. Dann müssen Sie unterscheiden, welcher Fall gerade eingetreten ist. Wenn sich ein Baum vorm Helden befindet, muss sich dieser nach links drehen, ansonsten geht er und sammelt Herzen auf.

**Knobelaufgabe für später**: Lösen Sie Level 18 und 19.

Ein Bild, das Pixel enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.2.6 Zugriff auf Objekte erhalten**

In unseren bisherigen Aufgaben konnten wir mithilfe des **Helden** und des **Knappen** alle Level lösen. Doch nicht immer ist der Zugriff auf wenige Objekte ausreichend, um das Ziel zu erreichen. Im Sinne der Objektorientierung können wir uns aber Zugriff auf weitere Objekte verschaffen, wenn die passenden Methoden gegeben sind.

**Aufgabe 1**

1. Öffnen Sie Level 20 und versuchen Sie dieses zu lösen. Erklären Sie knapp, weswegen Ihnen das nicht gelingen kann.

Durch eine kleine „*Mogelei*“ können Sie das Level dennoch lösen. Betrachten Sie die Objektdiagramme des Zettels mit dem Zauberspruch zettel und der Tür tuer:

|  |  |  |
| --- | --- | --- |
| **zettel** |  | **tuer** |
| x = 2  y = 4  gesammelt = False  spruch = "…" #zufall |  | x = 0  y = 3  offen = False  richtiger\_spruch = "…" #zufall |
| gib\_spruch() : String  spruch\_ausgeben() |  | ist\_passierbar() : Boolean  spruch\_anwenden(s : str) |

1. **Nutzen** Sie zunächst das Objekt zettel, um den darauf notierten Spruch mit print(…) auf der Konsole auszugeben.
2. **Lösen** Sie das Level, indem Sie den Spruch stattdessen der Tür übergeben.

*Tipp*:

Sie haben den Helden über die Objekt-Variable held bewegt, den Knappen über knappe und Methoden mit diesen Objekten aufgerufen, z.B. knappe.geh().

Hier müssen Sie jetzt die Objektvariablen **spruch** und **tuer** verwenden sowie die beiden Methoden gib\_spruch() für b) und c) sowie spruch\_anwenden(…) für c). Speichern Sie sich dazu den erhaltenen Spruch in einer Variable.

**Aufgabe 2**

Öffnen Sie Level 21 und versuchen Sie mit der gleichen Vorgehensweise das Level zu lösen. Es wird zu einer Fehlermeldung kommen. Notieren Sie die letzte Zeile der Fehlermeldung.

**Objekte** sind in der Regel **nicht direkt über Variablen referenzierbar**. Stattdessen müssen Sie Zugriff über andere Objekte erhalten, die bereits zur Verfügung stehen. In unserem Fall haben wir immer Zugriff auf das **Level** (vgl. Befehl level.lade()) sowie den **Helden**.

Die Objekte für den **Knappen**, die **Tür** und den **Zettel** existieren zwar im Speicher (schließlich werden Sie ja dargestellt), aber Sie besitzen noch keinen Zugriff auf diesen. Um Sie wie gewohnt nutzen zu können, rufen wir sie über Methoden referenzierbarer Objekte (held, level) ab und binden Sie an eine Variable.

Das kann man beispielhaft so darstellen:

|  |
| --- |
| **???** |
| x = 0  y = 3  offen = False  richtiger\_spruch = "…" #zufall |
| ist\_passierbar() : Boolean  spruch\_anwenden(s : str) |

t = objekt.erhalte\_tuer()

|  |
| --- |
| **t** |
| x = 0  y = 3  offen = False  richtiger\_spruch = "…" #zufall |
| ist\_passierbar() : Boolean  spruch\_anwenden(s : str) |

**Aufgabe 3**

Betrachten Sie die modifizierten Objektdiagramme des Helden und des Knappen. Diese haben jetzt eine Fähigkeit, um auf **das Objekt direkt vor ihnen** zuzugreifen. Der Held kann zudem jederzeit mit einer seiner Methoden auf das Knappen-Objekt zugreifen. **Lösen** Sie mit den beiden Methoden **Level 21**.

|  |  |  |
| --- | --- | --- |
| **held** |  | **???** (Knappen-Objekt) |
| x = 2  y = 4  richtung = "S" |  | x = 0  y = 3  richtung = "S" |
| geh() : String  ist\_auf\_herz() : Boolean  ...  gib\_knappe() : Knappe  gib\_objekt\_vor\_dir(): Objekt |  | geh() : String  ist\_auf\_herz() : Boolean  ...  gib\_objekt\_vor\_dir(): Objekt |

*Tipp:*

Betrachten Sie das Beispiel im Kasten oben, wie ein Objekt **an eine Variable** gebunden werden kann. Nutzen Sie dann das **Helden-Objekt**, um den **Knappen** an eine Variable zu binden (z.B. wieder an knappe). Außerdem kann er das Objekt vor sich erkennen (die Tür mit gib\_objekt\_vor\_dir()), welche Sie ebenfalls an eine Variable binden müssen. Zuletzt binden Sie mithilfe des Knappen-Objekts den Zettel an eine Variable und können das Level wie Level 20 lösen.

**Aufgabe 4**

Ergänzen Sie den folgenden Merksatz

**Merke**:

**Anfragen** können nicht nur \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Datentypen zurückliefern, sondern auch höhere Datentypen, wie z.B. \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_. Diese können zur weiteren Verwendung auch an \_\_\_\_\_\_\_\_\_\_\_\_\_\_ gebunden werden.

**Übung**

Lösen Sie Bonuslevel 104.

|  |
| --- |
| **???** (Monster-Objekte) |
| x = 2  y = 3  richtung = "W" |
| geh() : String  ...  angriff()  gib\_objekt\_vor\_dir(): Objekt |

**Aufgabe 4**

1. Öffnen Sie Level 22. Begründen Sie, weswegen Sie mit den jetzigen Mitteln das Level nicht lösen können.
2. Wie zuvor erwähnt haben Sie Zugriff auf das gesamte Level über die Variable level. Über das Level können Sie auf jedes Objekt zugreifen, wenn Sie dessen Koordinaten kennen. Dazu steht die Methode gib\_objekt\_bei(x,y) zur Verfügung. Lösen Sie Level 22 mit dessen Hilfe.

**8.2.6 Attribute von Objekten**

Ein Bild, das Pixel, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Wir haben bereits festgestellt, dass sich jedes Objekt in einem **Zustand** befindet und dass dieser Zustand über Methodenaufrufe **manipuliert** werden kann. Auf Grundlage des Zustands kann das Objekt über die Methoden auch Werte zurückliefern. Manchmal reichen diese Methoden allerdings nicht aus und wir müssen direkt auf den Zustand des Objekts zugreifen.

**Aufgabe 1**

1. Öffnen Sie **Level 23**. Begründen Sie knapp, weswegen Sie das Level nicht lösen können.

Wir können zwar über das **Level** Zugriff auf das Herz erhalten (zur Erinnerung: das ging über **level**.gib\_objekt\_bei(x,y)), aber bei einem Blick auf das Objektdiagramm des Herzens wird klar, dass dieses keine Möglichkeit bietet, es einzusammeln oder zu bewegen.

Methoden

Attribute

|  |
| --- |
| **herz1** |
| x = 5  y = 0 |
|  |

Sie haben jedoch gelernt, dass Objekte sowohl Eigenschaften (**Attribute**) als auch Fähigkeiten (**Methoden**) bündeln. Im Fall von unserem Herzen gibt es zwar keine Methoden, jedoch wird die Position des Herzens über dessen Attribute **x** und **y** bestimmt. Diese können wir manipulieren.

1. Erhalten Sie über die Level-Variable (s. oben) Zugriff auf das Herz und manipulieren Sie dessen Attribute so, dass der Held im Anschluss das Herz einsammeln kann.

Erhalten Sie zunächst Zugriff auf das Herz und speichern Sie dieses unter einer Variable ab, z.B. h=level.erhalte\_objekt\_bei(…). Manipulieren Sie dann zunächst die **x**-Koordinate des Herzens durch den Befehl **h.x =** …

*Tipp:*

1. Nutzen Sie diese Manipulation aus, um **Level 24** schnell zu lösen.

*Hinweis: Bei dieser Aufgabe steht Ihnen knappe direkt zur Verfügung.*

1. Testen Sie: Können Sie Objekte auch außerhalb des Levels „verschwinden“ lassen?

**Aufgabe 2**

Ergänzen Sie den Merksatz.

**Merke**:

Der Zugriff auf **Attribute** und \_\_\_\_\_\_\_\_\_\_\_\_ eines Objektes, z.B. mit dem Variablennamen **obj** erfolgt mithilfe eines \_\_\_\_\_\_\_\_\_\_\_\_\_\_, gefolgt vom Namen des Attributes oder der \_\_\_\_\_\_\_\_\_\_\_\_\_\_. Bei Attributen ist sowohl ein lesender als auch ein \_\_\_\_\_\_\_\_\_\_\_\_ Zugriff möglich. Allgemein also: obj**.**attribut oder obj**.**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Beispiele:**

print(held.x) ist ein \_\_\_\_\_\_\_\_\_\_\_\_\_\_ Zugriff auf das Attribut \_\_\_ von \_\_\_\_\_\_\_\_\_\_\_

\_\_\_\_\_\_\_\_\_\_\_ = 5 ist ein \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Zugriff auf das gleiche Attribut.

**Aufgabe 3**

1. Öffnen Sie **Level 25** und versuchen Sie das Level zu lösen. Notieren Sie hier Ihre Beobachtungen und versuchen Sie das Verhalten zu begründen.
2. Testen Sie, ob auch die Methoden des Helden eingeschränkt wurden, z.B. indem er sich drehen soll. Ist dies weiterhin möglich?
3. Testen Sie, ob alle Attribute des Helden nicht mehr zugänglich sind. Der Held besitzt ein bisher nicht verwendetes Attribut name. Testen Sie, ob das folgende noch möglich ist:
   1. Geben Sie den Namen auf der Konsole aus
   2. Ändern Sie den Namen auf einen Namen Ihrer Wahl

*Platz für Notizen:*

1. Vielleicht können Sie die Tür ja noch irgendwie manipulieren. Hier ist das Objektdiagramm der Tür. **Lösen** Sie mit dessen Hilfe **Level 25**.

|  |
| --- |
| **????** (Tür-Objekt) |
| x = 1  y = 2  offen = False  richtiger\_spruch = "…" #zufall |
| ist\_passierbar() : Boolean  spruch\_anwenden(s : str) |

**Aufgabe 4**

Begründen Sie, weswegen es durchaus Sinn machen könnte, nicht alle Attribute des Helden manipulieren zu können. Stellen Sie sich vor, Sie hätten diese Information bereits ab Level 1 gehabt. Hätten Sie dann anders gehandelt?

Ein Bild, das Pixel, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.2.7 Datenkapselung**

Nicht immer ist der **Zugriff** auf alle Attribute (oder auch Methoden) eines Objektes möglich. Dies kann je nach Objekt, Attribut und Methode unterschiedlich sein. So ist z.B. ab sofort der Zugriff auf die Position des Helden nicht mehr möglich, aber der Name kann weiterhin frei manipuliert werden. Doch wieso ist dies sinnvoll?

**Aufgabe 1**

Öffnen Sie **Level 26**. Ein simples Level, bei dem sich der Held um 180° drehen muss, bevor er das einsame Herz einsammelt. Kein Schnickschnack notwendig, aber vielleicht können wir es uns ja noch einfacher machen!

1. Sie haben bereits so viel Wissen angesammelt, dass Sie sich dazu entschließen den Helden einfach direkt über sein Richtungsattribut zu drehen. Keine Animation und der Befehl ist kürzer. Betrachten Sie den Zustand des Objekts im Level (einmaliges Ausführen) und setzen Sie das Attribut **richtung** des Helden direkt auf **Norden**. Lösen Sie dann das Level durch Vorwärtslaufen und einsammeln des Herzens. **Beschreiben** Sie, was passiert ist.

Ein Bild, das Pixel, Cartoon enthält.

KI-generierte Inhalte können fehlerhaft sein.

1. Möglicherweise hätte man etwas anderes dort eintragen sollen. Vielleicht wird die Richtung als Links, rechts gespeichert oder als Norden, Süden ausgeschrieben? Vielleicht sind es Zahlen von 0 bis 3 oder abegekürzt L,R,O,U? Testen Sie maximal 2 Minuten, ob Sie den korrekten Befehl finden.
2. Zum Gück stellt der Held eine Methode setze\_richtung(r) zur Verfügung, in welcher auch Norden in Form von „N“ eingetragen werden kann. Verwenden Sie diese Methode, um das Level zu lösen.
3. Falls Sie es in b) nicht herausgefunden haben sollten, die korrekte Angabe wäre „up“ gewesen. Testen Sie den Befehl held.richtung = „up“ aus, um das Level zu lösen. Welcher (vor allem optische) Nachteil hat diese Lösung im Vergleich zu setze\_richtung?

**8.2.8 Getter und Setter Methoden**

Ein Bild, das Animierter Cartoon, Clipart, Text, Cartoon enthält.

KI-generierte Inhalte können fehlerhaft sein.

In der Regel sind Attribute von Objekten nicht frei zugänglich, um zu verhindern, dass ungültige Werte eingetragen werden. Damit es dennoch möglich ist, diese Werte auszulesen und zu manipulieren, verwendet man sogenannte **getter**- und **setter** Methoden.

**Aufgabe 1**

Wir führen neue Objekte in unsere Spielwelt ein: farbige Schlüssel, die zu farbigen Schlössern passen. Es gibt die folgenden Farben: blue, golden, green, red, violet.

|  |
| --- |
| **????** (Tür-Objekt) |
| x = 2  y = 1  offen = False  farbe = "…" #zufall |
| ist\_passierbar() : String  schluessel\_verwenden(s : Schluessel)  get\_farbe() : String  get\_offen() : Boolean |

1. Öffnen Sie **Level 27**. Nutzen Sie die Informationen aus dem Objektdiagramm, um das Level zu lösen.

*Tipp: Erhalten Sie zunächst Zugriff auf das benötigte Schlüssel-Objekt über das Level.*

1. Öffnen Sie jetzt **Level 28** und lösen Sie es. Führen Sie Ihre Lösung erneut aus und beschreiben Sie, was passiert ist:
2. Sie benötigen das Objektdiagramm der Schlüssel-Objekte, um Level 29 konsistent lösen zu können:

|  |
| --- |
| **????** (Schlüssel-Objekt) |
| x = 3  y = 3  farbe = "…" #zufall |
| get\_farbe() : String |

1. Lösen Sie jetzt **Level 29**.

*Tipp: Betrachten Sie erneut das Objektdiagramm der Tür.*

**Aufgabe 2**

Öffnen Sie **Level 30**. Sie werden das Level mit den jetzigen Mitteln nicht lösen können. Die Schlüssel-Objekte haben jedoch setter-Methoden, die sich als nützlich erweisen könnten:

|  |
| --- |
| **????** (Schlüssel-Objekt) |
| x = 3  y = 3  farbe = "…" #zufall |
| get\_farbe() : String  set\_farbe(f : String)  setze\_position(x,y) |

Lösen Sie das Level.

**Aufgabe 3**

1. Öffnen Sie **Level 31** und versuchen Sie das Level auf herkömmliche Art zu lösen. Woran scheitern Sie?
2. Testen Sie die Methode setze\_position(x,y) des Helden. Testen Sie auch die gleiche Methode, welche dem Herzen zur Verfügung steht. Weswegen scheitert dies?
3. Sie müssen zu drastischeren Mitteln greifen. Öffnen Sie die held.py im Framework Unterordner und suchen Sie die Methode setze\_position. Vielleicht können Sie den Code mit minimalen Änderungen so manipulieren, dass Sie das Level lösen können.
4. Experimentieren Sie mit der setze\_position Methode herum, indem Sie verschiedene Werte ausprobieren.

**Aufgabe 4**

Ergänzen Sie den Lückentext.

**Merke:**

**Getter** (vom englischen *to get = holen*) dienen dazu, Informationen über den Zustand eines Objektes \_\_\_\_\_\_\_\_\_\_\_\_. Sie geben Werte zurück, ohne etwas zu \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

**Beispiele:**

zettel.gib\_spruch()

knappe.gib\_objekt\_vor\_dir()

held.gib\_knappe()

level.gib\_objekt\_bei(x,y)

**Setter** (vom englischen *to set = setzen*) dienen dazu, Attribute eines Objektes \_\_\_\_\_\_\_\_\_\_\_ zu verändern. Dabei kann das Objekt prüfen, ob der neue Wert \_\_\_\_\_\_\_\_\_\_, bzw. \_\_\_\_\_\_\_\_\_ ist.

**Beispiel**:

held.setze\_richtung(r)

held.setze\_position(x,y)

**Aufgabe 5**

1. Bewerten Sie die folgende Aussage:

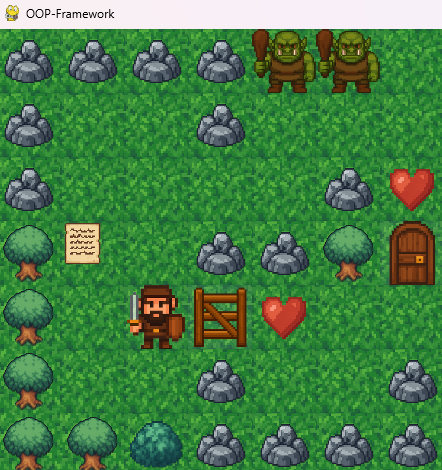
*„Setter sollten nur dann gesetzt werden, wenn mit diesen nicht geschmummelt werden kann. Die set\_position() Methode des Helden sollte also eigentlich komplett entfernt werden.“*

1. Bei Software unterscheiden wir zwischen der Benutzer-Ebene und der Entwickler-Ebene.

Beschreiben Sie, wie diese Ebenen in unserem Framework aussehen.

**Exkurs: Benutzer- vs. Entwickleransicht**

Als **Benutzer** eines Programms (z. B. Windows, PowerPoint, FireFox, Chrome, Minecraft,…) siehst du nur die **Benutzeroberfläche**, also Figuren, Aktionen und Reaktionen. Du kannst über **Eingaben** (z. B. Tastatur, Maus) etwas bewirken, hast aber **keinen direkten Zugriff** auf den Programmcode oder auf einzelne Daten im Hintergrund.



Wenn du ein Programm **entwickelst**, siehst du den **gesamten Code** und kannst **alles manipulieren**. Zugriffsregeln für Attribute kannst du also einfach umgehen und getter und setter hinzufügen, wie es dir beliebt.

Ein Bild, das Text, Pixel, Display, Screenshot enthält.

KI-generierte Inhalte können fehlerhaft sein.

Jedes Objekt wird jedoch mit einem **klaren Ziel** konzipiert und implementiert. Ein **unkontrolliertes Eingreifen** in diese Struktur kann leicht zu **Fehlern** und damit zu Abstürzen, ungewolltem Verhalten oder sogar Sicherheitslücken führen. Dies trifft umso mehr zu, wenn du deinen Code **anderen Entwicklern** zur Verfügung stellst und diese ihn weiterverwenden wollen.

Aus diesem Grund werden genau die getter und setter (und weitere Methoden) zur Verfügung gestellt, die benötigt werden, um Objekte wie intendiert zu verwenden. Ergänzen und ändern kann man diese im Nachhinein immer noch. Der Benutzer wird davon nichts merken.

**Beispiel:**

Der Held besitzt die Methode setze\_position(x,y), mit der er teleportiert werden kann. Diese Methode ist z.B. nützlich, wenn in das Spiel Portale eingeführt werden sollen. Insbesondere stellt sie sicher, dass der Held nicht an ungültige Stellen teleportiert werden kann. Ein Benutzer kann nie auf diese Methode zugreifen, wenn dies nicht über die **Benutzeroberfläche** (z.B. durch einen benutzbaren Teleportatationszauber) ermöglicht wird.

Ein Bild, das Pixel enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.3 Objekte und Klassen**

Wir haben bereits festgestellt, dass sich unsere Objekte zum Teil stark ähneln. Sie besitzen zum Teil gleiche Methoden und auch die Attribute heißen gleich. Dies werden wir jetzt genauer untersuchen.

**Aufgabe 1**

Betrachte die folgenden Objektdiagramme. Welcher Nachteil bietet diese Darstellung?

**Ein Bild, das Screenshot, Cartoon enthält.

KI-generierte Inhalte können fehlerhaft sein.**

|  |
| --- |
| **tuer1** |
| x = 3  y = 3  offen = False  farbe = "red" |
| ist\_passierbar() : Boolean  verwende\_schluessel(s : Schluessel)  get\_farbe() : String  get\_offen() : Boolean |

|  |
| --- |
| **orc2** |
| typ = "Monster"  x = 4  y = 2  richtung = "S"  ist\_ko = False |
| ist\_passierbar() : Boolean  setze\_richtung(r : String)  setze\_position(x,y : Integer)  geh()  zurueck()  links()  rechts()  attack()  gib\_objekt\_vor\_dir() : Objekt  was\_ist\_links() : String  was\_ist\_vorn() : String  was\_ist\_rechts() : String  herzen\_vor\_mir() : Integer  verbleibende\_herzen() : Integer |

|  |
| --- |
| **orc1** |
| typ = "Monster"  x = 3  y = 2  richtung = "S"  ist\_ko = False |
| ist\_passierbar() : Boolean  setze\_richtung(r : String)  setze\_position(x,y : Integer)  geh()  zurueck()  links()  rechts()  attack()  gib\_objekt\_vor\_dir() : Objekt  was\_ist\_links() : String  was\_ist\_vorn() : String  was\_ist\_rechts() : String  herzen\_vor\_mir() : Integer  verbleibende\_herzen() : Integer |

|  |
| --- |
| **tuer2** |
| x = 4  y = 3  offen = False  farbe = "red" |
| ist\_passierbar() : Boolean  verwende\_schluessel(s : Schluessel)  get\_farbe() : String  get\_offen() : Boolean |

**Aufgabe 2**

Ergänze:

Offensichtlich sind die beiden Monster und auch die beiden Türen sich jeweils sehr ähnlich. Sie besitzen die gleichen Attribute mit unterschiedlichen Werte. Die Methoden sind sogar komplett identisch.

Aus genau diesem Grund fassen wir **Objekte** zu einer **Klasse** zusammen. Diese sind wie ein Bauplan für gleichartige Objekte.

**8.3.1 Klassen- und Objektdiagramme**

**Merke**:

**Klassen** legen fest, welche **Attribute** und **Methoden** **Objekte** besitzen. **Objekte** sind dann **konkrete Instanzen** dieser **Klassen**. Dargestellt werden Klassen im **UML-Klassendiagramm**:

|  |
| --- |
| **Klassenname** |
| attribut\_1 : Datentyp  attribut\_2 : Datentyp  …  attribut\_n : Datentyp |
| methode\_1(…) : …  methode\_2(…) : …  …  methode\_n(…) : … |

|  |
| --- |
| **Monster** |
| x : Integer  y : Integer  richtung : String  ist\_ko : Boolean |
| ist\_passierbar() : Boolean  setze\_richtung(r : String)  setze\_position(x,y : Integer)  geh()  zurueck() |

Allgemein

Beispiel

**Aufgabe 3**

Zeichne Klassendiagramme für den Helden sowie für die Türen.

Da für jede Klasse festgelegt ist, welche Attribute und Methoden sie besitzen, können wir unsere Objektdiagramme nun stark kürzen:

|  |
| --- |
| **held : Held** |
| x = 1  y = 3  richtung = "down"  ist\_ko = False  gold = 0  weiblich = False  knappen = [] |

**Expertenfrage**

Erklären Sie, weswegen Variablennamen, z.B. orc1, x,y, richtung als Konvention klein geschrieben werden sollten:

**Aufgabe 5**

Stellen Sie die Objektdiagramme für tuer1 und orc1 in verkürzter Form dar:

**Ein Bild, das Screenshot, Cartoon enthält.

KI-generierte Inhalte können fehlerhaft sein.**

**Merke**:

Da alle **Objekte** einer **Klasse** die **gleichen Methoden** haben, geben wir diese im **Objektdiagramm** nicht mehr an. Stattdessen wird der Name der Klasse angegeben.

|  |
| --- |
| **objektname : Klasse** |
| attribut\_1 = Wert  attribut\_2 = Wert  …  attribut\_n = Wert |

|  |
| --- |
| **held : Held** |
| x = 1  y = 3  richtung = "down"  ist\_ko = False  gold = 0  weiblich = False  knappen = [] |

Allgemein

Beispiel

Ein Bild, das Screenshot, Pixel, Kugel enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.3.2 Eigene Objekte erzeugen**

Bisher haben wir nur mit den Objekten gearbeitet, die im Level vorhanden sind. Das werden wir nun ändern.

**Aufgabe 1**

1. Öffnen Sie **Level 32** und versuchen Sie es **mit den Pfeiltasten** zu lösen.
2. Möglicherweise kennen Sie bereits einige Tricks, um das Level dennoch zu lösen. Probieren Sie (maximal eine Minute) einige davon aus, machen Sie aber ggf. **Änderungen am Framework wieder rückgängig**! Ansonsten überspringen Sie diesen Teil einfach.
3. Sie können das Level lösen, indem Sie dem Bogenschützen Hindernisse in den Weg stellen. Dazu erzeugen Sie ein neues Hindernis-Objekt und stellen es an die gewünschte Stelle.

**Testen Sie:**

tor1 = Tor(1,2,True)

level.objekt\_hinzufuegen(tor1)

1. **Lösen** Sie jetzt **Level 32**, indem Sie vor die Bogenschütze Objekte setzen.

**Aufgabe 2**

1. Testen Sie den folgenden, verkürzten Befehl:

level.objekt\_hinzufuegen(Tuer(1,2))

1. Welcher Unterschied besteht zwischen der langen und der kurzen Fassung? Angenommen, Sie würden jetzt gerne die Farbe der Tür abfragen: Welche Variante würden Sie dann bevorzugen?

|  |
| --- |
| **Monster** |
| x : Integer  y : Integer  richtung : String  ist\_ko : Boolean |
| Monster(x,y : int, r : str)  ist\_passierbar() : Boolean  setze\_richtung(r : String)  … |

1. Öffnen Sie **Level 33** und versuchen Sie mithilfe des angepassten Klassendiagramms einige Monster hinter den Bogenschützen erscheinen zu lassen. Mal sehen, was dann wohl passiert.

**Merke:**

Objekte werden durch den Aufruf des **Konstruktors** einer Klasse erzeugt. Dieser heißt immer wie die Klasse selbst und verlangt in der Regel verschiedene Parameter, welche im Klassendiagramm ablesbar sind.

|  |
| --- |
| **Klassenname** |
| attribut\_1 : Datentyp  attribut\_2 : Datentyp  …  attribut\_n : Datentyp |
| Konstruktor(Parameter)  methode\_1(…) : …  methode\_2(…) : …  … |

|  |
| --- |
| **Monster** |
| x : Integer  y : Integer  richtung : String  ist\_ko : Boolean |
| Monster(x,y : int, r : str)  ist\_passierbar() : Boolean  setze\_richtung(r : String)  … |

Allgemein

Beispiel

Der **Aufruf** des Konstruktors **erzeugt** dann das Objekt, so dass es z.B. an eine **Variable** gebunden werden kann: **obj** = Konstruktor(…). Im Beispiel: **m** = Monster(1,2,"up").

**Aufgabe 3**

1. Öffnen Sie **Level 34**. Leider fehlt der Schlüssel, um das Level zu lösen.
2. Betrachten Sie das Klassendiagramm der Klasse Schluessel:

|  |
| --- |
| **Schluessel** |
| x : int  y : int  farbe : str |
| Schluessel(x,y : int)  get\_farbe() : String  set\_farbe(f : String)  setze\_position(x,y) |

1. Lösen Sie mit diesem Wissen **Level 34**, indem Sie selbst einen passenden Schlüssel erzeugen.

**Aufgabe 4**

„Bauen“ Sie das folgende Level nach. Laden Sie dazu Level 341 und nutzen Sie die folgenden Signaturen:

Ein Bild, das Screenshot, Cartoon, Animierter Cartoon, Animation enthält.

KI-generierte Inhalte können fehlerhaft sein.

* Knappe(x,y : int, r : str)
* Tuer(x,y : int, farbe : str)
* Schluessel(x,y : int)
* Herz(x,y : int)
* Hindernis(art : str, x,y : int)

Es gibt Bäume, Berge und Büsche im Level.

Exkurs: Mit Objekten in Listen umgehen

* Alle Objekte einer Liste bewegen

Ein Bild, das Pixel, Cartoon enthält.

KI-generierte Inhalte können fehlerhaft sein.**8.3.3 Eigene Klassen implementieren**

Bisher hat das Framework sich um unsere Klassen gekümmert, doch das ändern wir jetzt. In den folgenden Kapiteln werden Sie selbst einige Klassen des Frameworks neu implementieren. Wir beginnen dabei mit unserem Helden, für welchen wir nun das folgende, gekürzte Klassendiagramm verwenden.

|  |
| --- |
| **Held** |
| x : Integer  y : Integer  richtung : String  name : String  typ : String  weiblich : Boolean |
| Held(xp,yp : int, richtung : str, w : bool)  geh()  links()  rechts()  zurueck() |

**Aufgabe 1**

Laden Sie in der schueler.py Level 35. Ein leeres Level erscheint, in dem eigentlich unser Held stehen sollte. Das Framework kümmert sich selbst um die Erzeugung der Objekte anhand der Level-Datei, aber dazu muss die Helden-Klasse auch verfügbar sein.

1. Ergänzen Sie in Ihrem bekannten Programmierbereich die folgenden Zeilen:

**class** **Held**:

**def** **\_\_init\_\_**(self,xp,yp,richtung,w):

self.x = 0

Führen Sie das Programm aus.

1. Ergänzen Sie nun selbstständig die fehlenden Befehle, um der neuen Helden Klasse alle Attribute zuzuweisen. Verwenden Sie dabei sinnvolle Standardwerte, welche dem Klassendiagramm entsprechen. Den Standardnamen eines Helden dürfen Sie frei wählen, der Typ des Helden ist „Held“.
2. Möglicherweise ist Ihnen bereits aufgefallen, dass das Level nicht abgeschlossen werden kann, obwohl alle Attribute gesetzt und der Held sichtbar ist. Das liegt daran, dass im Level genau festgelegt ist, wo der Held steht, in welche Richtung er schaut und ob „er“ weiblich ist. Letzteres steuern Sie über level.lade(35,**weiblich=…**) sogar selbst!

Der Konstruktor des Helden erhält über Parameter genau diese Standardwerte:

Held(xp,yp : int, rp : str, w : bool)

Die Parameter sollten Sie dann auch verwenden. Ändern Sie die Zeilen daher wie folgt ab:

self.x = xp

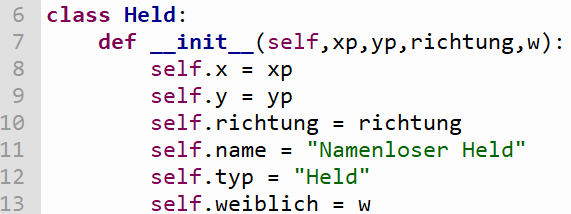
1. Der Parameter für die Richtung heißt wie das Attribut selbst. Der Parameter für weiblich enthält ebenfalls kein p. Macht dies Probleme?
2. Sobald Ihre Lösung **Level 35** löst („Level abgeschlossen“), laden Sie auch **Level 36** mit der gleichen Lösung. Ist dies ebenfalls direkt abgeschlossen, so haben Sie das Level gelöst und können fortfahren!

**Merke:**

Eine Klasse kann anhand des Klassendiagramms schnell implementiert werden. Eingeleitet wird eine Klassendefinition mit dem Schlüsselwort **class**, gefolgt vom Namen der Klasse und einem Doppelpunkt.

|  |
| --- |
| **Held** |
| x : Integer  y : Integer  richtung : String  name : String  typ : String  weiblich : Boolean |
| Held(xp,yp,richtung,w)  … |

Beispiel



Alle Methoden der Klasse werden dann eingerückt implementiert, beginnend beim **Konstruktor** der Klasse. Obwohl er beim Aufruf wie die Klasse selbst heißt, heißt er bei der Implementierung **\_\_init\_\_** und enthält als Parameter immer self. Die übrigen Parameter werden aus dem dem Klassendiagramm übernommen.

Alle Attribute werden dann in der Form self.attribut initialisiert. Entweder:

* Durch Verwendung eines Parameters, z.B. bei x oder weiblich
* Durch Setzen eines sinnvollen Standardwerts, z.B. bei typ oder name.

**Aufgabe 2**

Im Konstruktor können nicht nur stumpfe Werte oder Parameter gesetzt werden. Ändern Sie den Konstruktor so um, dass „Namenloser Held“ bei männlichen und „Namenlose Heldin“ bei weiblichen Helden als Name gesetzt wird.

*Hinweis: Obwohl es sinnvoll wäre, tun Sie dies nicht beim Attribut Typ, da in diesem Fall die Spiellogik des Frameworks gebrochen wird.*

**Aufgabe 3**

Jetzt fügen wir der Helden-Klasse die Fähigkeit hinzu, sich zu bewegen. Unser Ziel wird es sein, die Funtionalität des Frameworks Schritt für Schritt wiederherzustellen, so dass Sie nach Implementierung der Klassen(methoden) **die Level mit der Tastatur** lösen können. Sie müssen also ab sofort keine Programme zum Lösen der Level selbst mehr schreiben.

1. Öffnen Sie **Level 37**. Dieses Level erfordert zum Lösen, dass sich der Held zum markierten Zielfeld bewegt und alle Bewegungsmethoden implementiert sind.
2. Fügen Sie der Klasse Held eine neue Methode **geh** hinzu, welche die gleiche Einrückung wie der Konstruktor besitzt:

**def** geh(self):

**if** self.richtung == "up":

    self.y = self.y + ...

Implementieren Sie die Methode geh vollständig. Testen Sie Ihre Implementierung, indem Sie Level 37 starten und den Helden mit Pfeiltaste oben **⬆️** bewegen.

1. Obwohl die Methode **geh** laut Klassendiagramm keine Parameter besitzt, müssen Methoden einer Klasse in Python immer den Parameter self erhalten. Testen Sie jetzt den Befehl held.geh() im Hauptprogramm, also dort, wo Sie auch zuvor programmiert hatten (Einrückung 0, z.B. direkt vor der Zeile framework.starten()).
2. Implementieren Sie auf die gleiche Art und Weise die Methoden links, rechts und zurueck, um Level 37 mithilfe der Pfeiltasten zu lösen.

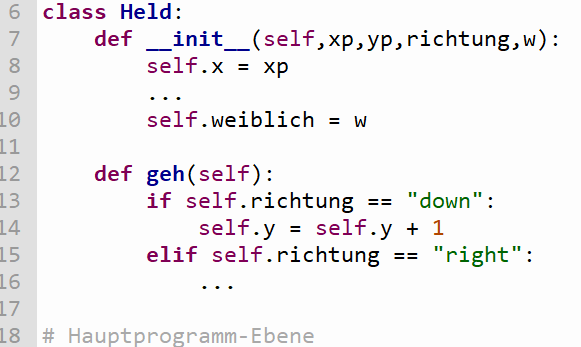
**Merke:**

Methoden einer Klasse können nach dem Konstruktor mit **def** methode(…) auf gleicher Einrückungsebene hinzugefügt werden.

Innerhalb der Klasse können die Attribute mit self.attribut referenziert werden. Das self ist dabei sozusagen eine **Selbstreferenz**, auf genau das Objekt, welches gerade die Methode ausführt. Dieses **self** muss als Parameter jeder Klassenmethode aufgeführt werden, obwohl dieser beim Aufruf später nicht mehr sichtbar ist.

|  |
| --- |
| **Held** |
| x : Integer  y : Integer  richtung : String  name : String  typ : String  weiblich : Boolean |
| Held(xp,yp,richtung,w)  geh()  links()  … |

Beispiel



Der erste Befehl, welcher sich wieder auf der Einrückungsstufe der Klassendefinition befindet, ist nicht mehr Teil der Klasse, sondern z.B. Teil des Hauptprogramms. Klassen können theoretisch auch auf Ebene eines Unterprogramms definiert werden.

**Übungsaufgaben**

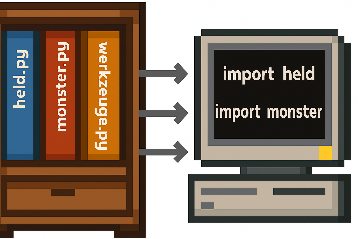
* Passen Sie den Konstruktor so an, dass nur up, down, left und right als Werte für richtung akzeptiert werden. Ansonsten wird down als Standard gesetzt
* Passen Sie den Konstruktor so an, dass nur positive Werte für x und y gesetzt werden können
* Schreiben Sie einen kleinen Namensgenerator, welcher beim Erzeugen des Helden einen zufälligen Namen erzeugt und setzt (entweder als Unterprogramm erzeuge\_name() oder direkt im Konstruktor

**8.3.4 Modularisierung**

Ein Bild, das Text, Cartoon, Puzzle enthält.

KI-generierte Inhalte können fehlerhaft sein.

Neben unserer Klasse Held gibt es noch Hindernis, Knappe, Monster, Schluessel, Tuer, Tor und weitere, welche wir noch nic ht genannt haben. Sie können sich vorstellen, dass dies sehr **unübersichtlich** wird, wenn der Code all dieser Klassen in der schueler.py liegen würde.



Aus diesem Grund lagern wir die Klassen in eigene Dateien aus, von wo aus wir sie später nach belieben in unsere Programme **importieren** können. Dies entspricht dem Prinzip der **Modularisierung**.

**Aufgabe 1**

1. Laden Sie **Level 38**. Das Level sollte leer sein.
2. **Ein Bild, das Text, Screenshot, Schrift, Dokument enthält.

   KI-generierte Inhalte können fehlerhaft sein.**Erstellen Sie eine leere Datei in Python und kopieren Sie dort die bisherige Klassendefinition des Helden hin. Speichern Sie diese Datei als **held.py** im Unterordner Klassen. Die Datei sollte dann (in etwa) so aussehen:
3. Löschen Sie jetzt die Klassendefinition aus der schueler.py. Sie sollte jetzt wieder so aussehen. Laden Sie erneut Level 38 und lösen Sie es mit der Tastatur.

**Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.**

**Aufgabe 2**

1. Laden Sie **Level 39**. In diesem Level wurde wohl vergessen, den Helden zu platzieren. Versuchen Sie jetzt durch Erzeugung eines passenden Helden-Objekts sowie Hinzufügen zum Level das Level abzuschließen. Welche Fehlermeldung erscheint?

*Tipp: Schauen Sie sich nochmal 8.3.2 an.*

Normalerweise kümmert sich das Framework um die Erzeugung der Objekte. Beim Programmieren des Frameworks wurde deswegen darauf geachtet, dass dieses Zugriff auf alle Klassen besitzt. Da Sie außerhalb des Frameworks ein Helden-Objekt Ihrer eigenen Helden-Klasse erzeugen möchten, müssen Sie sich selbst darum kümmern.

1. Um Ihre Heldenklasse zu importieren müssen Sie zunächst den folgenden Befehl ausführen, bevor Sie den Helden erzeugen.

**from** klassen.held **import** Held

Versuchen Sie nun erneut das Level zu lösen.

1. Testen Sie alternativ auch die folgenden Befehle. Welche funktionieren, welche nicht?

**from** klassen.held **import** **\***

**import** klassen.held

1. Was passiert, wenn der Import-Befehl nach dem Erzeugen des Helden steht?

**Erkenntnis**

Um Programmcode in unsere ausführende Datei A aus einer anderen Datei B zu \_\_\_\_\_\_\_\_\_\_\_\_\_, welche relativ zur ausführenden Datei in einem \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ liegt, so müssen wir diesen beim Import-Befehl mit angeben, gefolgt von einem \_\_\_\_\_\_\_\_\_\_\_\_ und dem Namen der \_\_\_\_\_\_\_\_\_\_\_\_.

Wäre Datei **B** im Unterordner **unter**, so wäre der passende Import-Befehl:

from import \*

**Aufgabe 3**

Wir möchten jetzt unseren Helden zufällig bewegen. Dazu steht im Ordner **framework** bereits eine Klasse für einen Würfel zur Verfügung, welche in der Datei werkzeug.py liegt. Hier ist das zugehörige Klassendiagramm:

|  |
| --- |
| Wuerfel |
| seiten : int |
| Wuerfel(s : int)  werfen() : int |

1. Bleiben Sie bei **Level 39**. Dort sollte noch Ihr Code zur Erzeugung des Helden liegen:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

1. Import-Befehle können zwar an (fast) beliebiger Position im Code stehen (vgl. Aufgabe 1 d) ), sollten aber eher gebündelt am Anfang des Programms stehen. Importieren Sie die Würfel-Klasse genau wie die Helden-Klasse aus dem Unterordner **framework** in Zeile 6, also nach dem Import der Helden-Klasse.
2. Erzeugen Sie jetzt ein neues Würfel-Objekt mit 6 Seiten und verwenden Sie es, um eine zufällige Zahl **zahl** zu erzeugen.
3. Unser Held soll dann genau so viele Schritte machen, wie gewürfelt wurde. Führen Sie das Programm einige Male aus. Ihnen sollten zwei Dinge auffallen:

6. Ersetzen Sie den Import-Befehl jetzt durch die Variante mit **\***, also

**from** framework.werkzeug **import** **\***

Führen Sie Ihr Programm erneut aus. Welche Beobachtung machen Sie? Öffnen Sie die werkzeug.py und finden Sie heraus, woher dieses Verhalten stammt.

**Merke:**

Unterprogramme und Klassen können über den **import** Befehl in andere Programme eingebunden werden. Dabei können entweder **integrierte Standardbibliotheken**, wie z.B. **random** eingebunden werden oder auch **eigene Dateien**, welche sich in **relativer Nähe** zur **importierenden Datei** befinden müssen.

Dabei gibt es grundsätzlich **drei Methoden**, um diese einzubinden.

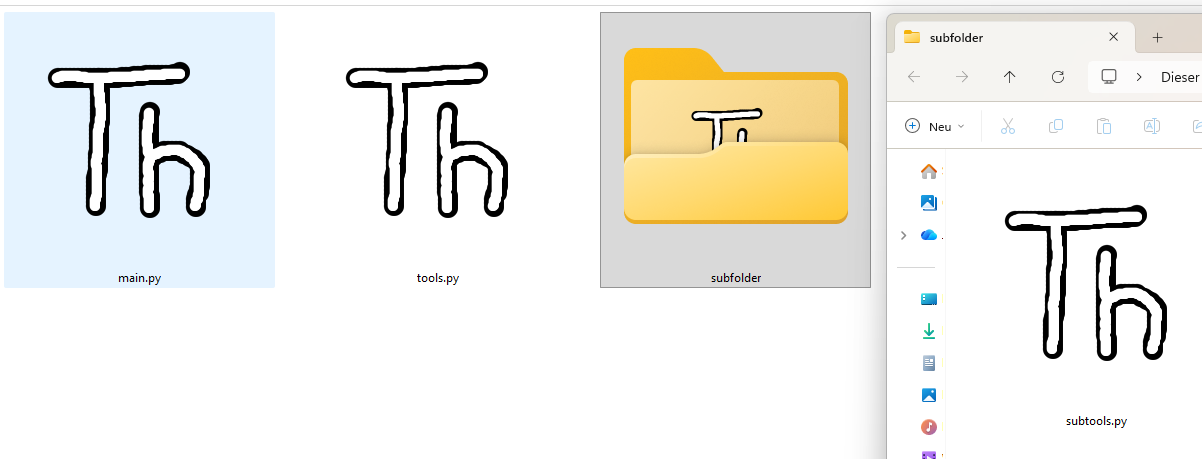
**Variante 1:**

Durch den Befehl **from** modul **import** **\*** wird im Grunde die gesamte Zieldatei in die importierende Datei eingefügt. Dies kann insbesondere dazu führen, dass bestehende Variablen, Unterprogramme oder Klassen durch den Import **überschrieben** werden. Man spricht auch davon, dass der **Namensraum** dann „**zugemüllt**“ wird (*engl. namespace pullution*). Es bedeutet, dass weniger Namen für Variablen, Methoden und Klassen zur Verfügung stehen und sollte daher nur bei kleinen Dateien verwendet werden.

**Beispiel:**

Der gesamte Inhalt der **tools.py** soll in die **main.py** importiert werden. Da sich die **tools.py** im gleichen Ordner wie die **main.py** befindet, schreiben wir

**from** tools **import** \*



**Variante 2:**

Möchte man nur einzelne Dinge importieren, so gibt man diese statt \* explizit an.

**Beispiel:**

Aus der subtools.py sollen die Variablen alter, das Unterprogramm zaehle und die Klasse Datum importiert werden. Sie befindet sich im Unterordner subfolder. Wir schreiben:

**from** subfolder.subtools **import** alter, zaehle, Datum

**Variante 3:**

Möchte man ohne die Nachteile von Variante 1 alles aus einem Modul importieren, so kann dieses mit **import modul** importiert werden. Dann muss der Modulname bei jedem Zugriff vorangeschrieben werden.

**Beispiel:**

**import** random importiert die gesamte random Bibliothek. Um die Methoden zu verwenden schreiben wir z.B. x = random.randint(1,5) oder print(random.uniform(a,b)).

**Übung**

Implementieren Sie im Unterordner klassen eine eigene Klasse Wuerfel. Importieren Sie diese dann auf verschiedene Arten in der schueler.py und nutzen Sie diese.

|  |
| --- |
| Wuerfel |
| seiten : int  bisherige\_summen : int |
| Wuerfel(s : int)  werfen() : int  reset()  mehrfach\_wurf(anzahl : int) : int |

**8.3.5 Implementierung der Datenkapselung**



Wie Sie bereits in den Kapiteln 8.2.7 und 8.2.8 gelernt haben, sollten Sie das Prinzip der **Datenkapselung** beachten: um fehlerhaftes Manipulieren von Attributen zu vermeiden, wird der Zugriff auf diese von „außerhalb“ der Klasse gesperrt und ein kontrollierter Zugriff über setter und getter Methoden ermöglicht.

**Merke:**

Attribute und Methoden können als privat markiert werden. Private Attribute und Methoden können nur **innerhalb der Klassendefinition** direkt adressiert werden und können somit nicht mehr von außen manipuliert werden. Attribute und Methoden, die nicht als privat markiert sind, sind automatisch öffentlich.

Dazu werden in der Klassendefinition dem Bezeichner privater Attribute zwei Unterstriche **\_\_** vorangestellt:

self.attribut = wert 🌐 **Öffentliches** Attribut

self.\_\_attribut = wert 🔒 **Privates** Attribut

**Beispiele:**

self.\_\_x = 5 ✅ Interne Zugriffe sind weiterhin gestattet

held.\_\_x = 5 ❌ Zugriff von außen ist gesperrt

Im UML-Klassendiagramm werden öffentliche Elemente mit **+** und private mit **–** markiert.

methode1 ist privat

methode2 und methode3 öffentlich

attribut1 ist öffentlich

attribut2 und attribut3 privat

|  |
| --- |
| **Klassenbezeichner** |
| + attribut1 : typ  - attribut2 : typ  - attribut3 : typ  … |
| Klassenbezeichner(Parameterliste)  - methode1(…)  + methode2(…)  + methode3(…)  … |

Der Konstruktor muss nicht als öffentlich deklariert werden.

**Aufgabe 1**

1. Laden Sie **Level 40**. Die Anzeige funktioniert, aber das Level kann nicht abgeschlossen werden.
2. Betrachten Sie jetzt das folgende Klassendiagramm des Helden und setzen Sie die entsprechenden Attribute auf privat. Dies sollte Level 40 lösen, aber der Held fehlt.

|  |
| --- |
| **Held** |
| - x : Integer  - y : Integer  - richtung : String  + name : String  - typ : String  - weiblich : Boolean |
| **Held**(xp,yp,richtung,w)  + geh()  + links()  … |

1. Laden Sie **Level 41**. Der Held kann nicht mehr angezeigt werden, weil kein Zugriff aus seine Position mehr möglich ist. Testen Sie dies einmal selbst, indem Sie in der schueler.py zuerst den Namen des Helden (öffentliches Attribut) ausgeben und dann versuchen, die x Koordinate des Helden auszugeben. Testen Sie dabei auch, ob der Schutz durch Verwendung von Unterstrichen umgangen werden kann.
2. Leeren Sie Ihr Programm in der schueler.py. Um den Zugriff auf die Attribute weiter zu ermöglichen, implementieren wir jetzt **getter** für alle privaten Attribute, da es in diesem Fall keinen Grund gibt, den lesenden Zugriff zu verhindern. Ergänzen Sie zunächst den getter für die x-Koordinate und führen Sie das Programm aus.

**def** **get\_x**(self):

**return** self.\_\_x

1. Im Objektinspektor sollte die x-Koordinate jetzt ausgelesen werden können. Ergänzen Sie nach der gleichen Namenskonvention die getter für alle privaten Attribute. Wenn Sie alles richtig gemacht haben, wird der Held wieder angezeigt und das Level gelöst.
2. Ergänzen Sie in der schueler.py ein kleines Programm, das die Position des Helden in der folgenden Form ausgibt:

„Namenloser Held steht an Position (3,2).“

1. Leeren Sie die schueler.py wieder und laden Sie **Level 42**. Sie können das Level noch nicht abschließen, weil Ihre Geh- und Drehmethoden noch nicht angepasst wurden. Auch beim Zugriff auf private Attribute innerhalb der Klasse müssen die Unterstriche angebracht werden. Ergänzen Sie daher die Unterstriche bei allen Zugriffen auf Attribute innerhalb der Methoden. Wenn Sie alles richtig gemacht haben, wird die Funktionalität wiederhergestellt und der Held kann bewegt werden.
2. Laden Sie jetzt abschließend **Level 43**. Das Framework erwartet noch einen **setter** für das Richtungsattribut. Fügen Sie zunächst einen setter für das Namensattribut hinzu:

**def** **set\_name**(self,name):

**if** len(name<20)

        self.name = name

**else**:

**print**("Name zu lang!")

1. Ergänzen Sie einen setter für die Richtungsangabe und stellen Sie dabei sicher, dass nur gültige Richtungsangaben übergeben werden.

**Übungsaufgabe**: Lassen Sie auch Richtungen der Form "S", "W", "O" und "M" zu, wobei diese natürlich zur richtigen Form korrigiert werden müssen.

**Reflexionsfrage:**

Aus welchem Grund muss der Konstruktor nicht als öffentliche Methode deklariert werden?

**Aufgabe 2**

1. Implementieren Sie die Klasse **Hindernis** nach dem folgenden Klassendiagramm vollständig in die **hindernis.py** im Klassenordner, um **Level 44** zu lösen.
2. Implementieren Sie die Klasse **Zettel** nach dem folgenden Klassendiagramm vollständig in der **zettel.py** im Klassenordner, um **Level 45** zu lösen.

|  |
| --- |
| **Hindernis** |
| - x : Integer  - y : Integer  - typ : String |
| **Hindernis**(x,y : int, art : str)  + get\_x() : int  + get\_y() : int  + get\_typ() : str  + set\_typ(art : str) |

|  |
| --- |
| **Zettel** |
| - x : Integer  - y : Integer  - typ : String  - spruch : String |
| **Zettel**(x,y : int)  + get\_x() : int  + get\_y() : int  + get\_typ() : int  + get\_spruch() : int  + set\_spruch(s : str) |

*Hinweise zu Hindernis:*

*Ein Hindernis kann als Typ Baum, Busch oder Berg sein. Dies wird in Form eines Strings mit dem Parameter art übergeben. Es muss natürlich entsprechend geprüft werden, dass nur gültige Arten gesetzt werden.*

*Bei einem Zettel gibt es keine Einschränkungen für den Zauberspruch. Dieser wird auf einen beliebigen „Standardspruch“ ihrer Wahl gesetzt.*

**Zusätzliche Aufgaben**

* Ergänzen Sie beim Hindernis eine private Methode randomize(), welches die Art des Hindernis zufällig setzt. Im Konstruktor wird dann diese Methode verwendet, um eine zufällige Art zu setzen, unabhängig vom Parameter (sinnvollerweise sollte die Methode aber nur aufgerufen werden, wenn ein ungültiger Wert übergeben wurde)
* Ergänzen Sie beim Zettel eine Methode zufalls\_spruch(), der einen zufälligen Zauberspruch erzeugt.

**Arbeitsblatt: Implementierung des Knappen**



Wir haben nun den Helden, Hindernisse und den Zettel in den Grundzügen implementiert. Einige Klassen fehlen noch, die sich allerdings mit einem Trick schneller implementieren lassen.

**Aufgabe 1**

Implementieren Sie den Knappen, um **Level 46** zu lösen.

|  |
| --- |
| **Knappe** |
| - x : Integer  - y : Integer  - richtung : String  + name : String  - typ : String |
| **Knappe**(x,y : int, art : str)  + geh()  + links()  + rechts()  + get\_x() : int  + get\_y() : int  + get\_typ() : str  + get\_richtung : str  + set\_richtung(r : str) |

*Tipp: Schauen Sie mal in der held.py vorbei.*

Ihnen sind bestimmt einige **Gemeinsamkeiten** zum Helden aufgefallen, aber auch zu anderen Objekten in unserer Spielwelt. So haben z.B. alle Objekte eine x- und y- Koordinate und passende getter, aber nicht alle Objekte haben eine Richtung oder können sich bewegen.

In der Objektorientierten Programmierung ist es möglich, Klassen aufeinander aufzubauen, so dass es eine **Basisklasse** gibt und eine **erweiterte Klasse**, welche alle Attribute und Methoden der Basisklasse „erbt“ und um weitere ergänzt. Diese erweiterte Klasse kann dann wieder als Basis dienen oder die Basisklasse kann zu anderen Klassen ergänzt werden. Dabei dürfen auch „Zwischenklassen“ verwendet werden, die gar nicht als richtige Klasse eingesetzt werden.

**Aufgabe 2**

Entfwerfen Sie eine solche Vererbungshierarchie für die Klassen unseres Frameworks: Held, Knappe, Monster, Bogenschütze, Hindernis, Herz, Tor, Tür (mit Spruch und farbige), Zettel und Schlüssel.

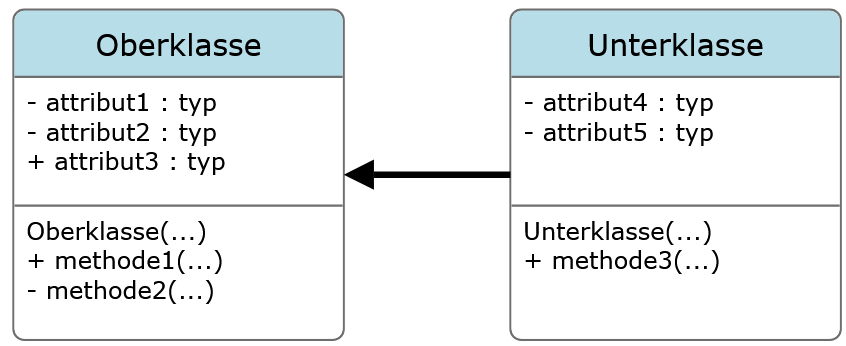
**8.4 Vererbung und Klassenbeziehungen**

Um Code bestehender Klassen wiederzuverwenden und zu ergänzen können Klassen durch das Prinzip der **Vererbung** aufeinander aufgebaut werden. Neben der Vermeidung von Codeduplizierung bietet dieses Vorgehen weitere Vorteile bei der Gestaltung von Projekten.

**Merke:**

Nach dem Prinzip der **Vererbung** kann eine Klasse **alle** **Attribute** und **Methoden** einer vorhandenen Klasse **übernehmen** (Fachbegriff: **erben**) und ergänzen. Die erbende Klasse nennen wir **Unterklasse**, abgeleitete Klasse oder Subklasse; die vererbende Klasse **Oberklasse**, Superklasse oder Basisklasse.

Im UML-Diagramm wird dies mit einem **Pfeil** mit einem Dreieck als Spitze dargestellt und wird als **IST**-Beziehung gelesen.



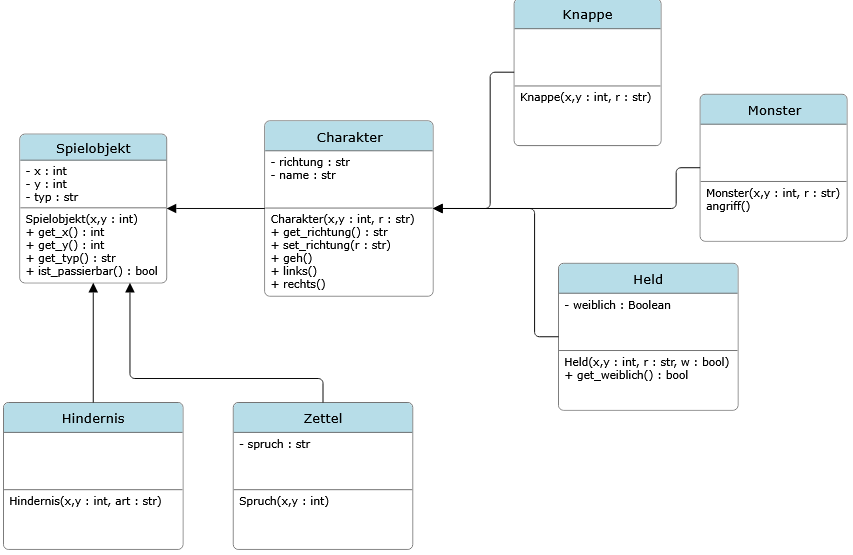
IST

Dies bedeutet, dass alle Objekte der Unterklasse automatisch auch als Objekte der Oberklasse gelten. Objekte der Unterklasse besitzen insgesamt fünf Attribute, von denen 3 von der Oberklasse geerbt wurden. Sie besitzen insgesamt drei Methoden, von denen zwei von der Oberklasse geerbt wurden.

**Beispiel:**

Eine Klasse Hund könnte als Unterklasse der Oberklasse Tier modelliert werden. Dann gilt: Jeder Hund ist ein Tier.

Für unser Framework verwenden wir die folgende Klassenhierarchie:



**8.4.1 Implementierung der Vererbung**

**Aufgabe 1**

Unsere neue Basisklasse ist **Spielobjekt**. Laden Sie **Level 47** und implementieren Sie die Klasse Spielobjekt in der spielobjekt.py. Da wir später prüfen müssen, wohin sich die Charaktere bewegen dürfen, benötigen wir zudem eine neue Methode **ist\_passierbar()**. Standardmäßig können Felder, auf denen sich Spielobjekte befinden, nicht betreten werden. Da ein generisches Spielobjekt keinen Typ besitzt, setzen wir dieses Attribut auf **None**.

**Aufgabe 2**

Wir implementieren jetzt die Klasse Hindernis als Unterklasse von Spielobjekt.

1. Laden Sie **Level 48**. Es sollte leer sein.
2. Öffnen Sie die **hindernis.py**. Zunächst müssen wir die Oberklasse importieren, damit sie dem Modul bekannt ist. Ergänzen Sie am Anfang der Datei (noch vor der Klassendefinition) den Befehl, um die Klasse Spielobjekt aus der Datei spielobjekt zu importieren. Da sich dort nur die gewünschte Klasse befindet, können Sie entweder gezielt die Klasse oder alles aus der Datei importieren.

*Tipp: Kapitel 8.3.4*

1. Um die Klasse von Spielobjekt erben zu lassen, ersetzen Sie die Klassendefinition durch:

**class** **Hindernis**(Spielobjekt):

Die Definition des Konstruktors bleibt davon unberührt, weil wir weiterhin die Position sowie die Art des Hindernis als Parameter benötigen. Da sich die Oberklasse um das Setzen aller geerbten Attribute „kümmern“ soll, löschen wir zunächst alle Befehle im Konstruktor und ersetzen diese dann durch:

**def** **\_\_init\_\_**(self,x,y,art):

super().\_\_init\_\_(x,y)

Der Befehl super().\_\_init\_\_(x,y) ruft den Konstruktor der Oberklasse auf und übergibt diesem die benötigten Attribute (in diesem Fall x und y). Jetzt müssen wir aber noch die Art des Hindernis setzen. Wir ergänzen:

self.\_Spielobjekt\_\_typ = art

1. Führen Sie erneut **Level 48** aus. Es sollte gelöst sein.

**Merke:**

Um eine Unterklasse von einer Oberklasse erben zu lassen, notieren wir in der Klassendefinition

**class** **Unterklasse**(Oberklasse):

Im Konstruktor muss dann der Konstruktor der Oberklasse mit allen benötigten Parametern aufgerufen werden. Im Anschluss können der Unterklasse neue Attribute hinzugefügt werden. Geerbte Attribute können ebenfalls verändert werden, bei privaten Attribute jedoch in besonderer Schreibweise:

**def** **\_\_init\_\_**(self,parameter\_1,parameter\_2,…,parameter\_n):

super().**\_\_init\_\_**(parameter1,wert,…)

self.attribut4 = parameter2

self.\_Oberklasse\_\_attribut1 = wert

…

**Aufgabe 2**

Implementieren Sie die Klasse Zettel in der zettel.py als Unterklasse von Spielobjekt, um **Level 49** zu lösen.

Ein Bild, das Text, Grafiken, Clipart, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.**Exkurs: Private Attribute in Python vererben**

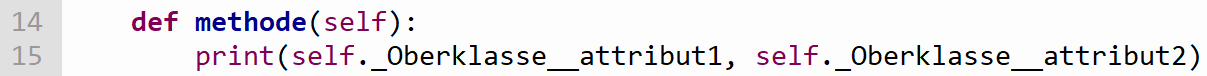
In anderen Programmiersprachen können private Attribute wie öffentliche Attribute vererbt und manipuliert werden. Der folgende Code würde in anderen Programmiersprachen (in deren Syntax) daher funktionieren, führt aber in Python beim Aufrufen der Unterklassen-Methode zu einem Fehler:

Ein Bild, das Text, Screenshot, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

**AttributeError: 'Unterklasse' object has no attribute '\_Unterklasse\_\_attribut1'**

Stattdessen muss in Python die folgende Schreibweise verwendet werden:



Eine Angabe der vererbenden Klasse ist dabei nicht ausreichend. Es muss die Klasse angegeben werden, welche die privaten Attribute erstmals gesetzt hat. Dies ist natürlich extrem unübersichtlich und unpraktikabel.

In der Praxis werden die Attribute in Python daher selten als privat markiert. Im Sinne der Anschaulichkeit bei der Umsetzung der Vererbung setzen wir daher alle Attribute unserer Klassen auf öffentlich. Die Verwendung von gettern und settern wird dennoch aus den bekannten Gründen empfohlen.

**Aufgabe**

Ersetzen Sie die spielobjekt.py, hindernis.py und zettel.py durch die Versionen in der OSS. Die Dateien behalten ihre ursprünglichen Namen. Lösen Sie damit **Level 50**.

**Reflexionsfrage**

Aus welchem Grund ist es im Grunde unwichtig, dass die Attribute in der Praxis öffentlich und nicht privat sind?

**8.4.2 Die Charakter Klasse**

TO-DO

<Einführungstext> <Abbildung>

Ein Bild, das Text, Screenshot, Diagramm, Schrift enthält.

KI-generierte Inhalte können fehlerhaft sein.

Um unsere Vererbungshierarchie weiter umzusetzen müssen wir jetzt die **Basisklasse** für unseren Helden, den Knappen und die Monster implementieren: die Klasse **Charakter**

Diese erbt vom bereits implementierten Spielbobjekt und besitzt eigentlich nur Methoden, welche wir bereits in der Helden Klasse implementiert haben.

**Aufgabe 1**

1. Laden Sie zur Vereinfachung der Implementierung die Helden Klasse mit öffentlichen Methoden aus der OSS herunter. Sie können Methoden aus dieser Datei in die neue Charakter-Klasse übernehmen. Alternativ können Sie Ihre eigene Helden oder Knappen Klasse verwenden und die Unterstriche entfernen.
2. Lösen Sie **Level 51**, indem Sie die Klasse Charakter nach dem folgenden UML-Klassendiagramm implementieren:

|  |
| --- |
| Charakter (Spielobjekt) |
| + richtung : str  + name : str |
| **Charakter** (x,y : int, r : str)  + set\_richtung(r : str)  + get\_richtung() : str  + geh()  + links()  + rechts() |

**Aufgabe 2**

1. Lösen Sie **Level 52**, indem Sie Ihre held.py so anpassen, dass die Helden-Klasse jetzt von **Charakter** erbt und um die Methode get\_weiblich() ergänzt.
2. Lösen Sie **Level 53**, indem Sie analog die Klasse **Knappe** anpassen.
3. Wenn Sie alles richtig gemacht haben, können Sie den Helden jetzt auch mit der Tastatur oder mit Befehlen steuern. Welches Problem tritt jedoch auf?

**8.4.3 Beziehungen zwischen Klassen**

Damit Charaktere erkennen können, ob eine Bewegung gültig ist oder nicht, benötigen sie Zugriff auf das Level und dessen Methoden. Das Framework ist so eingestellt, dass nach dem Erzeugen des Levels versucht wird, bei allen Charakteren das Level als Attribut zu setzen, damit ein Zugriff zu diesem besteht.

**Aufgabe 1**

1. Fügen Sie der Klasse Charakter ein neues Attribut **level** hinzu. Da es nicht mit dem Konstruktor übergeben wird und wir auf das Framework warten müssen, setzen wir es zunächst auf **None**. Dies ist ein Platzhalterwert und bedeutet wortwörtlich „Nichts“.
2. Fügen Sie jetzt die setter-Methode set\_level hinzu, welches das Attribut auf das als Parameter übergebene Level-Objekt setzt. Dies sollte **Level 54** lösen.

**Merke:**

**None** steht in Python für „keinen Wert“ und wird als klarer **Platzhalter** verwendet, wenn eine Variable oder ein Parameter noch keinen echten Wert besitzt.

Sie können damit also kennzeichnen, wenn ein konkreter Wert irgendwann verfügbar sein soll, aber zu Beginn noch unbekannt ist. Mit **None**-Objekten können im Grunde nur zwei sinnvolle Operationen durchgeführt werden:

**1.** Zuweisung: variable = None

**2.** Prüfen, ob None: **if** variable == **None**:

Jetzt kümmern wir uns darum, dass Charaktere keine ungültigen Bewegungen mehr durchführen können. Dazu nutzen wir die folgenden Methoden des Levels:

ist\_innerhalb(x,y): Liefert True zurück, wenn die übergebenen Zielkoordinaten noch innerhalb des Levels liegen, sonst False.

gib\_objekt\_bei(x,y): Liefert das Objekt an der angebenen Position zurück. Außerhalb des Levels sowie Stellen, wo begehbarer Weg ist, wird None zurückgeliefert, da dort kein Objekt ist.

Bei Objekten unterscheiden wir ebenfalls, ob sie betretbar sind oder nicht. Charaktere dürfen nicht in Hindernisse oder andere Charaktere laufen, aber können auf Zettel und Herzen laufen, um sie einzusammeln. Wir benötigen daher von Objekten die Methode

ist\_passierbar(): Liefert True zurück, wenn das Objekt „betreten“ werden darf

**Aufgabe 2**

Lösen Sie **Level 55**, indem Sie die geh-Methode von Charakter so anpassen, dass nur gültige geh-Aktionen durchgeführt werden können. Implementieren Sie selbstständig. Alternativ finden Sie weiter unten die dahinterstehende Logik. Testen Sie Ihre Lösung schrittweise mit der Tastatursteuerung. Eine Sache wird allerdings noch nicht funktionieren. Vielleicht können Sie dies bereits alleine lösen.

**Schritt 1**: Bestimmen Sie die Zielkoordinaten der Bewegung, also wohin der Charakter gehen möchte. Speichern Sie dies in Variablen.

**Schritt 2**: Prüfen Sie, ob die Zielkoordinaten innerhalb des Levels liegen.

**Schritt 3**: Bestimmen Sie, welches Objekt am Ziel vorliegt, bzw. ob dort überhaupt eines ist.

**Schritt 4**: Jetzt können Sie prüfen, ob das Objekt betretbar ist.

Wenn eine Prüfung ergeben hat, dass die Bewegung gültig war, setzen Sie die Koordinaten auf die Zielkoordinaten.

**Merke:**

Ist ein Objekt Attribut eines anderen Objekts, so stehen diese beiden Objekte, bzw. die jeweiligen Klassen in einer **Hat-Beziehung**. Im Framework hat die Klasse Charakter permanent Zugriff auf ein Objekt der Klasse Level, weswegen man sagt:

Die Klasse Charakter hat die Klasse Level.

Weiter wird unterschieden zwischen starker und schwacher Hat-Beziehung. Es sei die Situation gegeben: **Klasse A hat Klasse B**.

**Aggregation (schwache Hat-Beziehung):**

Existieren Objekte der Klasse B unabhängig von Klasse A, so handelt es sich um eine schwache Hat-Beziehung. Ein häufiger Indikator ist, dass Objekte von Klasse B nicht von Klasse A selbst erzeugt werden.

**Komposition (starke Hat-Beziehung):**

Ist die Existenz der Klasse B an die Existenz von Klasse A gebunden, so handelt es sich um eine starke Hat-Beziehung. Würden Objekte der Klasse A gelöscht werden, so bestünde danach kein Zugriff mehr auf die jeweiligen Objekte der Klasse B. Ein häufiger Indikator ist, dass Klasse A selbst Objekte der Klasse B erzeugt.

**Am Beispiel:**

Level-Objekte werden nicht von der Objekten der Klasse Charakter erzeugt und können auch ohne Charakter-Objekte existieren. Es handelt sich also um eine **Aggration**.

Im UML-Diagramm stellen wir die Aggration mit leeren Rauten dar:

Ein Bild, das Text, Screenshot, Schrift, Zahl enthält.

KI-generierte Inhalte können fehlerhaft sein.

Die Kompositon sieht ähnlich aus, verwendet aber ausgefüllte Rauten: 

**8.4.4 Komposition**

Held und Knappe können im Spiel nicht nur herumlaufen, sondern auch Gegenstände wie Schlüssel aufsammeln. Damit beide Klassen die gleichen Methoden zur Verwaltung der Gegenstände zu Verfügung haben (und ohne den Code duplizieren zu müssen), implementieren wir ein Inventar-System, also eine Art Rucksack, das die Gegenstände verwaltet.

Wir benötigen zwei neue Klassen

|  |
| --- |
| **Gegenstand** |
| + art : str  + typ : str  + im\_inventar : bool |
| **Gegenstand**(art : str)  + sammeln()  + gib\_typ() |

|  |
| --- |
| **Inventar** |
| + items : Gegenstand[]  + kapazitaet : int  + gold : int |
| **Inventar**()  + item\_hinzufuegen(i : Gegenstand)  + hat\_item(art : str)  + anzahl\_items() : int  + gib\_item\_nummer(index : int)  + ist\_voll() : bool  + gold\_sammeln(menge : int)  + gib\_gold() : int  + gib\_kapazitaet() : int |

**Aufgabe 1**

Implementieren Sie die beiden Klassen in die **gegenstand.py** und **inventar.py**, um **Level 56** zu lösen. Eine leere Liste erzeugen Sie z.B. mit l = []. Sie können mit l.append(x) etwas hinzufuegen und mit len(l) die Länge einer Liste bestimmen.

**Aufgabe 2**

Wir passen die Klassen **Held** und **Knappe** jetzt an. Sie erben weiter von Charakter und kümmern sich nun selbst um die Erzeugung ihreres Inventars.

|  |
| --- |
| **Held (Charakter)** |
| …  + rucksack : Inventar |
| **…** |

|  |
| --- |
| **Knappe (Charakter)** |
| …  + rucksack : Inventar |
| **…** |

Fügen Sie den beiden Klassen das Attribut rucksack hinzu und setzen Sie dies auf ein neu erzeugtes Inventar-Objekt. Der Held muss zusätzlich ein neues Gegenstand-Objekt (Schwert) erzeugen und es danach seinem Rucksack hinzfügen. Haben Sie alles richtig gemacht, wird **Level 57** gelöst.

**Aufgabe 3**

Füllen Sie den folgenden Lückentext aus:

Objekte der Klasse Held haben jetzt als Attribut ein Objekt der Klasse Inventar. Damit besteht zwischen den beiden Klassen eine Hat-Beziehung. Zusätzlich wird das Inventar-Objekt von der Klasse Held selbst erzeugt und ist nur über dieses referenzierbar. Würde man das Helden-Objekt löschen, so existiert auch keine Referenz mehr zum Inventar-Objekt. Damit handelt es sich hier um eine starke Hat-Beziehung, Fachbegriff: Komposition.

Ein weiterer Hinweis dafür ist, dass die Inventar-Objekte von Objekten der Klasse Held erzeugt werden. Dies muss jedoch nicht erfüllt sein, wie das folgende Beispiel zeigt:

Objekte der Klasse Held erzeugen Objekte der Klasse Gegenstand und übergeben diese an das Inventar-Objekt. Da sich der Held den Gegenstand nicht direkt als Attribut speichert, besteht zwischen den beiden Klassen keine Hat-Beziehung. Da die Gegenstände im Inventar als Attribut (in Form einer Liste) gespeichert werden, besteht also zwischen Inventar und Gegenstand eine starke Hat-Beziehung, obwohl die Gegenstände vom Helden erzeugt werden.

**8.4.5 Kennt-Beziehung**

Obwohl die Klasse Held Objekte der Klasse Gegenstand erzeugt, liegt hier keine Hat-Beziehung vor. Stattdessen handelt es sich um eine Kennt-Beziehung.

**Merke:**

Kennt ein Objekt ein anderes Objekt nur dadurch, dass es darauf **zugreifen** oder es **verwenden** kann, ohne es selbst zu besitzen oder dauerhaft zu verwalten, so spricht man von einer **Kennt-Beziehung**.

In diesem Fall ist das Objekt **kein Attribut**, sondern wird einer Methode **übergeben**, kurzzeitig gespeichert oder nur vorübergehend benutzt. Man sagt dann:

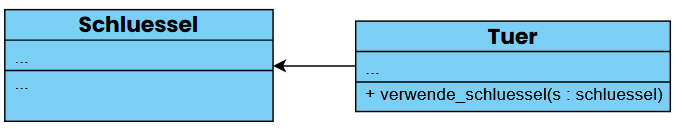
**Klasse A kennt Klasse B.**

Eine Kennt-Beziehung ist **weder** eine Aggregation **noch** eine Komposition, da Klasse A nicht für die Existenz von Objekten der Klasse B verantwortlich ist und diese auch nicht dauerhaft enthält.

**Am Beispiel:**

Helden Objekte erzeugen Objekte der Klasse Gegenstand, speichern sich diese jedoch nicht dauerhaft als Attribut. Ein weiteres Beispiel wäre das Verhältnis zwischen Tür und Schlüssel, da die Tür über einen Parameter ein Schlüssel-Objekt erhält. In beiden Fällen ist der Zugriff nur **temorär** und somit eine Kennt-Beziehung.

Im UML-Diagramm stellen wir eine Kennt-Beziehung mit einem **einfachen Pfeil** dar:



**Aufgabe 1**

Stellen Sie das Verhältnis zwischen Held und Gegenstand als UML-Diagramm dar.

**Aufgabe 2**

Identifizieren Sie in unserem Framework Beziehungsstrukturen, welche einer Kennt-Beziehung entsprechen.

**8.4.5 Polymorphie**

Bevor Sie damit beginnen können, die Funktionalität Schritt für Schritt vollständig wiederherzustellen, gibt es noch ein letztes Konzept, das Sie im Zuge der Vererbung kennen sollten: das Prinzip der **Polymorphie**.

**Aufgabe 1**

Öffnen Sie **Level 58**. Aus welchem Grund können Sie das Level nicht lösen? Argumentieren Sie anhand des Vererbungsschemas:

Ein Bild, das Text, Screenshot, Schrift, Diagramm enthält.

KI-generierte Inhalte können fehlerhaft sein.

**Aufgabe 2**

Öffnen Sie Ihre **zettel.py** und implementieren Sie in dieser Klasse die Methode ist\_passierbar() erneut. Zettel sind, ebenso wie Herzen, immer betretbar.

**Merke:**

Wird eine Methode in einer erbenden Klasse neu implementiert, um deren Verhalten anzupassen, so handelt es sich um eine **polymorphe Methode** und wir sprechen vom Prinzip der **Polymorphie**. Die ursprüngliche Methode kann weiterhin über super().methode() aufgerufen werden.

Im Beispiel muss die Methode ist\_passierbar in der Klasse Zettel, welche von Spielobjekt geerbt wird, neu implementiert werden. Damit ist ist\_passierbar für die Klasse Zettel eine polymorphe Methode.

**Aufgabe 3**

1. Implementieren Sie in der Klasse Charakter eine Methode zufallsName(), welche aus einer Liste an „normalen“ Namen einen zufälligen auswählt.

*Tipps:*

* Sie können eine Python Liste direkt in der Form liste = [5,6,7,8] erzeugen.
* Importieren Sie randint aus der random Bibliothek
* Der Zugriff auf Elemente einer Liste war mit liste[x] möglich

1. Implementieren Sie die Methode jetzt polymorph in den Klassen Held und Knappe neu. Seien Sie kreativ bei der Namensauswahl.

**8.5 Framework vervollständigen**

Ab diesem Punkt können Sie versuchen, das Framework vollständig neu zu implementieren. Dazu stehen Ihnen die **Level 200 bis 234** zur Verfügung. Es handelt sich jeweils um die orginalen Level **0 bis 34**, wobei jetzt Ihre eigenen Klassen verwendet werden.

Siegbedingungen der Level sind jedoch (Stand Version 1.0, Dezember 2025) in allen Fällen „Alle Herzen wurden gesammelt“. Es wird also nicht geprüft, ob Sie die Klassen korrekt implementiert haben, so dass z.B. Türen und Schlüssel korrekt funktionieren oder das Monster den Helden tatsächlich erschlägt.

Für viele Klassen und deren Methoden benötigen Sie die folgenden Methoden des Level-Objekts:

* ist\_weg(x,y : int) : boolean
* ist\_frontal\_zu\_monster(c : Charakter, m : Monster) : boolean
* gib\_objekt\_bei(x,y : int) : Spielobjekt
* objekt\_art\_an(x,y : int) : String
* entferne\_objekt(obj : Spielobjekt)
* gib\_objekte() : Spielobjekt-Liste
* gib\_groesse\_x() / gib\_groesse\_y() : int
* angriffs\_animation(c : Charakter)
* setze\_besiegt(c : Charakter)

Beachten Sie, dass Sie für vieles nun selbst sorgen müssen. Wenn sie beispielsweise später den Angriff des Monsters implementieren, dann können Sie zwar über das Level die Animation spielen, Sie müssen aber selbst dafür sorgen, dass der Held dann nichts mehr tun kann.

**To-Do 26/27**

**8.4.5 Komposition?**

Gegenstand\_hinzufuegen prüft, ob übergebener Gegenstand Schluessel ist

Rucksack durchsuchen nach Schlüssel, um Tür zu öffnen

Weitere Hat-Beziehung?

Rest: Implementierung der Spiellogik

(Türe, Tore, Spruch, …)