Technische Hochschule Augsburg

Programmieren 3, WS 25/26

Persönliche Beiträge zum Semesterprojekt

Tchinda Souleyman Sahlong

Inhalt

1 Individuelle Beiträge zum Projekt

1.1 Projektkontext

Im Semesterprojekt *PyMonster / Biester* wird ein einfaches, rundenbasiertes Spiel simuliert. In jedem Zug erhält ein sogenanntes **Beast** (Biest) ein **Sichtfeld von 7×7 Zeichen**. Darin können z. B. vorkommen:

- `B` – das eigene Biest in der Mitte
- `*` – Futter
- `.` – leere Felder
- `>` - Gegner
- `<` - Beast mit kleine Energie

Auf Basis dieses Sichtfeldes soll das Beast:

- seine Umgebung auswerten,
- sinnvolle Bewegungen planen (z. B. Futter jagen),
- und eine Liste von **Zugvektoren** `(dx, dy)` zurückgeben.

Die Positionen sind dabei immer **relativ zur Biest-Position**:
`dx` gibt an, wie weit das Biest nach links/rechts geht,
`dy` wie weit nach oben/unten.

1.2 Unterteilung in meheren Datei

Bei dieser Aufgabe ging es darum, einen *Biester-Client* zu erstellen, der mit unserer Strategie gegen die anderen Gruppen im Programmieren-Praktikum antritt. Dabei gibt es Werte- und Testläufe bei denen unsere Clients immer auf dem Server ins Rennen geschickt werden. Am Ende eines Laufs gab es *Rankings*, welche die Leistung der Clients/Gruppen widerspiegeln.

Die Grundidee unseres Clients war es, unsere Strategie in mehrere Module aufzuteilen, damit wir kleinere Teilprobleme und Aufgaben haben, die wir dann schön getrennt voneinander programmieren können. Durch diesen Gedanken können wir nicht nur Code gut separieren, sondern es macht unseren Code einfach erweiterbar da nur ein neues Modul hinzugefügt werden muss.

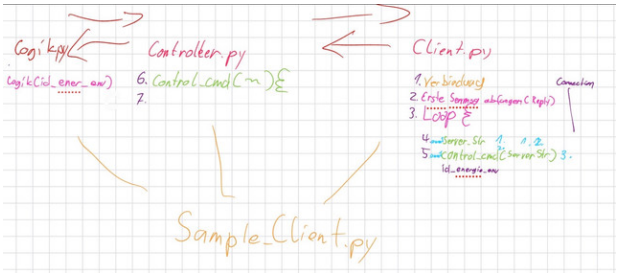


Figure 1: “Erste Skizze, wie wir die Aufteilung des SampleClients.py auf unsere neue Struktur geplant haben.”

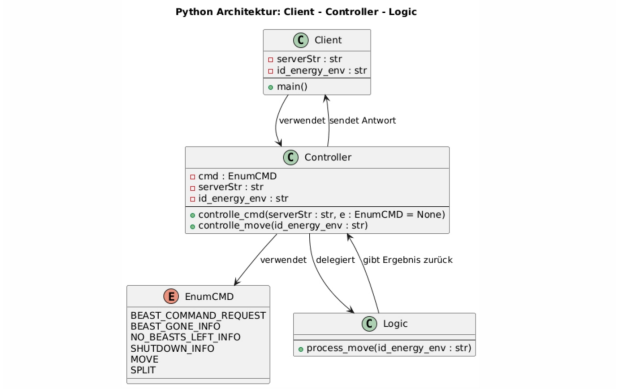


Figure 2: “Wie wir es als Anfangsimplementierung geplant haben”

Die konkrete Strategie wurde umgesetzt durch die Module (Food, Hunt, Kill, Escape, Split) durch Prioritäten, die den Biestern dynamisch zugeordnet wurden, konnten wir dann die einzelnen Module unterschiedlich gewichten und bewerten. Darum hat sich die Hauptlogik gekümmert.

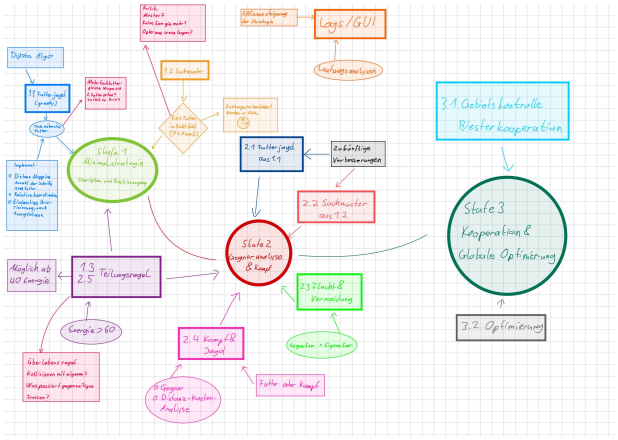


Figure 3: "Hier wird gezeigt, wie die Strategie geplant ist, dass sie in 3 Stufen erweitert werden kann. Man sieht deutlich die einzelnen Module und zu welchem Teil der Strategie sie gehören."

1.3 Architektur

Im Folgenden sind die wichtigsten Module unseres *Biester-Clients* und ihre Aufgaben aufgeführt:

- `client.py` – kümmert sich um die Kommunikation mit dem Server (Senden und Empfangen von Nachrichten).
- `controller.py` – wertet die vom Server empfangenen Informationen aus und legt fest, wie unser *Biester-Client* darauf reagiert.
- `logic.py` – bildet das zentrale „Gehirn" des Clients, ruft die benötigten Module auf und entscheidet den nächsten Zug.
- `beast.py` – bündelt alle Module und Metainformationen des aktiven Biests.
- `logger.py` – hier werden die wichtigsten Informationen unserer Biester geloggt, um diese in unserer GUI zu analysieren.
- `utils.py` – stellt Hilfsmodul und gemeinsame Attribute bereit, die in mehreren anderen Modulen verwendet werden.

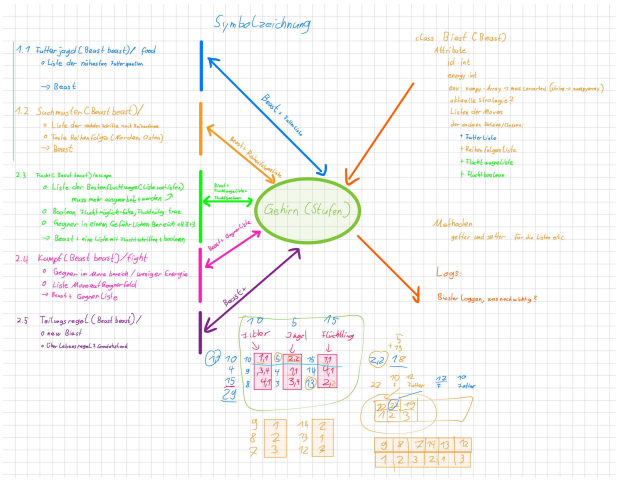
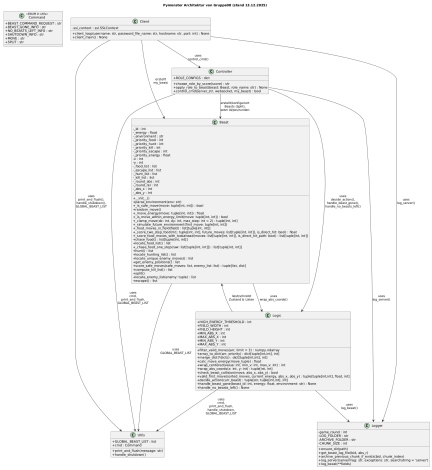


Figure 4: “Eine Skizze, wie die einzelnen Module zusammenspielen und die Logik unseren **perfekten Move** berechnet. Es wird auch gezeigt, wie die Beast-Klasse zum zentralen Helfer des **Gehirns** wird”

Klassedigramm fürs geasamte Projekt hier wird einfach dargestellt wie alle unsere Klasse und Module zusammenspielen und welche Komponente mit was kommuniziert.



1.4 Übersicht meiner individuellen Beiträge

Im Projekt war ich insbesondere für die **Beast-Logik** und die dazu passenden **Tests** zuständig.
Meine konkreten Beiträge waren:

- Implementierung von `parse_environment`
  – Umwandlung des 49-Zeichen-Strings in ein 7×7-Feld (Numpy-Array)

- Implementierung und Erweiterung von `locate_food_list`
  – Suchen aller Futterpositionen und Umrechnung in relative Koordinaten `(dx, dy)`
- Implementierung und schrittweise Erweiterung von `chase_food`
  – Basis-Strategie zur Futterjagd
  – Erweiterung um einen **Zwei-Schritt-Lookahead**, der zukünftige Futterchancen bewertet
  – Aufteilung der Logik in kleinere Hilfsmethoden (z. B. `_simulate_future_environment`, `_food_moves_in_field`, `_score_two_step_food`, `_chase_food_one_step`, `_score_food_moves_with_lookahead`)
- Entwicklung und Anpassung von **Tests** aller Methoden die Chase_food treffen.

1.4.1 Beitrag 1: Erster Entwurf der Beast-Klasse (`parse_environment`, `locate_food`, `random_move`)

Betroffener Commit: `433aca96 - Biest Klasse angefangen (parse_environment()) und noch chase_food()`

1.4.1.1 Ausgangssituation / Problemstellung / Ziel

- Zu Beginn des Projekts gab es noch **keine eigenständige Beast-Klasse**, die das lokale Sichtfeld auswertet.
- Für alle späteren Strategien (Food, Hunt, Escape, Split, …) brauchten wir aber:
  - eine Möglichkeit, das **49-Zeichen-Environment** in eine 7×7-Struktur umzuwandeln,
  - eine Funktion, die **Futterpositionen** im Feld findet,
  - eine einfache **Fallback-Bewegung**, falls keine andere Strategie greift.
- Ziel dieses Commits war es deshalb,
  - die **Grundstruktur der Beast-Klasse** anzulegen und
  - erste Kernfunktionen zu skizzieren:
    - `parse_environment` (Umwandlung des Strings in ein 7×7-Feld),
    - `locate_food` (Futter im Feld finden),
    - `random_move` (zufälliger Ein-Schritt-Move als Notlösung),
    - Platzhalter für `chase_food`, `food_direction` und `rest`.

Idee & Design

- Ich wollte sehr früh eine **kleine, aber lauffähige Version** des Beasts haben, mit der man bereits experimentieren kann.
- `parse_environment` sollte:

  - das 49-Zeichen-Sichtfeld in **7 Zeilen à 7 Zeichen** zerlegen,
  - intern eine 7×7-Matrix liefern, in der der Mittelpunkt das Biest markiert.

- **`locate_food(grid)`** sollte zunächst ganz einfach alle `'*'` im Grid finden und als **absolute Koordinaten `(x, y)`** zurückgeben.
  - Später konnte man diese Funktion noch erweitern (z. B. auf relative Koordinaten `(dx, dy)` umbauen – das hat dann u. a. mein Teamkollege **Rohit Saini** weitergeführt).

- **`random_move`** sollte ein sehr einfacher, aber immer gültiger Fallback sein:
  - zufälliger Schritt nach **oben, unten, links oder rechts**,
  - damit das Beast sich überhaupt bewegen kann, auch wenn `chase_food` noch nicht implementiert ist.

- Zusätzlich habe ich schon **Platzhalter** für spätere Funktionen eingefügt (`chase_food`, `food_direction`, `rest`), damit klar ist, wie die Schnittstelle der Klasse ungefähr aussehen soll.

Implementierung

Ein Ausschnitt der wichtigsten Methoden aus diesem Commit (noch ohne Numpy, Rückgabe von `parse_environment` ist eine einfache Liste von Listen):

```python
import random


class Beast:
    """
    The Beast Class is responsible for managing the state and behavior of a single
    beast in the game.
    """

    def parse_environment(self, env: str):
        """
        Nimmt den 49-Zeichen-String und formt ihn in ein 7x7-Grid um.
        """
        # according to to the size of our game (2N+1)x(2N+1), we need a initializer N
        # and the actual size: size = 2*N+1
        N: int = 3
        size_game: int = 2 * N + 1

        result = [
            list(env[element : element + size_game])
            for element in range(0, len(env), size_game)
```

```
        ]
        # Tell us where the Beast is
        result[N][N] = "B"

        return result

    def chase_food(self):
        """
        Platzhalter für die eigentliche Futterjagd-Strategie.
        """
        return

    def locate_food(self, grid):
        food_emplacement: list = []

        for y, row in enumerate(grid):
            for x, col in enumerate(row):
                if "*" in col:
                    food_emplacement.append((x, y))

        return food_emplacement

    def random_move(self):
        d_x, d_y = random.choice([(1, 0), (-1, 0), (0, 1), (0, -1)])
        return d_x, d_y
```

Verständliche Erklärung

- Das Spielfeld kommt als **ein langer String** von 49 Zeichen an.
- Weil das Sichtfeld immer 7×7 groß ist, setze ich `N = 3` und `size_game = 2 * N + 1 = 7`.
- In der List-Comprehension:
  - starte ich bei `i = 0` und gehe in Schritten von 7 (`range(0, len(env), size_game)`),
  - nehme jeweils ein 7-Zeichen-Stück: `env[i : i + size_game]`,
  - mache daraus eine Liste von Zeichen `list(...)`,
  - und sammle diese in `result`.
- Das ergibt eine Liste mit genau 7 Zeilen:
  - `result[0]` ist die erste Zeile,
  - `result[6]` die letzte Zeile.

- In der Mitte liegt der Index `N = 3`, also setze ich `result[3][3] = "B"`.
- Danach wandle ich die Liste in ein `np.array` um, damit man später bequem mit Numpy darauf zugreifen kann.

Dann sieht das Feld als Matrix so aus:

```
Zeile 0: . . . . . . .
Zeile 1: . . * . . . .
Zeile 2: . . . B . . .
Zeile 3: . . . . . . .
Zeile 4: . . . . . . .
Zeile 5: . . * . . . .
Zeile 6: . . . . . . .
               ^
               |
             Biest (3,3)
```

Pseudocode

```
FUNCTION parse_environment(env_string):
    N = 3
    size = 2*N + 1   # = 7

    result = leere Liste

    FOR i von 0 bis Länge(env_string) in Schritten von size:
        zeilen_string = env_string[i : i + size]
        zeile = Liste der Zeichen in zeilen_string
        füge zeile zu result hinzu

    # Beast in die Mitte setzen (Sicherheit)
    result[3][3] = 'B'

    numpy_env = numpy.array(result)
    RETURN numpy_env
```

Zusammenarbeit im Team

Mit diesem Commit habe ich den Startpunkt für die Beast-Logik geschaffen.

Darauf aufbauend hat **Rohit Saini** später:

locate_food zu locate_food_list weiterentwickelt (Umstieg auf relative Koordinaten (dx, dy)),

und meine erste chase_food-Version verbessert.

Ich selbst habe nach diesem Commit:

die Lookahead-Erweiterung für chase_food implementiert,

sowie passende Tests geschrieben (diese werden in späteren Beiträgen beschrieben).

Einsatz von KI

Diesen frühen Prototyp der Beast-Klasse habe ich hauptsächlich ohne KI-Unterstützung entwickelt.

1.4.2 Beitrag 2: Erste lauffähige chase_food-Strategie mit get_food_direction

Betroffener Commit: 7771b866 – chase_food Methode erledigt fehlt nun den Test

1.5 Ausgangssituation / Problemstellung / Ziel

Nach meinem ersten Commit gab es zwar: - eine Beast-Klasse, - `parse_environment`, - `locate_food` (erstes Gerüst), - und `random_move`,

aber die eigentliche Futterjagd-Strategie `chase_food` war noch leer.

Ziel dieses Commits war: - eine erste lauffähige Version von `chase_food` zu programmieren, - in der das Beast: - das 7×7-Sichtfeld auswertet, - das nächste Futter findet, - und einen Ein-Schritt-Move in diese Richtung macht, - und falls kein Futter gefunden wird → auf `random_move` zurückfällt.

Idee & Design

Die Idee für diese erste `chase_food`-Version war bewusst einfach:

1. **Sichtfeld einlesen**
   Mit `parse_environment(self.environment)` das 49-Zeichen-Environment in ein 7×7-Feld umwandeln.
2. **Biest-Mitte bestimmen**
   Die Mitte des Feldes ist `(x, y) = (len(feld) // 2, len(feld) // 2)`.
3. **Futterpositionen suchen**
   Mit `locate_food(feld)` alle Koordinaten `(x_food, y_food)` mit '*' einsammeln.
4. **Nächstes Futter wählen**
   Mit einer neuen Hilfsfunktion `get_food_direction`:
   - die Futterliste nach der Manhattan-Distanz `|x - x_food| + |y - y_food|` sortieren,

- das nächste Futter auswählen,
- die Richtung `(dx, dy)` zu diesem Ziel berechnen,
- und `dx, dy` auf -1, 0, 1 beschränken, damit das Beast nur einen Schritt macht.

5. **Fallback ohne Futter**
   Wenn `locate_food` kein Futter findet → `random_move()` aufrufen und denselben Mechanismus benutzen.

Zusätzlich habe ich vorgesehen, dass `chase_food`: - `self.x` und `self.y` entsprechend dem Move aktualisiert, - und `(dx, dy)` als Rückgabewert liefert.

Implementierung

```

def chase_food(self):
    """
    Futterjagd: Das Biest bewegt sich in Richtung des nächsten Futters,
    oder zufällig, wenn kein Futter sichtbar ist.
    """
    feld = self.parse_environment(self.environment)

    x = y = len(feld) // 2  # Mitte des 7x7-Felds
    food_pos = self.locate_food(feld)

    if food_pos:
        d_x, d_y = self.get_food_direction(food_pos, x, y)
        print(f"Biest {self.id} bewegt sich Richtung: d_x={d_x} d_y={d_y}")
    else:
        d_x, d_y = self.random_move()
        print(
            f"Biest {self.id} bewegt sich zufällig Richtung: d_x={d_x} d_y={d_y}"
        )

    # interne Koordinaten aktualisieren
    self.x += d_x
    self.y += d_y

    return d_x, d_y

def get_food_direction(self, food_dir, x: int, y: int):
    if not food_dir:
        return None

    # Futterliste nach Manhattan-Distanz sortieren
```

```
32      food_dir.sort(key=lambda pos: abs(pos[0] - x) + abs(pos[1] - y))
33      target_x, target_y = food_dir[0]
34
35      dx = target_x - x
36      dy = target_y - y
37
38      # nur -1, 0, 1 erlauben (ein Schritt pro Zug)
39      dx = max(-1, min(1, dx))
40      dy = max(-1, min(1, dy))
41
42      return dx, dy
```

Hinweis: In dieser Commit-Version war locate_food kurzzeitig fehlerhaft (sie gab versehentlich einen zufälligen Move zurück). Das wurde in späteren Commits korrigiert.

1.6 Einsatz von KI

- In dieser Phase habe ich KI genutzt um:
  - über mögliche Distanzmaße (Manhattan vs. euklidisch) zu diskutieren,
  - zu überprüfen, ob das Clamping von dx/dy auf $-1..1$ sinnvoll ist.
- Die konkrete Implementierung von chase_food und get_food_direction (Schleifen, Sortierung nach Distanz, Update von self.x/self.y) habe ich selbst geschrieben.

1.6.1 Beitrag 3: Erste Split-Idee mit einfacher Energielogik

Betroffener Commit: 38796857 – beast groß geschrieben und split logic

Kurz Was ich hier gemacht habe:

In diesem Commit habe ich meine erste Idee für das Split-Verhalten eingebaut. Die Logik war noch sehr einfach:

- Wenn ein Beast genug Energie hat (Schwellwert `SPLIT_ENERGY = 40.0`),
- dann soll es sich aufteilen und ein neues Beast in eine der vier Hauptrichtungen spawnen,
- die Richtung wird zufällig aus `(-1, 0), (1, 0), (0, -1), (0, 1)` gewählt,
- danach wird der entsprechende SPLIT-Befehl an den Server geschickt und geloggt.

Außerdem habe ich an dieser Stelle auch dafür gesorgt, dass die Klasse im Code einheitlich als `Beast` (groß geschrieben) verwendet wird.

Implementierung:

```
import math
import random
import numpy as np
from .utils import print_and_flush, cmd, handle_shutdown
from .beast import Beast

# ...
priority_food = beast.get_priority_food()
hunt_list = beast.get_hunt_list()
priority_hunt = beast.get_priority_hunt()

# Split energy
SPLIT_ENERGY: float = 40.0

if curr_beast.get_energy() >= SPLIT_ENERGY:
    splits_dir = ((-1, 0), (1, 0), (0, -1), (0, 1))
    dx, dy = random.choice(splits_dir)

log_beast(bid, "SPLIT", energ=energy, env=environment, move=(dx, dy))
server_command = f"{bid} {cmd.SPLIT} {dx} {dy}"
return server_command
```

Idee dahinter (ganz knapp)

Ziel: Überhaupt erstmal ein funktionierendes Split-Verhalten haben.

Die Bedingung war nur: „Energie 40 → splitte in eine zufällige der vier Hauptrichtungen“.

Später im Projekt wurde diese Logik deutlich komplizierter: - mehr Bedingungen (z. B. keine Gegner in der Nähe, genug Futter, Rundenanzahl), - intelligenteres Wählen der Split-Richtung, - und am Ende ist die Split-Logik in die `Beast.split()`-Methode gewandert.

Einsatz von KI

Keine Verwendung von KI

1.6.2 Beitrag 3/2: Split-Methode begonnen und erste Tests für Beast

Betroffener Commit: 1d2d390e – split methode angafangen und Test

Ausgangssituation / Problemstellung / Ziel

Bisher gab es nur eine grob skizzierte Split-Idee (z. B. im Controller mit zufälliger Richtung bei genügend Energie). Die Logik war aber: - noch nicht in der Beast-Klasse gebündelt, - nicht sauber testbar und - nicht klar definiert, was genau bei „zu wenig Energie" passieren soll.

Außerdem fehlten automatisierte Tests für `split`, `chase_food` und `parse_environment`.

Ziel dieses Commits: - erste Version der `split`-Methode direkt in der Beast-Klasse, - einfaches Verhalten für „genug Energie" vs. „zu wenig Energie", - erste Test-Datei `test_beast.py`.

Idee & Design

- Die Idee für split in dieser frühen Version:
  - Wenn das Beast genug Energie hat (z. B. > 40):
  - wähle eine der vier Richtungen `(-1, 0), (1, 0), (0, -1), (0, 1)` zufällig,
  - baue einen Server-Command-String wie **"{id} SPLIT dx dy"**,
  - gib diesen String zurück.
- Wenn die Energie nicht ausreicht:
  - soll kein Split stattfinden,
  - die Methode gibt None zurück,
  - optional wird eine Meldung geloggt/gedruckt.
- Parallel dazu:
  - habe ich für rest eine erste Idee ergänzt:
    - bei sehr wenig Energie < 10 bleibt das Beast stehen (dx = dy = 0) und schickt einen MOVE-Befehl mit (0,0),
  - und ich habe in einer ersten Fassung von test_beast.py sowohl split als auch chase_food und parse_environment in einem gemeinsamen Testmodul getestet.

Implementierung:

```python
def rest(self):
    """
    If the beast has less than 10 energy, the beast should rest
    and not move.
    """
    if self._energy < 10:
        dx, dy = 0, 0
        rest_korrdinaten = f"{self._id} {cmd.MOVE} {dx} {dy}"
        return rest_korrdinaten

    return None  # kein spezieller Rest-Command nötig

def split(self):
    """
    Erste Version der Split-Logik:
    - Bei genügend Energie: zufällige Richtung wählen und SPLIT-Befehl bauen.
    - Sonst: None zurückgeben.
    """
    # Beispiel-Schwelle (entspricht der damaligen Idee)
    SPLIT_ENERGY: float = 40.0

    if self._energy >= SPLIT_ENERGY:
        splits_dir = ((-1, 0), (1, 0), (0, -1), (0, 1))
        dx, dy = random.choice(splits_dir)

        serv_cmd = f"{self._id} {cmd.SPLIT} {dx} {dy}"
        return serv_cmd
    else:
        print(
            f"Beast {self._id} has insufficient energy ({self._energy}) to split."
        )
        return None
```

1.7 Erstes Test Datei:

In diesem Commit habe ich außerdem eine erste Testdatei angelegt, in der ich mehrere Dinge gleichzeitig getestet habe:

1. **parse_environment**:

   Form des Feldes (7×7),

`B` in der Mitte,

Environment-String korrekt gesetzt,

ID und Energie korrekt gesetzt,

2. **Split:**

   deterministische Richtung durch Monkeypatchen von random.choice,

   Rückgabe-String im richtigen Format,

   None bei zu wenig Energie,

3. **chase_food:**

   gibt eine Liste von Moves zurück,

   jedes Element ist ein Tupel (dx, dy),

   Moves liegen im erlaubten Bereich -2 dx, dy 2,

   die Food-Liste im Beast ist nicht leer, wenn Futter im Environment ist,

4. **chase_food_no_food:**

   wenn kein Futter im Sichtfeld ist, wird ein zufälliger Move zurückgegeben.

Ein Ausschnitt der wichtigsten Tests (etwas bereinigt dargestellt):

```
import pytest
import random
from pymonster.utils import cmd
from pymonster.beast import Beast

@pytest.fixture
def beast_inst():
    beast = Beast()
    beast.set_id(1)
    beast.set_energy(50)
    beast.set_environment(".<.........*....>...**.....<.........=...*....*..")
    return beast

def test_parse_environment(beast_inst):
    feld = beast_inst.parse_environment(beast_inst.get_environment())

    assert feld.shape == (7, 7)

    center = feld[3][3]
```

```
20      assert center == "B"
21
22      assert "*" in beast_inst.get_environment()
23      assert beast_inst.get_id() == 1
24      assert beast_inst.get_energy() == 50
25
26  def test_split(beast_inst, monkeypatch):
27      # Fixiere random.choice, damit Test deterministisch ist
28      monkeypatch.setattr(random, "choice", lambda seq: (1, 0))
29
30      command = beast_inst.split()
31      assert command == f"1 {cmd.SPLIT} 1 0"
32      assert "SPLIT" in command
33      assert command.startswith("1 ")
34
35  def test_split_insufficient_energy(beast_inst):
36      beast_inst.set_energy(10.3)
37      command = beast_inst.split()
38      assert command is None
39
40  def test_chase_food(beast_inst, monkeypatch):
41      monkeypatch.setattr(random, "choice", lambda seq: seq[0])
42      moves = beast_inst.chase_food()
43
44      assert isinstance(moves, list)
45      assert all(isinstance(m, tuple) for m in moves)
46
47      dx, dy = moves[0]
48      assert -2 <= dx <= 2
49      assert -2 <= dy <= 2
50
51      # Es sollte Futter im Environment erkannt worden sein
52      assert len(beast_inst.get_food_list()) > 0
53
54  def test_chase_food_no_food(monkeypatch):
55      beast = Beast()
56      beast.set_environment(".......................B.....................")
57
58      monkeypatch.setattr(random, "choice", lambda seq: (1, 0))
59      moves = beast.chase_food()
60
61      assert moves == [(1, 0)]
```

62

Einsatz von KI

- KI habe ich in dieser Phase eher unterstützend eingesetzt, z. B.:
  - um mir bei der Struktur von pytest-Tests Inspiration zu holen,
  - oder um einzelne Fehlermeldungen (z. B. AssertionError) besser zu verstehen.
- Die konkrete Ausformulierung der Tests (test_split, test_split_insufficient_energy, test_chase_food, test_chase_food_no_food) und ihr Bezug auf unsere konkrete Beast-Implementierung stammen von mir.

1.7.1 Beitrag 4: `chase_food` – Futterjagd mit Zwei-Schritt-Lookahead

Betroffener Commit: e4e064ae – implement chasefood Erweiteurng

Unterteilung der Aufgaben in Teilaufgaben

1. das Futter im Sichtfeld finden (`locate_food_list`),
2. daraus eine **Liste sinnvoller Bewegungen `(dx, dy)`** berechnen,
3. eine **Priorisierung** vornehmen:
   - Futter in der Nähe ist attraktiver,
   - Moves sollen im erlaubten Bewegungsradius bleiben,
4. bei Bedarf eine **Zukunftsabschätzung** machen:
   - Für einen ersten Zug `m1` wird simuliert, wie das Sichtfeld nach diesem Zug aussieht.
   - In diesem neuen Feld wird wieder Futter gesucht.
   - Ein 2. Zug `m2` wird bewertet.
   - Ein Score bewertet Kombinationen wie `(m1, bester m2)`.

So entsteht eine intelligente Strategie, die **nicht nur den sofortigen Vorteil**, sondern auch die **nächsten Möglichkeiten** berücksichtigt.

Distanzmaß: Chebyshev-Distanz

Für die Einteilung in Ringe verwende ich die **Chebyshev-Distanz**.
In Worten: Die Distanz `d_C` von einem Move `(dx, dy)` ist das Maximum aus den beiden Beträgen:

```
d_C(dx, dy) = max(|dx|, |dy|)
```

Damit gilt zum Beispiel:

- `d_C = 0` → Mitte (Biest selbst),
- `d_C = 1` → direkte Nachbarschaft (8 Felder um `B`),
- `d_C  2` → 5×5-Bereich,
- `d_C  3` → gesamtes 7×7-Sichtfeld.

Strategische Einteilung in Ringe

Aus den Roh-Futterpositionen `(dx, dy)` baue ich drei Mengen:

- **Nachbarn**: `max(|dx|, |dy|) == 1`
- **Ring 5×5**: `max(|dx|, |dy|)  2`
- **Ring 7×7**: ansonsten (bis zur Sichtfeldgrenze)

Textuelle Beschreibung des Ablaufs

```
Start -> locate_food_list()

Falls kein Futter gefunden:
    -> random_move() -> Return

Falls Futter gefunden:
    Prüfe Energie-Modus:
        - Wenn nur 1er-Moves erlaubt:
            -> _chase_food_one_step() -> Return
        - Sonst:
            -> Futter nach Ringen aufteilen (Neighbours, Ring5, Ring7)
               - Wenn Neighbours vorhanden:
                   Candidates = Neighbours
               - Sonst wenn Ring5 vorhanden:
                   Candidates = geclampte Moves aus Ring5
               - Sonst wenn Ring7 vorhanden:
                   Candidates = geclampte Moves aus Ring7
               - Sonst:
                   -> random_move() -> Return

            -> beste K Kandidaten auswählen
            -> _score_food_moves_with_lookahead()
            -> sortierte Moves zurückgeben
```

Codeausschnitt (dokumentierte Struktur von `chase_food`)

```python
def chase_food(self) -> list[tuple[int, int]]:
    """
    Futter-Suche mit Zwei-Schritt-Lookahead.

    Schritte:
        1) Alle Futter-Positionen relativ zum Beast holen.
        2) Bewegungen nach Distanz (Ringen) einteilen.
        3) Abhängig vom Energie-Modus entweder:
           - nur 1er-Moves betrachten, oder
           - 1er- und 2er-Moves mit Lookahead bewerten.
        4) Ergebnis ist eine sortierte Liste von (dx, dy)-Moves.

    Returns:
        list[tuple[int, int]]: sortierte Liste von Zugvektoren.
    """
    # 0) Rohdaten (relative Positionen)
    raw_food = self.locate_food_list()

    # 0a) Kein Food im Sichtfeld -> Random-Fallback
    if not raw_food:
        rx, ry = self.random_move()
        moves = [(rx, ry)]
        self.set_food_list(moves)
        return moves

    # 1) Je nach Energie-Level: 1er-Move-Strategie oder Normalmodus
    if self._priority_energy < 2.0:
        moves = self._chase_food_one_step(raw_food)
        self.set_food_list(moves)
        return moves

    # 2) Normalmodus: Food nach Ringen aufteilen
    neighbours: list[tuple[int, int]] = []
    ring5: list[tuple[int, int]] = []
    ring7: list[tuple[int, int]] = []

    for dx, dy in raw_food:
        cheby = max(abs(dx), abs(dy))
        if cheby == 1:
            neighbours.append((dx, dy))
        elif cheby <= 2:
            ring5.append((dx, dy))
```

```
43          else:
44              ring7.append((dx, dy))
45
46      def unique_sorted(moves: list[tuple[int, int]]) -> list[tuple[int, int]]:
47          """Hilfsfunktion: sortiert Moves und entfernt Duplikate."""
48          moves_sorted = sorted(
49              moves,
50              key=lambda m: (abs(m[0]) + abs(m[1]), m[0], m[1]),
51          )
52          uniq: list[tuple[int, int]] = []
53          for m in moves_sorted:
54              if not uniq or uniq[-1] != m:
55                  uniq.append(m)
56          return uniq
57
58      candidate_moves: list[tuple[int, int]] = []
59      is_direct_hit_path = False
60
61      # 3) Kandidaten je nach Ring bestimmen
62      if neighbours:
63          candidate_moves = unique_sorted(neighbours)
64          is_direct_hit_path = True
65      elif ring5:
66          clamped = []
67          seen = set()
68          for dx, dy in ring5:
69              mv = self._clamp_move(dx, dy, max_step=1)
70              if mv != (0, 0) and mv not in seen:
71                  seen.add(mv)
72                  clamped.append(mv)
73          candidate_moves = unique_sorted(clamped)
74      elif ring7:
75          clamped = []
76          seen = set()
77          for dx, dy in ring7:
78              mv = self._clamp_move(dx, dy, max_step=1)
79              if mv != (0, 0) and mv not in seen:
80                  seen.add(mv)
81                  clamped.append(mv)
82          candidate_moves = unique_sorted(clamped)
83
84      if not candidate_moves:
```

```
85          rx, ry = self.random_move()
86          moves = [(rx, ry)]
87          self.set_food_list(moves)
88          return moves
89
90      # 4) Lookahead nur für die besten K Kandidaten
91      best_sorted_food_list = candidate_moves[:7]  # oder [:6], je nach Definition
92
93      scored_moves = self._score_food_moves_with_lookahead(
94          best_sorted_food_list,
95          is_direct_hit_path=is_direct_hit_path,
96      )
97
98      self.set_food_list(scored_moves)
99      return scored_moves
```

2 Systematische Vorgehensweise(Allegemein)

Meine Arbeitsweise entsprach im Wesentlichen folgenden Schritten:

1. **Problem in eigenen Worten verstehen**
   - z. B. „Wie komme ich von einem flachen String zu einem 7×7-Feld¿‘
   - „Wie kann das Beast Futter nicht nur sehen, sondern auch sinnvoll bewerten¿‘
2. **Zerlegen in kleinere Teilaufgaben**
   - `parse_environment`: reines Umformen von Daten
   - `locate_food_list`: aus Feld → Liste von `(dx, dy)`
   - `chase_food`: Entscheidung, welche Moves gut sind
   - Lookahead-Funktionen: Simulation und Scoring
3. **Schrittweise Implementierung**
   - Zuerst einfache Versionen (`chase_food` ohne Lookahead),
   - später Erweiterung mit Zwei-Schritt-Lookahead und Scoring.
4. **Frühes Testen mit pytest**

- Nach jeder größeren Änderung habe ich neue Tests ergänzt oder bestehende Tests angepasst.

- Ziel war, Fehler möglichst früh zu finden und Verhalten zu stabilisieren.

2.1 Beitrag 5: Test-Suite für die Beast-Klasse strukturiert und erweitert

Betroffener Commit: `c65adfca - Tests beast.py laufen einwandfrei`

Ausgangssituation / Problemstellung / Ziel

- In früheren Commits hatte ich bereits **erste Tests** für `Beast` angelegt, aber:
  - vieles war noch in **einer einzigen Datei** vermischt,
  - die Testfälle waren **schwer zu überblicken**,
  - und es war nicht klar, **welcher Test genau welche Methode** abdeckt.
- Gleichzeitig war die Beast-Klasse inzwischen deutlich größer geworden:
  - Energie-Hilfsfunktionen (`_move_energy`, `_is_move_within_energy_limit`),
  - Klammer-Logik für Moves (`_clamp_move`),
  - Futter-Logik mit Lookahead (`chase_food`, `_simulate_future_environment`, `_score_two_step_food`, `_score_food_moves_with_lookahead`, `_food_moves_in_field`),
  - Jagd- und Kill-Logik (`hunt`, `locate_hunting_list`, `compute_kill_list`, `locate_enemy_list`),
  - Flucht-Logik und Gegner-Helfer (`get_enemy_positions`, `locate_unique_enemy_moves`, `escape`, `score_safe_moves`),
  - Split-Logik (`split` mit Normal- und Notfall-Split).
- Ziel dieses Commits war es deshalb:
  - die Tests **systematisch nach Themen zu trennen**,
  - für jede wichtige Methode der Beast-Klasse **gezielte Unit-Tests** zu schreiben,
  - und dafür zu sorgen, dass **alle Tests grün laufen** („Tests beast.py laufen einwandfrei“).

2.1.1 Einsatz von KI

- Für diese Test-Sammlung habe ich KI vor allem benutzt, um:
  - Ideen für sinnvolle Randfälle zu bekommen (z. B. „was passiert bei sehr viel Futter¿‘, „wie testet man Lookahead am besten¿‘),
  - einzelne Assertion-Ideen zu verfeinern (z. B. Nutzung von math.isclose bei Distanzberechnungen, Vergleich von Scores).

3 Zusammenarbeit im Team

Auch wenn diese Dokumentation meinen persönlichen Beitrag hervorhebt, war das Semesterprojekt insgesamt eine **Teamleistung**.
Wir haben uns sowohl **vor Ort** als auch **online** zu mehreren Teammeetings getroffen, in denen wir u. a. folgende Themen besprochen haben:

- Aufteilung der Aufgaben im Team
- Strukturierung des Codes und der Module
- Planung der nächsten Schritte und Meilensteine
- Analyse von Testläufen und Logs
- Diskussion von Fehlern und möglichen Bugfixes
- Bewertung und Anpassung der Strategie

Wichtige Ideen aus dem Team

- Die Idee, den ursprünglichen `SampleClient` in **drei getrennte Module** aufzuteilen – `client.py`, `controller.py` und `logic.py` – stammt von **Rohit Saini**.
  Dadurch wurde unsere Architektur deutlich übersichtlicher und besser testbar, weil Kommunikation, Steuerlogik und Strategie voneinander getrennt wurden.
- Die grundlegende **Strategie-Idee** (Food, Hunt, Kill, Escape, Split mit Prioritäten) stammt von **Gabriel Corazza**.
  Diese wurde anschließend in mehreren Teammeetings gemeinsam **verfeinert und erweitert**, z. B. durch Diskussion von Prioritäten, Sonderfällen und möglichen Erweiterungsstufen.

Gemeinsame Bugfixes und Verbesserungen

Neben den klar trennbaren Einzelaufgaben gab es Commits, die wir **bewusst als Team** erarbeitet haben:

- Commit `221330f5` – *Bugfixes*:
  In diesem Schritt haben wir im Team dafür gesorgt, dass der Code im ersten größeren Testrun überhaupt stabil läuft.
  Dazu gehörten u. a. kleinere Korrekturen an der Logik, Anpassungen an der Kommunikation mit dem Server und das Beheben offensichtlicher Laufzeitfehler.
- Commits `8d9c28a5`, `7d986422` und `3c8238b4` – *Bugfixes & Feintuning*:
  Hier haben wir gemeinsam:

- die **Logs** aus Testläufen analysiert,
- die **Prioritäten** der einzelnen Module (Food, Hunt, Escape, Split, etc.) angepasst,
- dafür gesorgt, dass **(0, 0)-Moves** (also „stehen bleiben“) in der Regel **nicht mehr vorkommen**,
- und weitere Feinjustierungen an der Strategie vorgenommen, damit sich die Biester sinnvoller bewegen.

Diese Commits spiegeln gut wider, dass viele Verbesserungen nicht von einer einzelnen Person stammen, sondern aus **Diskussionen im Team**, gemeinsamer Fehlersuche und iterativen Anpassungen an unserer Strategie.

4 Hinweise

Details zu den Anforderungen an die persönliche Dokumentation
finden sich in der Aufgabenstellung zum Semesterprojekt, Kapitel 6.2 (Verständliche Darstellung der Lösungsansätze, Qualität der Implementierung,
Einsatz von KI-Werkzeugen, Reflexion des eigenen Beitrags).

5 Quellenverzeichnis