Bausteine für größere Programme

Bisher behandelt: OOP-Bausteine

– Klassen und Objekte
– Methoden, Attribute und Properties
– Dunder Methods

Nun folgt: OOP zur Strukturierung größerer Programme

– Vererbung: Funktionalität wiederverwenden
– Duck Typing: Fokus auf Verhalten, um Codemenge zu begrenzen
– Inversion of Control: Code-Organisation für modularere Systeme

Klassen

– bündeln Daten (Attribute) und Verhalten (Methoden)

– Objekte sind Instanzen von Klassen

```
class BankAccount:
    def __init__(self, owner, balance=0):
        self.owner = owner
        self.balance = balance


    def deposit(self, amount):
        self.balance += amount
```

Unerwünschte Code-Duplikation

```python
class Circle:
    def __init__(self, radius):
        self.radius = radius

    def area(self):
        return 3.14159 * self.radius ** 2

    def perimeter(self):
        return 2 * 3.14159 * self.radius


class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)
```

```python
class Shape:
    def area(self):
        pass  # Platzhalter - wird von konkreten Formen implementiert

    def perimeter(self):
        pass  # Platzhalter - wird von konkreten Formen implementiert


class Circle(Shape):  # Circle erbt von Shape
    def __init__(self, radius):
        self.radius = radius


    def area(self):
        return 3.14159 * self.radius ** 2


    def perimeter(self):
        return 2 * 3.14159 * self.radius
```

Vererbung

Mechanismus zur Organisation
von geteilter Funktionalität

Vererbung: Terminologie

– *Elternklasse (Basisklasse)*: definiert gemeinsames Verhalten

– *Kindklasse (Unterklasse, Subklasse)*: erbt und kann Verhalten überschreiben

– *Methodenüberschreibung:* Ersetzen der Implementierung der Elternklasse

– Vererbung sinnvoll bei **„ist-ein“-Beziehung**:
Kindklasse spezifiziert eine (besondere Form) der Elternklase

```
class Shape:  # normale Klasse, erbt von keiner anderen
    def area(self):
        pass

class Circle(Shape):  # Circle ist eine Subklasse von Shape
    def __init__(self, radius):
        self.radius = radius

    def area(self):  # Methode wird überschrieben
        return 3.14159 * self.radius ** 2
```

Beispiel

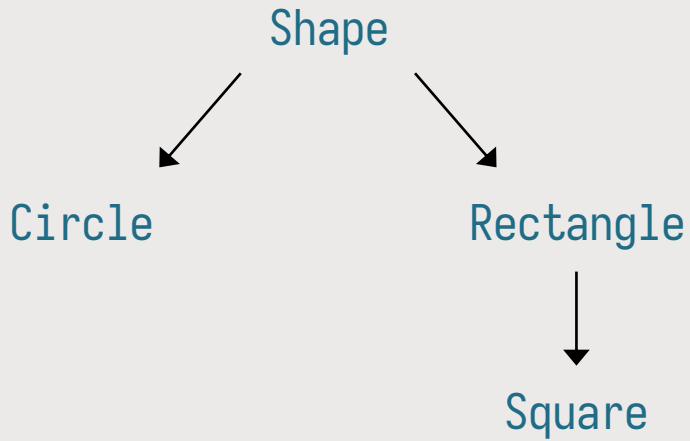
```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)


# Wir können alle Subklassen gleich behandeln:

shapes = [Circle(5), Rectangle(3, 4)]
for shape in shapes:
    print(f"Fläche: {shape.area()}")
    print(f"Umfang: {shape.perimeter()}")
```

Klassenhierarchien

– Klassen können Vererbungsbäume bilden

– Jede Ebene stellt spezialisierteres Verhalten dar.

– Ein `Square` *ist ein* (besonderes) `Rectangle`.
  Ein `Rectangle` *ist eine* (besondere) `Shape`



```python
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def area(self):
        return self.width * self.height

    def perimeter(self):
        return 2 * (self.width + self.height)


class Square(Rectangle):
    def __init__(self, side_length):
        # Direktes Setzen der Rectangle-Attribute
        self.width = side_length
        self.height = side_length
```

Erster Versuch zur Definition von Square

```
class Rectangle(Shape):
    def __init__(self, width, height):
        self.width = width
        self.height = height

    # ...

# So lieber nicht (mangelnde Kapselung):
class Square(Rectangle):
    def __init__(self, side_length):
        self.width = side_length
        self.height = side_length


# Besser:
class Square(Rectangle):
    def __init__(self, side_length):
        super().__init__(side_length, side_length)
```

Bessere Variante

Rectangle kümmert sich durch Aufruf von *super()* selbst um seine Attribute

Beispiel für die Nutzung überschriebener Methoden

```python
# Formen erstellen
square = Square(5)
rectangle = Rectangle(3, 4)
circle = Circle(2)


# Square nutzt automatisch Rectangle-Methoden
print(f"Quadratfläche: {square.area()}")          # 25
print(f"Quadratumfang: {square.perimeter()}")     # 20


# Alle Formen nutzen ihre eigenen Berechnungen
shapes = [square, rectangle, circle]
for shape in shapes:
    print(f"{type(shape).__name__}:")
    print(f"  Fläche: {shape.area()}")
    print(f"  Umfang: {shape.perimeter()}\n")
```

Beispiel für super()

```python
class Student:
    def __init__(self, name, student_id):
        self.name = name
        self.student_id = student_id

    def get_info(self):
        return f"Student {self.name} (ID: {self.student_id})"


class GraduateStudent(Student):
    def __init__(self, name, student_id, research_area):
        # Erst die Basis-Studierenden-Initialisierung
        super().__init__(name, student_id)
        # Dann graduate-spezifische Infos
        self.research_area = research_area

    def get_info(self):
        # Auf Basis-Info aufbauen
        basic_info = super().get_info()
        return f"{basic_info} - Forschungsgebiet: {self.research_area}"
```

Drei gängige Vererbungsmuster

– Erweiterung: neue Fähigkeiten hinzufügen
– Einschränkung: Verhalten begrenzen
– Spezialisierung: bestehendes Verhalten verfeinern

Muster 1
Erweiterung

```
class EnhancedList(list):
    def average(self):
        if not self:
            raise ValueError("Liste ist leer - Durchschnitt nicht berechenbar")
        return sum(self) / len(self)

    def clear_zeros(self):
        while 0 in self:
            self.remove(0)  # sucht und entfernt (nur) erstes Element, das 0 ist


numbers = EnhancedList([1, 0, 2, 0, 3, 4])
print(numbers.average())  # normale Listen-Methoden funktionieren weiterhin!
numbers.sort()            # geerbt von list
numbers.clear_zeros()     # unsere neue Methode
print(numbers)  # [1, 2, 3, 4]
```

Muster 2
Einschränkung

```
class PositiveList(list):
    def append(self, item):
        if item <= 0:
            raise ValueError("Nur positive Zahlen erlaubt")
        super().append(item)

    def extend(self, items):
        if any(x <= 0 for x in items):  # sog. List Comprehension (kommt später)
            raise ValueError("Nur positive Zahlen erlaubt")
        super().extend(items)


pos_nums = PositiveList([1, 2, 3])
pos_nums.append(4)  # funktioniert
try:
    pos_nums.append(-1)  # ValueError (Exceptions kommen später)
except ValueError as e:
    print(e)
```

Muster 3
Spezialisierung

```
class SortedList(list):
    def append(self, item):
        super().append(item)
        self.sort()

    def extend(self, items):
        super().extend(items)
        self.sort()


sorted_nums = SortedList([3, 1, 4])
print(sorted_nums)  # [1, 3, 4]
sorted_nums.append(2)
print(sorted_nums)  # [1, 2, 3, 4]
```

System zur Kursverwaltung
zeigt wie die Konstrukte verwendet werden

1/4

```python
class Course:
    def __init__(self, code, title, capacity):
        self._code = code
        self._title = title
        self._capacity = capacity
        self._enrolled = set()  # eingeschriebene Studierende

    def enroll(self, student):
        if len(self._enrolled) < self._capacity:
            self._enrolled.add(student)
            return True
        return False

    def get_enrollment_percentage(self):
        return (len(self._enrolled) / self._capacity) * 100

    def get_code(self):
        return self._code
```

```
class LabCourse(Course):
    def __init__(self, code, title):
        # Labs sind klein
        super().__init__(code, title, capacity=15)

class LectureCourse(Course):
    def __init__(self, code, title):
        # Vorlesungen sind groß
        super().__init__(code, title, capacity=100)

class Registry:
    def __init__(self):
        self._courses = []

    def add_course(self, course):
        self._courses.append(course)

    def get_student_courses(self, student):
        student_courses = []
        for course in self._courses:
            if student in course._enrolled:
                student_courses.append(course)
        return student_courses
```

System zur Kursverwaltung
zeigt wie die Konstrukte verwendet werden

2/4

```
class Student:
    def __init__(self, name, student_id, registry):
        self._name = name
        self._id = student_id
        self._registry = registry

    def get_my_courses(self):
        return self._registry.get_student_courses(self)

    def get_name(self):
        return self._name


registry = Registry()

python_lab = LabCourse("CS101L", "Python Programming Lab")
java_lecture = LectureCourse("CS102", "Java Programming")

registry.add_course(python_lab)
registry.add_course(java_lecture)

alice = Student("Alice", "001", registry)
bob = Student("Bob", "002", registry)
```

System zur Kursverwaltung
zeigt wie die Konstrukte verwendet werden

3/4

```python
# Anwesenheitsbericht ausgeben
for course in [python_lab, java_lecture]:
    print(f"Course: {course.get_code()}")
    print("Attendance: "
          f"{course.get_enrollment_percentage()}%")

# Alice's Kurse ausgeben
print(f"\n{alice.get_name()}'s courses:")
for course in alice.get_my_courses():
    print(course.get_code())
```

```
Beispielausgabe:
Course: CS101L
Attendance: 13.33%
Course: CS102
Attendance: 1%

Alice's courses:
CS101L
CS102
```

System zur Kursverwaltung
zeigt wie die Konstrukte
verwendet werden

4/4

Eine überraschende Beobachtung

```python
# All diese Varianten funktionieren in einer for-Schleife:

for letter in "hello":        # String
    print(letter)

for number in [1, 2, 3]:      # Liste
    print(number)

for elem in {4, 5, 6}:        # Set
    print(elem)

for i in range(3):            # range
    print(i)


# Gemeinsames Verhalten!
# Also Subklassen von Iterable o.ä.?
```

```python
print(isinstance("hello", str))        # True
print(isinstance([1,2,3], list))       # True
print(type("hello").__bases__)         # (object,)
print(type([1,2,3]).__bases__)         # (object,)


# Diese Typen haben keine gemeinsame Elternklasse!
# Und trotzdem funktionieren sie in for-Schleifen...
```

Vererbung ist nicht immer sinnvoll.

Manchmal sollen sich Objekte Verhalten teilen, es liegt aber keine „ist-ein“-Beziehung vor.

Es wäre irreführend, solche Objekte in eine Vererbungshierarchie zu zwingen.

Vererbung ist nicht immer sinnvoll.

Manchmal sollen sich Objekte Verhalten teilen, es liegt aber keine „ist-ein“-Beziehung vor.

Es wäre irreführend, solche Objekte in eine Vererbungshierarchie zu zwingen.

Lösung: Duck Typing

Wenn es wie eine Ente watschelt und wie eine Ente quakt, ist es eine Ente.

Der Fokus liegt darauf, dass bestimmte Methoden implementiert werden.

Beispiel für Duck Typing

```python
class Dog:
    def make_sound(self):
        return "Wuff!"

class Duck:
    def make_sound(self):
        return "Quack!"

class Cat:
    def make_sound(self):
        return "Miau!"

# Python interessiert sich nicht für Typen, nur für Verhalten
def animal_chorus(animals):
    for animal in animals:
        print(animal.make_sound())

# Verschiedene Tiere erstellen
pets = [Dog(), Cat(), Duck()]
animal_chorus(pets)  # funktioniert problemlos!
```

Vererbung und Duck Typing im Vergleich

```python
# Vererbungsansatz:
class Animal:
    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        return "Wuff!"



# Duck-Typing-Ansatz:
class Dog:  # Keine Vererbung nötig!
    def make_sound(self):
        return "Wuff!"
```

Eigene iterierbare Objekte erstellen

```python
class CountByTwo:
    def __init__(self, limit):
        self.limit = limit
        self.current = 0

    def __iter__(self): # wird benötigt um iterierbar zu sein
        return self

    def __next__(self): # wird benötigt um iterierbar zu sein
        if self.current >= self.limit:
            raise StopIteration
        self.current += 2
        return self.current

# Funktioniert wie eingebaute iterierbare Objekte!
counter = CountByTwo(10)
for num in counter:
    print(num)  # gibt aus: 2, 4, 6, 8, 10
```

Iterieren ist ein sogenanntes *Protokoll*

regelt, was erwartet wird und
was wie aufgerufen wird

```
# wie funktioniert das?
counter = CountByTwo(10)
for num in counter:
    print(num)  # gibt aus: 2, 4, 6, 8, 10
```

```
# Pythons for-Schleife macht im Wesentlichen das:
iterator = counter.__iter__()
while True:
    try:
        value = iterator.__next__()
        print(value)
    except StopIteration:
        break
```

```python
class Character:
    def __init__(self, x, y, health=100):
        self.x = x
        self.y = y
        self.health = health

    def take_damage(self, amount):
        self.health -= amount
        if self.health <= 0:
            print("Character besiegt!")

class Building:
    def __init__(self, x, y, health=1000):
        self.x = x
        self.y = y
        self.health = health

    def take_damage(self, amount):
        self.health -= amount * 2  # Gebäude nehmen doppelten Schaden
        if self.health <= 0:
            print("Gebäude eingestürzt!")

class DecorationProp:  # Manche Objekte können keinen Schaden nehmen
    def __init__(self, x, y):
        self.x = x
        self.y = y
```

Spielwelt-Beispiel
mit Duck Typing

Jedes Objekt weiß, wie es Schaden nimmt – falls es beschädigt werden kann

Spielwelt-Beispiel
mit Duck Typing

Mit hasattr *prüfen ob ein Objekt das Protokoll unterstützt*

```python
def calculate_distance(x1, y1, x2, y2):
    # Luftlinie auszurechnen mitSatz von Pythagoras
    # (Potenzieren mit 0.5 entspricht der Quadratwurzel)
    return ((x1 - x2)**2 + (y1 - y2)**2) ** 0.5


def explosion(center_x, center_y, radius, game_objects):
    for obj in game_objects:
        # Abstand zur Explosion berechnen
        distance = calculate_distance(center_x, center_y, obj.x, obj.y)

        if distance <= radius:
            # Prüfen ob Objekt Schaden nehmen kann
            if hasattr(obj, 'take_damage'):
                # Mehr Schaden je näher am Zentrum
                damage = 100 * (1 - distance/radius)
                obj.take_damage(damage)
```

Spielwelt-Beispiel
mit Duck Typing

Beispielhafte Verwendung

```
# Spielobjekte erstellen
player = Character(0, 0)
house = Building(10, 10)
tree = DecorationProp(5, 5)

# In Spielwelt einfügen
game_objects = [player, house, tree]

# Eine Explosion bei Position (5, 5) erzeugen
explosion(5, 5, 20, game_objects)
```

```
Mögliche Ausgabe:
Character nimmt 75% Schaden
Gebäude stürzt ein!
Der Baum steht unbeeindruckt da.
```

Spielwelt-Beispiel
mit Duck Typing

Beispielhafte Verwendung

```
# Spielobjekte erstellen
player = Character(0, 0)
house = Building(10, 10)
tree = DecorationProp(5, 5)

# In Spielwelt einfügen
game_objects = [player, house, tree]

# Eine Explosion bei Position (5, 5) erzeugen
explosion(5, 5, 20, game_objects)
```

```
Mögliche Ausgabe:
Character nimmt 75% Schaden
Gebäude stürzt ein!
Der Baum steht unbeeindruckt da.
```

Erster Versuch einer Anpassung

```python
class Character:
    # ...
    def take_damage(self, amount):
        self.health -= amount
        print(f"Character nimmt {amount:.0f}% Schaden")
        if self.health <= 0:
            print("Character besiegt!")

class Building:
    # ...
    def take_damage(self, amount):
        self.health -= amount * 2
        if self.health <= 0:
            print("Gebäude stürzt ein!")

player = Character(0, 0)
house = Building(10, 10)
tree = DecorationProp(5, 5)
game_objects = [player, house, tree]

explosion(5, 5, 20, game_objects)
```

```
Mögliche Ausgabe:
Character nimmt 75% Schaden
Gebäude stürzt ein!
Der Baum steht unbeeindruckt da.
```

```python
class Character:
    # ...
    def take_damage(self, amount):
        self.health -= amount
        print(f"Character nimmt {amount:.0f}% Schaden")
        if self.health <= 0:
            print("Character besiegt!")

class Building:
    # ...
    def take_damage(self, amount):
        self.health -= amount * 2
        if self.health <= 0:
            print("Gebäude stürzt ein!")

player = Character(0, 0)
house = Building(10, 10)
tree = DecorationProp(5, 5)
game_objects = [player, house, tree]

explosion(5, 5, 20, game_objects)
```

Erster Versuch einer Anpassung

Mögliche Ausgabe:

```
Character nimmt 75% Schaden
Gebäude stürzt ein!
```

Der Baum steht unbeeindruckt da.

Designproblem: Statusmeldungen bei Explosionen 🤔

Option 1: *take_damage in DecorationProp implementieren*

😀 Ausgabe bei allen Objekten an gleicher Stelle; jedes Objekt hat Kontrolle über die Ausgabe

😕 Irreführend: suggeriert dass *DecorationProp* schaden nehmen kann

Option 2: alle *Statusmeldungen in explosion()-Funktion ausgeben*

😀 Zentraler Ort für alle Meldungen

😕 *explosion()* muss alle Objekttypen kennen (nicht vergessen, bei neuen Typen anzupassen!)

Option 3: *Separate Methode output_explosion_status() in Objekten implementieren*

😀 Klare Trennung von Schaden-Logik und Statusmeldungen

😕 Logik dann aber auf zwei Methoden verteilt

Elegantere Lösung möglich: **Inversion of Control**

Bevor wir zu Inversion of Control (IoC) kommen…

… sehen wir uns an, wie eine Lösung ohne IoC aussehen würde.

Ohne IoC


```python
class GameObject:
    def notify_explosion_effect(self):
        pass  # Standardverhalten: keine Ausgabe


class Character(GameObject):
    def __init__(self, x, y, health=100):
        self.x = x
        self.y = y
        self.health = health

    def take_damage(self, amount):
        self.health -= amount
        if self.health <= 0:
            print("Character besiegt!")

    def notify_explosion_effect(self):
        print(f"Character hat noch {self.health:.0f}% Gesundheit")
```

Ohne IoC


```
class Building(GameObject):
    def __init__(self, x, y, health=1000):
        self.x = x
        self.y = y
        self.health = health

    def take_damage(self, amount):
        self.health -= amount * 2
        if self.health <= 0:
            print("Gebäude stürzt ein!")

    def notify_explosion_effect(self):
        if self.health <= 0:
            print("Gebäude liegt in Trümmern")
        else:
            print("Gebäude hat Schaden genommen")
```

```
class DecorationProp(GameObject):
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def notify_explosion_effect(self):
        print("Dekorationsobjekt steht unbeeindruckt da")
```

Ohne IoC

4/4

```
def explosion(center_x, center_y, radius, game_objects):
    for obj in game_objects:
        distance = calculate_distance(center_x, center_y, obj.x, obj.y)

        if distance <= radius:
            # Schadensberechnung und -anwendung
            if hasattr(obj, 'take_damage'):
                damage = 100 * (1 - distance/radius)
                obj.take_damage(damage)
            # Statusmeldung
            obj.notify_explosion_effect()
```

```python
class ExplosionEffect:
    def notify_damage(self, obj, damage):
        pass

    def notify_immune(self, obj):
        pass

class ConsoleExplosionEffect(ExplosionEffect):
    def notify_damage(self, obj, damage):
        if isinstance(obj, Character):
            print(f"Character nimmt {damage:.0f}% Schaden")
        elif isinstance(obj, Building):
            print("Gebäude hat Schaden genommen")

    def notify_immune(self, obj):
        print(f"{type(obj).__name__} steht unbeeindruckt da")


def explosion(center_x, center_y, radius, game_objects, effect_handler):
    for obj in game_objects:
        distance = calculate_distance(center_x, center_y, obj.x, obj.y)

        if distance <= radius:
            if hasattr(obj, 'take_damage'):
                damage = 100 * (1 - distance/radius)
                obj.take_damage(damage)
                effect_handler.notify_damage(obj, damage)
            else:
                effect_handler.notify_immune(obj)
```

Inversion of Control

So lagern wir die Ausgabelogik aus

Besser lesbarer Code

Methoden machen (nur) das, was ihr Name suggeriert

Inversion of Control

erleichert spätere Erweiterungen

```
class ChristmasExplosionEffect(ExplosionEffect):
    def notify_damage(self, obj, damage):
        print("Ho ho ho! Festliche Explosion!")
        if isinstance(obj, Character):
            print("🎄 Character stärkt sich mit einem Zimtstern!")
        elif isinstance(obj, Building):
            print("🎅 Es funkeln die Lichterketten ganz weihnachtlich!")

    def notify_immune(self, obj):
        print("❄ Schneegestöber umgibt", type(obj).__name__)
```

```python
class CompositeExplosionEffect(ExplosionEffect):
    def __init__(self, *handlers):
        self.handlers = handlers

    def notify_damage(self, obj, damage):
        for handler in self.handlers:
            handler.notify_damage(obj, damage)

    def notify_immune(self, obj):
        for handler in self.handlers:
            handler.notify_immune(obj)


# Verwendung
effects = CompositeExplosionEffect(
    ConsoleExplosionEffect(),
    ChristmasExplosionEffect()
)
explosion(5, 5, 20, game_objects, effects)
```

Inversion of Control

ermöglicht Flexibilität ohne Änderung der Klassen

```python
# Schlechtes Design: vermischte Zuständigkeiten
class Character:
    def take_damage(self, amount):
        self._health -= amount
        print(f"Schaden: {amount}")  # Vermischt Logik mit Ausgabe
        self._update_graphics()      # Vermischt Logik mit Darstellung


# Gutes Design: getrennte Zuständigkeiten
class Character:
    def take_damage(self, amount):
        self._health -= amount       # Nur Kernlogik

class DamageNotifier:
    def notify_damage(self, character, amount):
        print(f"Schaden: {amount}")  # Nur Benachrichtigung

class GraphicsUpdater:
    def update(self, character):
        # nur Darstellung aktualisieren
        # ...
```

Inversion of Control

hilft bei Umsetzung von
Separation of Concerns

Bereits bekanntes Ziel von OOP: Kapselung (Encapsulation)

– Innere Details werden verborgen
– Zugriff nur über definierte Schnittstelle
– Beispiel: *_health* ist privates Attribut

Weiteres Ziel: Separation of Concerns

– Jede Klasse hat eine klare, einzelne Verantwortung
– Verschiedene Aspekte werden getrennt behandelt
– Beispiel: Spielobjekte kümmern sich um Kernverhalten, *ExplosionEffect* um Benachrichtigungen

Bereits bekanntes Ziel von OOP: Kapselung (Encapsulation)

– Innere Details werden verborgen

– Zugriff nur über definierte Schnittstelle

– Beispiel: *_health* ist privates Attribut

🤔 *Sollten wir auch* `take_damage()` *von den Spielobjekten separieren?*

Weiteres Ziel: Separation of Concerns

– Jede Klasse hat eine klare, einzelne Verantwortung

– Verschiedene Aspekte werden getrennt behandelt

– Beispiel: Spielobjekte kümmern sich um Kernverhalten, *ExplosionEffect* um Benachrichtigungen

```python
# Option 1: Methoden in den Klassen (aktueller Ansatz)
class Character:
    def take_damage(self, amount):
        self.health -= amount


# Option 2: Vollständig ausgelagerte Schadenslogik
class DamageHandler:
    def apply_damage(self, obj, amount):
        if isinstance(obj, Character):
            obj.health -= amount
        elif isinstance(obj, Building):
            obj.health -= amount * 2


# Option 3: Hybrider Ansatz mit Handler
class Character:
    def accept_damage(self, handler):
        handler.handle_character_damage(self)
```

Welche Variante ist besser?

Designentscheidungen
sind kontextabhängig

Vererbung

- Eltern-/Kindklassen-Beziehungen
- Code-Wiederverwendung bei „ist-ein“-Beziehungen
- Beispiel: *Circle* ist eine *Shape:* `class Circle(Shape):`

Duck Typing

- Fokus auf Verhalten, nicht auf Typ
- „Wenn es wie eine Ente watschelt...“
- *Iterator-Protokoll* als Beispiel: `for item in my_list:`
- Spielobjekte, die Schaden nehmen können

Inversion of Control

- Verhalten an spezialisierte Klassen delegieren
- Flexibilität indem Verhalten in Klassen *injiziert* wird
- Beispiel: `game = GameBoard(ChristmasExplosionHandler())`

Klassen machen durch Vererbung Code wiederverwendbar.

Es sollte eine echte "ist-ein"-Beziehung bestehen:

Square ist ein *Rectangle*

Gemeinsames Interface ermöglicht einheitliche Behandlung verschiedener Objekte (explosion()-Funktion behandelt alle Objekte gleich)

Mit der Methode super() können Objekte einer Subklasse auf die Implementierung in der Basisklasse zugreifen und diese erweitern.

Inversion of Control (IoC) lagert Verhalten aus.

Beispiel: unsere ExplosionEffects

Duck Typing erlaubt es Objekten, die gleiche Funktionalität anzubieten, ohne verwandt zu sein.

Beispiel: Iterator-Protokoll

IoC erleichtert es, das Ziel Separation of Concerns zu erreichen, also klar getrennte Zuständigkeiten.

Beispiel: Spiellogik und Benachrichtigungen