Inhalt

[Objekte und Klassen : Objects and classes 2](#_Toc194994083)

[Ausblenden von Details und Erstellen der öffentlichen Schnittstelle : Hiding details and creating the public interface 3](#_Toc194994084)

[Abstraktion 3](#_Toc194994085)

[Komposition und Aggregation : Composition & Aggregation 3](#_Toc194994086)

[UML 5](#_Toc194994087)

[Ziel von UML 5](#_Toc194994088)

[Vorteile der UML 5](#_Toc194994089)

[UML-Diagrammtypen (wichtigste Kategorien) 5](#_Toc194994090)

[1. Strukturdiagramme (beschreiben den statischen Aufbau) 5](#_Toc194994091)

[2. Verhaltensdiagramme (beschreiben dynamisches Verhalten) 6](#_Toc194994092)

[Klassendiagramm 7](#_Toc194994093)

[Objektdiagramm 10](#_Toc194994094)

[Komponentendiagramm 10](#_Toc194994095)

[Verteilungsdiagramm (Deployment Diagram) 10](#_Toc194994096)

[Paketdiagramm 10](#_Toc194994097)

[Anwendungsfalldiagramm 11](#_Toc194994098)

[Aktivitätsdiagramm 11](#_Toc194994099)

[Sequenzdiagramm 11](#_Toc194994100)

[Zustandsdiagramm (State Machine Diagram) 12](#_Toc194994101)

[Vererbung – Inheritance 13](#_Toc194994102)

[Wie funktioniert Vererbung? 13](#_Toc194994103)

[Wichtige Aspekte der Vererbung 13](#_Toc194994104)

[Vorteile der Vererbung 13](#_Toc194994105)

[Mehrfache Vererbung - Multiple inheritance 13](#_Toc194994106)

[4+1 Views Modell 15](#_Toc194994107)

[1. Logische Sicht (Logical View) 15](#_Toc194994108)

[2. Prozesssicht (Process View) 15](#_Toc194994109)

[3. Entwicklungssicht (Development View) 16](#_Toc194994110)

[4. Physikalische Sicht (Physical View) 16](#_Toc194994111)

[5. Szenariosicht (Use-Case View) 16](#_Toc194994112)

***Die objektorientierte Analyse (OOA) ist der Prozess der Betrachtung eines Problems, eines Systems oder Aufgabe und die Identifizierung der Objekte und der Interaktionen zwischen diesen Objekten.*** In der Analysephase geht es darum, ***was*** getan werden muss. Eine bessere Formulierung wäre vielleicht objektorientierte Erkundung. In der Softwareentwicklung gehören zu den ersten Phasen der Analyse ***die Befragung von Kunden, die Untersuchung ihrer Prozesse und das Ausschließen von Möglichkeiten.***

***Objektorientierter Entwurf (OOD) ist der Prozess der Umwandlung solcher Anforderungen in eine Implementierungsspezifikation***. Der Designer muss die Objekte benennen, die Verhaltensweisen definieren und formal festlegen, welche Objekte bestimmte Verhaltensweisen bei anderen Objekten aktivieren können. In der Entwurfsphase geht es darum, das, ***was*** getan werden soll, in die Art und Weise umzuwandeln, ***wie*** es getan werden soll. ***Objektorientierte Programmierung (OOP) ist der Prozess der Umwandlung eines Entwurfs in ein funktionierendes Programm, das das tut, was der Produktinhaber ursprünglich wollte.***

Bei der iterativen Entwicklung wird ein kleiner Teil der Aufgabe modelliert, entworfen und programmiert, und dann wird das Produkt überprüft und erweitert, um jede Funktion zu verbessern und neue Funktionen in einer Reihe von kurzen Entwicklungszyklen aufzunehmen.

# Objekte und Klassen : Objects and classes

***„Ein Objekt ist ein Ding mit Eigenschaften (Attribute) und Fähigkeiten (Methoden).“***

***Ein Objekt ist eine Sammlung von Daten mit zugehörigen Verhaltensweisen.***

***„Eine Klasse ist der Bauplan, ein Objekt ist das fertige Haus.“***

***In der objektorientierten Modellierung wird der Begriff „Klasse“ für eine Art von Objekt verwendet.***

***Objekte sind Instanzen von Klassen, die miteinander verbunden werden können. Eine Klasseninstanz ist ein bestimmtes Objekt mit einem eigenen Satz von Daten und Verhaltensweisen; eine bestimmte Orange auf dem Tisch vor uns wird als Instanz der allgemeinen Klasse der Orangen bezeichnet.***

***Daten stellen die individuellen Merkmale eines bestimmten Objekts dar. Eine Klasse kann bestimmte Sätze von Merkmalen definieren, die für alle Objekte gelten, die Mitglieder dieser Klasse sind. Jedes spezifische Objekt kann verschiedene Datenwerte für die gegebenen Merkmale haben. In Python kann man ein Attribut auch als Instanzvariable bezeichnen.***

***Verhaltensweisen sind Aktionen, die an einem Objekt ausgeführt werden können***. Die ***Verhaltensweisen, die an einer bestimmten Objektklasse ausgeführt werden können, werden als Methoden der Klasse ausgedrückt***. Auf der Programmierebene sind Methoden wie Funktionen in der strukturierten Programmierung, aber sie haben ***Zugriff auf die Attribute*** - insbesondere auf die ***Instanzvariablen*** mit den Daten, die mit diesem Objekt verbunden sind. ***Wie Funktionen können auch Methoden Parameter und Rückgabewerte annehmen.***

***Bei der objektorientierten Analyse und Entwicklung geht es darum, herauszufinden, was diese Objekte sind und wie sie zusammenwirken sollen. Jede Klasse hat Zuständigkeiten und Kooperationen.***

# Ausblenden von Details und Erstellen der öffentlichen Schnittstelle : Hiding details and creating the public interface

***„Kapselung schützt die Innereien einer Klasse vor der Außenwelt.“***

***Der Hauptzweck der Modellierung eines Objekts im objektorientierten Design besteht darin, die öffentliche Schnittstelle dieses Objekts zu bestimmen.*** ***Die Schnittstelle ist die Sammlung von Attributen und Methoden, auf die andere Objekte zugreifen können, um mit diesem Objekt zu interagieren.*** Andere Objekte müssen nicht auf das Innenleben des Objekts zugreifen, und in einigen Sprachen ist dies auch nicht erlaubt.

***Ziel von Information Hiding ist es, interne Implementierungsdetails einer Klasse vor der Außenwelt zu verbergen, sodass Nutzer der Klasse nur das benutzen können, was ausdrücklich als "öffentlich" vorgesehen ist.***

***Encapsulation (zu Deutsch: Kapselung) :Es bezeichnet den Ansatz, Daten (Attribute) und die zugehörige Logik (Methoden) in einer einzelnen Einheit – der Klasse – zu bündeln und gleichzeitig den Zugriff auf die Daten zu kontrollieren.***

## Abstraktion

**„Abstraktion zeigt, was ein Objekt tut – nicht, wie es das tut.“**

Es bedeutet, dass eine Klasse nur die **wesentlichen Merkmale** eines Objekts nach außen sichtbar macht und **alle unwesentlichen Details ausblendet**.

Abstraktion ist der Prozess der Kapselung von Informationen durch eine eigene öffentliche Schnittstelle. Alle privaten Elemente können Gegenstand von Information Hiding sein. In UML-Diagrammen können wir ein führendes - anstelle eines führenden + verwenden, um anzudeuten, dass es nicht Teil einer öffentlichen Schnittstelle ist.

# Komposition und Aggregation : Composition & Aggregation

Komposition und Aggregation sind zwei Arten von "Hat-ein"-Beziehungen in der objektorientierten Programmierung (OOP). Sie beschreiben, wie Objekte miteinander verbunden sind, insbesondere wie ein Objekt ein anderes enthält oder verwendet.

Komposition:

***„Ohne das Ganze gibt es den Teil nicht.“***

* Ein Objekt **besteht aus** anderen Objekten.
* Die Lebensdauer der enthaltenen Objekte ist **an das Hauptobjekt gebunden**.
* Wenn das Hauptobjekt gelöscht wird, werden auch die enthaltenen Objekte gelöscht.

Aggregation:

***„Der Teil lebt auch ohne das Ganze.“***

* Ein Objekt nutzt ein anderes, aber ist nicht dessen Eigentümer.
* Die Lebensdauer des enthaltenen Objekts ist unabhängig vom Hauptobjekt.
* Mehrere Objekte können dasselbe Teilobjekt teilen.

# UML

***„UML ist der Bauplan für Software – visuell, strukturiert, objektorientiert.“***

**UML (Unified Modeling Language)** ist eine standardisierte **grafische Notationssprache**, mit der man **Softwarearchitektur, Design und Prozesse** visuell darstellen kann.  
Sie dient der **Planung, Kommunikation und Dokumentation** von Softwaresystemen – besonders in der objektorientierten Entwicklung.

## Ziel von UML

* **Verständliche Visualisierung** von Systemen
* **Abstraktion komplexer Zusammenhänge**
* **Kommunikation zwischen Entwicklern, Architekten, Fachabteilungen**
* **Standardisierung von Entwürfen**
* **Dokumentation vor oder neben dem Code**

## Vorteile der UML

* **Standardisierte Sprache** für Entwickler und Stakeholder
* **Tool-unabhängig**
* **Fördert saubere Architektur und Planung**
* **Hilft bei Wartung und Erweiterung**
* **Nützlich für Teamkommunikation**

## UML-Diagrammtypen (wichtigste Kategorien)

UML-Diagramme werden in zwei Hauptgruppen unterteilt:

### **Strukturdiagramme** (beschreiben den statischen Aufbau)

***„Strukturdiagramme zeigen,* was *existiert – nicht,* wie *es sich verhält.“***

* **Klassendiagramm**: zeigt Klassen, Attribute, Methoden, Beziehungen (z. B. Vererbung, Assoziation)
* **Objektdiagramm**: zeigt konkrete Objekte (Instanzen) zur Laufzeit
* **Komponentendiagramm**: zeigt Software-Komponenten und deren Abhängigkeiten
* **Paketdiagramm**: zeigt Gruppierungen von Klassen (Module, Namespaces)

Ein Bild, das Text, Screenshot enthält.

Automatisch generierte Beschreibung

### **Verhaltensdiagramme** (beschreiben dynamisches Verhalten)

***„Verhaltensdiagramme zeigen, was passiert – Strukturdiagramme zeigen, was existiert.“***

* **Anwendungsfalldiagramm (Use Case)**: zeigt, wie Benutzer (Akteure) mit dem System interagieren
* **Sequenzdiagramm**: zeigt zeitlichen Ablauf von Nachrichten/Methoden zwischen Objekten
* **Aktivitätsdiagramm**: zeigt logische Abläufe / Workflows (ähnlich Flussdiagramm)
* **Zustandsdiagramm**: zeigt Zustände eines Objekts und Übergänge bei bestimmten Ereignissen
* **Verteilungsdiagramm**: zeigt physische Verteilung auf Hardware (Server, Geräte)

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

Automatisch generierte Beschreibung

# Klassendiagramm

***Die Assoziation ist die grundlegendste Art und Weise, wie Instanzen zweier Klassen miteinander in Beziehung stehen können. UML ist äußerst nützlich, um eine schnelle, einfache und konsistente Kommunikation zu gewährleisten.*** Die UML umfasst weit mehr als Klassen- und Objektdiagramme; sie verfügt auch über eine Syntax für Anwendungsfälle, Bereitstellung, Zustandsänderungen und Aktivitäten.

Zeigt:

* Klassen (mit Namen, Attributen, Methoden)
* Beziehungen:
  + **Vererbung (Generalization)**: Dreieck mit Linie
  + **Assoziation**: einfache Linie
  + **Aggregation**: leerer Rhombus
  + **Komposition**: gefüllter Rhombus
  + **Abhängigkeit (Dependency)**: gestrichelte Linie mit Pfeil

**Verwendet für:**

* Datenmodellierung
* Architekturentwurf
* Objektorientierte Planung

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

Automatisch generierte Beschreibung

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

Automatisch generierte Beschreibung

# Objektdiagramm

**Was zeigt es?**

* **Konkrete Instanzen** von Klassen zu einem bestimmten Zeitpunkt
* Beispielhafte Ausprägung des Klassendiagramms

**Verwendet für:**

* Testfälle, Laufzeitbeobachtung, Beispielkonfigurationen

**Unterschied zum Klassendiagramm:**

* Zeigt **konkrete Objekte** statt allgemeiner Klassen

# Komponentendiagramm

**Was zeigt es?**

* **Softwarekomponenten** (z. B. Module, Bibliotheken, Services) und deren **Abhängigkeiten**

**Verwendet für:**

* **Modularisierung** von großen Systemen
* Beschreibung der **technischen Architektur**

**Typisch bei:**

* Microservices, Plugins, Frameworks

# Verteilungsdiagramm (Deployment Diagram)

**Was zeigt es?**

* **Physische Verteilung** von Softwarekomponenten auf **Hardware-Knoten** (z. B. Server, Geräte)

**Verwendet für:**

* Systembereitstellung
* Netzwerktopologien
* Laufzeitumgebung

# Paketdiagramm

**Was zeigt es?**

* Gruppierungen von Klassen in **Pakete / Namespaces / Module**

**Verwendet für:**

* Strukturierung des Codes
* Modularisierung
* Übersicht großer Projekte

# Anwendungsfalldiagramm (Use Case)

## **Zweck und Nutzen**

* Darstellung der **funktionalen Anforderungen** eines Systems.
* Kommunikation mit **Stakeholdern**, Kunden, Fachexperten.
* Grundlage für **Testfälle**, **Systemdesign** und **Implementierung**.
* Verdeutlicht, **wer** was **mit dem System macht**.

### **Akteur (Actor)**

* Externes Objekt (Person, System, Organisation), das **mit dem System interagiert**.
* Wird **außerhalb** des Systems dargestellt.
* Symbol: **Strichmännchen**
* Beispiel: „Kunde“, „Administrator“, „Zahlungssystem“

### **Anwendungsfall (Use Case)**

* Beschreibt eine **funktionale Einheit** – eine Handlung oder Dienstleistung, die das System einem Akteur anbietet.
* Symbol: **Oval**
* Beispiel: „Bestellung aufgeben“, „Produkt suchen“, „Bericht generieren“

### **Systemgrenze**

* Ein **Rechteck**, das das **zu modellierende System** abgrenzt.
* Enthält alle Use Cases, aber **nicht** die Akteure.

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

Automatisch generierte Beschreibung

**Was zeigt es?**

* **Interaktionen** zwischen Benutzern (Akteuren) und dem System
* Welche **Funktionen / Anwendungsfälle** dem Nutzer zur Verfügung stehen

**Verwendet für:**

* **Anforderungsanalyse**
* Erste **Überblicksdarstellung** für Fachabteilungen

**Beispiel:**

* Ein Kunde kann sich registrieren, Produkte ansehen, eine Bestellung aufgeben

# Aktivitätsdiagramm

Es beschreibt den **Ablauf von Aktivitäten (Prozessen)** in einem System. Man kann damit **Geschäftsprozesse**, **Workflows**, **Programmlogik** oder **Systemverhalten** modellieren.

**Ziel:**  
Zeigen, **wie ein Prozess oder ein Ablauf funktioniert** – inklusive Entscheidungen, paralleler Abläufe und Enden.

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

Automatisch generierte Beschreibung

**Was zeigt es?**

* **Abläufe**, **Prozesse** und **Workflows**
* Alternative und parallele Wege sind darstellbar

**Verwendet für:**

* **Geschäftsprozesse**
* Algorithmen / Prozesslogik
* Bedingte Abläufe

**Beispiel:**

* Bestellprozess: Produkt auswählen → Warenkorb → Bezahlen → Bestellbestätigung

# Sequenzdiagramm

**Was zeigt es?**

* Den **zeitlichen Ablauf** von Nachrichten / Methodenaufrufen zwischen Objekten

**Verwendet für:**

* Detailanalyse von **Abläufen zwischen Klassen**
* **Kommunikationsverläufe** in Echtzeit oder verteilten Systemen

**Besonderheit:**

* Zeitachse von oben nach unten
* Lebenslinien der Objekte

**Beispiel:**

* Kunde ruft login() auf → Authentifizierungsservice → Datenbank → Antwort zurück

# Zustandsdiagramm (State Machine Diagram)

**Was zeigt es?**

* **Zustände eines Objekts** und **Übergänge** bei bestimmten Ereignissen

**Verwendet für:**

* Objekte mit **komplexem Lebenszyklus** oder **veränderlichem Verhalten**

**Typische Anwendung:**

* Tür: offen → (schließen) → geschlossen → (verriegeln) → verriegelt
* Zustandswechsel durch Events (z. B. "Benutzer klickt auf Button")

# Vererbung – Inheritance

***„Vererbung ist wie ein Kind, das von den Eigenschaften und Fähigkeiten der Eltern erbt, aber auch eigene Fähigkeiten entwickeln kann.“***

**Vererbung** ist ein zentrales Konzept der objektorientierten Programmierung (OOP), das es einer **Unterklasse** ermöglicht, **Eigenschaften** (Attribute) und **Verhalten** (Methoden) einer **Oberklasse** zu **erben**. Sie ermöglicht eine **hierarchische Struktur** und fördert **Wiederverwendbarkeit** sowie **Erweiterbarkeit** von Code.

## **Wie funktioniert Vererbung?**

* **Oberklasse (Superklasse)**: Die Klasse, die die gemeinsamen Eigenschaften und Methoden bereitstellt.
* **Unterklasse (Subklasse)**: Die Klasse, die von der Oberklasse erbt und deren Eigenschaften und Methoden übernimmt.
* **Vererbung ermöglicht Erweiterung**: Die Unterklasse kann die geerbten Methoden und Eigenschaften verwenden oder **überschreiben**, um spezifisches Verhalten hinzuzufügen oder zu ändern.

### **Wichtige Aspekte der Vererbung**

1. **Methodenüberschreibung (Override)**:  
   Die Unterklasse kann geerbte Methoden überschreiben, um ihr eigenes Verhalten zu implementieren.
2. **Super-Schlüsselwort**:  
   Mit super() kann die Unterklasse auf die Methoden und den Konstruktor der Oberklasse zugreifen.
3. **Mehrfachvererbung**:  
   In einigen Programmiersprachen (z. B. Python) ist Mehrfachvererbung möglich, d. h. eine Klasse kann von mehr als einer Oberklasse erben.
4. **Polymorphismus**:  
   Vererbung ermöglicht Polymorphismus, d. h. Objekte der Unterklasse können an Stellen verwendet werden, die eigentlich für Objekte der Oberklasse gedacht sind.

### **Vorteile der Vererbung**

* **Code-Wiederverwendbarkeit**: Einmal definierte Funktionen in der Oberklasse können von allen Unterklassen wiederverwendet werden.
* **Erweiterbarkeit**: Neue Funktionalitäten können durch Erstellen neuer Unterklassen hinzugefügt werden, ohne den bestehenden Code zu verändern.
* **Hierarchie**: Vererbung erleichtert das Erstellen von Klassenhierarchien, die die realen Beziehungen von Objekten besser widerspiegeln.

# Mehrfache Vererbung - Multiple inheritance

***„Mehrfache Vererbung ist wie ein Kind, das Merkmale von mehreren Eltern übernimmt und diese kombiniert.“***

**Mehrfache Vererbung** tritt auf, wenn eine **Unterklasse von mehr als einer Oberklasse erbt**. Das bedeutet, dass eine Klasse **mehr als eine Elternklasse** hat und somit die Eigenschaften und Methoden von mehreren Klassen übernehmen kann.

Ein häufiges Problem bei der Mehrfachvererbung ist der **Methode- oder Attribut-Konflikt**. Wenn zwei Oberklassen dieselbe Methode oder dasselbe Attribut haben, weiß die Unterklasse möglicherweise nicht, welche Methode oder welches Attribut sie verwenden soll.

Die **Method Resolution Order (MRO)** legt die Reihenfolge fest, in der die Methoden aus den Oberklassen aufgerufen werden.

# 4+1 Views Modell

***„Das 4+1 Views Modell hilft, ein Softwaresystem aus verschiedenen Perspektiven zu betrachten – als Ganzes und in seinen Teildimensionen.“***

Das **4+1 Views Modell** ist ein Architekturmodell, das von **Philippe Kruchten** entwickelt wurde, um die Architektur eines Softwaresystems zu beschreiben. Es bietet eine strukturierte Sichtweise auf ein System aus verschiedenen Perspektiven, die helfen, die Systemarchitektur klar zu verstehen und zu kommunizieren.

Das Modell besteht aus **5 verschiedenen Sichtweisen** (4+1), die alle Aspekte eines Systems beschreiben:

1. **Logische Sicht**
2. **Prozesssicht**
3. **Entwicklungssicht**
4. **Physikalische Sicht**
5. **Szenariosicht** (oder Use-Case-Sicht)

Jede dieser Sichten konzentriert sich auf verschiedene Aspekte der Softwarearchitektur, und zusammen bieten sie eine umfassende Sicht auf das System.

## **1. Logische Sicht (Logical View)**

* **Ziel**: Die **Funktionalität des Systems** aus Sicht der **Benutzer** und **Geschäftslogik**.
* Diese Sicht beschreibt, **wie das System funktioniert**, ohne sich mit der Implementierung oder den technischen Details zu beschäftigen. Sie konzentriert sich auf die **Datenmodelle**, **Klassendiagramme**, und **Schnittstellen** der Software.
* **Wer benötigt es?**: Architekten, Entwickler, und Benutzer.

**Beispiel:**

* In einem E-Commerce-System könnten hier Klassen wie Bestellung, Kunde, Zahlung und ihre Beziehungen beschrieben werden.

## **2. Prozesssicht (Process View)**

* **Ziel**: Beschreibt die **dynamische Interaktion** zwischen den **Prozessen**, die das System während der Ausführung ausführt.
* Diese Sicht befasst sich mit **Concurrency**, **Threads**, **Prozesskommunikation** und der **Verwaltung von Ressourcen**. Sie zeigt auf, **wie Prozesse zusammenarbeiten**, um die Funktionalität zu erreichen.
* **Wer benötigt es?**: Architekten, Systemadministratoren, Entwickler.

**Beispiel:**

* Ein Webserver, der Anfragen verarbeitet, und der zugehörige Prozess zur Bearbeitung der Anfragen.

## **3. Entwicklungssicht (Development View)**

* **Ziel**: Beschreibt die **Systemarchitektur aus Sicht der Entwickler** und konzentriert sich auf **Module, Komponenten und deren Beziehungen**.
* Diese Sicht zeigt, **wie das System in verschiedene Softwareeinheiten (Module/Komponenten)** aufgeteilt wird, wie diese in der Entwicklung verwendet werden, und wie die **Codebasis** organisiert ist.
* Sie beschreibt **Abhängigkeiten**, **Pakete**, **Bibliotheken** und **Entwicklungsumgebungen**.
* **Wer benötigt es?**: Entwickler und Softwarearchitekten.

**Beispiel:**

* Ein Modul für die **Datenbankkommunikation**, ein anderes für die **Benutzerauthentifizierung** und wieder ein anderes für die **Benutzeroberfläche**.

## **4. Physikalische Sicht (Physical View)**

* **Ziel**: Beschreibt die **Hardwarekomponenten** und deren **Verteilung**, die das System ausführen.
* Diese Sicht stellt dar, **wie Softwarekomponenten auf physischen Maschinen** (Server, Datenbanken, Netzwerke etc.) laufen. Sie befasst sich mit **Deployment**, **Netzwerkarchitektur** und **Hardwareanforderungen**.
* Sie geht auf die **Verteilung von Software auf Hardware** und die **Kommunikation zwischen Maschinen** ein.
* **Wer benötigt es?**: Systemadministratoren, Netzwerkingenieure.

**Beispiel:**

* Ein Webserver könnte auf einem physischen Server laufen, während die Datenbank auf einem anderen Server gehostet wird.

## **5. Szenariosicht (Use-Case View)**

* **Ziel**: Veranschaulicht das Systemverhalten aus Sicht der **Benutzerinteraktionen**.
* Diese Sicht zeigt, **wie Benutzer mit dem System interagieren** und wie das System auf **konkrete Anwendungsfälle oder Szenarien** reagiert. Sie ist besonders nützlich, um **non-funktionale Anforderungen** wie **Leistung**, **Fehlerbehandlung** oder **Skalierbarkeit** zu verstehen.
* **Wer benötigt es?**: Alle Beteiligten (Stakeholder, Architekten, Entwickler).

**Beispiel:**

* Ein Benutzer loggt sich ein, stöbert in Produkten und tätigt eine Bestellung. Dieses Szenario wird beschrieben, um zu verstehen, welche Funktionalitäten das System bieten muss und wie es darauf reagieren soll.

# Abstrakte Klassen

**Abstrakte Klassen** sind Klassen, die **nicht direkt instanziiert** werden können. Sie dienen als **Basisklassen** und definieren ein gemeinsames **Interface**, das von abgeleiteten Klassen **implementiert** werden muss. In Python verwendet man dazu das Modul abc (*Abstract Base Classes*).

### **Zweck abstrakter Klassen**

* Strukturierung des Codes
* Definition von Schnittstellen
* Vermeidung unvollständiger Implementierungen
* Unterstützung für **Polymorphie** in der objektorientierten Programmierung

### **Wichtige Begriffe**

* **Abstrakte Klasse**: Eine Klasse, die mindestens eine abstrakte Methode enthält.
* **Abstrakte Methode**: Eine Methode ohne Implementierung, die in Unterklassen überschrieben werden muss.

### **Implementierung in Python**

Python nutzt das Modul abc:

* ABC: Basisklasse für abstrakte Klassen
* @abstractmethod: Dekorator zum Kennzeichnen abstrakter Methoden

### **Besonderheiten**

* Eine Unterklasse **muss alle abstrakten Methoden** implementieren, sonst bleibt auch sie abstrakt.
* Abstrakte Klassen können **normale Methoden und Attribute** enthalten.
* Mehrere abstrakte Klassen können vererbt werden (Mehrfachvererbung).

### **Praktischer Nutzen**

* Ideal für **Plugin-Systeme**, **Frameworks**, **APIs**.
* Sicherstellung, dass bestimmte Methoden in Subklassen implementiert werden.
* Vereinheitlichung von Schnittstellen bei verschiedenen Implementierungen.

# Polymorphismus

**Polymorphismus** bedeutet wörtlich „Vielgestaltigkeit“. In der Programmierung bezeichnet es die Fähigkeit, dass **unterschiedliche Objekte über dieselbe Schnittstelle angesprochen werden können**, obwohl sie intern unterschiedlich funktionieren.

### **Grundidee von Polymorphismus**

Man schreibt Code, der mit **Oberklassen oder Schnittstellen** arbeitet, aber zur Laufzeit mit **Objekten der Unterklassen** umgehen kann – ohne zu wissen, welche konkreten Klassen es sind.

### **Arten von Polymorphismus**

* **Subtypen-Polymorphismus (klassisch, durch Vererbung):**  
  Eine Methode ist in der Basisklasse definiert (oft als abstrakt), aber in den Unterklassen unterschiedlich implementiert.
* **Duck Typing (dynamisch, Python-spezifisch):**  
  Wenn ein Objekt **"sich wie etwas verhält"**, kann es wie dieses Objekt verwendet werden – unabhängig von der Vererbungshierarchie.

Ein Bild, das Text, Screenshot, Software, Multimedia-Software enthält.

Automatisch generierte Beschreibung

Ein Bild, das Text, Elektronik, Screenshot, Software enthält.

Automatisch generierte Beschreibung

# k-NN (k-Nearest Neighbors)

**k-NN** steht für **"k-Nearest Neighbors"** und ist ein **einfacher, aber effektiver Algorithmus** aus dem Bereich des **überwachten maschinellen Lernens**. Er wird hauptsächlich für **Klassifikations-** und **Regressionsprobleme** eingesetzt.

## **Grundidee**

Beim k-NN-Algorithmus wird eine Vorhersage für ein neues Datenobjekt getroffen, indem man sich dessen **„nächste Nachbarn“** im Merkmalsraum anschaut – also die **k** Objekte, die ihm am ähnlichsten sind.

**Prinzip:**

* **"Sag mir, was deine Nachbarn sind, und ich sage dir, was du bist."**

## **Ablauf des Algorithmus (Klassifikation)**

Angenommen, du hast eine Menge von Objekten mit bekannten Klassenlabels, z. B. Punkte in einem Koordinatensystem.

### Schritte:

1. **Wähle eine Zahl k** (Anzahl der Nachbarn, z. B. 3 oder 5).
2. **Berechne die Distanz** zwischen dem neuen Punkt und allen Trainingspunkten.
3. **Finde die k nächsten Nachbarn** mit den geringsten Distanzen.
4. **Stimme ab**: Die Klasse, die unter diesen k Nachbarn am häufigsten vorkommt, wird als Ergebnis genommen.

## **Beispiel (Klassifikation)**

Ein neues Tier hat folgende Merkmale:

* Gewicht: 10 kg
* Größe: 40 cm

Die bekannten Tiere in der Datenbank sind Hunde und Katzen mit diesen Merkmalen. Der Algorithmus vergleicht die Merkmale des neuen Tieres mit allen vorhandenen und wählt z. B. die 5 nächsten Tiere. Wenn davon 3 Hunde und 2 Katzen sind, sagt der Algorithmus: **"Das ist ein Hund."**

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

Automatisch generierte Beschreibung

## **Wahl von k**

* **Kleinere Werte von k**: sensibel gegenüber Ausreißern (z. B. k=1).
* **Größere Werte von k**: glattere Grenzen, aber möglicherweise ungenauer bei stark gemischten Daten.
* Typisch: **ungerade Zahl**, um bei Klassifikation **Stimmengleichheit zu vermeiden**.

## **Vor- und Nachteile**

### Vorteile:

* Einfach zu verstehen und zu implementieren
* Keine Annahmen über die Verteilung der Daten
* Funktioniert gut bei kleinerem Datenvolumen

### Nachteile:

* **Langsam bei großen Datensätzen** (muss alle Distanzen berechnen)
* Sensitiv auf irrelevante Features oder Skalierung (Feature-Skalierung oft notwendig)
* Speicherintensiv (speichert alle Trainingsdaten)

## **Anwendungsbeispiele**

* Bilderkennung
* Empfehlungssysteme
* Textklassifikation
* Betrugserkennung
* Medizinische Diagnosen