EUROPA-FACHBUCHREIHE
für IKT

IT Grund- und Fachstufe für Technische IT-Berufe

Bearbeitet von Lehrern und Ingenieuren an beruflichen Schulen

8. Auflage

VERLAG EUROPA-LEHRMITTEL · Nourney, Vollmer GmbH & Co. KG
Düsselberger Straße 23 · 42781 Haan-Gruiten

Europa-Nr.: 36519

Autoren der IKT-Fachkunde

Monika Burgmaier	Durbach
Patricia Burgmaier	Melsungen
Fréderique Chauffer	Offenburg
Elmar Dehler	Ulm
Ute Jansen	Sindelfingen
Hermann Münch	Stuttgart
Jan Quast	Berlin
Bernd Schiemann	Durbach

Bildbearbeitung:
Zeichenbüro Verlag Europa-Lehrmittel, Ostfildern

Lektorat:
Bernd Schiemann, Durbach

8. Auflage 2025
Druck 5 4 3 2 1
Alle Drucke derselben Auflage sind parallel einsetzbar, da sie bis auf die Korrektur von Druckfehlern identisch sind.

ISBN 978-3-8085-3301-7

Diesem Buch wurden die neuesten Ausgaben der DIN-Blätter und der VDE-Bestimmungen zugrunde gelegt. Verbindlich sind jedoch nur die DIN-Blätter und VDE-Bestimmungen selbst.
Die DIN-Blätter können von der Beuth-Verlag GmbH, Burggrafenstraße 4–7, 10787 Berlin 30, und Kamekestraße 2–8, 50672 Köln, bezogen werden.

Umschlag: braunwerbeagentur, 42477 Radevormwald
Umschlagfoto: © Michael Traitor - stock.adobe.com
Satz: Dipl. Des. Susanne Beckmann, 59514 Welver
Druck: Akontext s.r.o., 141 00 Prag 4 (CZ)

Vorwort zur 8. Auflage

Liebe Leserin, lieber Leser,

mit der **Grund- und Fachstufe für technische IT-Berufe** besitzen Sie **in einem Band** ein Fachbuch für die Erstausbildung zum/zur Fachinformatiker/-in in allen Fachrichtungen sowie zum/zur IT-System-Elektroniker/-in, das prüfungsrelevante Inhalte in kompakter und übersichtlicher Form darstellt und das selbstständige Lernen fördert.

Projekte und Testaufgaben zu **Prüfungsthemen** finden Sie unter
Testen Sie Ihre Handlungskompetenz. Diese Auflage bietet Ihnen 38 Seiten zum Üben und Vertiefen. Ausführliche Lösungsvorschläge finden Sie dazu kostenlos in der EUROPATHEK.

Die **8. Auflage** wurde neu bearbeitet und um zahlreiche neue Inhalte erweitert:

Qualitätsmanagement, Scrum, PPS, Kundenberatung und Angebotserstellung, Kalkulation, Single Pair Ethernet (SPE), VPN, Cloud im digitalen Umfeld, IT-Grundschutz, Turing-Maschine, Programmstrukturvergleich für C#, C++, Java, Python, OOP mit UML für C#, C++, Java, Problemlösungstechniken, Cyber-Physische Systeme, Raspberry Pi, Übersicht Serverbetriebssysteme, NoSQL-Datenbanken, Container mit Docker, Netzwerke mit Linux, Dienstgüte QoS, Webhosting.

Hinweise auf verwendete Tabellenbücher werden im Buch abgekürzt verwendet, z. B. **TabIGSA** für „Tabellenbuch Informations-, Geräte-, System- und Automatisierungstechnik“ oder **TabIT** für „IT-Tabellenbuch“.

Digitalisierung und Industrie 4.0, IT-Security, IT-Systemtechnik, Mobile Computing, Mobile Devices, Mensch-Maschine-Schnittstelle (HMI), Virtuelle Welten, Produktionsplanungs- und Steuerungssysteme sind auch wichtige Inhalte in anderen Bildungsplänen. Wir empfehlen deshalb dieses Buch auch Lernenden von Informationstechnischen Gymnasien, Fachgymnasien, Fachoberschulen, Berufskollegs, Technikerschulen und Studierenden als eine kompakte Übersicht über die verschiedenen Themengebiete der Informatik.

Den Autoren ist es wichtig, schwierige Zusammenhänge in verständlicher Sprache zu formulieren und durch mehrfarbige Bilder, Diagramme und Tabellen zu veranschaulichen.

Ihre Meinung zum Buch interessiert uns. Teilen Sie uns Ihre Verbesserungsvorschläge, Ihre Kritik, aber auch Ihre Zustimmung zum Buch mit. Schreiben Sie eine E-Mail an
lektorat@europa-lehrmittel.de

Das Autorenteam und der Verlag Europa-Lehrmittel Sommer 2025

Lernfelder[1] IT-Grund- und Fachwissen im Überblick

[1] **In den Lernfeldern 1 bis 9 sind die Informationen für das Arbeiten in den erweiterten projektbezogenen Lernfeldern 10, 11 und 12 der Fachrichtungen Anwendungsentwicklung und Systemintegration zu finden.**

Lernfeld 8

Daten systemübergreifend bereitstellen

1.2 Agile Softwareentwicklung

1.2.1 Grundlagen

Die rasante Zunahme von Informationen und Daten, die Unternehmensentscheidungen beeinflussen, benötigt schnelle und flexible Anpassungsprozesse der benutzten Software. Dabei ist ein modernes Führungskonzept, das im Unternehmen den Mitarbeitern agiles[1] Handeln ermöglicht, hilfreich. Bei immer mehr Projekten kommen agile Methoden zur Anwendung.

2001 erstellten Softwareentwickler in Utah ein agiles Manifest.

Sie formulierten vier Grundwerte:

1 **Menschen** vor Prozessen und Werkzeugen
2 **Funktionierende Software** vor umfangreicher Dokumentation
3 **Zusammenarbeit mit dem Kunden** vor Vertragsverhandlungen
4 **Flexibilität** vor Plantreue.

Aus diesen Grundwerten leiteten sie 12 agile Prinzipien ab **(Bild 1)**.

Alle agilen Methoden basieren auf den Grundwerten und Grundprinzipien des agilen Manifestes.

Gemeinsamkeiten aller agilen Methoden sind:

- Arbeiten in selbstorganisierten Teams,
- intensive Kommunikation im Team und mit dem Kunden,
- iterative[2] Arbeitsweise zur ständigen Verbesserung des Ergebnisses,
- Eigenverantwortung des Teams für die Erreichung eines qualitativ hochwertigen Ergebnisses.
- Verwendung verschiedener agiler Techniken.

Typische Beispiele von agilen Techniken sind:

- Taskboards **(Bild2a)**,
- Burn-Down-Charts **(Bild 2b)**,
- Daily-Standup-Meetings **(Bild 2c)**,
- User Story,
- Definition of Done.

Die **User Story** ersetzt in der agilen Vorgehensweise das Lastenheft. Im Lastenheft wurden bis ins Detail die Anforderungen des Kunden vor Projektbeginn erstellt. In einer User-Story sollen ausschließlich die drei Fragen „**Wer** will **was** und **warum**“ beantwortet werden. Die Kommunikation zwischen Entwickler und Kunden während des Projekts zur gegenseitigen Rückmeldung ist von Bedeutung.

Beispiele für agile[1] Methoden zur Softwareentwicklung
SCRUM
KANBAN
Extreme Programming (XP)
Dynamic System Development Method (DSDM)

Agile Prizipien

1 Kunden zufriedenstellen
2 Änderungen sind willkommen
3 Häufige Auslieferung
4 Fachübergreifende Zusammenarbeit
5 Unterstützen und Vertrauen
6 Direkte Kommunikation
7 Funktionierende Lösungen
8 Nachhaltige Geschwindigkeit
9 Streben nach Qualität
10 Einfachheit ist von Bedeutung
11 Selbstorganisiert Handeln
12 Reflektieren und anpassen

In Anlehnung an die 12 agilen Prinzipen nach agilemanifesto.org

Bild 1: Agile Prinzipien

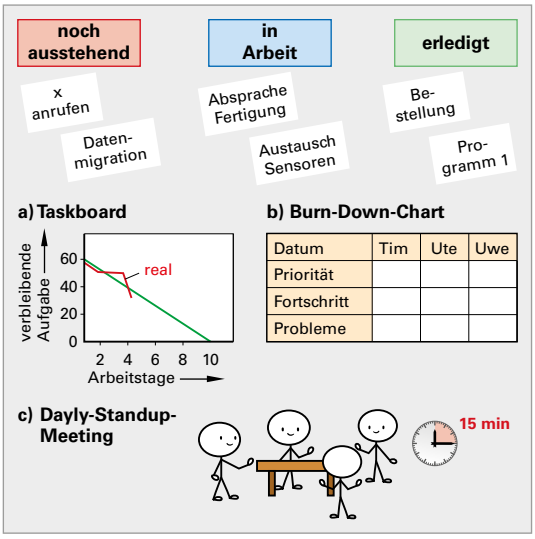

Bild 2: Beispiele für agile Techniken

Die **Definition of Done (DoD)** enthält z. B. in Checklistenform alle Aktivitäten, die für die User Story abgehakt werden müssen. Sie dient zur Abstimmung zwischen Auftraggeber und Entwicklungsteam. Außerdem sorgt DoD im Entwicklungsteam für Klarheit, welche Aufgaben schon erledigt sind.

[1] agil von agere (lat) = handeln, im Sinne von beweglich, tätig
[2] iterativ = wiederholend, sich schrittweise der exakten Lösung anpassen

1.2.2 Scrum

Entstehung

Die Ursprünge von **Scrum** legten die beiden Wissenschaftler Takeuchi und Nonaka aus Japan. Sie entwickelten den Ansatz der japanischen Unternehmen weiter, mit der diese in der Automobilproduktion und in der Unterhaltungselektronik innovative Produkte erzeugten.

Zur bildhaften Beschreibung ihres Modells verwendeten sie einen Begriff aus dem Rugby. Das englische Wort Scrum steht für das Gedränge beim Rugby. Die Mannschaft ist dann erfolgreich, wenn beim Spiel der Ball flexibel zwischen den Teammitgliedern wandert und alle in Bewegung sind.

Später übertrugen die Amerikaner Sutherland und Schwaber diese Ideen auf die Softwareentwicklung. In regelmäßigen Zeitabständen überarbeiten sie ihren „Scrum Guide", der in vielen Sprachen die Grundzüge von Scrum erklärt.

Scrum Entwicklungsbeteiligte:
1986 **Ikujirō Nonaka** und **Hirotaka Takeuchi**
1993 Jeff Sutherland
1995 Ken Schwaber
➔ aktueller Scrum Guide unter
https://scrumguides.org

5 Grundsätze des Lean Thinking:
- Im Zentrum steht der Kundenmehrwert
- Prozessfluss wird analysiert und optimiert
- Arbeitsablauf verläuft reibungslos ➔ (Flow)
- Absprachen mit Auftraggeber im Flow ➔ (Pull)
- Streben nach Perfektion

Grundsätze

Die Grundideen für das Arbeiten von Teams mit der Scrum-Methode sind aus Erfahrungswissen (Empirie) und **Lean Thinking** (Schlankes Denken) abgeleitet.

Für die gemeinsame schrittweise Annäherung an die bestmögliche Lösung eines Projekts muss das damit betraute Team die geltenden Spielregeln bei seiner Arbeit und die jeweiligen Rollen und den Gesamtprozessablauf kennen.

Die Einhaltung der Scrum-Grundsätze **(Bild 1)** unterstützen die erfolgreiche Lösung des Projekts.

So sind alle Projektbeteiligten über den Stand des Projektfortschritts, zu lösende Probleme oder Anwendung bestimmter Richtlinien zu informieren.

Bewertungen und Feedbacks der Zusammenarbeit des Teams und des Projektfortschritts erfolgen in regelmäßigen Abständen.

Die Erkenntnisse aus den Feedbacks und Bewertungen sind Grundlage für die weiteren Bearbeitungsphasen im Projekt.

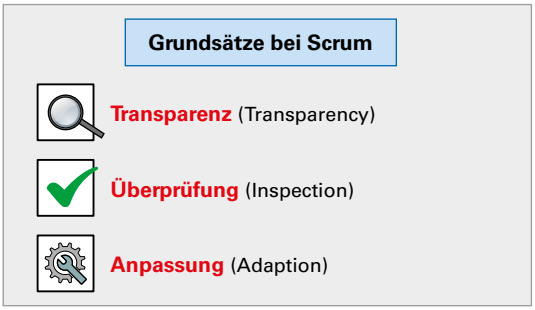

Bild 1: Grundsätze bei Scrum

Scrum-Werte

Für die erfolgreiche Arbeit mit der Scrum-Methode „leben" die Teammitglieder bei der Projektarbeit folgende fünf Scrum-Werte:
- Selbstverpflichtung (Commitment),
- Fokus (Fokus),
- Offenheit (Openness),
- Respekt (Respect),
- Mut (Courage).

Beispiele für das Handeln und Verhalten des Teams nach den fünf Scrum-Werten enthält die **Tabelle 1**.

Tabelle 1: Beispiele für Verhalten/Handeln des Scrum-Teams

Selbstverpflichtung	Verantwortung für Verpflichtungen, Verantwortung für Fehler, Entscheidung im Interesse von Unternehmen, Umwelt und öffentlicher Sicherheit, Interessenkonflikte werden allen Parteien vollständig und vorausplanend kommuniziert.
Fokus	Blick auf das vereinbarte Ziel, frühzeitige Kommunikation über Nichterreichen, qualitativ hochwertige Arbeit für den Kunden im Blick.
Offenheit	Ehrliche Kommunikation, Wahrheit akzeptieren und verstehen.
Respekt	Direkte Klärung des Sachverhalts mit der betreffenden Person, professionelles Verhalten.
Mut	Lieber Nein als falsche Versprechen, auch Scheitern bringt Fortschritt für zukünftige Aufgaben, Offenheit, wenn Hilfe nötig.

Basis für dieses Handeln und Verhalten ist die vertrauensvolle Zusammenarbeit im Team. Dazu gehört eine lösungsorientierte und wertschätzende Kommunikation.

Scrum Rollen

Um agil zu arbeiten, sollte das Team aus maximal 10 Personen bestehen. Innerhalb des Teams gibt es keine Hierarchien. Für alle projektbezogenen Aufgaben ist das Scrum-Team umsetzungsverantwortlich und steuert seine Arbeit selbst. Dazu gehört auch die Zusammenarbeit mit den Stakeholdern. Das gesamte Team ist ergebnis-verantwortlich, d. h. es muss sich mit jeder eingesetzten Zeiteinheit dem bestmöglichen Projektergebnis annähern.

Im Scrum Team sind drei spezifische Ergebnisverantwortlichkeiten **(Bild 1)** definiert:

- das Entwicklerteam (Developer),
- ein Product Owner,
- ein Scrum-Master.

Im **Entwicklerteam** arbeiten bis zu 9 Fachkräfte interdisziplinär, d. h. sie dürfen sich nicht ausschließlich auf ihren Fachbereich beschränken. Ziel muss es sein, dass sich alle gleichermaßen für die Zielerreichung verantwortlich fühlen. Somit werden typische Schuldzuweisungen der Teampartner vermieden, die als No Gos (Geht nicht) bei der Arbeit im Scrum Team gelten **(Bild 2)**.

Product Owner sind die Hauptverantwortlichen im Scrum-Team. Sie sind die Schnittstelle zwischen internen und externen Stakeholdern. Sie haben sowohl die Bedürfnisse der Kunden und Nutzer sowie den Geschäftserfolg des Unternehmens im Blick. Sie müssen die Projektziele jederzeit allen Beteiligten klar kommunizieren und agieren damit als Multiplikatoren und Vermittler. Dazu benötigen sie besondere Eigenschaften **(Bild 3)**.

Der **Scrum-Master** sorgt dafür, dass das Entwicklungsteam beste Arbeitsbedingungen hat. Er stellt die notwendigen Ressourcen zu Verfügung und ist Moderator im Projektablauf. Typische Fragestellungen für ihn sind:

- Was übersehen wir aktuell?
- Gibt es einen anderen Weg?
- Wie können wir noch besser miteinander arbeiten?
- Was müssen wir ändern, dass uns nicht gelungenes das nächste Mal nicht mehr passiert?
- Was ist zu tun, um das Scrum Team bestmöglich zu unterstützen?
- Wenn wir das nicht erledigen, was würde dann passieren?
- Mit welchem minimalen Umfang erreichen wir das bestmögliche Ergebnis?

Er ist auch eine Art „Schiedsrichter" im Scrum Prozess. So behält er z. B. im Blick, ob die Regeln oder die Prozessschritte eingehalten werden.

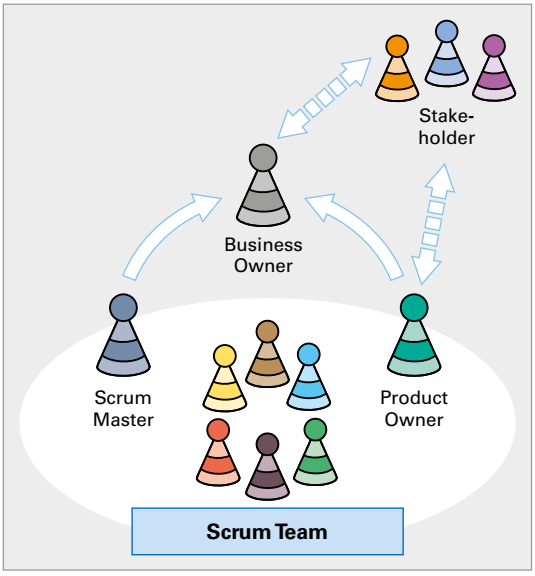

Bild 1: Rollen in Scrum

Stakeholder: alle Personengruppen, die am Projektausgang interessiert sind

Business Owner: Geschäftsinhaber

Keiner kann mir sagen, worin das Problem liegt.

Ich warte jetzt schon die ganze Zeit auf ...

Ich weiß gerade nicht, was XXX tut.

Das ist nicht meine Aufgabe, dafür ist ...

Das haben wir schon immer so gemacht.

Das hat noch nie funktioniert.

Kein Kunde wird das jemals haben wollen.

Bild 2: No Gos im Scrum Team

Bild 3: Eigenschaften Product Owner

Scrum Events

In einem Kick-off-Meeting wird mit allen Personengruppen nach **Bild 1** die Anforderungen an das Produkt aus der User-Story **(Seite 443)** auf einer Story-Card festgehalten. So soll eine gemeinsame Vision von dem zu erstellenden Produkt entwickelt werden.

Dann beginnt der Scrum-Prozess, dessen Herzschlag der **Sprint, Bild 2,** ist. Dies ist ein Ereignis von einer festen Dauer, bis maximal einem Monat. So ist sichergestellt, dass mindestens jeden Monat eine Überprüfung und Anpassung der Fortschritte in Richtung des Produktzieles erreicht wird. Kürzere Sprints können das Risiko von Kosten und Aufwand minimieren, um so neue Erkenntnisse für nachfolgende Sprints zu verwerten. Nur der Product Owner hat die Befugnis einen Sprint abzubrechen. Während des Sprints nimmt die Qualität nicht ab. Außerdem kann der Produkt Backlog, **Seite 447**, verfeinert werden. Weiterhin können bei neuen Erkenntnissen mit dem Product Owner Inhalt und Umfang der Arbeit geklärt werden.

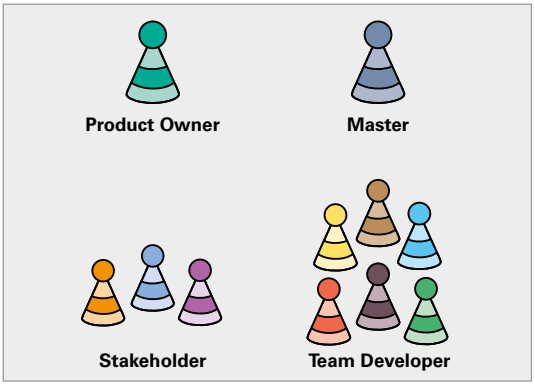

Bild 1: Rollen bei Scrum

Begriffe im Sprint
Sprint Planning: Planen der Sprintziele
Daily Scrum: tägliches Treffen von 15 Minuten
Sprint Review: Rückschau bezogen auf Produkt
Sprint Retrospektive: Rückschau bezogen auf das Team

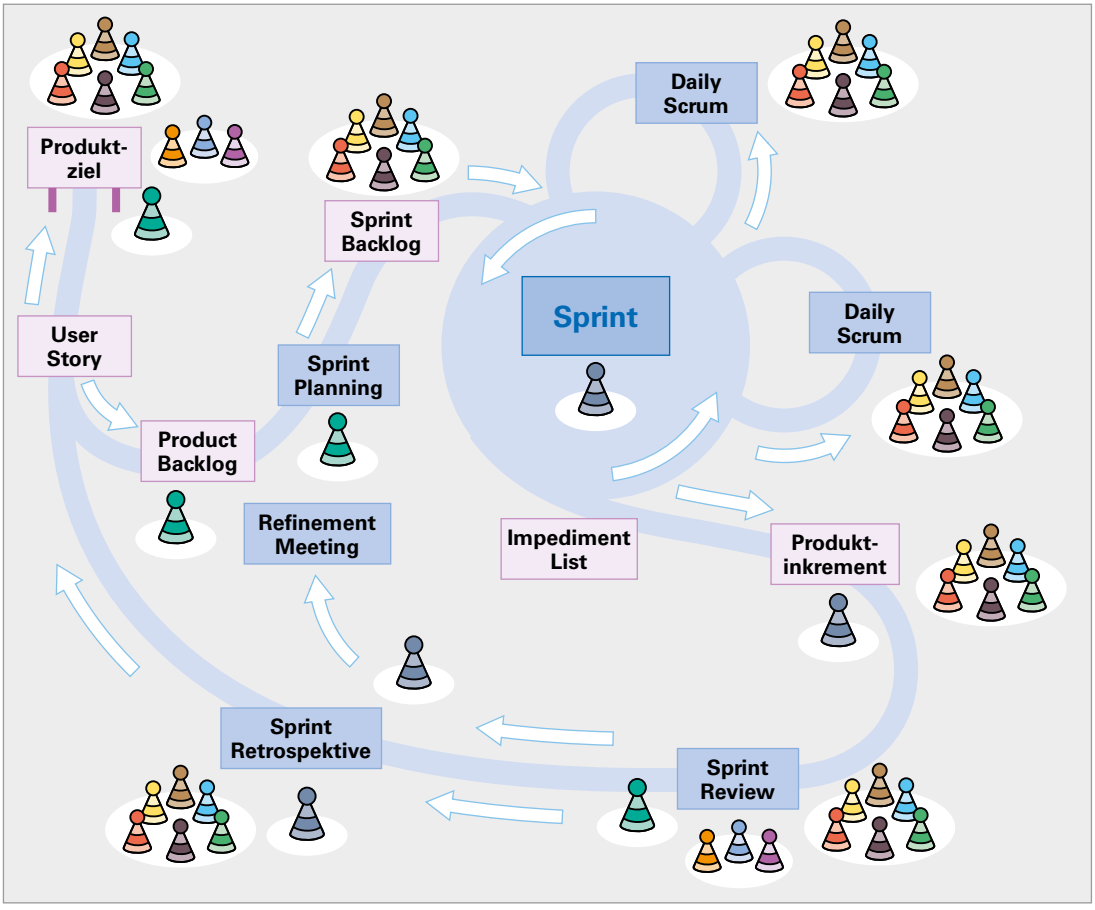

Bild 2: Scrum-Prozess

1.2.3 Kanban

Kanban entstand aus der Produktionssteuerung von Toyota in den 1950er-Jahren. David Anderson hat diese Elemente 2007 auf die Softwareentwicklung übertragen.

Bei Kanban wird die Gesamtaufgabe, die in einer vorgegeben Zeit zu erledigen ist, in einzelne Schritte zerlegt und deren Bearbeitungstand auf einem Kanban-Board **(Bild 1)** festgehalten.

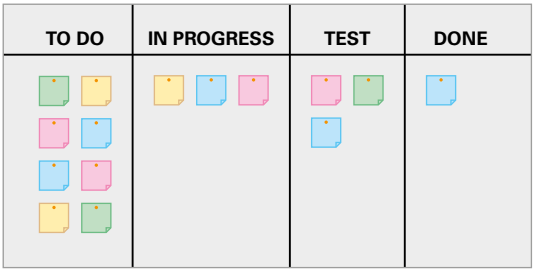

Bild 1: Kanban Board

Das Board besteht mindestens aus 3 Spalten:

- To Do (zu erledigen)
- In Progress (in Bearbeitung)
- Done (erledigt).

Häufig werden weitere Spalten vom Team eingeführt, z. B. in Prüfung, wird entworfen, wird geschrieben, wird überarbeitet, in Wartestellung, ist gesperrt.

Die Teammitglieder wählen Ihre Aufgabe vom Board aus der Spalte to Do. In Kanban gibt es keinen festen Zeitplan und keine Fälligkeitstermine. Sobald die Aufgabe erledigt ist, wird diese auf dem Board unter Done veröffentlicht. Es liegt in der gemeinsamen Verantwortung des Teams bei den Aufgaben am Board zusammenzuarbeiten und diese zum Abschluss zu bringen.

1.2.4 Extreme Programming (XP)

Die Methode wurde von Kent Beck Ende der 90ger Jahre bei der Entwicklung einer Software für die Gehaltsabrechnung des Daimler-Chrysler-Konzerns angewandt. Die einzelnen Planungs- und Feedbackschleifen **(Bild 2)** unterscheiden sich in ihren Entwicklungszeiten. So kann z. B. durch Paarprogrammierung die Qualität der Software verbessert werden. Dabei bedient der Pilot den Rechner und der Navigator denkt über den Code nach.

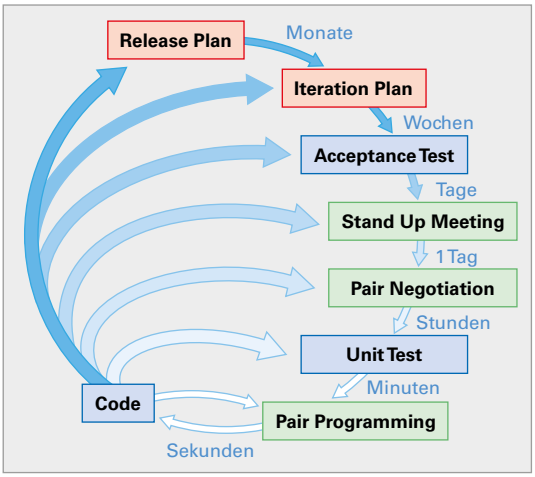

Bild 2: Planungs-Feedbackschleifen bei XP

1.2.5 Vergleich Scrum, Kanban, XP

Tabelle 1 zeigt den Vergleich der 3 agilen Methoden.

Tabelle 1: Vergleich Scrum, Kanban, XP

Scrum	Kanban	Extreme Programming
Rollen definiert: Product Owner, Scrum Master, Developer	Keine Rollen vorgegeben.	Kunde gibt Auftragsreihenfolge vor.
Sprint-Planing, Spint, Daily Scrum, Sprint Review, Sprint Retrospektive sind verbindliche Elemente.	Visualisierung der Arbeitsabläufe, Feedbackschleifen.	Kurze Entwicklungszeiten, Wartungs- und Erweiterbarkeit der Software gewährleistet.
Arbeitsrhythmus durch Sprints, max. 4 Wochen, bestimmt.	Fortlaufendes Arbeiten.	Iterationen in kurzen Abständen, z. B. 1 - 2 Wochen, unmittelbares Feedback, schnelle Reaktion auf Kundenwünsche.
Umfangreiche Schulung des Teams.	Einfache Einführung.	Fachlichkeit des Entwicklers steht im Vordergrund.
Kreativität liegt beim Team.	Projekt muss vorweg in teilbare Aufgaben zerlegt werden.	Technische Praktiken werden vorgeschrieben.
Geeignet für große Projekte.	Für große Projekte nicht geeignet.	Geeignet für große Projekte und kleine Entwicklungsteams

3 OOP mit UML umsetzen

3.1 OOP in C#

Klassen mit Attributen und Methoden

In C# werden im namespace zuerst Klassen mitsamt Attributen und Methoden **(Bild 1)** deklariert. Im Hauptprogramm Main werden anschließend Objekte der Klasse Person, z. B. person1, angelegt, denen mit dem new-Operator Speicherplatz zugewiesen wird. Da die Attribute der Klasse `public`, also „öffentlich“ sind, kann man auch außerhalb der Klasse Person im Hauptprogramm direkt durch `person1.name` und `person1.alter` auf die Attribute zugreifen. Die Klasse besitzt eine **nichtstatische Methode** `datenAnzeigen()`. Solche Methoden werden im Hauptprogramm direkt vom Objekt (person1) aufgerufen. Beim Aufruf wird eine Verweisvariable this erzeugt, die auf das Objekt zeigt, von dem aus die Methode aufgerufen wird.

Mit this kann man in der Methode auf die Attribute name und alter von person1 zugreifen.

Kapselung von Attributen

Um den schreibenden und lesenden Zugriff auf die Attribute zu kontrollieren, werden diese in der objektorientierten Programmierung als private deklariert **(Bild 2)**. Dies nennt man Kapselung. Um außerhalb der Klasse Person auf private Attribute zuzugreifen, verwendet man **set- und get-Methoden**. Diese Methoden sind nicht-statisch und werden im Hauptprogramm vom Objekt person1 aus aufgerufen.

Set-Methoden setzen das Attribut des Objekts, von dem aus sie aufgerufen werden, auf den Wert des übergebenen Parameters. Sie besitzen keinen Rückgabewert (void).

Get-Methoden geben den Inhalt des Attributs zurück und besitzen als Rückgabedatentyp den Datentyp des Attributs. Sie haben keine Parameter.

Ein Vorteil der Verwendung von set-Methoden ist z. B. die Wertebereichsüberprüfung für Attribute **(Bild 3)**. In der Methode `setAlter()` wird der als Parameter übergebene Wert neuesAlter nur dann dem Attribut alter zugewiesen, wenn er in einem sinnvollen Wertebereich liegt.

Indem man gezielt manche der set- und get-Methoden weglässt, kann man außerdem den lesenden oder schreibenden Zugriff auf bestimmte Attribute verhindern.

Kapselung regelt den lesenden und schreibenden Zugriff auf Attribute und ermöglicht Wertebereichsüberprüfungen.

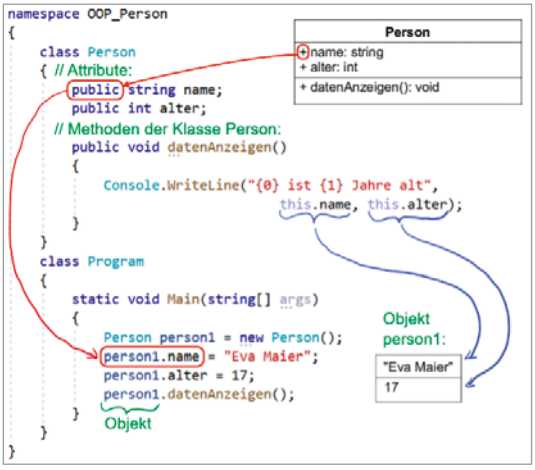

Bild 1: Klasse mit Attributen und Methoden

Bild 2: Zugriff auf private Attribute einer Klasse

```
public void setAlter(int neuesAlter)
{
    if(neuesAlter > 0 && neuesAlter <= 130)
      this.alter = neuesAlter;
    else
      Console.WriteLine("Das Alter ist ungültig!");
}
```

Bild 3: Beispiel einer set-Methode

Konstruktoren

Konstruktoren sind Methoden, die denselben Namen wie die Klasse haben und keinen Rückgabedatentyp besitzen, nicht einmal `void`.

> Ein Konstruktor wird aufgerufen, wenn ein Objekt einer Klasse erzeugt wird und dient der Speicherplatzreservierung und Vorbelegung der Attribute mit Werten.

Wird in der Klasse kein Konstruktor deklariert, so wird bei der Erzeugung eines Objekts automatisch ein Standardkonstruktor aufgerufen. Dieser weist sämtlichen Attributen Standardwerte zu: Beim Datentyp int den Wert 0, bei double 0.0 und bei Strings null.

Durch die Deklaration von Konstruktoren in einer Klasse ist eine Vorbelegung der Attribute mit individuellen Werten möglich. In einer Klasse können mehrere Konstruktoren erstellt werden, die sich nur durch Anzahl und/oder Datentyp der Parameter unterscheiden **(Bild 1)**.

Der **Standardkonstruktor** besitzt keine Parameter und setzt die Attribute auf festgelegte Werte, der **Parameter-Konstruktor** weist einem oder mehreren Attributen Werte zu, die als Parameter übergeben werden.

Im Hauptprogramm wird das Objekt `person1` mit dem Standardkonstruktor erzeugt und das Objekt `person2` mit dem Parameter-Konstruktor.

Bild 1: Klasse mit Konstruktoren

Konto
– kontoNr: int – Inhaber: string – kontostand: double
+ Konto(neueNr: int, neuerInhaber: string) + getKontoNr(): int + setInhaber(neuerInhaber: string): void + getInhaber(): string + setKontostand(neuerKontostand: double) + getKontostand(): double + einzahlen(betrag: double): void + abheben(betrag: double): bool

Bild 2: UML-Diagramm der Klasse Konto

Beispiel 1: Kontoprogramm

Erstellen Sie das Konto-Programm zum UML-Diagramm **Bild 2**.
Deklarieren Sie im Hauptprogramm ein Objekt meinKonto der Klasse Konto und testen Sie sämtliche Methoden.

Lösung: **Bild 3**

Die set-Methode für die Kontonummer und damit der schreibende Zugriff auf die Kontonummer wurde bewusst weggelassen, da die Kontonummer nur einmalig beim Erstellen des Objektes mit dem Parameter-Konstruktor zugewiesen und anschließend nicht mehr verändert werden soll.

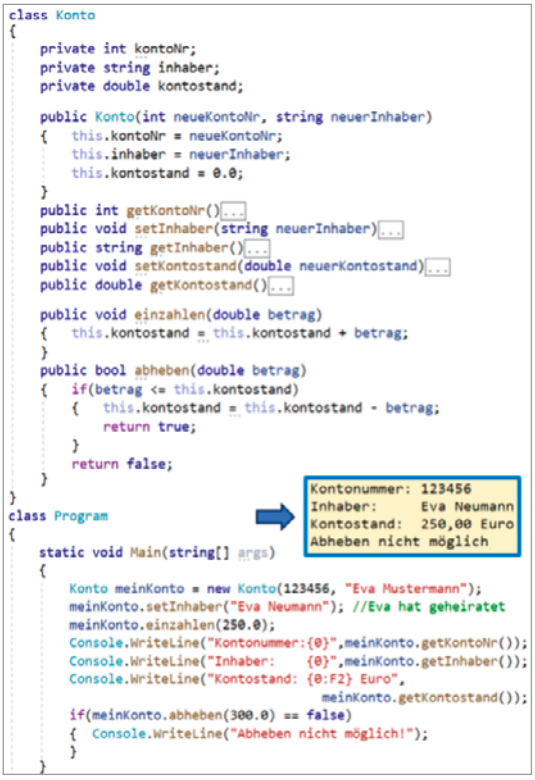

Bild 3: Kontoprogramm

Vererbung

Gemeinsame Attribute und Methoden können in einer Oberklasse definiert werden und an die Unterklassen vererbt werden.

Verwaltet eine Firma z. B. Daten von Kunden und Mitarbeitern **(Bild 1)**, werden die gemeinsamen Eigenschaften und Methoden dieser Personen in einer Oberklasse festgelegt **(Bild 2)**. Das Attribut name wird als `protected` deklariert, d. h. in den Unterklassen, die von der Klasse `Person` erben, kann man direkt darauf zugreifen, in allen anderen Klassen mit set-und get-Methoden.

Die Unterklassen erben die Eigenschaften und Methoden von der Oberklasse. Das wird im C#-Quelltext durch einen Doppelpunkt angegeben **(Bild 3)**. Die Unterklasse `Kunde` erbt von der Klasse `Person` das Attribut name und die Methode `getName()`.

Der Standardkonstruktor `Kunde()` der Unterklasse ruft mit `base()` als erstes den Standardkonstruktor der Oberklasse `Person` auf und setzt dabei `name="--leer--"`. Anschließend wird `kundenNr=0` zugewiesen.

Der Parameter-Konstruktor `Kunde(string name, int nr)` reicht mit `base(name)` den Namen an den Parameter-Konstruktor der Oberklasse weiter und setzt dabei den Namen des damit erzeugten Kundenobjektes auf den übergebenen Wert. Anschließend wird die Kundennummer zugewiesen.

Beispiel 1: Unterklasse mit Vererbung

Erstellen Sie für das UML-Diagramm Bild 1 eine weitere Unterklasse `Mitarbeiter`, die von `Person` erbt.

Lösung: **Bild 4**

Im Hauptprogramm werden die Objekte `einePerson`, `einKunde` und `einMitarbeiter` erzeugt und durch den Aufruf der Parameter-Konstruktoren direkt mit Startwerten initialisiert **(Bild 5)**. Die geerbte Methode `getName()` kann von Objekten jeder der Klassen aufgerufen werden.

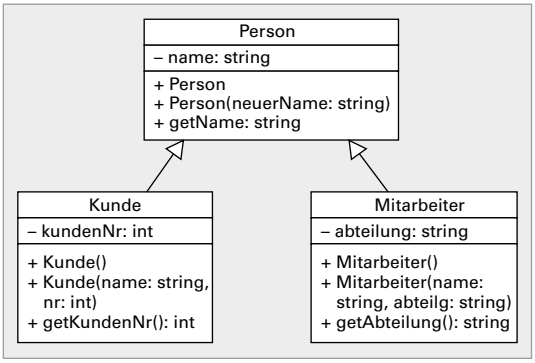

Bild 1: Vererbung

```
class Person                          //Basisklasse
{
    protected string name;

    public Person()                   //Standardkonstruktor
    {  this.name = "--leer--"; }
    public Person(string neuerName) //Parameter-Konstruktor
    {  this.name = neuerName; }
    public string getName()
    {  return this.name; }
}
```

Bild 2: Die Oberklasse „Person“

```
class Kunde : Person          //Kunde erbt von Person
{
    private int kundenNr;

    public Kunde() : base()
    {  this.kundenNr = 0; }
    public Kunde(string name, int nr) : base(name)
    {  this.kundenNr = nr;  }
    public int getKundenNr()
    {  return this.kundenNr;  }
}
```

Bild 3: Die Unterklasse „Kunde“ erbt von „Person“

```
class Mitarbeiter : Person    //Mitarbeiter erbt von Person
{
    private string abteilung;

    public Mitarbeiter() : base()
    {  this.abteilung = "--keine Angabe--";
    }
    public Mitarbeiter(string n, string abteilg) : base(n)
    {  this.abteilung = abteilg;
    }
    public string getAbteilung()
    {  return this.abteilung;
    }
}
```

Bild 4: Die Unterklasse „Mitarbeiter“

```
static void Main(string[] args)
{
    Person einePerson = new Person("Winter");
    Kunde einKunde = new Kunde("Müller", 123);
    Mitarbeiter einMitarbeiter =
                        new Mitarbeiter("Schmidt","Verkauf");
    Console.WriteLine("Person: {0}", einePerson.getName());
    Console.WriteLine("Kunde: {0}  Nr:{1}",
                einKunde.getName(), einKunde.getKundenNr());
    Console.WriteLine("Mitarbeiter: {0}  Abteilung:{1}",
    einMitarbeiter.getName(), einMitarbeiter.getAbteilung());
}
```

```
Person: Winter
Kunde: Müller  Nr:123
Mitarbeiter: Schmidt  Abteilung:Verkauf
```

Bild 5: Hauptprogramm

Polymorphie

Polymorphie (Vielgestaltigkeit) tritt in Zusammenhang mit Vererbung auf.

> Eine Methode ist polymorph, wenn sie mit gleichem Namen und gleichen Parametern in mehreren Klassen, die z. B. voneinander erben, implementiert ist.

Die polymorphe Methode muss in der Oberklasse mit **virtual** gekennzeichnet werden und in den Unterklassen mit **override**, um beim Methodenaufruf den Inhalt der Methode der Oberklasse zu überschreiben.

Beim Methodenaufruf wird abhängig vom Objekttyp erst zur Laufzeit bestimmt, welche der Methoden verwendet wird.

Übung 1: Erstellen einer polymorphen Methode

Die Klassen `Person`, `Kunde` und `Mitarbeiter` sollen um eine polymorphe Methode `datenanzeigen()` erweitert werden **(Bild 1)**, die jeweils die spezifischen Daten der unterschiedlichen Objekte ausgibt. Einzelne Ausgaben der Attribute im Hauptprogramm und die dafür benötigten get-Methoden sind nun nicht mehr nötig.

Lösung: **Bild 2**

Die Methode `datenanzeigen()` wird in der Oberklasse Person mit `virtual` gekennzeichnet ①, in den Unterklassen mit `override` (② und ③). Im Hauptprogramm werden durch Aufruf der Parameter-Konstruktoren Objekte der drei Klassen erzeugt. Die Bildschirmausgabe durch Aufruf der polymorphen Methode `datenanzeigen()` ist in Bild 2 zu sehen.

Übung 2: Aufruf einer polymorphen Methode von einem Array von Personen

Verändern Sie das Hauptprogramm von Übung 1, indem sie durch Aufruf des Parameter-Konstruktors jeweils ein Objekt der Klassen `Person`, `Kunde` und `Mitarbeiter` erstellen. Erzeugen Sie anschließend ein Array von drei Personen und weisen Sie diesem die Objekte zu.

Rufen Sie in einer Schleife die polymorphe Methode `datenanzeigen()` von den Arrayelementen aus auf.

Lösung: **Bild 3**

Erstellt man ein Array der Oberklasse Person, das Objekte der drei unterschiedlichen Klassen enthält, entscheidet sich beim Aufruf der polymorphen Methode datenanzeigen() erst zur Laufzeit, welche Version der Methode ausgeführt wird.

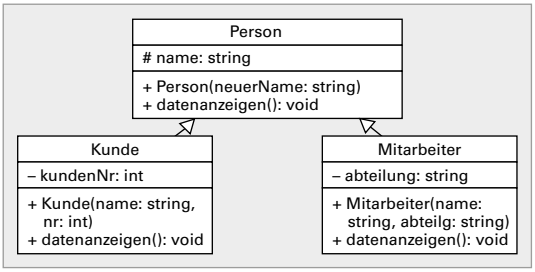

Bild 1: Polymorphie bei der Vererbung

```
class Person
{
    protected string name;
    public Person(string neuerName)...
    public virtual void datenanzeigen()          ①
    {
      Console.WriteLine("Person:{0}",this.name);
    }
}
class Kunde : Person
{
    private int kundenNr;
    public Kunde(string name, int nr) ...
    public override void datenanzeigen()         ②
    {
      Console.WriteLine("Kunde:{0} Kundennr:{1}",
                        this.name,this.kundenNr);
    }
}
class Mitarbeiter : Person
{
    private string abteilg;
    public Mitarbeiter(string n, string abteilung) ...
    public override void datenanzeigen()         ③
    {
      Console.WriteLine("Mitarbeiter:{0} Abteilg:{1}",
                        this.name,this.abteilg);
    }
}
class Program
{
    static void Main(string[] args)
    {
        Person einePerson = new Person("Winter");
        Person einKunde = new Kunde("Müller", 123);
        Person einMitarbeiter =
                new Mitarbeiter("Schmidt", "Verkauf");
      ①einePerson.datenanzeigen();
      ②einKunde.datenanzeigen();
      ③einMitarbeiter.datenanzeigen();
    }
}
```

```
Person: Winter
Kunde: Müller  Kundennr: 123
Mitarbeiter: Schmidt  Abteilg: Verkauf
```

Bild 2: Beispiel einer polymorphen Methode

```
class Program
{
    static void Main(string[] args)
    {
        Person einePerson = new Person("Winter");
        Kunde einKunde = new Kunde("Müller", 123);
        Mitarbeiter einMitarbeiter =
                        new Mitarbeiter("Schmidt", "Verkauf");
        Person[] personen =
            new Person[3] {einePerson,einKunde,einMitarbeiter};
        for (int i = 0; i < personen.Length; i++)
        {   personen[i].datenanzeigen();  }
    }
}
```

Bild 3: Array einer Klasse mit polymorpher Methode

3.2 OOP in C++

Eine UML-Klasse kann durch eine C++-Klasse realisiert werden. Am besten legt man für jede Klasse eine separate Datei an, damit unterschiedliche Programme dieselbe Klasse verwenden können.

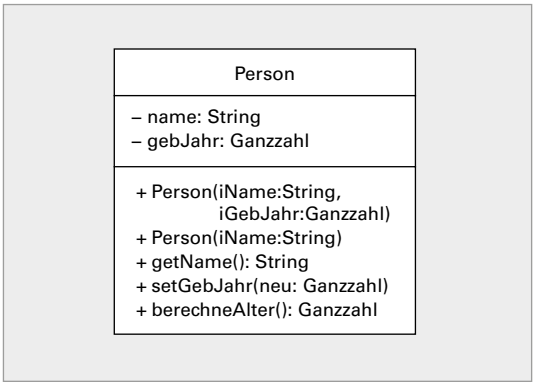

Bild 1: Klasse Person in UML

Beispiel 1: UML-Klasse in C++ umsetzen

Setzen Sie die UML-Klasse Person **(Bild 1)** in C++ um.

Lösung: **Bild 2**

```
#include <string>
#include <ctime>
using namespace std;
class Person{
private:
    string name;
    int gebJahr;
public:
    Person(string iName, int iGebJahr){
        name = iName;
        gebJahr = iGebJahr; }
    //Überladener Konstruktor
    Person(string iName){
        name = iName;}
    string getName() {
       return name; }
    void setGebJahr(int neu) {
       gebJahr = neu; }
    int berechneAlter() {
       if (gebJahr < 1920) {
          return 0; }
       else {
          // Aus der C++-Bibliothek
          time_t t = time(0);
          struct tm* now = localtime(&t);
          int aktJahr=now->tm_year+1900;
          return aktJahr - gebJahr; }
    }
};
```

Bild 2: Klasse Person in C++

Die Methoden haben direkten Zugriff auf die Attribute, z. B. liest getName() den Namen der Person aus und gibt ihn zurück.

Zur Bestimmung des aktuellen Jahrs verwendet die Methode berechneAlter() Funktionen aus der C++-Bibliothek. Bei einem ungültigem Geburtsjahr (vor 1920) wird 0 als Alter zurückgegeben.

Beispiel 2: Objekte erzeugen

Erstellen Sie ein Testprogramm, in dem folgende Objekte erzeugt werden.

person1
name = "Joe" gebJahr = 2006

person2
name = "Zoe" gebJahr = 2005

Lösung: **Bild 3**

```
#include <iostream>
#include "Person.cpp"
using namespace std;
int main(){
  Person person1("Joe", 2006);
  cout <<"Name: "<< person1.getName();
  cout <<" Alter: "<< person1.berechneAlter();
  //Objekt mit dem anderen Konstruktor erzeugen
  Person person2("Zoe");
  cout <<endl<<"Name: "<< person2.getName();
  cout <<" Alter: "<< person2.berechneAlter();
  person2.setGebJahr(2005);
  cout <<" Alter: "<< person2.berechneAlter();
  return 0;
}
```

```
Name: Joe Alter: 18
Name: Zoe Alter: 0 Alter: 19
```

Bild 3: Programm mit Objekterzeugung

Zur Verwendung der Klasse Person muss eine include-Anweisung angegeben werden.

Für person1 wird der erste Konstruktor mit allen Attributwerten als Parameter aufgerufen, für person2 der zweite Konstruktor nur mit dem Namen. Das Geburtsjahr von person2 hat zuerst keinen gültigen Wert, es wird später durch Aufruf der set-Methode gesetzt.

Da das Attribut private ist, kann man das Geburtsjahr nicht direkt zuweisen:

~~person2.gebJahr = 2005;~~

sondern benötigt dafür die set-Methode:

person2.setGebJahr(2005);

Klassenattribute und Klassenmethoden

Klassenattribute (statische Attribute) und Klassenmethoden (statische Methoden) beziehen sich auf die Klasse als Ganzes, nicht auf einzelne Objekte. In UML werden sie unterstrichen und in C++ mit dem Schlüsselwort static versehen.

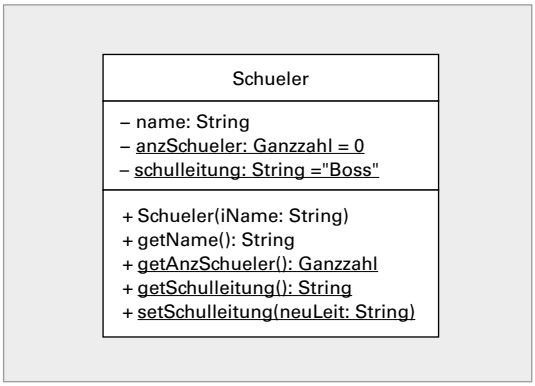

Bild 1: UML-Klasse Schueler

Beispiel 1: Klassenattribute und Klassenmethoden verwenden

Setzen Sie die UML-Klasse Schueler **(Bild 1)** in C++ um und verwenden Sie Klassenmethoden in einem Testprogramm.

Lösung: **Bild 2**, **Bild 3**

Die Klassenattribute anzSchueler und schulleitung müssen außerhalb der Klasse Schueler initialisiert werden (Bild 3).

Bei jeder Erzeugung eines Objekts wird die aktuelle Anzahl Schüler im Konstruktor automatisch um 1 erhöht.

Klassenmethoden können auf Klassenattribute zugreifen. Sie werden mit dem Klassennamen anstatt einem Objektnamen aufgerufen, z. B.: „Schueler::“ statt „schueler1“ vor der Methode.

Vererbung

Beispiel 2: Vererbung umsetzen

Erklären Sie am Beispiel Person/Lehrkraft **(Bild 4)**, wie die Vererbung in C++ umgesetzt wird.

Lösung:
public class Lehrkraft: **public Person** {
  ...
}

Aufgrund der Vererbung verfügt die Klasse Lehrkraft über

name, gebJahr, getName()

und

ausgebenDaten().

Der Konstruktor wird nicht vererbt, er muss in Lehrkraft neu geschrieben werden.

```
#include <string>
#include <ctime>
using namespace std;
class Schueler{
private:
     string name;
     static int anzSchueler;
     static string schulleitung;
public:
     Schueler(string iName){
          name = iName;
          anzSchueler++;}
     string getName() {
        return name;}
     static int getAnzSchueler(){
        return anzSchueler;}
     static string getSchulleitung(){
        return schulleitung;}
     static void setSchulleitung
                      (string neuLeit) {
        schulleitung = neuLeit;}
};
```

Bild 2: Klasse mit Klassenattributen und -methoden

```
#include <iostream>
#include "Schueler.cpp"
using namespace std;
// Initialisierung der statischen
// Variablen aus Schueler
int Schueler::anzSchueler = 0;
string Schueler::schulleitung = "Boss";
int main(){
     Schueler schueler1("Ada");
     Schueler schueler2("Flo");
     cout << Schueler::getAnzSchueler()
          << endl;
     cout << Schueler::getSchulleitung()
          << endl;
     Schueler::setSchulleitung("Neue Leitung");
     cout << Schueler::getSchulleitung();
     return 0;}
```

```
2
Boss
Neue Leitung
```

Bild 3: Programm mit Verwendung von Klassenmethoden

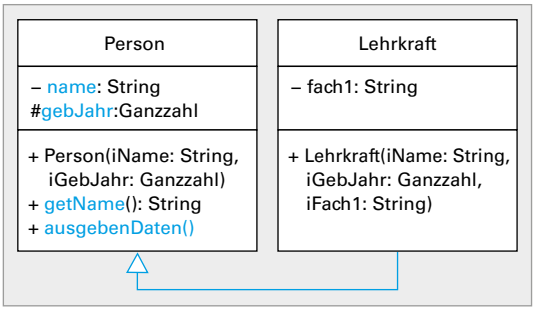

Bild 4: Klassenhierarchie mit Vererbung

Beispiel 1: Konstruktor einer Unterklasse schreiben

Programmieren Sie den Konstruktor der Klasse Lehrkraft.

Lösung: **Bild 1**

```
Lehrkraft(string iName,int iGebJahr,
  string iFach1):
  Person(iName, iGebJahr){
    fach1 = iFach1;
}
```

Bild 1: Konstruktor einer Unterklasse

Im Konstruktor der Klasse Lehrkraft muss zuerst der Konstruktor der Oberklasse Person aufgerufen werden. Dies geschieht in Form einer Initialisierungsliste zwischen dem Doppelpunkt und der ersten geschweiften Klammer. Danach wird das für Lehrkräfte spezielle Attribut fach1 gesetzt.

Überschreiben

Die Methode ausgebenDaten() der Klasse Person gibt den Namen und das Geburtsjahr aus. Für die Lehrkraft soll zusätzlich das Fach ausgegeben werden. Die geerbte Methode ausgebenDaten() muss in Lehrkraft überschrieben werden, sie erscheint im UML-Diagramm der Klasse Lehrkraft **(Bild 2)**.

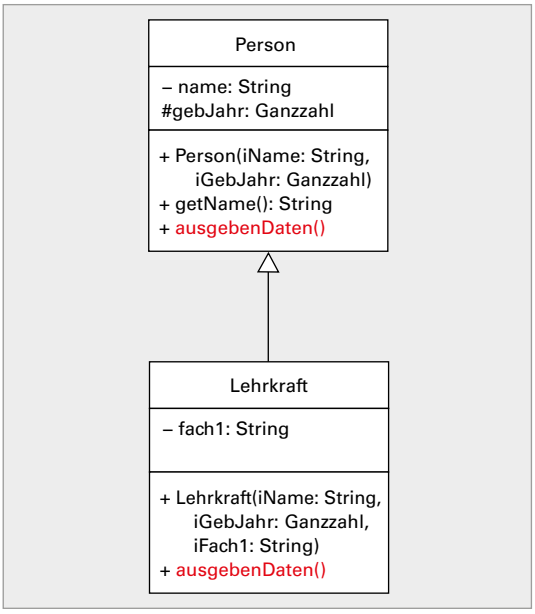

Bild 2: Klassenhierarchie mit überschriebener Methode

Beispiel 2: Methode überschreiben

Programmieren Sie die Methoden ausgebenDaten() a) der Klasse Person und b) der Klasse Lehrkraft.

Lösung: **Bild 3**

Eine Methode, die überschrieben wird, muss als *virtual* in der Oberklasse gekennzeichnet werden, in der Unterklasse muss *override* (= überschreiben) vor { angegeben werden.

a) In der virtuellen Methode ausgebenDaten() der Oberklasse Person werden die Werte der Attribute name und gebJahr ausgegeben.

b1) In ausgebenDaten() der Klasse Lehrkraft werden die geerbten personenbezogenen Attribute ausgegeben und dann das Fach. Das geerbte Attribut gebJahr ist protected, es kann direkt angesprochen werden. Das Attribut name ist privat, es kann nicht direkt ausgelesen werden. Sein Wert muss durch Aufruf der getName()-Methode ermittelt werden.

b2) In ausgebenDaten() der Klasse Lehrkraft kann als alternative Lösung zu b1 die Methode

ausgebenDaten()

aus der Oberklasse Person aufgerufen werden. Dadurch werden zuerst die Anweisungen aus a) ausgeführt und dann das Fach ausgegeben. Im Gegensatz zu b1) ist diese Lösung unabhängig von der Sichtbarkeit. Außerdem bleibt das Erscheinungsbild einheitlich: Änderungen bei ausgebenDaten() der Oberklasse werden auch bei der Ausgabe für die Lehrkraft wirksam.

a) In der Klasse Person:

```
virtual void ausgebenDaten() {
   cout << endl << name << " ";
   cout << gebJahr;
}
```

b1) In der Klasse Lehrkraft:

```
void ausgebenDaten() override {
    // name ist private
    cout << endl << getName();
    // gebJahr ist protected
    cout << " " << gebJahr;
    cout << " " << fach1;
}
```

b2) In der Klasse Lehrkraft:

```
void ausgebenDaten() override {
    Person::ausgebenDaten();
    cout << " " << fach1;
}
```

Bild 3: Realisierung der Methode ausgebenDaten()

Abstrakte Klasse und abstrakte Methode

Beispiel 1: Gegebenes Programm verstehen

Untersuchen Sie den gegebenen Code auf Besonderheiten **(Bild 1)**.

Lösung: In Person ist holePersonTyp() rein virtuell, d. h. abstrakt: sie endet mit = 0 und hat keine geschweiften Klammern. Dadurch ist die Klasse Person automatisch abstrakt. holePersonTyp() muss in der Unterklasse Lehrkraft überschrieben werden (override).

Eine Klasse ohne abstrakte Methoden kann nicht explizit als abstrakt definiert werden.

Von der abstrakten Klasse Person können keine Objekte erstellt werden, nur von Lehrkraft:

~~Person person("Joe", 2006)~~

Lehrkraft lehrer("Schlau", 1980, "IT")

Die Klasse Person hat trotzdem einen Konstruktor. Dieser wird im Konstruktor der Unterklasse

Lehrkraft

aufgerufen, um Speicherplatzreservierung und Initialisierungen der personenbezogenen Daten durchzuführen.

```cpp
#include <iostream>
using namespace std;
class Person{
protected:
    string name;
    int gebJahr;
public:
    Person(string iName, int iGebJahr){
        name = iName;
        gebJahr = iGebJahr; }
    virtual string holePersonTyp()=0;
};
```

```cpp
#include "Person.cpp"
class Lehrkraft : public Person {
private: string fach1;
public:
    Lehrkraft(string iName,int iGebJahr,
      string iFach1):
      Person(iName, iGebJahr){
        fach1 = iFach1;}
    string holePersonTyp() override {
        return "Lehrkraft";}
};
```

```cpp
#include <iostream>
#include "Lehrkraft.cpp"
using namespace std;
int main(){
   Lehrkraft lehrer("Schlau", 1980, "IT");
   cout << lehrer.holePersonTyp();
   return 0; }
```

Lehrkraft

Bild 1: Klassen mit Besonderheiten

Interfaces (= Schnittstellen)

Die Klasse Lehrkraft erbt von zwei Interfaces **(Bild 2)**, muss kannFachUnterrichten(), eintragenFehlend() und austragenFehlend() implementieren **(Bild 3)**. In C++ gibt es keine Interfaces, sie werden mithilfe von abstrakten Klassen realisiert. Dies ist möglich, da C++ Mehrfachvererbung unterstützt **(Bild 4)**.

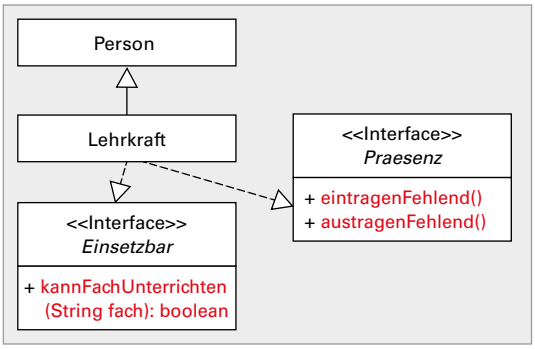

Bild 2: Vererbung bei Schnittstellen

```cpp
#include "Person.cpp"
#include "Einsetzbar.cpp"
#include "Praesenz.cpp"
class Lehrkraft : public Person,
   public Einsetzbar, public Praesenz {
private:
    string fach1;
    bool fehlend;
public:
    // Konstruktor fehlt hier
    void eintragenFehlend() {
       fehlend = true;}
    void austragenFehlend() {
       fehlend = false;}
    bool kannFachUnterrichten(string fach)
    {
      if (fach == fach1 && !fehlend)
           return true;
      else return false;
    } };
```

Bild 3: Klasse mit Interface-Implementierung

```cpp
class Praesenz{
public:
   virtual void eintragenFehlend()=0;
   virtual void austragenFehlend()=0;
};
```

Bild 4: Beispiel einer abstrakten Klasse als Interface

Assoziationen

Bei der Umsetzung einer gerichteten Assoziation muss nur die Ausgangsklasse angepasst werden.

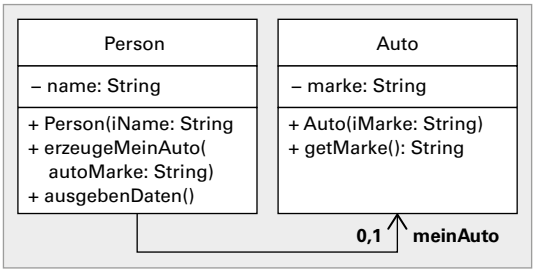

Bild 1: Beispiel einer gerichteten Assoziation

Beispiel 1: Assoziation umsetzen

Setzen Sie die Klasse Person aus **Bild 1** in C++ um und schreiben Sie ein Testprogramm.

Lösung: **Bild 2, Bild 3**

In der Klasse Person wird ein zusätzliches Attribut meinAuto als Zeiger auf ein Objekt vom Typ Auto eingefügt. Dieses Attribut wird in UML normalerweise nicht als Attribut angegeben, da es die Umsetzung der Assoziation darstellt.

Wenn meinAuto private ist, sind Methoden notwendig, mit denen auf meinAuto zugegriffen werden kann, z. B. die set-Methode. Die set-Methode wird eher nicht im UML-Diagramm angegeben.

Bei setMeinAuto() wird ein Auto außerhalb der Klasse Person erstellt und ein Zeiger auf dieses Auto übergeben. Eine andere Möglichkeit ist der Aufruf von erzeugeMeinAuto() mit dem Markennamen als Parameter: innerhalb von Person wird ein Auto durch Aufruf des Konstruktors mit new erzeugt, es liefert einen Zeiger zurück, der in meinAuto gespeichert wird. Falls eine Person davor schon ein Auto hatte, muss es explizit gelöscht werden, der Speicherplatz wird sonst nicht automatisch freigegeben.

Bei einer Person ohne Auto hat meinAuto die Kennzeichnung *nullptr.* In ausgebenDaten() muss überprüft werden, ob meinAuto leer ist *(nullptr)*, in diesem Fall ist der Zugriff auf die Marke nicht zulässig: es würde das Programm zum Absturz bringen.

Nach meinAuto ist ein Pfeil notwendig, kein Punkt, weil meinAuto ein Zeiger ist.

~~meinAuto.getMarke()~~

meinAuto->getMarke().

```
#include <string>
#include <iostream>
#include "Auto.cpp"
using namespace std;
class Person{
private:
  string name;
  Auto* meinAuto;//Zeiger aufs Auto
public:
  Person(string iName){
    name = iName;
    meinAuto = nullptr;
    }
  ~Person(){
    delete meinAuto;
  }
  void setMeinAuto(Auto* zeigerAuto){
    if (meinAuto != nullptr){
       delete meinAuto;
    }
    meinAuto = zeigerAuto;}
  void erzeugeMeinAuto(string autoMarke){
    if (meinAuto != nullptr){
       delete meinAuto;
     }
     meinAuto = new Auto(autoMarke);
     }
  void ausgebenDaten() {
    if (meinAuto != nullptr) {
      cout << name << " hat einen "
       << meinAuto->getMarke() << endl;}
      else { cout << name <<" hat kein Auto";
        cout << endl;}}
};
```

Bild 2: Umsetzung der Assoziation meinAuto

```
// Auszug aus main()
Person person1("Maxi");
person1.ausgebenDaten();
// Auto erzeugen und zuordnen
Auto* neuesAuto = new Auto("Renault");
person1.setMeinAuto(neuesAuto);
person1.ausgebenDaten();
// Auto wird in Person erzeugt
person1.erzeugeMeinAuto("Mercedes");
person1.ausgebenDaten();
```

```
Maxi hat kein Auto
Maxi hat einen Renault
Maxi hat einen Mercedes
```

Bild 3: Beispiel einer Person mit Auto

In Bild 3 wird eine Person angelegt und bekommt ein Auto zugewiesen, indem das Auto zuerst erzeugt und der Methode setMeinAuto() übergeben wird.

Danach wird ein neues Auto (Mercedes) durch Übergabe der Marke an erzeugeMeinAuto() **(Bild 3)** erzeugt und zugewiesen.

4.6.5 NoSQL-Datenbanken

NoSQL-Datenbanken sind nicht-relationale Datenbanken, die große Mengen unstrukturierter Daten flexibel speichern können. Sie eignen sich für die Verarbeitung von grafischen, hierarchischen oder sich stark verändernden Daten. Es lassen sich vier Typen von NoSQL-Datenbanken unterscheiden:

1. **Dokument-Datenbanken**, auch **DocumentStore-DB**, deren Daten in Form von schemalosen JSON-Strukturen oder Dokumenten vorliegen, die Ganzzahlen, Zeichenketten oder freien Text enthalten **(Bild 1)**. Dieses Modell eignet sich für Kataloge, Benutzerprofile und Management-Systeme. Ein Beispiel ist **MongoDB** (von hu**mongo**us = gigantisch), ein dokumentorientiertes Datenbanksystem mit JSON-ähnlichen Strukturen und einer Javascript-Schnittstelle. Dabei werden schemafreie Dokumente in einer Collection zusammengefasst, die ihrerseits in einer Datenbank liegen. Eine Collection ist eine Sammlung von Dokumenten, deren Daten jedoch keine gemeinsame Struktur aufweisen müssen.

2. **Schlüssel-Werte-Datenbanken**, auch **Key-Value-Based DB**, speichern die Daten unter einem Schlüssel ab, über den man auf die Daten zugreifen kann **(Bild 2)**. Die Daten können als Freiformwerte einfache Ganzzahlen, Zeichenketten oder komplexe JSON-Strukturen enthalten. Ein bekanntes Datenbanksystem dieser Art ist **Redis**.

3. **Spaltenorientierte Datenbanken**, auch **Wide-Column-Based DB**, ermöglichen eine effiziente Speicherung von hochdimensionalen Daten und Datensätzen mit unterschiedlichen Attributen in Spalten **(Bild 3)**.

> Ein **hochdimensionierter Datensatz** liegt vor, wenn die Anzahl der Merkmale größer ist als die Anzahl der Beobachtungen.

   Verwendung findet dieses System z. B. bei der „Big Table“ bei Google-Anwendungen. Ein spaltenorientiertes Datenbanksystem ist **Cassandra**, das eine sehr hohe Skalierbarkeit und Ausfallsicherheit besitzt.

4. **Graph-Datenbanken**, die Daten als Netzwerk oder Graph von Entitäten und deren Beziehungen speichern **(Bild 4)**. Die Dokumente enthalten direkte Verknüpfungen zwischen den Objekten, um so ein schnelles Durchsuchen stark verbundener Datensätze zu ermöglichen. Eingesetzt werden Graph-Datenbanken bei der Routenplanung von Navigationssystemen oder für das Beziehungsgeflecht in Facebook. Ein System dieses Datenbank-Typs ist **Neo4j**.

```
{
  "Name": "Max Mayer",
  "Adresse"{
              "Strasse": "Hauptstraße",
              "Hausnr": 16,
              "PLZ": "70180",
              "Ort": "Stuttgart"
            }
  "Noten": [ 2.1, 1.7, 3.4, 1.5, 2.3 ]
}
{
  "Name": "Eva Schmidt",
  "Adresse"{
              "Strasse": "Waldweg",
              "Hausnr": 3,
              "PLZ": "10315",
              "Ort": "Berlin"
            }
  "Noten": [ 1.8, 2.9, 3.3, 1.4, 2.1 ]
}
```

Bild 1: Beispiel einer Dokument-DB

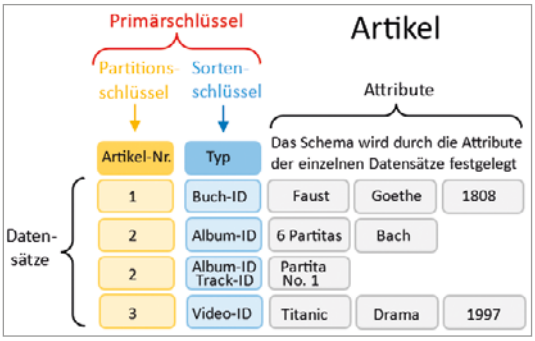

Bild 2: Beispiel einer Key-Value-Based DB

```
Row Key: "User_1"
 Column "Name"      - Wert: "Maximilian"
 Column "Alter"     - Wert: "19"
 Column "Nickname"  - Wert: "Max"
Row Key: "User_2"
 Column "Name"      - Wert: "Eva-Maria"
 Column "Alter"     - Wert: "18"
 Column "Nickname"  - Wert: "Evi"
```

Bild 3: Beispiel einer Wide-Column-Based DB

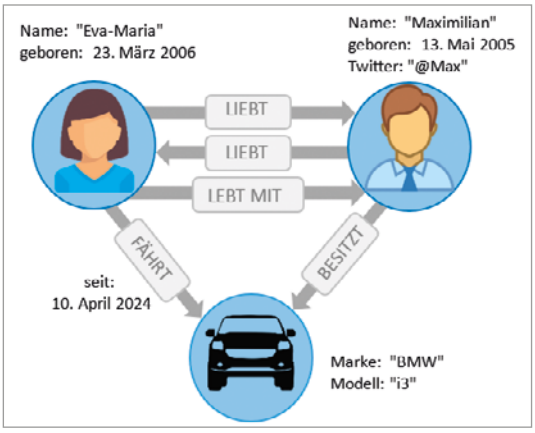

Bild 4: Beispiel einer Graph-DB