PR2 – Formular für Lesenotizen
SS2021

Nachname	Vorname	Matrikelnummer	Abgabedatum:
Lushaj	Detijon	1630149	29.03.21

L.2.5 astah UML

Ziele und Bedeutung

- Sprache zur Spezifikation (Beschreibung) von Softwaresystemen
- UML ist **die** Modellierungssprache für objektorientierte Softwareentwicklung

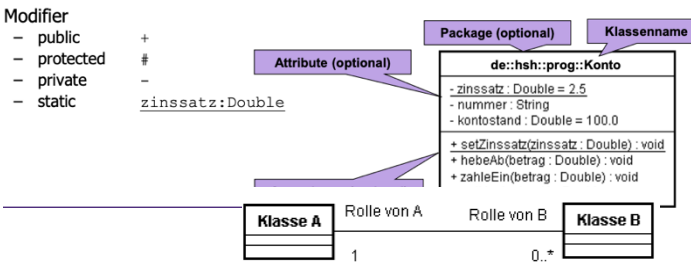
Klassendiagramm: Grafische Darstellung der statischen Struktur eines Softwaresystems

- Enthält Klassen, Attribute, Methoden und Beziehungen

Notation von Attributen:
attributName: Typ [= Initialwert]

- Der Typ steht hier also hinter dem Attributnamen

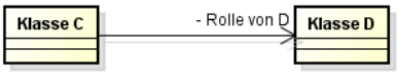
Notation von Methoden:
methodenName(Parameterliste):Ergebnistyp



Beziehungen

Beziehungen in UML

– Objekte der Klasse A haben eine Beziehung zu Objekten der Klasse B und umgekehrt.
– **Rolle**: (zwei) Namen für eine Beziehung
– **Multiplizität**: gibt an, mit wie vielen Objekten man in Beziehung steht (sog. Kardinalität)
• Notation: einfache Zahl (0,1,*) oder Intervall der Form 0..*, 1..5
• Das * steht dabei für eine beliebige Zahl

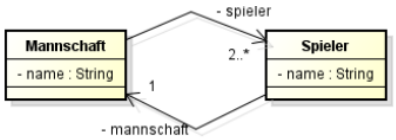


Navigierbarkeit

Beziehungen können in einer Richtung gerichtet sein Darstellung durch einfachen Pfeil (→)
• Die Beziehung wird nur aus einer Richtung benutzt:
– Implementierung: Quelle hat Referenz auf Ziel, aber nicht umgekehrt

Einwertige / Mehrwertige Beziehungen

Ein Objekt kann mit einem oder mehreren Objekten einer anderen Klasse in Beziehung stehen.

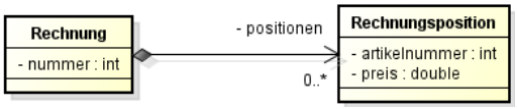


Beziehungsstärke

Assoziation - relativ lose Kopplung, dauerhaft oder temporär - “kennt”
Einsetzen, wenn ein Attribut Objekte einer anderen Klasse referenziert

Abhängigkeit -“nutzt vorübergehend” - Schwächste Form der Beziehung
• Temporäre Abhängigkeit **(z. B. nur für die Dauer eines Methodenaufrufes)** notiert man in UML mit der gestrichelten Linie.

Komposition - stärkste Form der Beziehung - “besteht aus“
• Enthalten sein via Wert (containment by value):
• Notation in UML:
– Ausgefüllte Raute auf der Seite des Ganzen
– Multiplizität auf der Seite des Ganzen ist immer 1 und wird daher häufig weggelassen

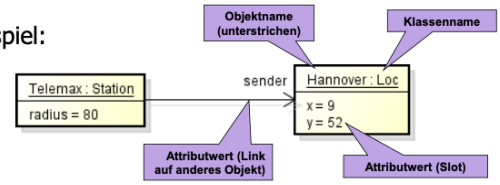


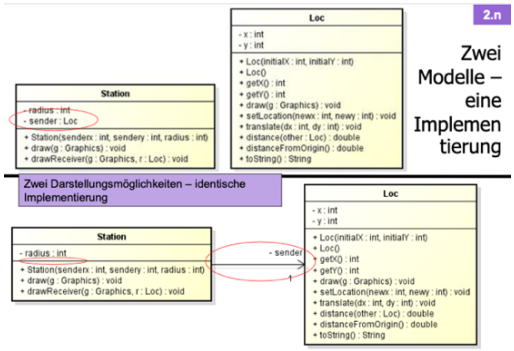
Objektdiagramme

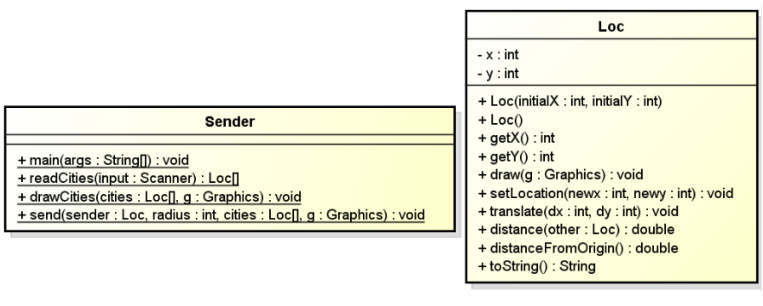
Zur Darstellung der Klassen verwendet man Klassendiagramme.

Zur Darstellung von Objekten verwendet man (selten) Objektdiagramme.
– Deutlich seltener eingesetzt, z. B. zur Verdeutlichung eines komplizierten Objektgeflechts
– Immer noch statisches Diagramm (Objektzusammenhänge), keine dynamischen Abläufe

- Beispiel:







```
public class Sender {
	public static void main(String[ ] args) throws FileNotFoundException {
		Scanner input = new Scanner(new File("Cities.txt"));
		Loc[ ] cities = readCities(input);
		input.close();

		DrawingPanel panel= new DrawingPanel(200, 200);
		Graphics g= panel.getGraphics();
		drawCities(cities,g);

		Scanner console = new Scanner(System.in) ;
		Loc sender = new Loc();
		System.out.print("Sender x?");
		int x = console.nextInt();
		System.out.print("Sender y?");
		int y= console.nextInt();
		sender.setLocation(x, y);
		System.out.print("Radius?");
		int r = console.nextInt();
		console.close();
		check(sender , r, cities,g);
	}
	public static void  check(Loc sender, int r, Loc[ ] cities , Graphics g) {
		g.setColor(Color.YELLOW);
		g.drawOval(sender.getX() - r, sender.getY() -r , r*2, r*2);
		for (int i =0 ; i< cities.length; i++) {
			double distance = cities[i].distance(sender);
			if (distance <= r) {
				cities[i].draw(g, 3);
			}
		}
	}
	public static void drawCities(Loc[ ] cities, Graphics g) {
		g.setColor(Color.BLACK);
		for (int i = 0; i < cities.length;i++) {
			cities[i].draw(g, 3);
		}
	}
	public static Loc[ ] readCities(Scanner input) {
		int num = input.nextInt();
		Loc[ ] cities = new Loc[num];
		for (int i = 0; i<num; i++) {
			Loc loc = new Loc();
			int x = input.nextInt();
			int y = input.nextInt();
			loc.setLocation(x, y);
			cities[i] = loc;
		}
		return cities;
	}
```

```
public class Loc {
	private int x;
	private int y;

	public Loc(int initialX, int initialY) {
		setLocation(initialX,initialY);
	}
	public Loc() {
		this(0, 0); // Verkettung von Konstruktoren
		//x = 0;   //Standardkonstruktor
		//y = 0;
	}

	public int getX() {
		return x;
	}
	public int getY() {
		return y;
	}

	public void draw(Graphics g , int r) {
		g.fillOval(this.x, this.y, r, r);
	//this greift auf das Objekt in dem  ich mich befinde zu
		g.drawString(toString(), x, y);
	}

	public void setLocation(int x, int y) {
		if ( x < 0 || y < 00) {
		    throw new IllegalArgumentException("x und y
				muessen groesser null sein");
		}
		this.x = x;		//Hier ist ein This notwendig
		this.y = y;
	}

	public void transelate(int dx, int dy) {
		setLocation(x + dx,y + dy);
	}
	public double distance(Loc b) {
		return (Math.sqrt(Math.pow((b.x-x), 2)+Math.pow(b.y-y, 2)));
	}
	public double distanceFromOrgin(Graphics g , int r) {
		return (Math.sqrt(Math.pow(x, 2)+Math.pow(y, 2)));

	}
	public String toString() {
		return "(" + x + ", " + y + ")";
	}
```

L.3 Vererbung

Code reuse / Code-Wiederverwendung: Die Praxis, Code so zu schreiben, dass man ihn in verschiedenen Kontexten einsetzen kann.

- Allgemeine Regeln sind nützlich (großes Handbuch).
- Spezielle Regeln, die Vorrang vor allgemeinen Regeln haben, sind ebenfalls nützlich.

Ist-ein-Beziehung und Hierarchien

„Ist-ein"-Beziehung (is-a relationship): Eine hierarchische Verbindung zwischen Kategorien, wobei eine Kategorie eine spezielle Version der anderen ist.

Vererbungshierarchie (inheritance hierarchy): Eine Menge von Klassen, die durch Ist-ein-Beziehungen verbunden sind und dadurch gemeinsamen Code teilen.

Vererbung – einmal schreiben und Wiederverwenden **nur bei „Ist-ein"-Beziehung!**

Vererbung (inheritance): Bildung einer neuen Klasse, die auf einer existierenden Klasse basiert, wobei

Attribute und Verhalten der existierenden Klasse übernommen werden.

• Vererbung ist ein Mechanismus,

- um verwandte Klassen zu gruppieren, um Code verwandter Klassen gemeinsam zu nutzen: **Code-Redundanz vermeiden**
- Eine Klasse erweitert eine andere Klasse um spezielles Verhalten.