Universität Bremen

Praktische Informatik 1
Objektinteraktion 2

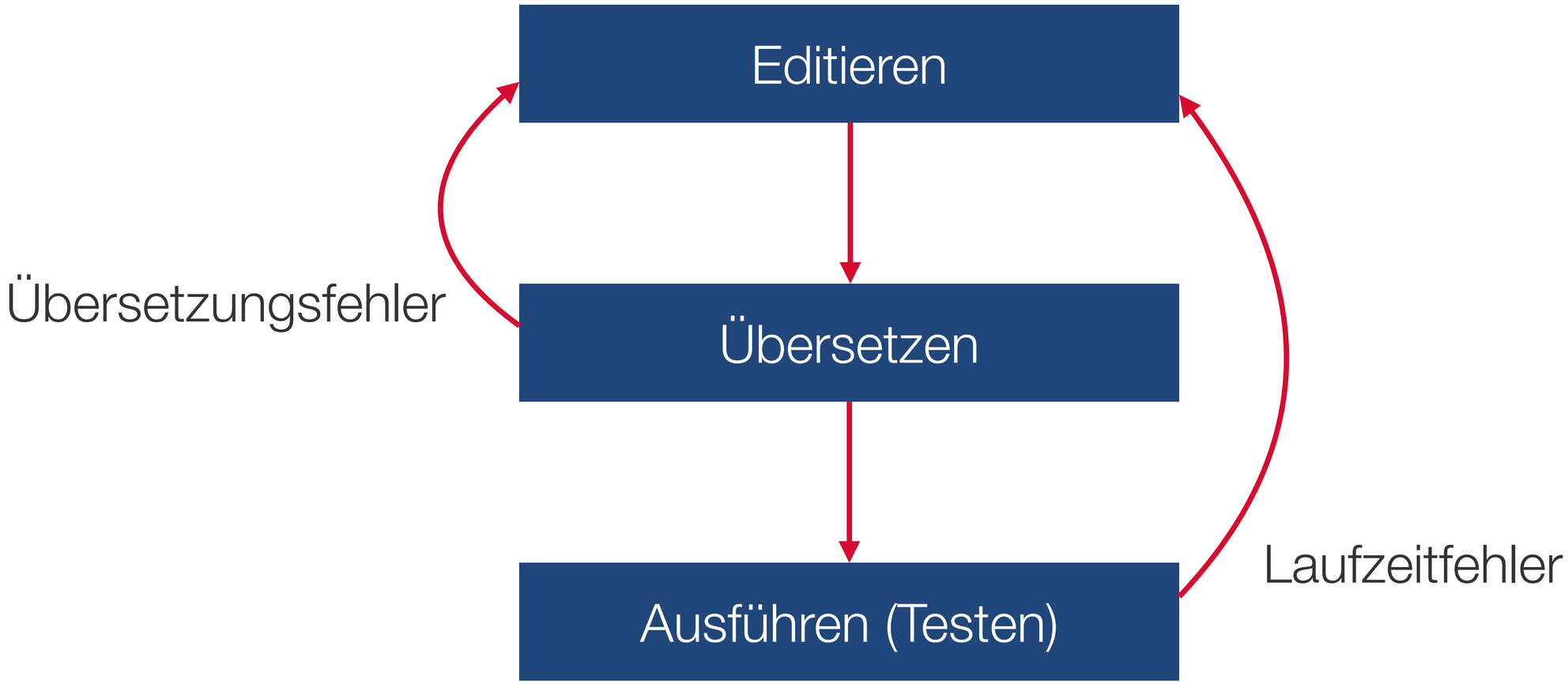
Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Entwicklungszyklus



Fehler beim Programmieren

```
int i = 0, s = 0
while (i < 10); {
    s = s + i;
    i = i + 1;
}
```

- **Tippfehler**: Semikolon vergessen, Klammern passen nicht, Code kopiert und nicht richtig angepasst
  - Können vom Compiler entdeckt werden, müssen aber nicht

```
if (getNextKey() == VK_RIGHT)
    if (p.getX() < 5)
        p.setLocation(p.getX() + 1, p.getY());
else
    playSound("error");
```

- **Strukturfehler**: Compiler verwendet andere Zuordnung als gedacht

```
int i = "1";
float d = 2 / 3;
```

- **Typfehler**: Werden bei Zuweisungen erkannt, bleiben aber in Ausdrücken teilweise unerkannt

- **Logische Fehler**: Sind erst zur Laufzeit sichtbar

Fehlerbehebung: Übersetzungsfehler

- Nur den ersten Fehler beachten, andere könnten Folgefehler sein
  - BlueJ unterstreicht Fehler und zeigt Zähler rechts unten an
  - Klick auf Zähler springt zum nächsten Fehler
- **Fehlermeldung lesen**
  - Dokumentation zu Fehlermeldung lesen
- Quelltext bei der angegebenen Zeilennummer ansehen
  - Dokumentation zu falschem Konstrukt lesen

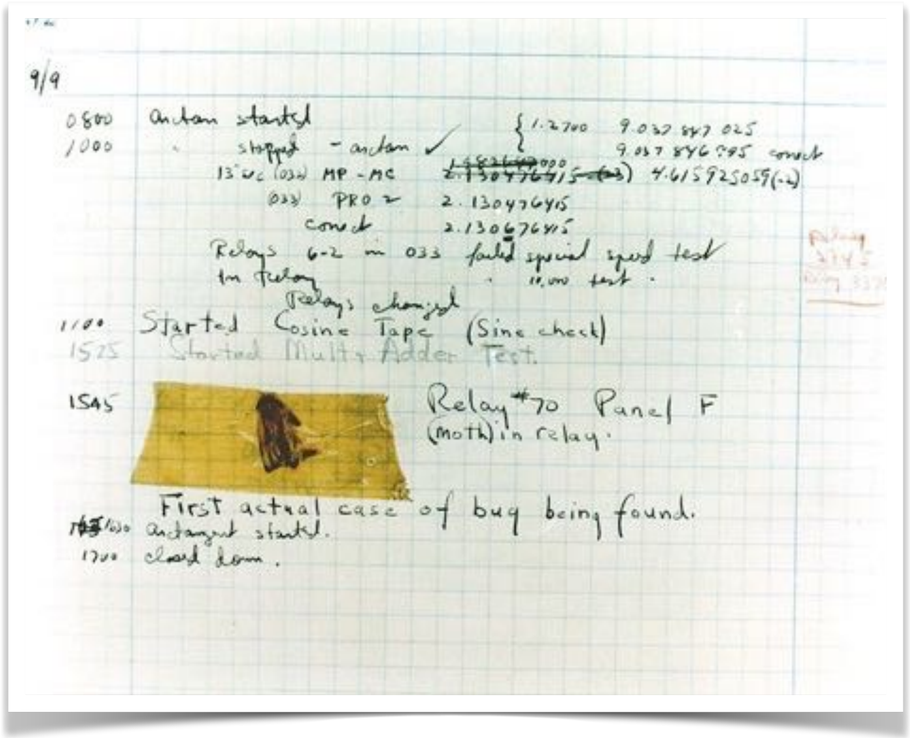
Universität Bremen

Fehlerbehebung: Laufzeitfehler

- Falls eine Fehlermeldung angezeigt wird, diese lesen und den Quelltext bei der angegebenen Zeilennummer ansehen
- **Debugger** verwenden oder **Testausgaben einbauen**
  - Aber keine weiteren Fehler durch solche Ausgaben einbauen!
- Verschiedene Testfälle durchspielen
  - Wann tritt der Fehler auf, wann nicht?
- **Nur ein verstandener Fehler ist ein beseitigter Fehler!**

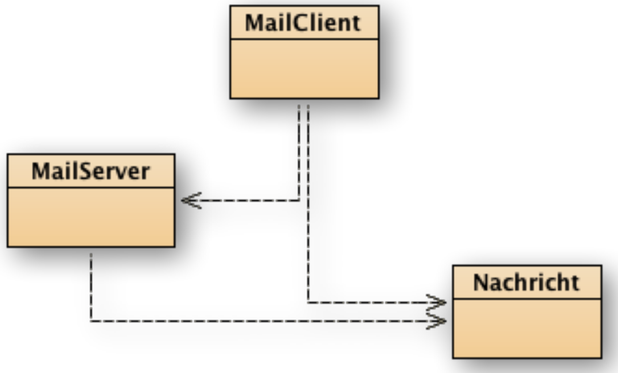
Debugger

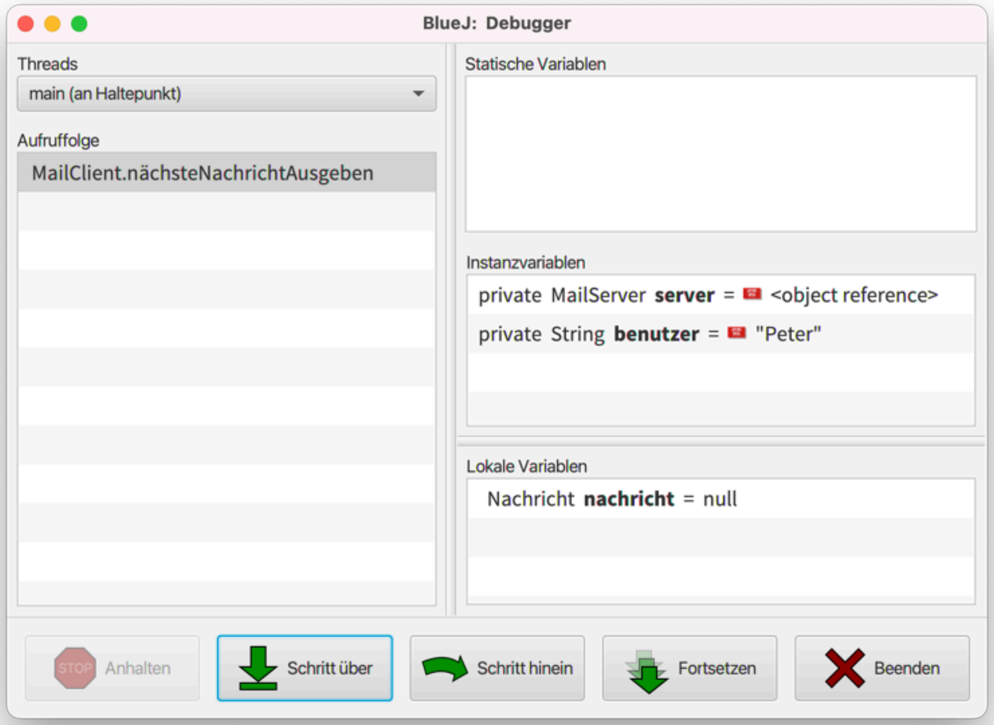
- Programme enthalten oft Fehler (**Bugs**), die sich erst zur **Laufzeit** zeigen
- Ein **Debugger** hilft, das Verhalten eines Programms während seiner Laufzeit zu untersuchen
  - Schrittweise Ausführung
  - Gezieltes Anhalten
  - Sichten des Programmzustands

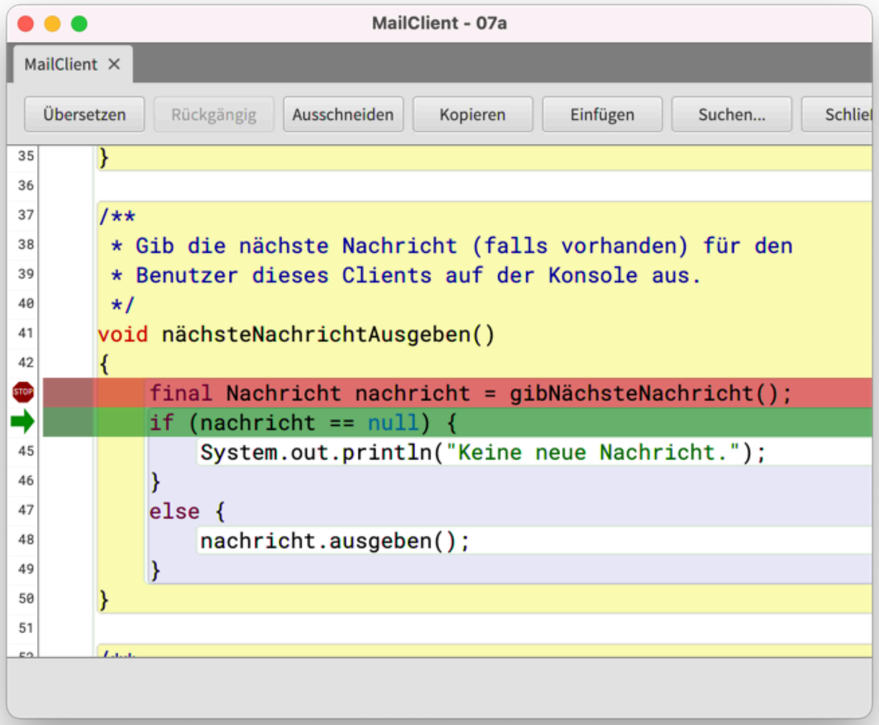


Logbuch-Seite des Mark II Aiken Relay Calculator mit dem ersten Bug (1947)

Mail-System: Demo







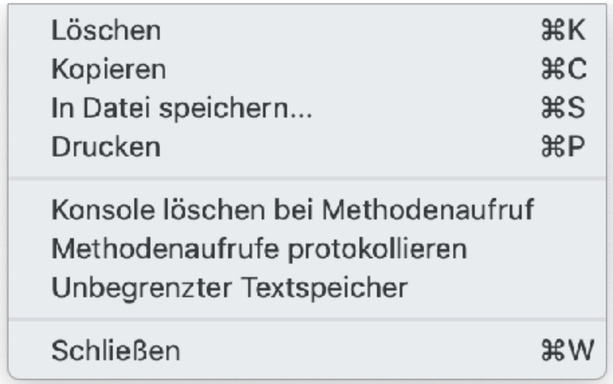
Universität Bremen

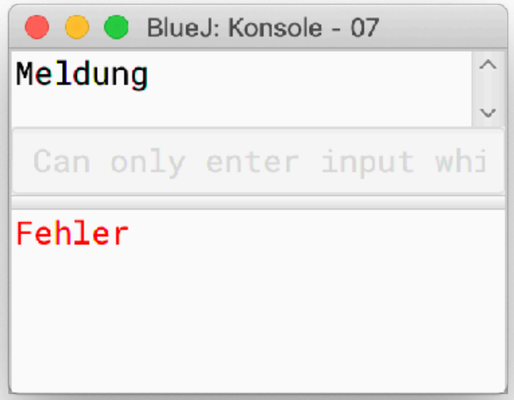
Textausgaben in Java

- Mit **System.out.println** und **System.out.print** kann Text auf der Konsole ausgegeben werden
  - **System.out.println** gibt nach der Ausgabe einen Zeilenumbruch aus
- Beide Methoden sind für viele Datentypen überladen
- Mit **System.err.println** usw. können Fehlermeldungen ausgegeben werden
  - In BlueJ eigener Bereich mit rotem Text
- BlueJ-Konsole kann konfiguriert werden

```
void ausgeben()
{
    System.out.println("Von: " + absender);
    System.out.print("An: ");
    System.out.println(empfänger);
}
```

```
System.out.println("Meldung");
System.err.println("Fehler");
```



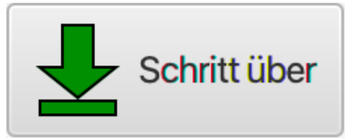


Universität Bremen

Benutzung des Debuggers

- Hält das Programm an, wenn es gerade läuft

- Führt die nächste Anweisung aus. Methodenaufrufe werden am Stück ausgeführt oder bis zum nächsten Haltepunkt



- Führt die nächste Anweisung aus. In Methodenaufrufe wird abgestiegen, wenn der Quelltext der Methode bekannt ist

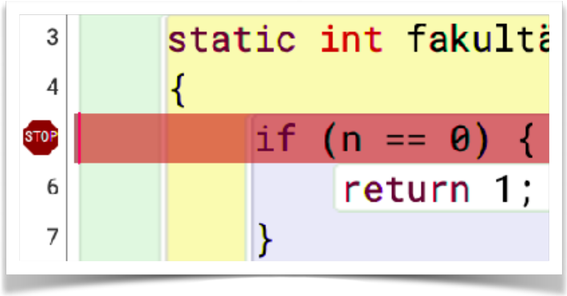
- Führt das Programm bis zu seinem Ende oder dem nächsten Haltepunkt aus

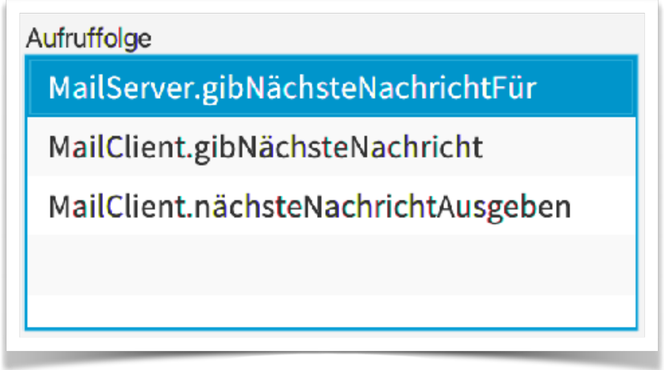
- Bricht das Programm ab und startet die virtuelle Maschine neu (dasselbe wie  rechts unten im Hauptfenster)

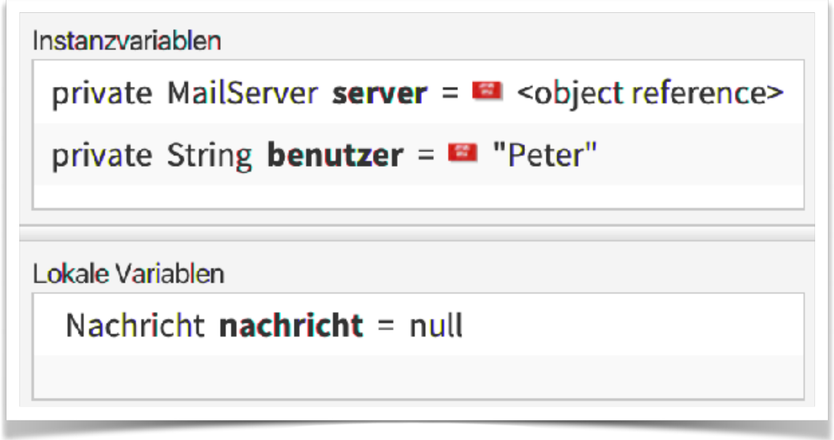
Universität Bremen

Debugger

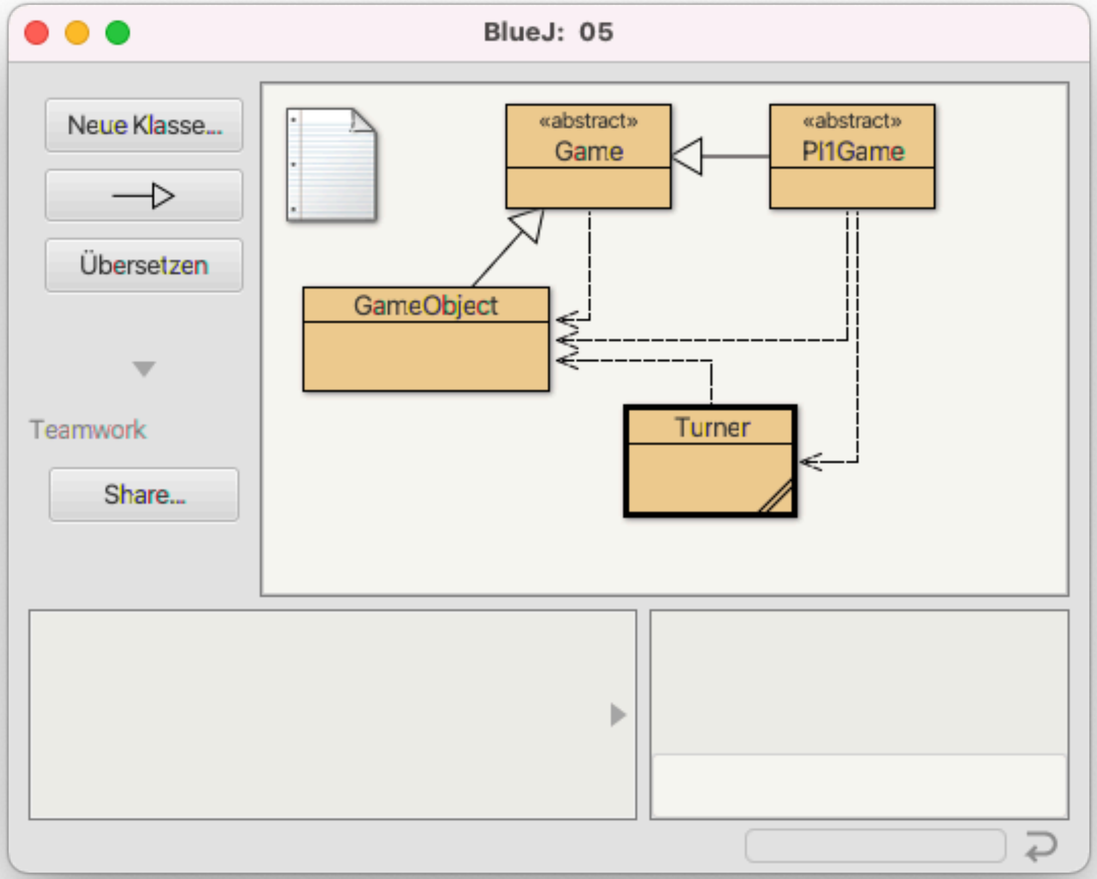
- **Haltepunkt** (Breakpoint): Hält das Programm **vor** der markierten Anweisung an
- **Aufruffolge** (Call Stack): Zeigt die Hierarchie der aufgerufenen Methoden an
- **Ausführungskontext** einer Methode: Alle lokalen Variablen und Parameter
  - **Instanzvariablen**: Attribute des aktuellen Objekts, also von **this**







Fehler finden: Demo



Rekursion und Iteration: Beispiel

- **3!**

= 3 · 2!

= 3 · (2 · 1!)

= 3 · (2 · (1 · 0!))

= 3 · (2 · (1 · 1))

= 3 · (2 · 1)

= 3 · 2

= 6

$$n! = \begin{cases} 1 & \text{wenn } n = 0 \\ n \cdot (n-1)! & \text{sonst} \end{cases}, n \in \mathbb{N}_0$$

$$n! = \prod_{i=1}^{n} i, n \in \mathbb{N}_0$$

```
static int fakultät(final int n)
{
    if (n == 0) {
        return 1;
    }
    else {
        return n * fakultät(n - 1);
    }
}
```

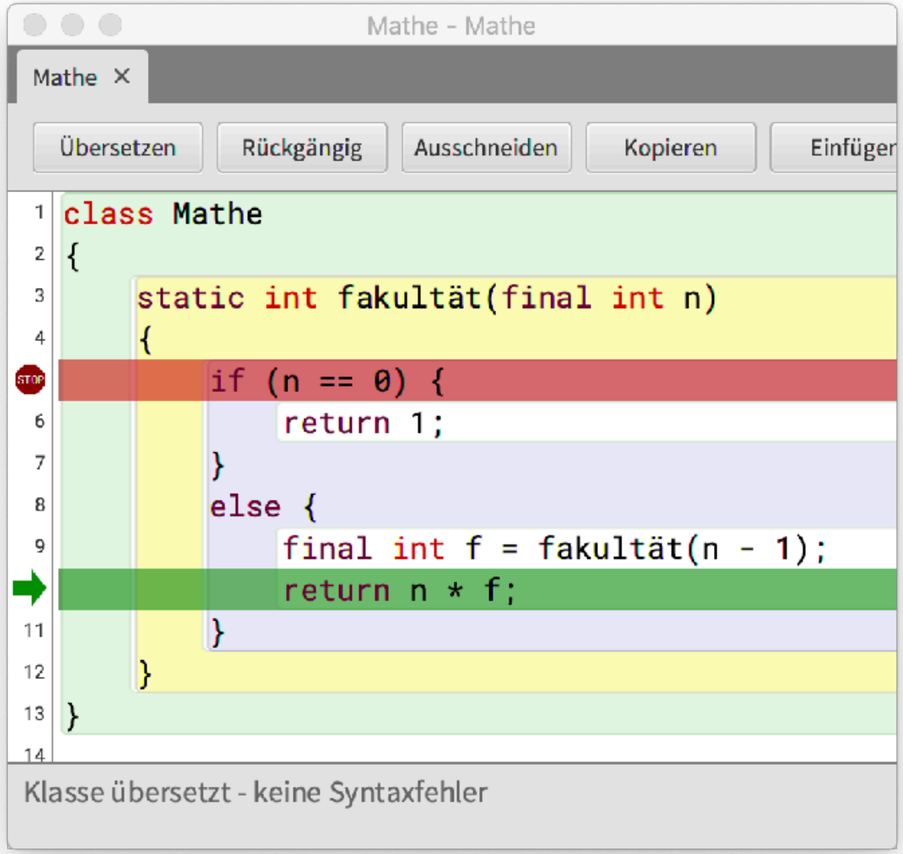
Rekursion

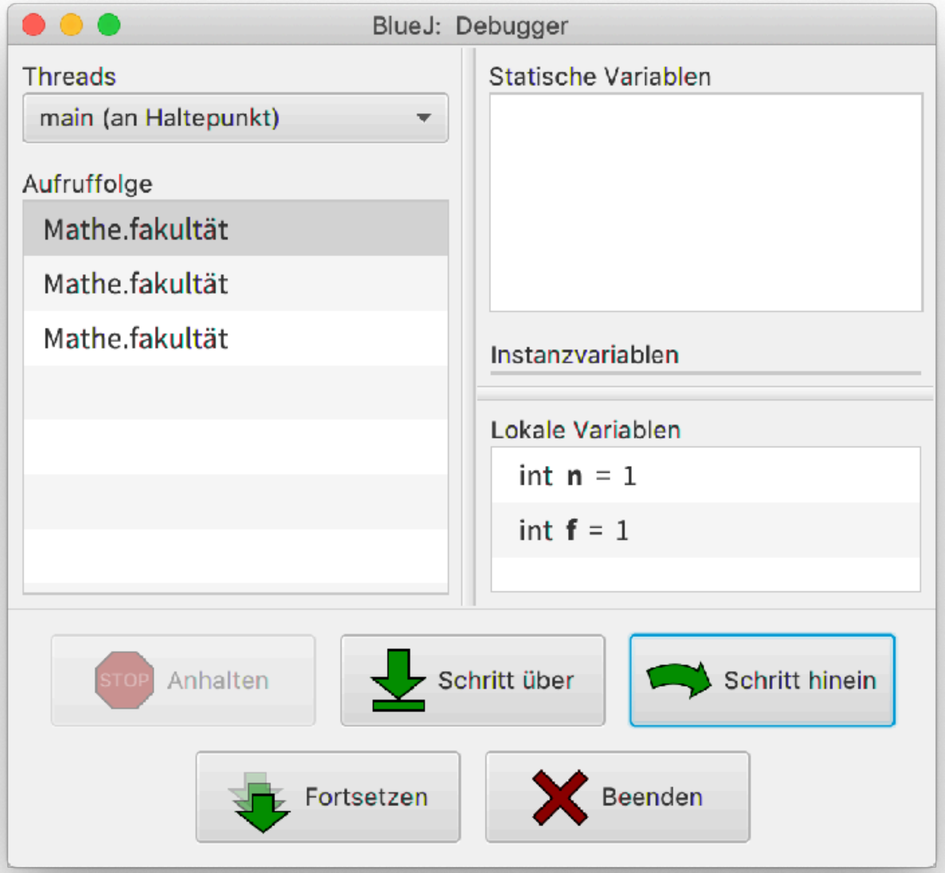
```
static int fakultät(final int n)
{
    int p = 1, i = 1;
    while (i <= n) {
        p = p * i;
        i = i + 1;
    }
    return p;
}
```

Iteration

Universität Bremen

Rekursion: Demo





Universität Bremen

Rekursion

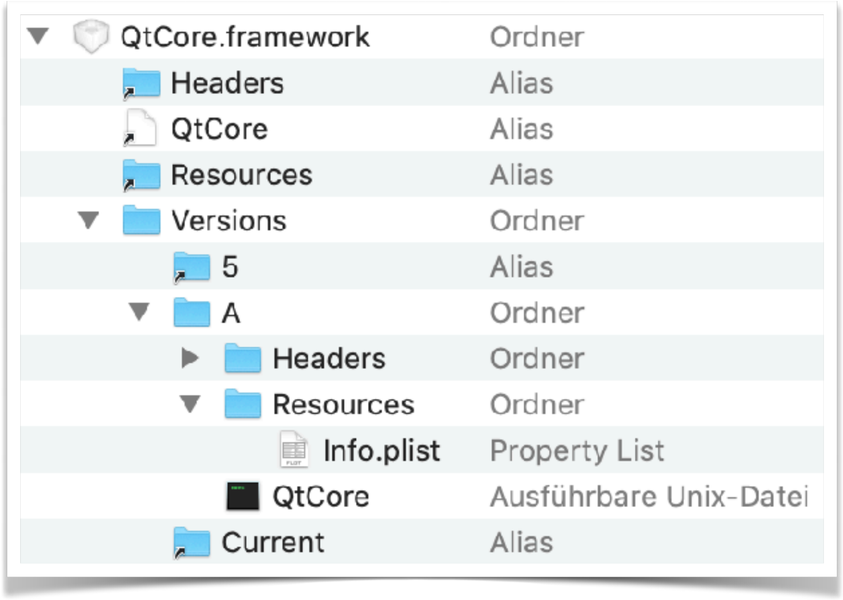
- Es gibt zwei unterschiedliche Ansätze zur Problemlösung
  - **Rekursion**: Methode löst einen Teil des Problems und **ruft sich selbst** wieder **auf**, um den Rest zu lösen
  - **Iteration**: Problem wird in einer **Schleife** gelöst, wobei jeder Durchlauf einen Teil des Problems löst

```
static int fakultät(final int n)
{
    if (n == 0) {
        return 1;
    }
    else {
        return n * fakultät(n - 1);
    }
}
```

- Lokale Variablen existieren **pro** rekursivem **Aufruf einmal** und können **unterschiedliche Werte** haben
- In mindestens einem Ausführungszweig darf sich die Methode nicht selbst aufrufen (**Rekursionsende**)
- **Rekursiver Abstieg**: Was auf dem Weg bis zum Rekursionsende gemacht wird
- **Rekursiver Aufstieg**: Was zwischen Rekursionsende und Ende des ursprünglichen Aufrufs passiert

Universität Bremen

Einschub: Dateisystem

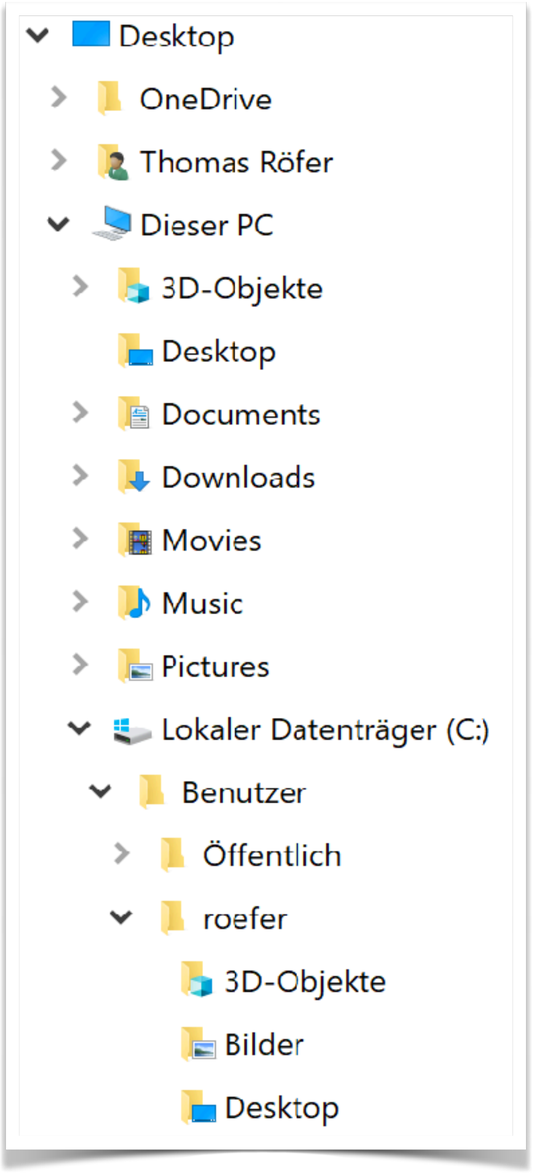


- **Dateien**, z.B. Dokumente, Programme usw.
- **Verzeichnisse** enthalten Dateien, Verzeichnisse und Querverweise
- **Querverweise** zeigen auf Dateien oder Verzeichnisse, die an einem anderen Ort im Dateisystem abgelegt sind
  - Manche passen sich an, wenn das Ziel verschoben wird, manche nicht
- Alle Dateisystemobjekte haben **Namen**, bei denen mal zwischen Groß- und Kleinschreibung unterschieden wird (Linux) und mal nicht (Windows, macOS)
- **Dateisystemwurzel**(n): Laufwerke **A**-**Z** und Netzwerklaufwerke **\\<computer>\<freigabe>** unter Windows, **/** unter Linux und macOS

Universität Bremen

Dateisystem: Fallstricke

- Dargestellte Struktur in graphischer Benutzungsoberfläche teilweise anders als tatsächliche
- Verzeichnisnamen in graphischer Benutzungsoberfläche in Landessprache übersetzt, im Dateisystem aber nicht
- **Virtuelle Verzeichnisse**, die z.B. Dateien anzeigen, die einem Suchmuster entsprechen
- **Temporäre Verzeichnisse**, die z.B. den Inhalt eines **zip**-Archivs anzeigen, aber nach dem Schließen wieder weg sind
- Verzeichnisse und Dateien können versteckt sein
  - Unter **Linux** und **macOS** ist z.B. alles versteckt, das mit **.** beginnt



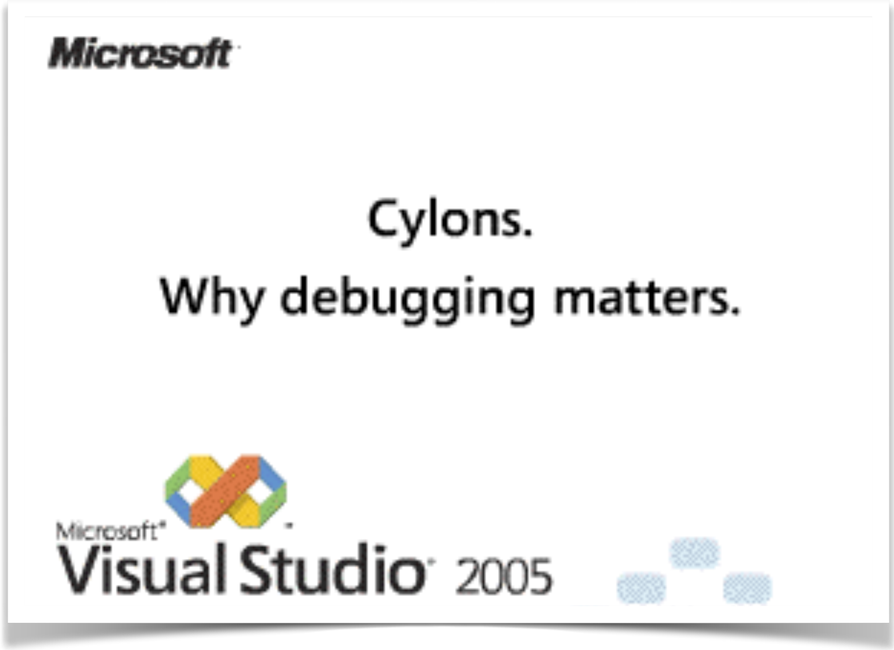
Universität Bremen

Heimverzeichnis

- Verzeichnis, das für jede Nutzer:in des Computers/Computernetzes getrennt existiert
  - Unterverzeichnisse für Dokumente, Bilder, Musik, Videos, Downloads, den Desktop usw.
- **Windows**: **C:\Benutzer\<name>** (eigentlich **C:\Users\<name>**)
- **Linux**: **/home/<name>** (oder **~** für aktuelle und **~<name>** für andere Nutzer:innen)
- **macOS**: **/Users/<name>** (oder **~** für aktuelle und **~<name>** für andere Nutzer:innen)

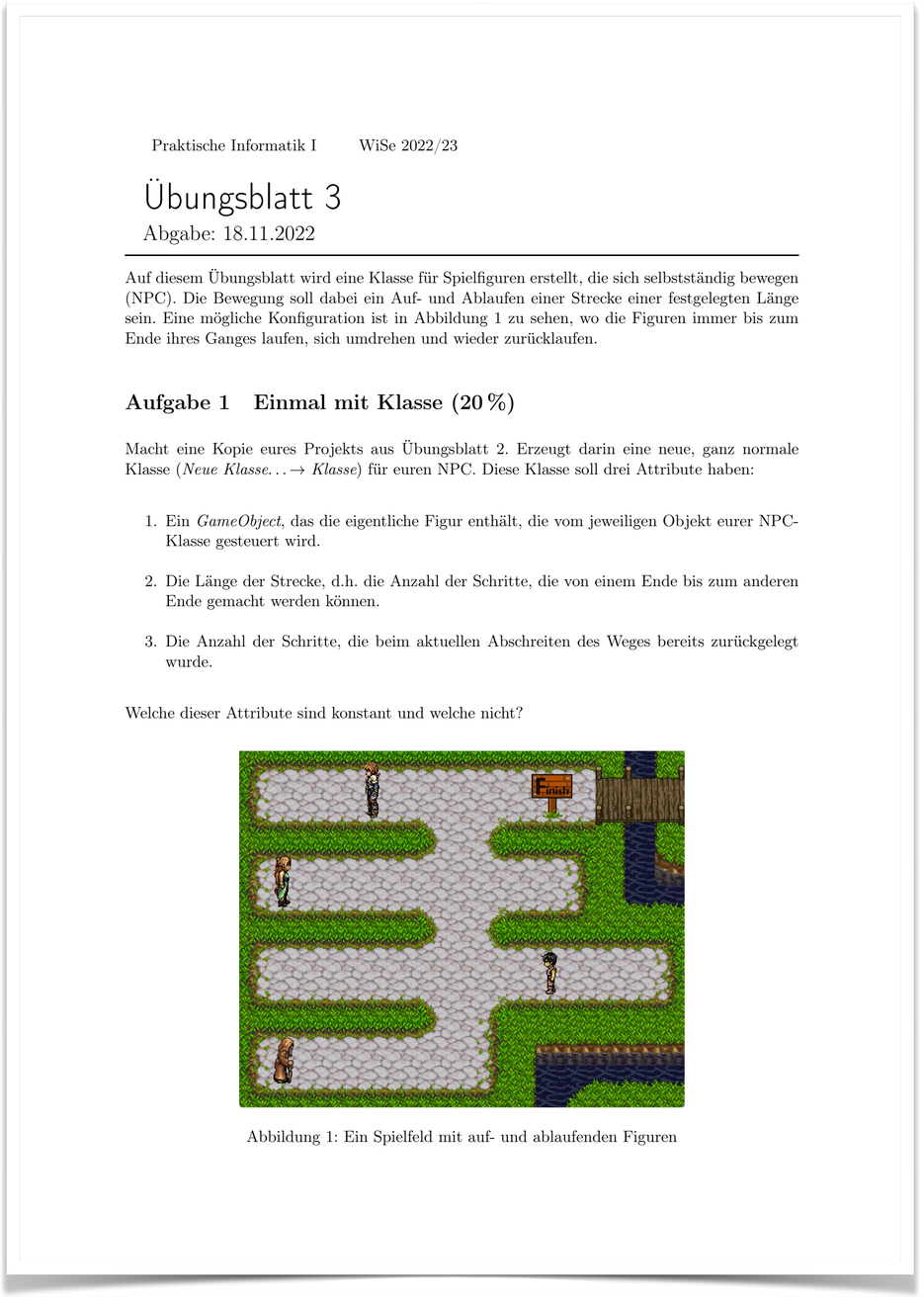
Zusammenfassung der Konzepte

- **Entwicklungszyklus**
- **Fehlerbehebung**
- **Debugger**
- **Rekursion**
- **„Nur ein verstandener Fehler ist ein beseitigter Fehler“**
- **Dateisystem**



Übungsblatt 3

- Patrouillierende Figur implementieren
- Aufgabe 1: Klasse mit Attributen
- Aufgabe 2: Konstruktor
- Aufgabe 3: Schritt ausführen
- Aufgabe 4: Einbindung in Spiel
- Bonusaufgabe: Spielfigur bei Positionsgleichheit verschwinden lassen und Spielabbruch
- **Nicht Entpacken**, Lizenz **Selbst verfasstes … Werk**



Praktische Informatik I WiSe 2022/23

Übungsblatt 3

Abgabe: 18.11.2022

Auf diesem Übungsblatt wird eine Klasse für Spielfiguren erstellt, die sich selbstständig bewegen (NPC). Die Bewegung soll dabei ein Auf- und Ablaufen einer Strecke einer festgelegten Länge sein. Eine mögliche Konfiguration ist in Abbildung 1 zu sehen, wo die Figuren immer bis zum Ende ihres Ganges laufen, sich umdrehen und wieder zurücklaufen.

Aufgabe 1 Einmal mit Klasse (20 %)

Macht eine Kopie eures Projekts aus Übungsblatt 2. Erzeugt darin eine neue, ganz normale Klasse (*Neue Klasse…* → *Klasse*) für euren NPC. Diese Klasse soll drei Attribute haben:

1. Ein *GameObject*, das die eigentliche Figur enthält, die vom jeweiligen Objekt eurer NPC-Klasse gesteuert wird.
2. Die Länge der Strecke, d.h. die Anzahl der Schritte, die von einem Ende bis zum anderen Ende gemacht werden können.
3. Die Anzahl der Schritte, die beim aktuellen Abschreiten des Weges bereits zurückgelegt wurde.

Welche dieser Attribute sind konstant und welche nicht?

Abbildung 1: Ein Spielfeld mit auf- und ablaufenden Figuren