Universität Bremen

Praktische Informatik 1
Fehler vermeiden 1

Thomas Röfer

Cyber-Physical Systems
Deutsches Forschungszentrum für Künstliche Intelligenz

Multisensorische Interaktive Systeme
Fachbereich 3, Universität Bremen

Bild: ©Michael Ihle

Universität Bremen

Testen und Fehlerbeseitigung

- **Übersetzungsfehler**: Quelltext entspricht nicht der Sprachspezifikation (werden vom Compiler entdeckt)
- **Logische Fehler**: Programm zeigt nicht das gewünschte Verhalten
- **Testen**: Überprüfung, ob ein Stück Software (Methode, Klasse, Programm) das gewünschte Verhalten zeigt
- **Fehlerbeseitigung**: Suche nach der **Ursache** eines Fehlers und seine Beseitigung
  - Nur ein **verstandener** Fehler ist ein beseitigter Fehler!

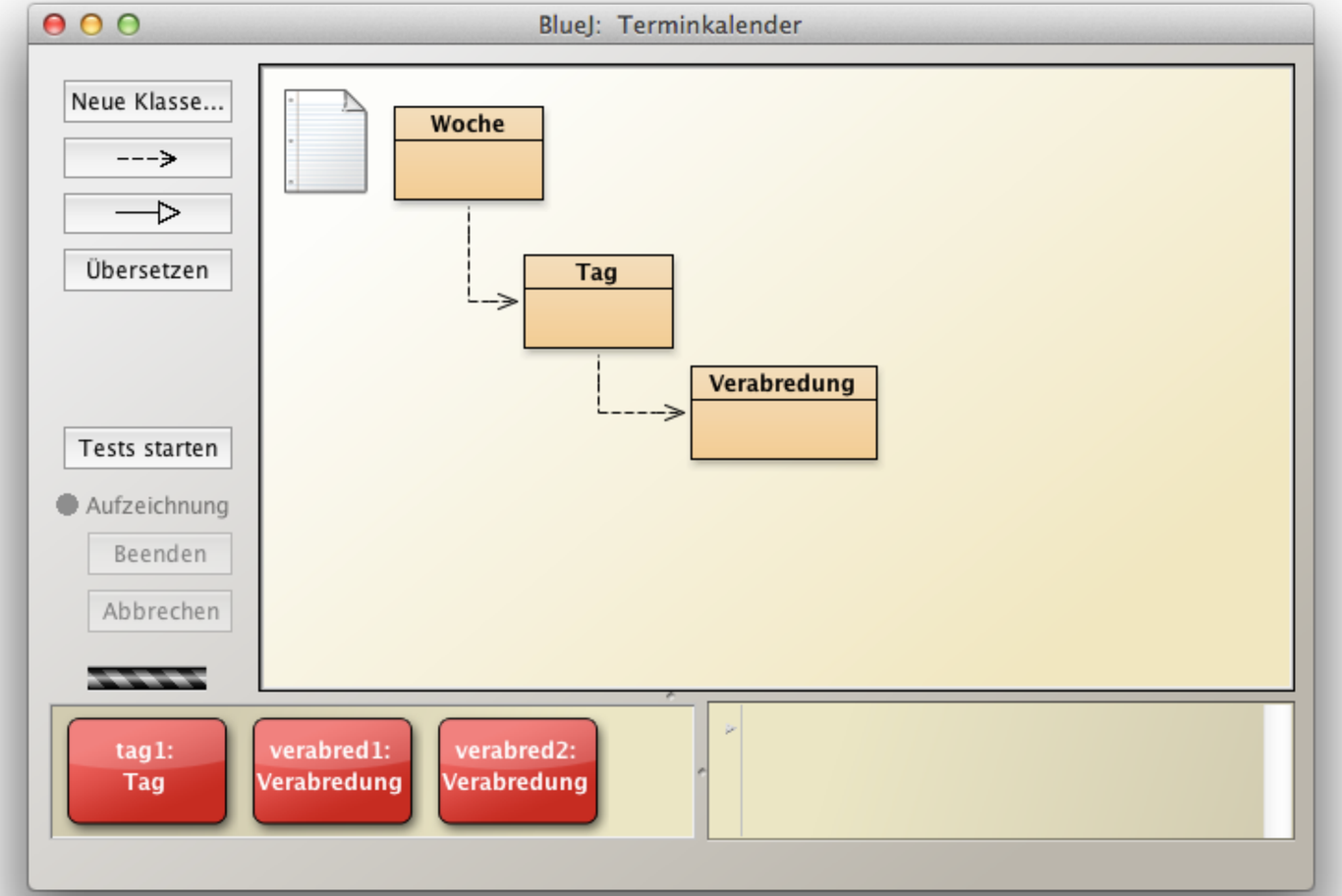
Universität Bremen

Modultests (Unit-Tests)

- **Modultests**: Tests einzelner Einheiten einer Anwendung
- **Akzeptanztests**: Test einer gesamten Anwendung
- Frühes Testen einzelner Komponenten → kein Gesamtsystem aus lauter fehlerhaften Komponenten
  - Sonst beeinflussen sich Fehler gegenseitig
  - Viel schwerer zu finden
- **BlueJ** erlaubt interaktives Testen von Klassen

Universität Bremen

Terminkalender: Demo



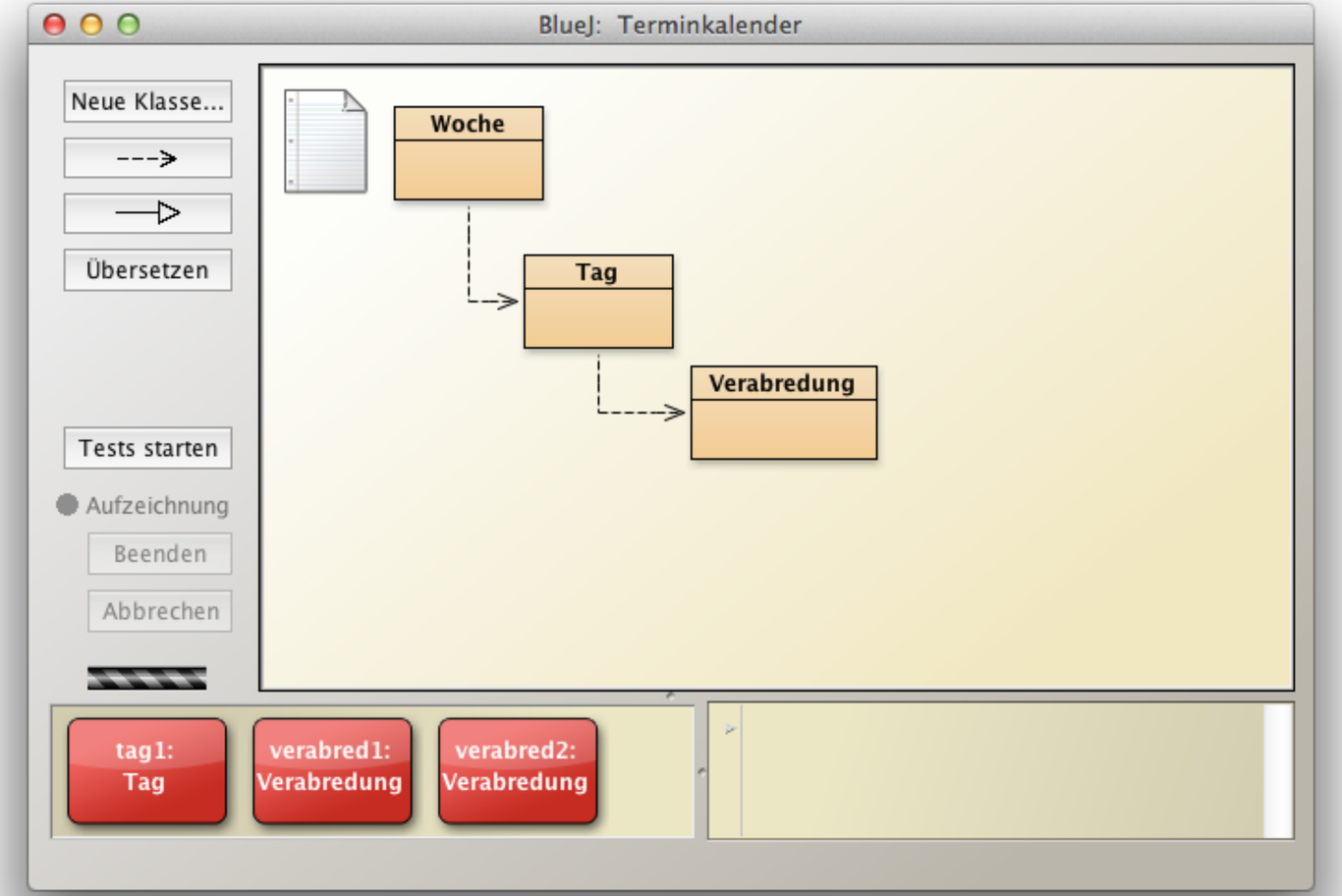
Universität Bremen

Terminkalender-Tests

- Genügend Platz im Attribut **termine** für Termine von 8-17h?
- Gibt **termineAusgeben** korrekt die Liste der vereinbarten Termine aus?
- Aktualisiert **terminVereinbaren** das Attribut **termine** korrekt?
- Liefert **findePlatz** das korrekte Ergebnis?

Universität Bremen

Terminkalender: Demo 2



Universität Bremen

Testfälle

- **Trivialfälle**: Fälle, in denen das Ergebnis ohne sich wiederholende Berechnung zurückgeliefert wird, z.B. vereinbaren eines Termins der Dauer 0
- **Grenzfälle**: Eingaben an den Grenzen des erlaubten Wertebereichs, z.B. Termine vor/nach 8h oder 17h
- **Fehlerfälle**: Wenn Eingaben Fehler erzeugen, z.B. **playSound("missingFile")**
- **Normalfälle**: Alles andere

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

Universität Bremen

Positives und negatives Testen

- **Positives Testen**: Überprüft die Funktionalität, die wir erwarten
  - Z.B. erfolgreiches Eintragen eines Termins
- **Negatives Testen**: Überprüfung der Fälle, die scheitern sollten
  - Z.B. doppeltes Buchen eines Termins oder Buchen außerhalb des gültigen Zeitbereichs

Universität Bremen

Regressionstests

- Tests, die bereits erfolgreich durchgeführt wurden und während der Weiterentwicklung der Software wiederholt werden
- Wahrscheinlichkeit der Durchführung steigt durch **Automatisierung** der Tests
- Implementierung von **Testgerüstklassen**, die andere Klassen testen

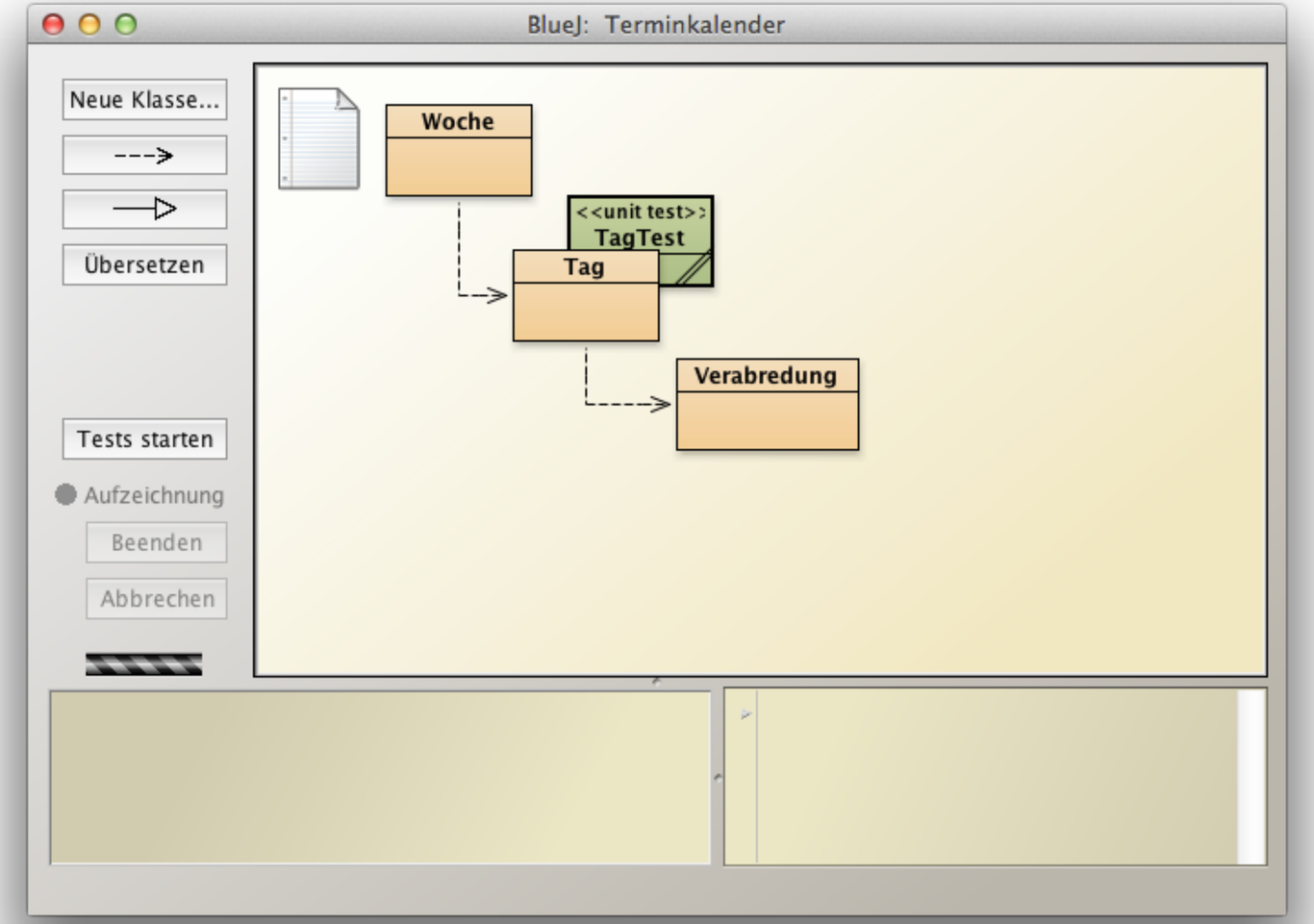
```
class TagTest
{
    private Tag tag;

    void leerenTagAusgeben()
    {
        tag = new Tag(1);
        tag.termineAusgeben();
    }
}
```

Universität Bremen

Automatisiertes Prüfen von Testergebnissen

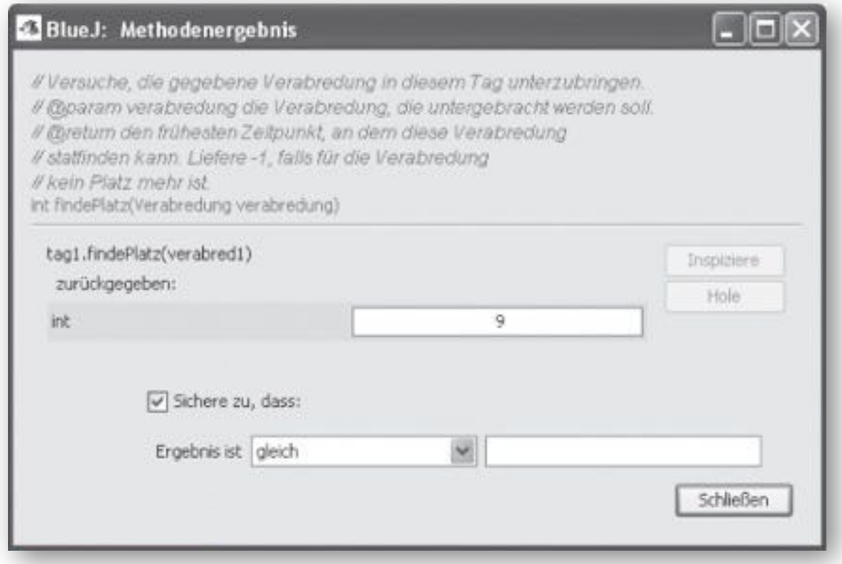
- Manuelles Überprüfen der Richtigkeit von Testergebnissen ist aufwändig und fehlerträchtig
- Daher sollten Tests **selbstüberprüfend** sein
- **JUnit**: Rahmenwerk zur Unterstützung von Modul- und Regressionstests in Java
- In **BlueJ** integriert
- Achtung: **JUnit** ruft unsere Tests aus einem anderen Paket auf, weshalb unsere Testklassen und -methoden **public** sein müssen

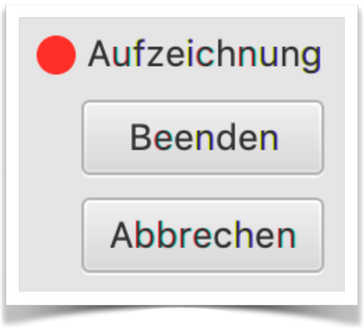
Terminkalender: Demo 3



Tests aufzeichnen

- Kontextmenu der zu testenden Klasse: **Testklasse erzeugen**
- Kontextmenu der Testklasse: **Erzeuge Testmethode...**
- Objekte erzeugen, Methoden aufrufen
- Bei Methoden mit Ergebnis **Zusicherungen** für Rückgabewerte definieren
- Schaltfläche **Aufzeichnung beenden**





Universität Bremen

Testgerüste

- Ein **Testgerüst** besteht aus einer Menge von Objekten in einem definierten Zustand, die als Grundlage für einen Modultest dienen

- In BlueJ

  - Objekte erzeugen

  - Kontextmenu der Testklasse: **Objektzustand speichern**

```
public class TagTest
{
    private Tag tag1;
    private Verabredung verabred1;

    @BeforeEach
    public void setUp()
    {
        tag1 = new Tag(1);
        verabred1 = new Verabredung(
                "Vorlesung", 1);
    }
}
```

Universität Bremen

Teststärke: Testabdeckung

- Alle Tests laufen erfolgreich durch, aber testen sie auch alles?
- Es kann eine Statistik über den Code erstellt werden, der während der Tests tatsächlich ausgeführt wird (**Testabdeckung**, Ziel: 100 %)
- **Methodenabdeckung**: Anteil der Methoden, die ausgeführt werden
- **Anweisungsabdeckung**: Anteil der Anweisungen (manchmal auch Zeilen), die ausgeführt werden
- **Zweigabdeckung**: Anteil der Zweige (z.B. auch leere **else**-Zweige), die ausgeführt werden
- **Bedingungsabdeckung**: Anteil der Teilbedingungen, die sowohl **true** als auch **false** werden
- **Pfadabdeckung**: Anteil der möglichen Pfade (d.h. Abfolgen aus verschiedenen Zweigen), die durchlaufen werden

Universität Bremen

Teststärke: Mutationstests

- Mutationstests testen die Stärke von Tests
- Sie verändern den ursprünglich erfolgreich getesteten Code an einzelnen Stellen und erwarten, dass diese Änderungen durch die Tests als Fehler aufgedeckt werden
  - Allerdings macht nicht jede Änderung die Ursprungsimplementierung falsch
- Mutationen können z.B. sein:
  - Bedingungen auf **true** oder **false** setzen
  - Operatoren durch ähnliche ersetzen, z.B. **>** durch **>=** oder **+** durch **-**
  - Teile von Ausdrücken oder ganze Anweisungen weglassen
- Im zweiten Semester werden wir ein Werkzeug einsetzen, das dies automatisiert

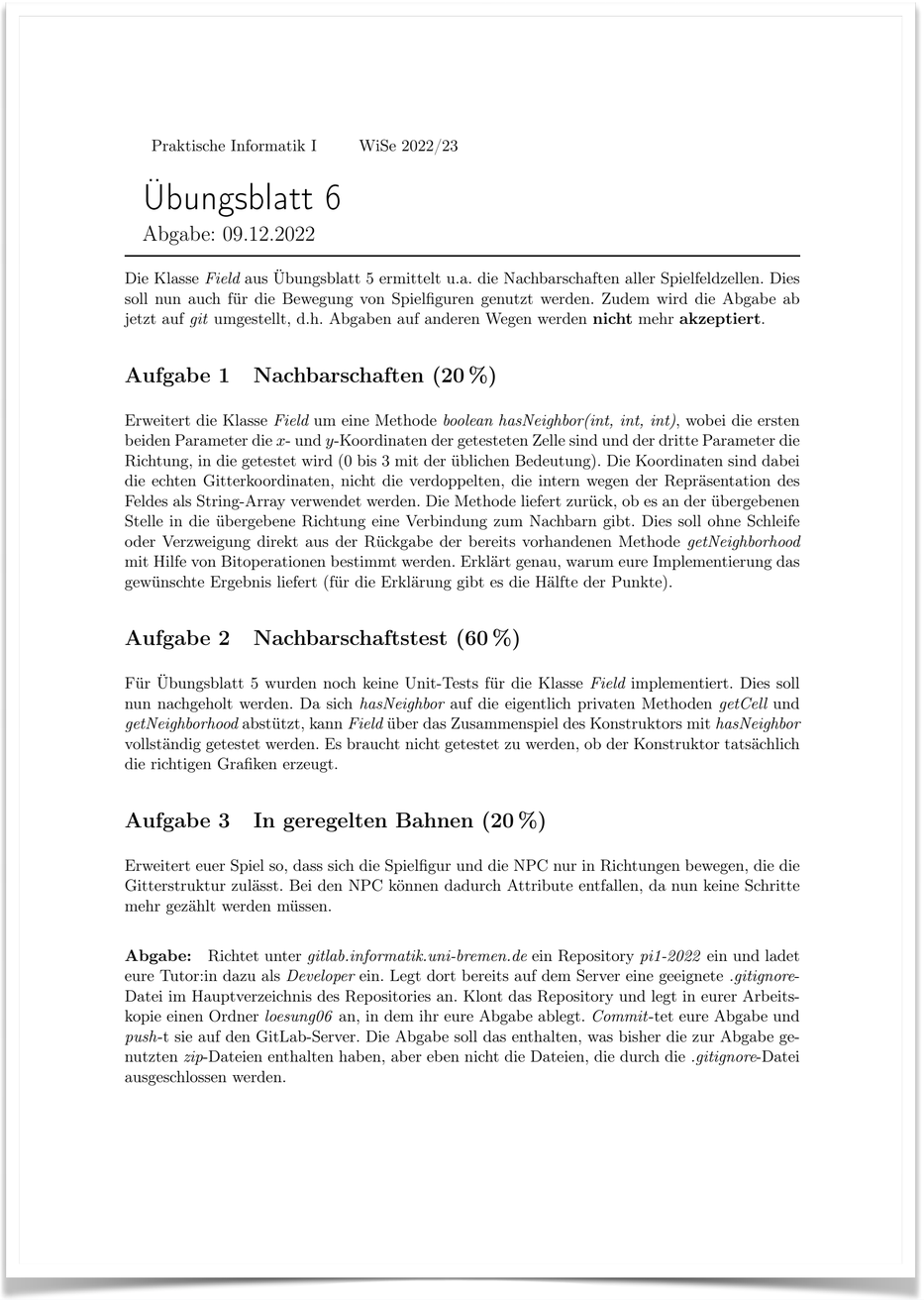
Universität Bremen

Zusammenfassung der Konzepte

- **Testen** (**positiv**/**negativ**)
- **Trivialfall**, **Normalfall**, **Grenzfall**, **Fehlerfall**
- **Regressionstest**
- **Zusicherung**
- **Testgerüst**
- **Teststärke**, **Testabdeckung**, **Mutationstest**

Übungsblatt 6

- Aufgabe 1: Klasse **Field** um Nachbarschafts-Test erweitern (Ohne Schleife/Verzweigung)
  - Mit Bitoperationen!
- Aufgabe 2: Modul-Tests für die Klasse **Field**
- Aufgabe 3: Bewegung auf Feldgitter beschränken
- Abgabe im GitLab der Uni



Praktische Informatik I WiSe 2022/23

Übungsblatt 6

Abgabe: 09.12.2022

Die Klasse *Field* aus Übungsblatt 5 ermittelt u.a. die Nachbarschaften aller Spielfeldzellen. Dies soll nun auch für die Bewegung von Spielfiguren genutzt werden. Zudem wird die Abgabe ab jetzt auf *git* umgestellt, d.h. Abgaben auf anderen Wegen werden **nicht** mehr **akzeptiert**.

Aufgabe 1 Nachbarschaften (20 %)

Erweitert die Klasse *Field* um eine Methode *boolean hasNeighbor(int, int, int)*, wobei die ersten beiden Parameter die x- und y-Koordinaten der getesteten Zelle sind und der dritte Parameter die Richtung, in die getestet wird (0 bis 3 mit der üblichen Bedeutung). Die Koordinaten sind dabei die echten Gitterkoordinaten, nicht die verdoppelten, die intern wegen der Repräsentation des Feldes als String-Array verwendet werden. Die Methode liefert zurück, ob es an der übergebenen Stelle in die übergebene Richtung eine Verbindung zum Nachbarn gibt. Dies soll ohne Schleife oder Verzweigung direkt aus der Rückgabe der bereits vorhandenen Methode *getNeighborhood* mit Hilfe von Bitoperationen bestimmt werden. Erklärt genau, warum eure Implementierung das gewünschte Ergebnis liefert (für die Erklärung gibt es die Hälfte der Punkte).

Aufgabe 2 Nachbarschaftstest (60 %)

Für Übungsblatt 5 wurden noch keine Unit-Tests für die Klasse *Field* implementiert. Dies soll nun nachgeholt werden. Da sich *hasNeighbor* auf die eigentlich privaten Methoden *getCell* und *getNeighborhood* abstützt, kann *Field* über das Zusammenspiel des Konstruktors mit *hasNeighbor* vollständig getestet werden. Es braucht nicht getestet zu werden, ob der Konstruktor tatsächlich die richtigen Grafiken erzeugt.

Aufgabe 3 In geregelten Bahnen (20 %)

Erweitert euer Spiel so, dass sich die Spielfigur und die NPC nur in Richtungen bewegen, die die Gitterstruktur zulässt. Bei den NPC können dadurch Attribute entfallen, da nun keine Schritte mehr gezählt werden müssen.

Abgabe: Richtet unter *gitlab.informatik.uni-bremen.de* ein Repository *pi1-2022* ein und ladet eure Tutor:in dazu als *Developer* ein. Legt dort bereits auf dem Server eine geeignete *.gitignore*-Datei im Hauptverzeichnis des Repositories an. Klont das Repository und legt in eurer Arbeitskopie einen Ordner *loesung06* an, in dem ihr eure Abgabe ablegt. *Commit*-tet eure Abgabe und *push*-t sie auf den GitLab-Server. Die Abgabe soll das enthalten, was bisher die zur Abgabe genutzten *zip*-Dateien enthalten haben, aber eben nicht die Dateien, die durch die *.gitignore*-Datei ausgeschlossen werden.