Karlsruher Institut für Technologie

Software Design and Quality

Programmieren Saalübung WS 24/25

Nils Niehues, Tobias Thirolf, Moritz Rimpf
KASTEL – Institute of Information Security and Dependability

Sommersemester

- **Keine** Vorlesung!
- Übungsbetrieb:
  - Tutorien
  - Übungsblätter
  - Präsenzübung
- Abschlussaufgaben

SS25

Plagiarismus

Es werden nur selbstständig angefertigte Lösungen akzeptiert. [1] [2] [3] [4]

- Nicht erlaubt ist z.B.
  - …die Verwendung von fremden Lösungen
  - …das Weitergeben oder Teilen eigener Lösung
  - …das Generieren von Lösungen (z.B. mit ChatGPT)
- Erlaubt ist z.B.
  - …das Verwenden von Beispiellösungen des **aktuellen** Semesters
  - …das Verwenden eigener Lösungen von den Übungsblättern

[1] §11 Informatik Prüfungsordnung
[2] §19 Satzung zur Sicherung guter wissenschaftlicher Praxis am KIT
[3] Richtlinien der Informatik-Fakultät zu Generativer KI
[4] Plagiarismus-Hinweise auf den Übungsblättern und Abschlussaufgaben

Methodik-Abzüge bei AAs

- Neben Funktionalität wird auch Methodik geprüft
- Großer Teil der Bewertung von Übungsblättern **und** Abschlussaufgaben
- Alle Bewertungskriterien sind im Wiki: *s.kit.edu/wiki*
- Für AAs gelten die Kriterien **aller** bisherigen Blätter

Bewertungsrichtlinien [Einklappen]

Blatt 1 Abzüge
- Einheitliche Sprache
- Systemabhängiger Zeilenumbruch

Blatt 2 Abzüge
- Dokumentation
- Instanceof außerhalb der equals-Methode
- Komplexität
- Leerer Block/Leerer Konstruktor
- Nur Main
- Scanner
- Schlechter Bezeichner
- Schwieriger Code
- Unbenutztes Element

Blatt 3 Abzüge
- Assertions
- Bedeutungslose Konstanten
- Datenkapselung
- Duplikate
- Enum
- Fehlermeldungen
- Final
- Hartcodieren
- Hilfsklasse
- IO/UI
- JavaDoc
- JavaDoc Trivial
- Konstanten-Klasse
- Magic Literal
- Object statt konkreter Klasse
- Pakete
- Polymorphie
- Raw Types
- Reimplementierung
- Runtime Exceptions
- Sichtbarkeit
- Stringreferenzen
- Ungeeigneter Schleifentyp

Blatt 4 Abzüge
- Cast außerhalb der equals-Methode
- Exceptions Kontrollfluss
- Große Try-Catch Blöcke
- Interface statt konkreter Klasse
- Statische Methoden und Attribute
- Verweigerte Vererbung

Häufigste Fehler — Blatt 3

■■■■■■■■□□ 85%	Unnötige Komplexität
■■■■■■□□□□ 61%	Enum/Mengen
■■■■■■□□□□ 56%	Sichtbarkeit
■■■■■□□□□□ 52%	Bedeutungslose Konstanten
■■■■□□□□□□ 44%	Final
■■■■□□□□□□ 40%	Scanner
■■■□□□□□□□ 35%	IO/UI
■■■□□□□□□□ 33%	Unbenutztes Element
■■■□□□□□□□ 28%	Statisches Attribut
■■■□□□□□□□ 28%	Schlechte Bezeichner
■■□□□□□□□□ 21%	Kommentare
■■□□□□□□□□ 21%	JavaDoc Trivial
■■□□□□□□□□ 20%	Falscher Schleifentyp
■■□□□□□□□□ 19%	Schwieriger Code
■■□□□□□□□□ 17%	Getter/Setter für Listen
■■□□□□□□□□ 17%	Statische Methode

Häufigste Fehler — Blatt 4

■■■■■■■■□□ 77%	Unnötige Komplexität
■■■■■■■□□□ 67%	Magic Literal
■■■■■□□□□ 63%	Sichtbarkeit
■■■■■□□□□ 63%	Enum/Mengen
■■■□□□□□□ 44%	Getter/Setter für Listen
■■■□□□□□□ 42%	Final
■■■□□□□□□ 39%	Unbenutztes Element
■■□□□□□□□ 33%	IO/UI
■■□□□□□□□ 33%	Scanner
■■□□□□□□□ 33%	JavaDoc Trivial
■■□□□□□□□ 30%	Interface
■■□□□□□□□ 29%	Statisches Attribut
■■□□□□□□□ 29%	Statische Methode
■■□□□□□□□ 27%	Bedeutungslose Konstantenbezeichner
■■□□□□□□□ 27%	Schlechte Bezeichner
■□□□□□□□□ 22%	Falscher Schleifentyp

Unnötige Komplexität

- Redundanz/Komplexität erhöht visuelles Rauschen
  => einfachste Lösung wählen

```java
boolean isValid() {
    if (this.sold == false && !(this.price <= 0)) {
        return true;
    } else {
        return false;
    }
}
```

```java
boolean isValid() {
    return !this.sold && this.price > 0;
}
```

Sichtbarkeit und Datenkapselung

- Die Sichtbarkeit sollte so restriktiv wie möglich gesetzt werden
- Interner Zustand soll nur, falls wirklich nötig, herausgegeben werden
- Die Möglichkeit zu ungeplanten Zugriff führt zu Bugs

```
public final class Book {
    public String title;
    long isbn;

    protected Book(String title, long isbn) { // protected sinnlos, da Klasse final
        if (!isIsbnValid(isbn)) {
            throw new InvalidIsbnException("The isbn is not valid");
        }
        this.title = title;
        this.isbn = isbn;
    }

    public boolean isIsbnValid(long isbn) { // interne Hilfsmethode
        // ...
    }
}
```

Sichtbarkeit und Datenkapselung

- Die Sichtbarkeit sollte so restriktiv wie möglich gesetzt werden
- Interner Zustand soll nur, falls wirklich nötig, herausgegeben werden
- Die Möglichkeit zu ungeplanten Zugriff führt zu Bugs

```
public final class Book {
    private String title;
    private long isbn;
    public Book(String title, long isbn) {
        if (!isIsbnValid(isbn)) {
            throw new InvalidIsbnException("The isbn is not valid");
        }
        this.title = title;
        this.isbn = isbn;
    }
    public String getTitle() { return this.title; }
    public long getIsbn() { return this.isbn;}
    private boolean isIsbnValid(long isbn) {
        // ...
    }
}
```

Zugriffsmethoden für Listen

- Zugriffsmethoden auf Listen o.ä. dürfen nicht die Kapselung verletzen
- Dies gilt für den Zugriff auf alle Arrays, Collections und Maps

```
private List<String> names;

public void setNames(List<String> names) {
   this.names = names;
}

public List<String> getNames() {
   return names;
}
```

Zugriffsmethoden für Listen

- Zugriffsmethoden auf Listen o.ä. dürfen nicht die Kapselung verletzen
- Dies gilt für den Zugriff auf alle Arrays, Collections und Maps

```
private List<String> names;

public void setNames(List<String> names) {
    this.names = new ArrayList<>(names);
}

public List<String> getNames() {
    return new ArrayList<>(names); // Also possible: immutable list
}

public void addName(String name) {
    names.add(name); // Fine-grained access is preferable
}
```

Final-Modifizierer

- Attribute sollten immer, wenn möglich, final sein
- Verhindert unabsichtliche Änderungen der Attribute (nicht ihrer Zustände)

```
class Event {
    private static int BASE_FEE = 1000;
    private List<String> participants;
    public Event() {
        participants = new ArrayList<>();
    }
    public void addParticipant(String name) {
        participants.add(name);
    }
    public int calculateProfit(int ticketPrice, int venueCost) {
        int fixedCost = BASE_FEE + venueCost;
        return participants.size() * ticketPrice - fixedCost;
    }
}
```

Final-Modifizierer

- Attribute sollten immer, wenn möglich, final sein
- Verhindert unabsichtliche Änderungen der Attribute (nicht ihrer Zustände)

```
class Event {
    private final static int BASE_FEE = 1000; // constants should be final
    private final List<String> participants; // fields should be final if possible
    public Event() {
        participants = new ArrayList<>();
    }
    public void addParticipant(String name) { // not required for parameters or local variables
        participants.add(name); // (list is final, but not the content)
    }
    public int calculateProfit(int ticketPrice, int venueCost) {
        int fixedCost = BASE_FEE + venueCost;
        return participants.size() * ticketPrice - fixedCost;
    }
}
```

Karlsruher Institut für Technologie

Software Design and Quality

Programmieren Saalübung WS 24/25

Nils Niehues, Tobias Thirolf, **Moritz Rimpf**

KASTEL – Institute of Information Security and Dependability

Wie werden Testfälle gebaut?

- Aus den selben Informationen gelesen, die Ihnen zur Verfügung stehen:
  - Übungsblatt
  - Genaues lesen lohnt sich also

- Artemis-Tests testen Funktionale Anforderungen
  - Funktionen
  - Fehlerbehandlung
  - Randfälle

- Methodik wird manuell in Korrektur nach Abgabe bewertet

Basis-Testfälle

- Beispielinteraktion

Beispielinteraktion

```
> start X1V
OK
> place I1R 2 X1V
OK
> place I1V 6 X1V
OK
> place E1R 1 I1R
OK
> place E1V 6 I1V
OK
> print
E1R 4 I1R
I1R 1 E1R 5 X1V
X1V 2 I1R 6 I1V
E1V 3 I1V
I1V 3 X1V 6 E1V
> place S2R 1 E1R
OK
> pass
OK
> print
S2R 4 E1R
E1R 1 S2R 4 I1R
I1R 1 E1R 5 X1V
X1V 2 I1R 6 I1V
E1V 3 I1V
I1V 3 X1V 6 E1V
> place S1R 6 S2R
Error, place not allowed
> place X1R 6 E1R
OK
> place A1V 6 E1V
OK
> print
X1R 2 S2R 3 E1R
S2R 4 E1R 5 X1R
E1R 1 S2R 4 I1R 6 X1R
I1R 1 E1R 5 X1V
X1V 2 I1R 6 I1V
A1V 3 E1V
E1V 3 I1V 6 A1V
I1V 3 X1V 6 E1V
> place S1R 6 S2R
OK
> move A1V 0 A1R
```

Beispielinteraktion

```
Error, move not allowed
> pass
OK
> move X1R 5 S1R
OK
> pass
OK
> place A1R 6 X1R
OK
> pass
OK
> move A1R 5 X1R
OK
> move A1V 0 A1R
OK
> print
X1R 2 S1R 5 A1V
A1R 0 A1V 2 X1R 4 E1V
S1R 3 S2R 5 X1R
S2R 4 E1R 6 S1R
E1R 1 S2R 4 I1R
I1R 1 E1R 5 X1V
X1V 2 I1R 6 I1V
A1V 2 X1R 4 E1V
E1V 1 A1V 3 I1V
I1V 3 X1V 6 E1V
> pass
OK
> move A1V 1 I1V
OK
> pass
OK
> move A1V 3 I1R
Error, move not allowed
> move A1V 1 X1V
OK
> pass
OK
> place S1V 5 I1V
OK
> pass
OK
> move S1V 2 E1R
OK
```

Beispielinteraktion

```
> print
X1R 2 S1R 5 A1R
A1R 2 X1R 4 E1V
S1R 3 S2R 5 X1R
S2R 3 S1V 4 E1R 6 S1R
E1R 1 S2R 2 S1V 4 I1R 5 A1V
I1R 1 E1R 5 X1V 6 A1V
X1V 1 A1V 2 I1R 6 I1V
A1V 2 E1R 3 I1R 4 X1V 5 I1V
S1V 5 E1R 6 S2R
E1V 1 A1R 3 I1V
I1V 2 A1V 3 X1V 6 E1V
> place E2R 1 S1R
OK
> pass
OK
> move E2R 1 S2R 1 S1V 2 S1V
OK
> move S1V 5 X1V
Error, move not allowed
> pass
OK
> move E1R 4 X1V
Error, move not allowed
> move X1R 1 A1V
OK
> print
X1R 1 S1R 2 S2R 3 E1R 4 A1V
A1R 4 E1V
S1R 3 S2R 4 X1R
S2R 3 S1V 4 E1R 5 X1R 6 S1R
E1R 1 S2R 2 S1V 4 I1R 5 A1V 6 X1R
E2R 5 S1V
I1R 1 E1R 5 X1V 6 A1V
X1V 1 A1V 2 I1R 6 I1V
A1V 1 X1R 2 E1R 3 I1R 4 X1V 5 I1V
S1V 2 E2R 5 E1R 6 S2R
E1V 1 A1R 3 I1V
I1V 2 A1V 3 X1V 6 E1V
> pass
OK
> move A1R 2 E1V
OK
> move I1V 4 E1V 5 E1V 6 E1V 1 E1V 1 A1R
WINNER ULTRAVIOLET
> quit
```

Basis-Testfälle

- Spiele
  - Alle „realistischen Spielabläufe“
    - Spieler 1 Sieg
    - Spieler 2 Sieg
    - Unentschieden

> Wenn beide Spieler allerdings direkt hintereinander passen endet das Spiel unentschieden.

5A, „Jagd nach Mister X“

- CLI-Programme
  - Kommandos einzeln
    - Soweit möglich
    - Manche Kommandos haben andere als „Voraussetzung“

```
>new 36 15
[…]
>debug
[…]
```

4A, „Perserverance“

- Interface-Aufgaben
  - Methoden einzeln
    - Soweit möglich

```
// Test A:
StringUtility.removeVowels(„Hello World“)

// Test B:
StringUtility.wordCount(„Hello World“)
```

2A, „String-Utility“

    - Auch hier Voraussetzungen oder Abhängigkeiten möglich

```
AdventCalendar.getDay() // => 0
AdventCalendar.nextDay()
AdventCalendar.getDay() // => 1
```

3B, „Adventskalender“

Einzelne Funktionalitäten

A.3 Arten von Geheimdienstlern

Jede Art von Geheimdienstler (Spielstein) hat ihre eigenen Regeln, nach denen er ziehen darf.

A.2.3 Einsetzen von Mister X

Mister X kann in einem der ersten vier Züge eingesetzt werden. Spätestens im vierten Zug muss er eingesetzt worden sein.

A.2.4 Bewegung der Spielsteine

Bis zum Einsetzen des Mister X dürfen andere Spielsteine zwar eingesetzt, aber nicht bewegt werden. Erst wenn der eigene Mister X im Spiel ist, kann ein Spieler bei jedem Zug entscheiden, ob er einen neuen Spielstein einsetzt oder einen bereits vorhandenen bewegt.

A.6.2 Der `start`-Befehl

Der Befehl startet ein neues aktives Spiel. Wird der Befehl während eines laufenden Spiels eingebeben, wird dieses abgebrochen und ein neues Spiel gestartet.

Randfälle

A.2.1 Einsetzen der Spielsteine

Ein neuer Spielstein kann bei jedem Zug eingesetzt werden. Außer im ersten Spielzug darf ein Spielstein niemals angrenzend an einen gegnerischen Spielstein eingesetzt werden.

A.4 Einschränkungen

A.4.1 Der Schauplatz

Die Spielsteine auf dem Spielfeld müssen jederzeit miteinander verbunden sein. Kein Spielstein darf alleine (ohne Verbindung zum Schauplatz) sein. Der Schauplatz darf auch nicht geteilt werden.

- Es müssen mindestens 2.147.483.647 initiale Lebenspunkte (`<Punkte>`) unterstützt werden.

- Nicht zu prüfen:

Sie können davon ausgehen, dass alle Vektoren zweidimensional sind. Sie können davon ausgehen, dass `args` nicht null und immer mindestens drei Argumente enthält (der Grad und min. zwei Vektoren) und kein Vektor doppelt auftritt. Sie können zudem davon ausgehen, dass die textuelle Darstellung der Argumente korrekt ist und müssen dies nicht überprüfen.

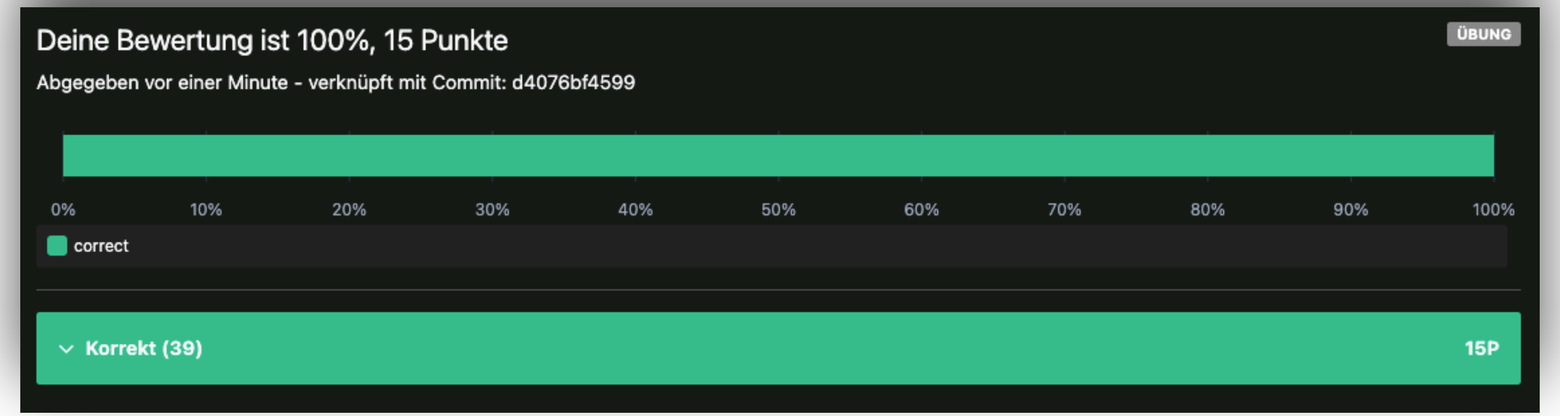
IO-Testfälle

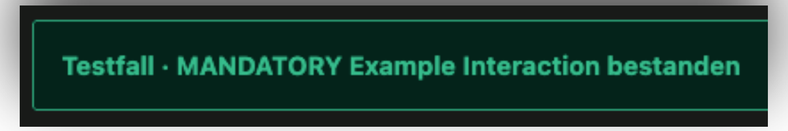
- Einzelne Befehle mit…
  - zu vielen Argumenten (z.B. „`quit argumentA argumentB`„)
  - zu wenigen Argumenten (z.B. „`start`„)
  - nicht definierten Argumenten (z.B. „`place X1R 4 NonexistantToken`„)
  - ungültigen Argumenten (z.B. „`place X1R 4 I1R`„, obwohl X1R bereits platziert)

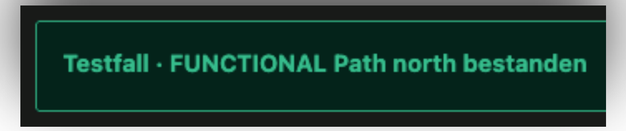
- Kommandos in ungültiger Reihenfolge
  - z.B. „`place`„ vor „`start`„

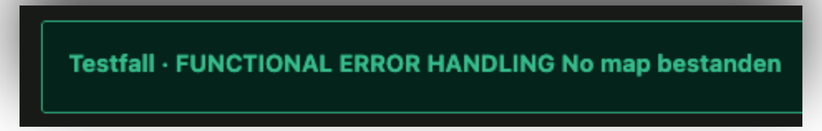
- Fehlernachrichten müssen korrektem Schema entsprechen!
  - z.B. „`Error, `„
  - z.B. „`ERROR: `„

Testfälle auf Artemis



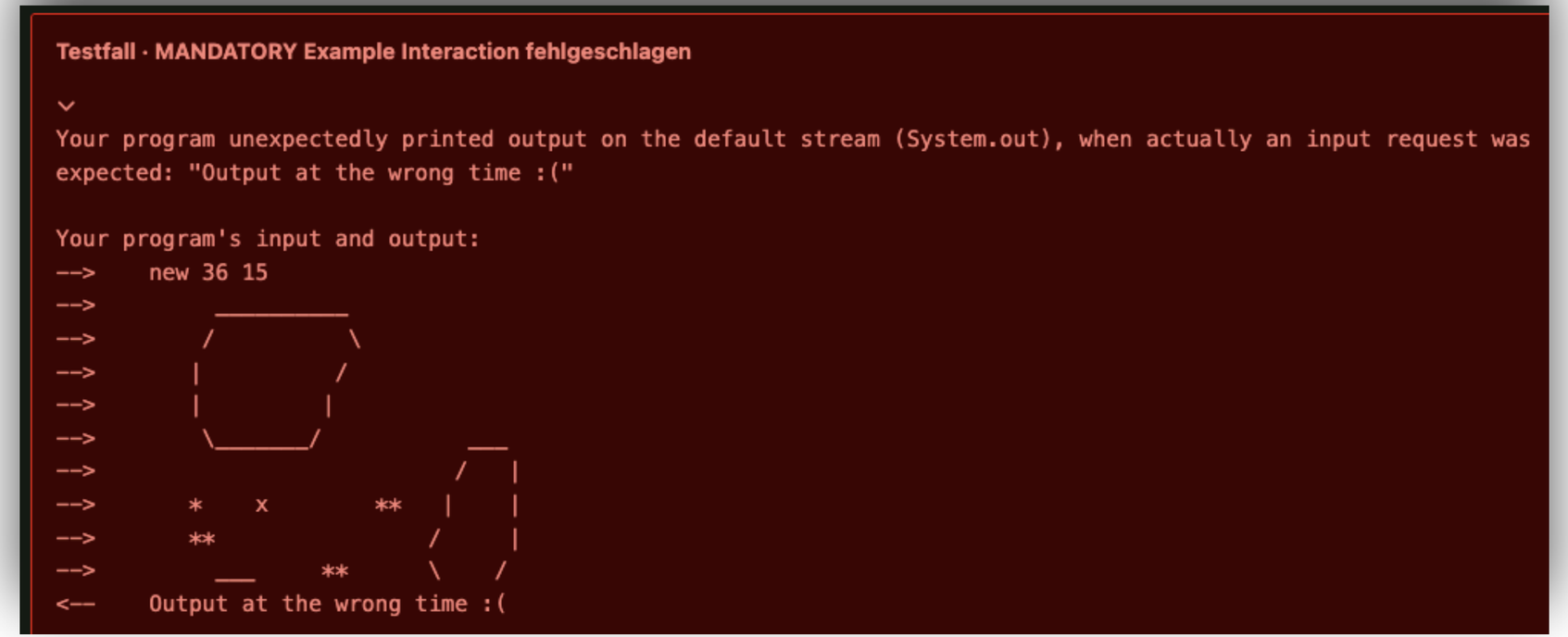






Fehlernachrichten auf Artemis

Fehlernachrichten auf Artemis



Fehlernachrichten auf Artemis

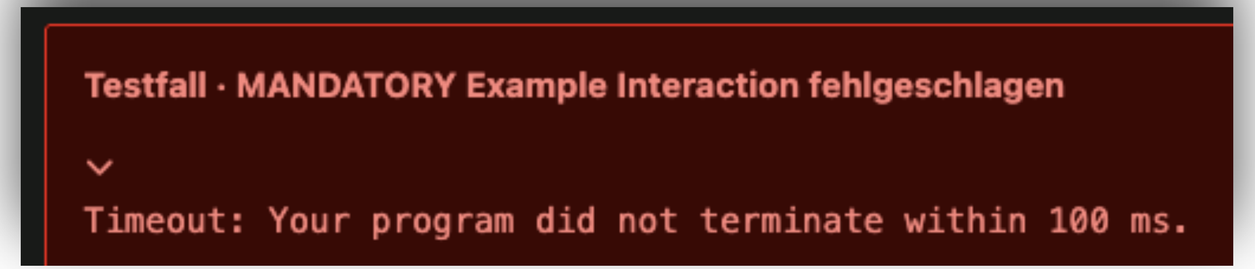
Fehlernachrichten auf Artemis

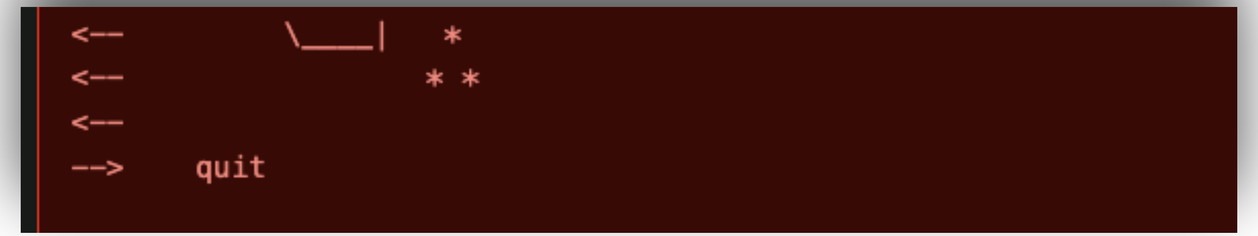
Testfall · MANDATORY Example Interaction fehlgeschlagen

```
Your program unexpectedly printed output on the default stream (System.out), when it should have terminated instead:
"Look at me, I'm quitting!"
```

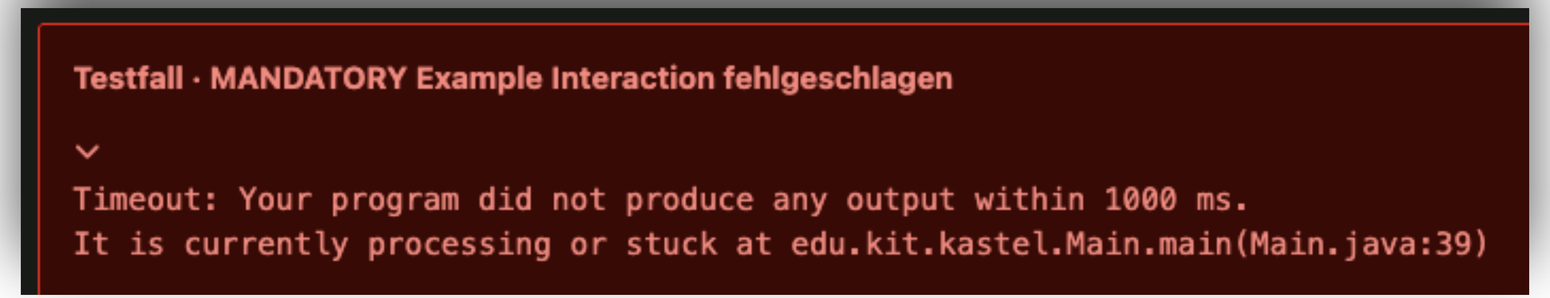
```
-->    quit
<--    Look at me, I'm quitting!
<--    Look at me, I'm quitting!
```

Fehlernachrichten auf Artemis

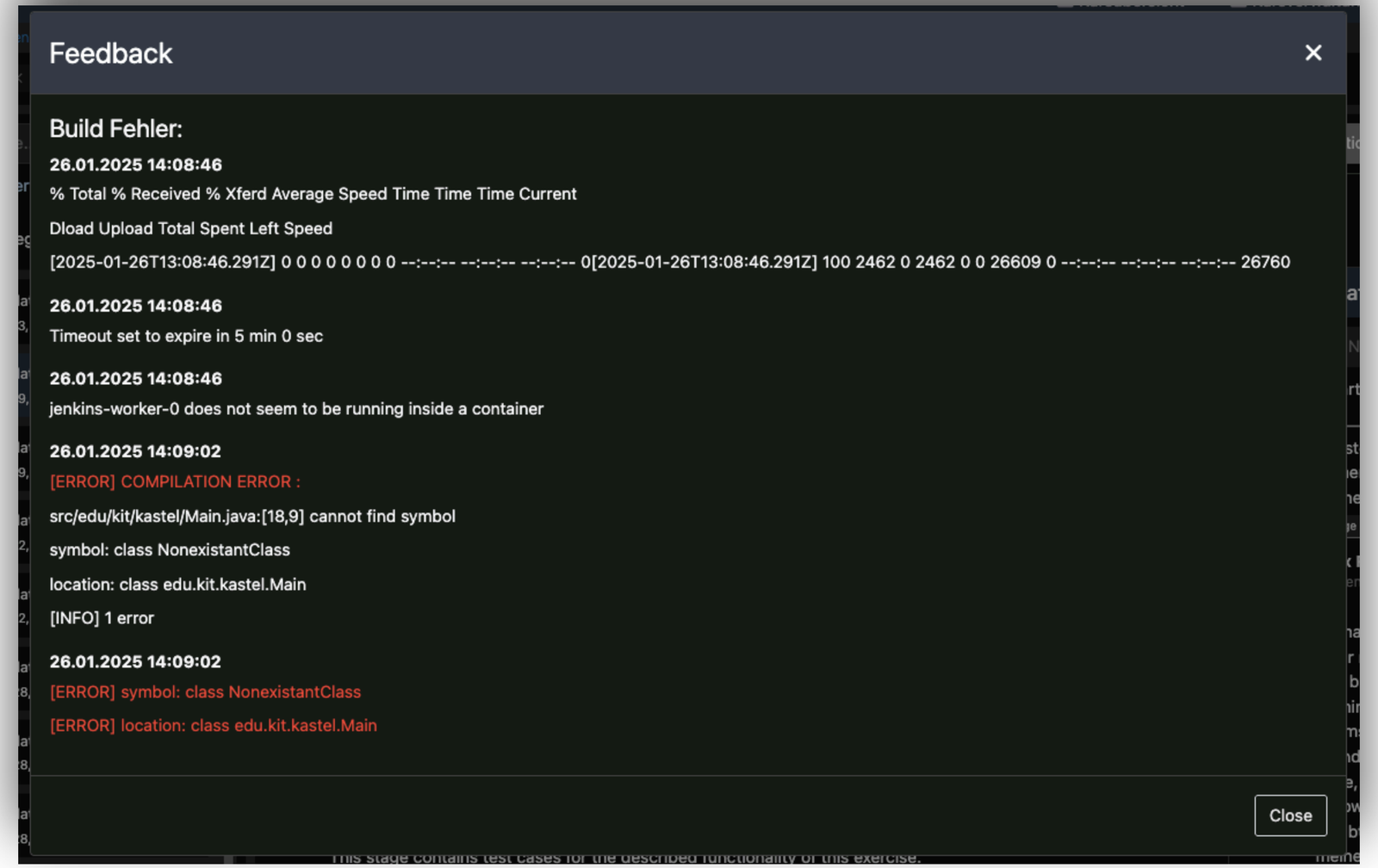




Fehlernachrichten auf Artemis



Fehlernachrichten auf Artemis



Fragen stellen & beantwortet bekommen

- Grundsätzlich: Gerne Fragen stellen

- Vor neuer Frage stellen: Artemis Forum durchsuchen!
  - Gleiche oder ähnliche Frage möglicherweise bereits gestellt
  - Neue Fragen werden eher priorisiert als Duplikate

- Fragen richtig stellen

Fragen richtig stellen

0 Zugstein wird auf den Zielstein (Oben) gelegt.

1 Zugstein wird an die oberste Kante (Norden) des Zielsteins gelegt.

2 Zugstein wird an die linke oberste Kante (Nordost) des Zielsteins gelegt.

3 Zugstein wird an die linke untere Kante (Südost) des Zielsteins gelegt.

4 Zugstein wird an die unterste Kante (Süden) des Zielsteins gelegt.

5 Zugstein wird an die rechte untere Kante (Südwest) des Zielsteins gelegt.

6 Zugstein wird an die rechte oberste Kante (Nordwest) des Zielsteins gelegt.

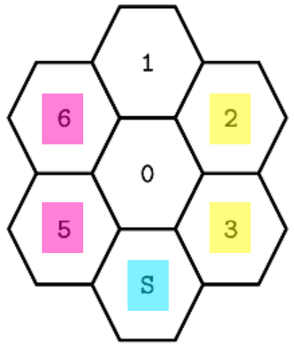


Abbildung A.2: Die sieben Berührungskanten eines Spielsteins.

Fragen richtig stellen

- Schlecht:
  - „Ich verstehe Abbildung A.2 nicht.“
    - Antwort: „Lesen sie das Übungsblatt aufmerksam durch, dort sind alle Informationen enthalten“

0 Zugstein wird auf den Zielstein (Oben) gelegt.
1 Zugstein wird an die oberste Kante (Norden) des Zielsteins gelegt.
2 Zugstein wird an die linke oberste Kante (Nordost) des Zielsteins gelegt.
3 Zugstein wird an die linke untere Kante (Südost) des Zielsteins gelegt.
4 Zugstein wird an die unterste Kante (Süden) des Zielsteins gelegt.
5 Zugstein wird an die rechte untere Kante (Südwest) des Zielsteins gelegt.
6 Zugstein wird an die rechte oberste Kante (Nordwest) des Zielsteins gelegt.

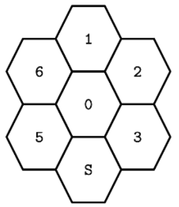


Abbildung A.2: Die sieben Berührungskanten eines Spielsteins.

- Gut:
  - „In Abbildung A.2 sind 2&3 rechts und 5&6 links, aber in dem Text darüber ist dies umgekehrt. Ist dies beabsichtigt und welche Angabe ist korrekt?“

Fragen richtig stellen

- Annahmen mit in Frage schreiben
- Schlecht:
  - „Testfall „Draw“ hat einen Fehler.“
    - Antwort: „Nein.“
- Besser:
  - „Im Testfall „Draw“ wird „I3R 2 I2R 3 X1R“ erwartet, obwohl der Output „I3R 6 I2R 5 X1R“ sein müsste.“
    - Antwort: „Output „I3R 2 I2R 3 X1R“ ist korrekt.“
- Gut:
  - „Im Testfall „Draw“ wird in Zeile 34 „I3R 2 I2R 3 X1R“ erwartet, obwohl der Output nach Abbildung A.2 „I3R 6 I2R 5 X1R“ sein müsste.“
    - Antwort: „Output „I3R 2 I2R 3 X1R“ ist korrekt, Abbildung A.2 ist fehlerhaft.“

0 Zugstein wird auf den Zielstein (Oben) gelegt.
1 Zugstein wird an die oberste Kante (Norden) des Zielsteins gelegt.
2 Zugstein wird an die linke oberste Kante (Nordost) des Zielsteins gelegt.
3 Zugstein wird an die linke untere Kante (Südost) des Zielsteins gelegt.
4 Zugstein wird an die unterste Kante (Süden) des Zielsteins gelegt.
5 Zugstein wird an die rechte untere Kante (Südwest) des Zielsteins gelegt.
6 Zugstein wird an die rechte oberste Kante (Nordwest) des Zielsteins gelegt.

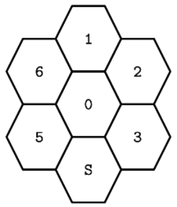
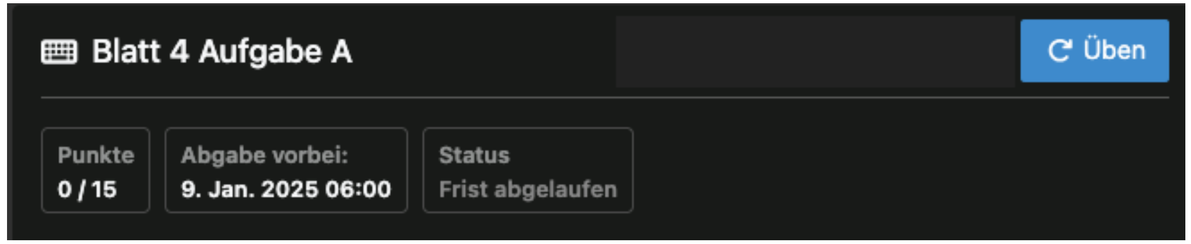


Abbildung A.2: Die sieben Berührungskanten eines Spielsteins.

Weitere Vorbereitung auf die Abschlussaufgaben

- Übungsmodus auf Artemis



- Alte Beispiellösungen lesen & verstehen

- Mögliche Erweiterungen für aktuelle Aufgaben (Blatt 4, Blatt 5) überlegen / implementieren
  - z.B. mehrere Spiele parallel laufen lassen

Karlsruher Institut für Technologie

Software Design and Quality

Programmieren Saalübung WS 24/25

Nils Niehues, **Tobias Thirolf**, Moritz Rimpf
KASTEL – Institute of Information Security and Dependability

Informationsträger

Grundlegender Gedanke eines Programms:

- Verarbeitung von Information

- Code ist dafür verantwortlich Informationen weiterzureichen
- **Guter** Code stellt Informationen anschaulich für Programmierer dar

- Objektorientierung nutzt typisierte Informationsträger durch Klassen

Insbesondere das Modell/die Logik sollte so wenig wie möglich mit *String*s, oder primitiven Datentypen arbeiten.

Implizite Informationsträger

Gängige Praktiken:

- (Nicht-) Existenz eines Objektes ausnutzen
  - Achtung: unerwartetes **null** führt oft zu Fehlern
- Mehrere stark gekoppelte Informationen als (**private**) Klasse wrappen
  - z. B. als **Record** (Kapselung beachten)
  - Siehe Beispiellösung 5A *Result* Klasse
- Exceptions für Fehlerzustand
  - Achtung: wird gerne missbraucht, um Kontrollfluss zu steuern!

Exceptions

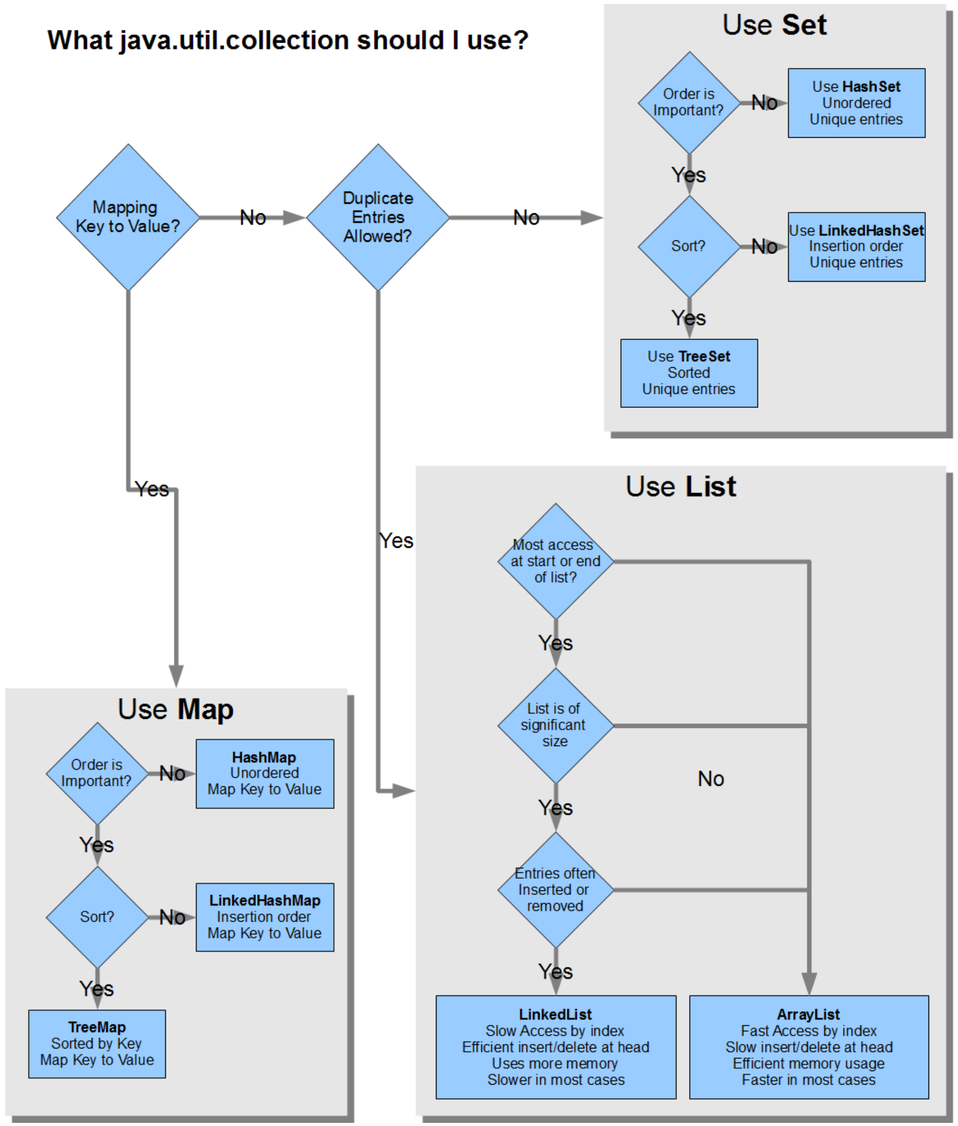
Sinnvolle Verwendung (alles sollte erfüllt sein):

- Fehlerzustand ist zu komplex, um Rückgabewert miteinzubeziehen
- Exception wird so früh wie möglich **sinnvoll** verarbeitet
- Der Typ der Exception stellt den Fehlerzustand dar, nicht die Message
- Nachvollziehbar welche Codestelle welche Exception auslöst
- Der Code an der Stelle nicht **sinnvoll** weitergeführt werden kann

*Beispiele für **schlechte** Verwendung:*

- Die gleiche Exception für jedes Problem werfen/fangen
- Exception, statt **sinnvolle** typisierte Lösung
  - z. B. beim Beenden des Programms, oder logisch falschen Eingaben

Collections



https://stackoverflow.com/questions/21974361/which-java-collection-should-i-use

Magic Literals

Motivation:

- Literale, die eine Bedeutung besitzen (abhängig sind von einem Kontext) sollten nicht frei im Code definiert sein

Mögliche Prüfung:

- *Kann sich die Zahl ändern, indem die Aufgabenstellung/der Kontext verändert wird?*

Z. B. nicht “magic”:

- Schleifeninitialisierung mit erstem Index -> durch Java vorgegeben

- Lediglich bei Zahlen gibt es Ausnahmen sowie der leere String (“”)

Allgemeine Tipps

- Von Anfang an **Benutzerinteraktion** und **Logik** durch Pakete trennen
- Im Logik Paket so wenig wie möglich mit **Strings** arbeiten
- Mit **Checkstyle** und **Inspection Profil** am *Ende* aufräumen
- Beispiellösungen verstehen/verwenden
- Problem visualisieren
- Frühzeitig beginnen!

Strukturiertes Vorgehen - Vorbereitung

Ziel:

- Vermeidung von zusätzlichem Arbeitsaufwand
- Hilfestellung bei Problemen

- Vergleichen der Aufgabe mit Übungsaufgaben
  - Wird eine Benutzerinteraktion benötigt?
  - Wurden vergleichbare algorithmische Probleme bereits gelöst?
  - Gibt es ähnliche Strukturen?

- Hilfstools vorbereiten

Hinweis: Die **Beispiellösungen** können hierzu hilfreich sein :)

Strukturiertes Vorgehen - Ansätze

Grundansätze beim Programmieren:

- **Divide and Conquer** (schon aus Vorlesung bekannt)
- **Dummy Methods** abstrahieren ein ungelöstes Problem
- **“Rückwärts Programmieren”** schränkt den Lösungsraum ein

• Die Ansätze können gemischt und willkürlich angewandt werden

Denken Sie an Bausteine, die zusammengesetzt komplexe Strukturen ergeben. Programmieren ist die Erstellung dieser Bausteine sowie die Ausarbeitung der Anleitung zum Zusammenbauen.

Strukturiertes Vorgehen - Dummy Methods

Ziel: Abstraktion eines Problems

Anwendung:

- Beschreibe ungelöstes **Teil**problem als Blackbox durch Eingabe (*Parameter*) und Ausgabe (*Rückgabewert*)

- Methodensignatur allein ausreichend, um Methode zu verwenden
- Ermöglicht den Fokus auf das übergeordnete Problem nicht zu verlieren
- Lagert ein Problem als wiederverwendbaren Baustein aus

Strukturiertes Vorgehen - “Rückwärts Programmieren”

Ziel: Einschränken des Lösungsraums durch Definition von Gegebenheiten

Anwendung:

- Lösbares **Teil**problem (ggf. unabhängig) implementieren

- Methodensignatur gibt danach Gegebenheiten für Verwendung vor
- Übergeordnetes Problem reduziert sich darauf Gegebenheiten zu erfüllen
- Lagert ein Problem als wiederverwendbaren Baustein aus

- Implementierte Methode gibt vor, was benötigt wird, um **Teil**problem zu lösen
- Sozusagen vom Ziel ausgehend “rückwärts” zum Start

Strukturiertes Vorgehen - Tipps

- Agiles Arbeiten
  - Eigene Einschränkungen dienen als Orientierung und sind ggf. anzupassen

- Überblick behalten
  - Welche Funktionalität wurde bereits implementiert? Was fehlt noch?
  - Objektorientierte Aufteilung in Klassen

- Erlaubte Materialien als Referenzen und Hinweise nicht vernachlässigen

Wie identifiziert man ein Problem?

Grundkonzept:

- Vergleiche Erwartung mit tatsächlichem Wert
- Erstes Auftreten einer Unstimmigkeit ist mögliche Ursache
- Ursache für kritische Unstimmigkeit ist Ursache des Problems

Häufige Fehlerquelle:

- Falsche Annahme -> Annahmen überprüfen/hinterfragen!

Besonders nützlich:

- Debugger
- *Redirect Input* bei vielen Nutzereingaben