Einführung in die Objektorientierte Programmierung
Vorlesung 22: Fehlerbehandlung & -Vermeidung

Sebastian Küpper

Fehlererkennung und Beseitigung: Ein teures Unterfangen

- World Testing Survey 2015: 45% der Software-Projekte beanspruchen 30% der Zeit für Tests
- Nicht erkannte Fehler können hohe finanzielle Kosten oder gar Lebensgefahr bedeuten

Fehlererkennung und Beseitigung: Ein teures Unterfangen

- World Testing Survey 2015: 45% der Software-Projekte beanspruchen 30% der Zeit für Tests
- Nicht erkannte Fehler können hohe finanzielle Kosten oder gar Lebensgefahr bedeuten
- Drei Ansätze zur Fehlerbehandlung:
  - Verifikation:Korrektheit der Software beweisen ($\Rightarrow$ Seminar Modellierung und Verifikation)
  - Testen:Systematisch Fehler suchen und beheben ($\Rightarrow$ Kapitel 41, Software-Technik-Veranstaltungen)
  - Fehlervermeidung:Häufige Fehler kennen und von vorneherein vermeiden (Thema dieser Einheit)

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden
- Schreibfehler in Variablen- Klassen und Methodennamen

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden
- Schreibfehler in Variablen- Klassen und Methodennamen
- Eine Anweisung ist nicht mit Semikolon beendet

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden
- Schreibfehler in Variablen- Klassen und Methodennamen
- Eine Anweisung ist nicht mit Semikolon beendet
- Fehlende öffnende / schließende Klammern in arithmetischen oder booleschen Ausdrücken

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden
- Schreibfehler in Variablen- Klassen und Methodennamen
- Eine Anweisung ist nicht mit Semikolon beendet
- Fehlende öffnende / schließende Klammern in arithmetischen oder booleschen Ausdrücken
- Fehlende explizite Typumwandlung, bes. bei nicht verlustfreien arithmetischen Zuweisungen

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden
- Schreibfehler in Variablen- Klassen und Methodennamen
- Eine Anweisung ist nicht mit Semikolon beendet
- Fehlende öffnende / schließende Klammern in arithmetischen oder booleschen Ausdrücken
- Fehlende explizite Typumwandlung, bes. bei nicht verlustfreien arithmetischen Zuweisungen
- Ein Feld wurde nicht initialisiert aber zugegriffen: `String[] namen; namen[3] = "Kraemer";`

Häufige Programmierfehler (Compilezeit)

- Ein Paket oder eine Klasse vergessen einzubinden
- Schreibfehler in Variablen- Klassen und Methodennamen
- Eine Anweisung ist nicht mit Semikolon beendet
- Fehlende öffnende / schließende Klammern in arithmetischen oder booleschen Ausdrücken
- Fehlende explizite Typumwandlung, bes. bei nicht verlustfreien arithmetischen Zuweisungen
- Ein Feld wurde nicht initialisiert aber zugegriffen: `String[] namen; namen[3] = "Kraemer";`
- Eine Methode der Oberklasse wird unbeabsichtigt nicht überschrieben (abweichende Signatur) und mit dem Keyword `@Overrride` versehen

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:

```
for (int i = 0; i < LOOPS; i++); { doSomething(); }
```

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:
  `for (int i = 0; i < LOOPS; i++); { doSomething(); }`
- Zuweisungsoperator (=) an Stellen verwendet, an denen Vergleichsoperator (==) gedacht ist: `if(b=true)...`

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:
  `for (int i = 0; i < LOOPS; i++); { doSomething(); }`
- Zuweisungsoperator ($=$) an Stellen verwendet, an denen Vergleichsoperator ($==$) gedacht ist: `if(b=true)...`
- Vergleich von Gleitkommazahlen mit $==$ oder $!=$: Nur Näherungen, besser ε-Vergleich

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:
  `for (int i = 0; i < LOOPS; i++); { doSomething(); }`
- Zuweisungsoperator (=) an Stellen verwendet, an denen Vergleichsoperator (==) gedacht ist: `if(b=true)...`
- Vergleich von Gleitkommazahlen mit == oder ! =: Nur Näherungen, besser ε-Vergleich
- Zeichenketten oder Objekte mit == verglichen wo `equals` gedacht ist

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:
  `for (int i = 0; i < LOOPS; i++); { doSomething(); }`
- Zuweisungsoperator (=) an Stellen verwendet, an denen Vergleichsoperator (==) gedacht ist: `if(b=true)...`
- Vergleich von Gleitkommazahlen mit == oder ! =: Nur Näherungen, besser ε-Vergleich
- Zeichenketten oder Objekte mit == verglichen wo `equals` gedacht ist
- Fehlende Klammern in arithmetischen oder boolschen Ausdrücken

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:
  `for (int i = 0; i < LOOPS; i++); { doSomething(); }`
- Zuweisungsoperator (=) an Stellen verwendet, an denen Vergleichsoperator (==) gedacht ist: `if(b=true)...`
- Vergleich von Gleitkommazahlen mit == oder ! =: Nur Näherungen, besser ε-Vergleich
- Zeichenketten oder Objekte mit == verglichen wo `equals` gedacht ist
- Fehlende Klammern in arithmetischen oder boolschen Ausdrücken
- Versehentliche Ganzzahldivision (z.B. `float x = 1 / 2;`)

Häufige Programmierfehler (Laufzeit)

- Geschweifte Klammern um Anweisungsblöcke fehlen
- Zusätzliches Semikolon, das eine Anweisung vorzeitig beendet:
  `for (int i = 0; i < LOOPS; i++); { doSomething(); }`
- Zuweisungsoperator (=) an Stellen verwendet, an denen Vergleichsoperator (==) gedacht ist: `if(b=true)...`
- Vergleich von Gleitkommazahlen mit == oder ! =: Nur Näherungen, besser ε-Vergleich
- Zeichenketten oder Objekte mit == verglichen wo `equals` gedacht ist
- Fehlende Klammern in arithmetischen oder boolschen Ausdrücken
- Versehentliche Ganzzahldivision (z.B. `float x = 1 / 2;`)
- Schleifenvariable wird nicht angepasst, z.B. in `while(i<5) doSomething();`

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`
- Bei einer `do-while`-Schleife nicht berücksichtigt, dass sie mindestens ein Mal ausgeführt wird (kann z.B. zu `NullPointerExceptions` führen)

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`
- Bei einer `do-while`-Schleife nicht berücksichtigt, dass sie mindestens ein Mal ausgeführt wird (kann z.B. zu `NullPointerExceptions` führen)
- Ein `break` oder `continue` fehlt

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`
- Bei einer `do-while`-Schleife nicht berücksichtigt, dass sie mindestens ein Mal ausgeführt wird (kann z.B. zu `NullPointerExceptions` führen)
- Ein `break` oder `continue` fehlt
- `break` und `continue` wurden verwechselt

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`
- Bei einer `do-while`-Schleife nicht berücksichtigt, dass sie mindestens ein Mal ausgeführt wird (kann z.B. zu `NullPointerExceptions` führen)
- Ein `break` oder `continue` fehlt
- `break` und `continue` wurden verwechselt
- Zugriff auf einen illegalen Feldindex, z.B. `int[] zahlen = new int[3]; zahlen[3] = 0;`

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`
- Bei einer `do-while`-Schleife nicht berücksichtigt, dass sie mindestens ein Mal ausgeführt wird (kann z.B. zu `NullPointerExceptions` führen)
- Ein `break` oder `continue` fehlt
- `break` und `continue` wurden verwechselt
- Zugriff auf einen illegalen Feldindex, z.B. `int[] zahlen = new int[3]; zahlen[3] = 0;`
- Eine Methode der Oberklasse wird unbeabsichtigt überschrieben

Häufige Programmierfehler (Laufzeit)

- Versehentlich wird Schleifenvariable einer umgebenden Schleife verändert: `for(int i=0; i<5; ++i) for(int j=0; j<5;++i) doSomething();`
- Bei einer `do-while`-Schleife nicht berücksichtigt, dass sie mindestens ein Mal ausgeführt wird (kann z.B. zu `NullPointerExceptions` führen)
- Ein `break` oder `continue` fehlt
- `break` und `continue` wurden verwechselt
- Zugriff auf einen illegalen Feldindex, z.B. `int[] zahlen = new int[3]; zahlen[3] = 0;`
- Eine Methode der Oberklasse wird unbeabsichtigt überschrieben
- Eine Methode der Oberklasse wird unbeabsichtigt nicht überschrieben (abweichende Signatur)

Strategien zur Fehlerfindung

Es gibt einen Fehler im Programm aber Sie wissen nicht wo?
Folgende Stragien können bei der Lokalisation helfen:

- Testausgaben
- Programmteile ausschalten
- Verwendung eines Debuggers

Testausgaben

An verschiedenen Punkten im Programm Ausgaben erzeugen, die anzeigen

- dass die entsprechende Stelle im Code ausgeführt wird
- welche Werte bestimmte Variablen haben

```
for (int i = 0; i < LOOPS; i++) {
    ...
    a = ...
    System.out.println("i hat den Wert " + i + ", a hat "
        + "den Wert " + a);
}
```

Programmteile ausschalten

Einzelne Zeilen oder Methodenaufrufe auskommentieren um die problematische Befehlsgruppe einzugrenzen

```
public void methodeA() {
    ...
    this.methodeB();
    // this.methodeC();
    // tritt das Problem auch auf,
    // wenn methodeC() nicht
    // ausgefuehrt wird?
}
```

Verwendung eines Debuggers

Debugger ermöglichen es:

- Stopppunkte zu definieren
- Den Code schrittweise zu durchlaufen
- Attributwerte, Klassenvariablen und lokale Variablen zum Ausführungszeitpunkt einzusehen

Allgemeine Strategie zur Fehlerbeseitigung

1. Fehler reproduzieren durch Nachvollziehen der Eingaben
2. Fehlerstelle durch Auskommentieren oder mittels Debugger lokalisieren
3. Fehler erkennen: Ermitteln, welche Werte Variablen vor und nach Befehl haben sollten; vergleichen mit ist-Werten
4. Fehler verstehen durch Code-Analyse

Allgemeine Strategie zur Fehlervermeidung

- Aussagekräftige Variablennamen wählen
- Ausdrücke übersichtlich halten und ggf. in Teilausdrücke unterteilen
- Jede Anweisung in eine eigene Zeile setzen
- Sparsam mit Sichtbarkeit umgehen: Möglichst alle Attribute privat setzen

Allgemeine Strategie zur Fehlervermeidung in Methoden

- Funktionalität komplexer Methoden auf mehrere Einzelmethoden verteilen
- Code-Redundanz vermeiden; widerholenden Code ggf. in eigene Methode auslagern
- Methodenaufruf mit unsinnigen Parametern abfangen (`Exception` werfen, `assert` verwenden)
- Hinterfragen, ob man eine Methode auch versteht, wenn man nicht der Autor ist
- Vor- und Nachbedingungen einer Methode verständlich dokumentieren (Javadoc!)

Abschließende Hinweise

- Stellt sich eine Methode als sehr fehlerhaft heraus, ist neu programmieren oft einfacher als alle Fehler beseitigen
- Vor dem Umsetzen einer eigenen Lösung schauen ob es bewährte Lösungen gibt: In der Java-API oder als frei verfügbare oder (bes. in wirtschaftlichem Kontext) kommerzielle Pakete.

Fragen zur Vorlesungseinheit

1. Welche drei Methoden gibt es, die Fehlerfreiheit eines Programms zu gewährleisten?
2. Warum sind Fehler zur Compilezeit gegenüber solchen zur Laufzeit zu bevorzugen?
3. Was sind die vier Schritte zur Identifikation von Fehlern?