

Informatik I: Einführung in die Programmierung

11. Programmentwicklung: Testen und Debuggen

Albert-Ludwigs-Universität Freiburg



**UNI
FREIBURG**

Prof. Dr. Peter Thiemann

29. Dezember 2022



Programmentwicklung

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Wie kommen Fehler ins Programm?



- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Wie kommen Fehler ins Programm?



- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Besonders in einfach erscheinenden Fällen: Schreibfehler, zu kurz gedacht, falsche Annahmen, ...

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Wie kommen Fehler ins Programm?



- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Besonders in einfach erscheinenden Fällen: Schreibfehler, zu kurz gedacht, falsche Annahmen, ...
- “Rund **50%** des Programmieraufwands wird für die Identifikation und Beseitigung von Fehlern aufgewendet.”

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Wie kommen Fehler ins Programm?



- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Besonders in einfach erscheinenden Fällen: Schreibfehler, zu kurz gedacht, falsche Annahmen, ...
- “Rund **50%** des Programmieraufwands wird für die Identifikation und Beseitigung von Fehlern aufgewendet.”
- “The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.” — Tom Cargill, Bell Labs

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Wie kommen Fehler ins Programm?



- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Besonders in einfach erscheinenden Fällen: Schreibfehler, zu kurz gedacht, falsche Annahmen, ...
- “Rund **50%** des Programmieraufwands wird für die Identifikation und Beseitigung von Fehlern aufgewendet.”
- “The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.” — Tom Cargill, Bell Labs
- Auch “fertige” Software hat noch **1–18 Fehler pro 1000 Zeilen Code!**

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Wie kommen Fehler ins Programm?



- Beim Schreiben von Programmen wird nicht immer alles auf Anhieb richtig gemacht.
- Besonders in einfach erscheinenden Fällen: Schreibfehler, zu kurz gedacht, falsche Annahmen, ...
- “Rund **50%** des Programmieraufwands wird für die Identifikation und Beseitigung von Fehlern aufgewendet.”
- “The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.” — Tom Cargill, Bell Labs
- Auch “fertige” Software hat noch **1–18 Fehler pro 1000 Zeilen Code!**
- Wichtig: **Werkzeuge** für die Fehlersuche und für die Qualitätskontrolle durch automatisches Testen

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Ausdrucks über den ganzen Zahlen errechnet. Der Ausdruck wird durch einen Ausdrucksbaum repräsentiert.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Ausdrucks über den ganzen Zahlen errechnet. Der Ausdruck wird durch einen Ausdrucksbaum repräsentiert.
- Beispiel: $\text{Node} ('*', \text{Node} ('+', \text{leaf}(2), \text{leaf}(5)), \text{leaf}(6)) \mapsto 42$

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Ausdrucks über den ganzen Zahlen errechnet. Der Ausdruck wird durch einen Ausdrucksbaum repräsentiert.
- Beispiel: $\text{Node} ('*', \text{Node} ('+', \text{leaf}(2), \text{leaf}(5)), \text{leaf}(6)) \mapsto 42$
- Methode: Traversierung des Ausdrucksbaums.

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Wir wollen ein Programm entwickeln, das den Wert eines arithmetischen Ausdrucks über den ganzen Zahlen errechnet. Der Ausdruck wird durch einen Ausdrucksbaum repräsentiert.
- Beispiel: $\text{Node}('*', \text{Node}('+', \text{leaf}(2), \text{leaf}(5)), \text{leaf}(6)) \mapsto 42$
- Methode: Traversierung des Ausdrucksbaums.
- Annahme: der Baum ist nicht leer

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Evaluating an Expression Tree

```
def expeval(tree)
  match tree:
    case Node('+', left, right):
      return expeval(left)+expval(right)
    case Node('-', left, right):
      return expeval(left)-expeval(right)
    case Node('*', left, right):
      return expeval(left)*expeval(right)
    case Node('/', left, right):
      return expeval(left)/expeval(right))
```

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler
Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Syntaxfehler

Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung. Meist einfach zu finden und zu reparieren.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Syntaxfehler

Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung. Meist einfach zu finden und zu reparieren.

Laufzeitfehler

Während der Ausführung passiert nichts (das Programm hängt) oder es gibt eine Fehlermeldung (**Exception**).

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Syntaxfehler

Das Programm entspricht nicht der formalen Grammatik. Solche Fehler bemerkt der Python-Interpreter vor der Ausführung. Meist einfach zu finden und zu reparieren.

Laufzeitfehler

Während der Ausführung passiert nichts (das Programm hängt) oder es gibt eine Fehlermeldung (**Exception**).

Logische Fehler

Alles „läuft“, aber die Ausgaben und Aktionen des Programms sind anders als erwartet. Das sind die gefährlichsten Fehler. Beispiel: *Mars-Climate-Orbiter*.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.

Programm-
entwicklung

Fehlertypen

**Syntaktische
Fehler**

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
 - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
 - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)
 - Unbalancierte Klammern

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
 - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)
 - Unbalancierte Klammern
 - `=` statt `==` in Booleschen Ausdrücken

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
 - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)
 - Unbalancierte Klammern
 - `=` statt `==` in Booleschen Ausdrücken
 - Die Einrückung!

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
 - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)
 - Unbalancierte Klammern
 - `=` statt `==` in Booleschen Ausdrücken
 - Die Einrückung!
- Oft helfen Editoren mit Syntaxunterstützung.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Interpreter gibt an, wo der Fehler festgestellt wurde.
- Das tatsächliche Problem kann aber mehrere Zeilen vorher liegen!
- Typische Fehler:
 - Schlüsselwort als Variablennamen benutzt
 - Es fehlt ein ':' für ein mehrzeiliges Statement (`while`, `if`, `for`, `def`, usw.)
 - Nicht abgeschlossener Multi-Zeilen-String (drei öffnende Anführungszeichen)
 - Unbalancierte Klammern
 - `=` statt `==` in Booleschen Ausdrücken
 - Die Einrückung!
- Oft helfen Editoren mit Syntaxunterstützung.
- Wenn nichts anderes mehr hilft: Sukzessives Auskommentieren

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Dieses Programm enthält 2 Syntaxfehler.

Evaluating an Expression tree

```
def expeval(tree)
  match tree:
    case Node('+', left, right):
      return expeval(left)+expeval(right)
    case Node('-', left, right):
      return expeval(left)-expeval(right)
    case Node('*', left, right):
      return expeval(left)*expeval(right)
    case Node('/', left, right):
      return expeval(left)/expeval(right))
```

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Dieses Programm enthält 2 Syntaxfehler.
- Das syntaktisch korrekte Programm:

Evaluating an Expression tree

```
def expeval(tree):  
    match tree:  
        case Node('+', left, right):  
            return expeval(left)+expeval(right)  
        case Node('-', left, right):  
            return expeval(left)-expeval(right)  
        case Node('*', left, right):  
            return expeval(left)*expeval(right)  
        case Node('/', left, right):  
            return expeval(left)/expeval(right) )
```

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“



- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“



- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggf. Timeout vorsehen).

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“



- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggf. Timeout vorsehen).
- Es befindet sich in einer **Endlosschleife**.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“



- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).
- Es wartet auf Daten aus anderer Quelle (ggf. Timeout vorsehen).
- Es befindet sich in einer **Endlosschleife**.
 - **Beispiel:** in einer `while`-Schleife wird die Schleifenvariable nicht geändert!

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“



- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).
 - Es wartet auf Daten aus anderer Quelle (ggf. Timeout vorsehen).
 - Es befindet sich in einer **Endlosschleife**.
 - **Beispiel:** in einer `while`-Schleife wird die Schleifenvariable nicht geändert!
- **Abbrechen** mit Ctrl-C.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Laufzeitfehler: Das Programm „hängt“



- Das Programm wartet auf eine Eingabe (→ kein Fehler, Eingabe machen).
 - Es wartet auf Daten aus anderer Quelle (ggf. Timeout vorsehen).
 - Es befindet sich in einer **Endlosschleife**.
 - **Beispiel:** in einer `while`-Schleife wird die Schleifenvariable nicht geändert!
- **Abbrechen** mit Ctrl-C.
- Dann Fehler einkreisen und identifizieren (siehe **Debugging**)

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



■ Typische Fehler:

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Typische Fehler:
 - `NameError`: Benutzung einer nicht initialisierten Variablen.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Typische Fehler:
 - `NameError`: Benutzung einer nicht initialisierten Variablen.
 - `TypeError`: Wert hat einen anderen Typ als erwartet.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



■ Typische Fehler:

- `NameError`: Benutzung einer nicht initialisierten Variablen.
- `TypeError`: Wert hat einen anderen Typ als erwartet.
- `IndexError`: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



■ Typische Fehler:

- `NameError`: Benutzung einer nicht initialisierten Variablen.
- `TypeError`: Wert hat einen anderen Typ als erwartet.
- `IndexError`: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
- `KeyError`: Ähnlich wie `IndexError`, aber für *Dictionaries* (lernen wir noch kennen).

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



■ Typische Fehler:

- `NameError`: Benutzung einer nicht initialisierten Variablen.
- `TypeError`: Wert hat einen anderen Typ als erwartet.
- `IndexError`: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
- `KeyError`: Ähnlich wie `IndexError`, aber für *Dictionaries* (lernen wir noch kennen).
- `AttributeError`: Versuch ein nicht existentes Attribut anzusprechen.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



■ Typische Fehler:

- `NameError`: Benutzung einer nicht initialisierten Variablen.
- `TypeError`: Wert hat einen anderen Typ als erwartet.
- `IndexError`: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
- `KeyError`: Ähnlich wie `IndexError`, aber für *Dictionaries* (lernen wir noch kennen).
- `AttributeError`: Versuch ein nicht existentes Attribut anzusprechen.
 - **Beispiel**: Zugriff auf Attribut `rigt`

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Typische Fehler:
 - `NameError`: Benutzung einer nicht initialisierten Variablen.
 - `TypeError`: Wert hat einen anderen Typ als erwartet.
 - `IndexError`: Zugriff auf Sequenz über einen Index, der zu klein oder zu groß ist.
 - `KeyError`: Ähnlich wie `IndexError`, aber für *Dictionaries* (lernen wir noch kennen).
 - `AttributeError`: Versuch ein nicht existentes Attribut anzusprechen.
 - **Beispiel**: Zugriff auf Attribut `rigt`
- Es gibt einen **Stack-Backtrace** und eine genaue Angabe der Stelle.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche **kleinere Eingabe**, bei der Fehler ebenfalls auftritt.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche **kleinere Eingabe**, bei der Fehler ebenfalls auftritt.
 - **Beispiel:** Die Eingabe ist ein Baum

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche **kleinere Eingabe**, bei der Fehler ebenfalls auftritt.
 - **Beispiel:** Die Eingabe ist ein Baum
 - Tritt der Fehler bereits bei einem Teilbaum auf?

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche **kleinere Eingabe**, bei der Fehler ebenfalls auftritt.
 - **Beispiel:** Die Eingabe ist ein Baum
 - Tritt der Fehler bereits bei einem Teilbaum auf?
 - Liegt es an der Markierung der Wurzel?

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche **kleinere Eingabe**, bei der Fehler ebenfalls auftritt.
 - **Beispiel:** Die Eingabe ist ein Baum
 - Tritt der Fehler bereits bei einem Teilbaum auf?
 - Liegt es an der Markierung der Wurzel?
 - Schneide Teilbäume (auf sinnvolle Art und Weise) ab um kleinere Eingaben zu erhalten.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Der Fehler tritt bei einer bestimmten Eingabe auf.
- Suche **kleinere Eingabe**, bei der Fehler ebenfalls auftritt.
 - **Beispiel:** Die Eingabe ist ein Baum
 - Tritt der Fehler bereits bei einem Teilbaum auf?
 - Liegt es an der Markierung der Wurzel?
 - Schneide Teilbäume (auf sinnvolle Art und Weise) ab um kleinere Eingaben zu erhalten.
- Erstelle einen **Testfall** aus der kleinen Eingabe und der erwarteten Ausgabe (s.u.).

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Führe den so erstellten Testfall aus.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Führe den so erstellten Testfall aus.
- Das Ergebnis sollte fehlerhaft, d.h. anders als die erwartete Ausgabe, sein.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Führe den so erstellten Testfall aus.
- Das Ergebnis sollte fehlerhaft, d.h. anders als die erwartete Ausgabe, sein.
- Gehe von der Fehlerstelle schrittweise **rückwärts** bis alles (Inhalte von Variablen und Attributen) wieder richtig erscheint.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Führe den so erstellten Testfall aus.
 - Das Ergebnis sollte fehlerhaft, d.h. anders als die erwartete Ausgabe, sein.
 - Gehe von der Fehlerstelle schrittweise **rückwärts** bis alles (Inhalte von Variablen und Attributen) wieder richtig erscheint.
- ⇒ Der Fehler wurde durch die letzte Anweisung manifestiert.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Führe den so erstellten Testfall aus.
 - Das Ergebnis sollte fehlerhaft, d.h. anders als die erwartete Ausgabe, sein.
 - Gehe von der Fehlerstelle schrittweise **rückwärts** bis alles (Inhalte von Variablen und Attributen) wieder richtig erscheint.
- ⇒ Der Fehler wurde durch die letzte Anweisung manifestiert.
- Enthält die Anweisung selbst einen Fehler?

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Führe den so erstellten Testfall aus.
 - Das Ergebnis sollte fehlerhaft, d.h. anders als die erwartete Ausgabe, sein.
 - Gehe von der Fehlerstelle schrittweise **rückwärts** bis alles (Inhalte von Variablen und Attributen) wieder richtig erscheint.
- ⇒ Der Fehler wurde durch die letzte Anweisung manifestiert.
- Enthält die Anweisung selbst einen Fehler?
 - Falls nicht: Warum wurde sie ausgeführt? Das kann an umschließenden (fehlerhaften) bedingten Anweisungen liegen!

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Start mit dem Beispielausdruck

```
>>> e = Node('*', Node('+', leaf(2), leaf(5)), leaf(6))
>>> print(expeval(e))
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

File "<stdin>", line 7, in expeval

File "<stdin>", line 3, in expeval

NameError: name 'expval' is not defined

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler

Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Start mit dem Beispielausdruck

```
>>> e = Node('*', Node('+', leaf(2), leaf(5)), leaf(6))
```

```
>>> print(expeval(e))
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 7, in expeval
```

```
  File "<stdin>", line 3, in expeval
```

```
NameError: name 'expval' is not defined
```

Verkleinern! Probiere linken Teilbaum

```
>>> print(expeval(e.left))
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
  File "<stdin>", line 3, in expeval
```

```
NameError: name 'expval' is not defined
```

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Weiter verkleinern! Probiere linken Teilbaum vom linken Teilbaum

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
None
```

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Weiter verkleinern! Probiere linken Teilbaum vom linken Teilbaum

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
None
```

- Hoppla, ein anderer Fehler!

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Weiter verkleinern! Probiere linken Teilbaum vom linken Teilbaum

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
None
```

- Hoppla, **ein anderer Fehler!**
- Offenbar wird der Fall, dass der Baum ein Blatt ist, nicht korrekt behandelt!

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Weiter verkleinern! Probiere linken Teilbaum vom linken Teilbaum

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
None
```

- Hoppla, **ein anderer Fehler!**
- Offenbar wird der Fall, dass der Baum ein Blatt ist, nicht korrekt behandelt!
- Abhilfe: Einfügen von `return tree.mark` am Ende.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Weiter verkleinern! Probiere linken Teilbaum vom linken Teilbaum

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
None
```

- Hoppla, **ein anderer Fehler!**
- Offenbar wird der Fall, dass der Baum ein Blatt ist, nicht korrekt behandelt!
- Abhilfe: Einfügen von `return tree.mark` am Ende.

Nach der Korrektur

```
>>> print(e.left.left)
Node(2, None, None)
>>> print(expeval(e.left.left))
2
```

Programm-

entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler

Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

- Kleinstes Beispiel, das den Fehler verursacht.

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler

Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

- Kleinstes Beispiel, das den Fehler verursacht.
- `e.left` beginnt mit '+', also muss dort der Fehler sein.

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler

Laufzeitfehler
Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

- Kleinstes Beispiel, das den Fehler verursacht.
- `e.left` beginnt mit '+', also muss dort der Fehler sein.
- Korrigiere dort `expval` nach `expeval`.

Programm-

entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum linken Teilbaum

```
>>> print(expeval(e.left))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 3, in expeval
NameError: name 'expval' is not defined
```

- Kleinstes Beispiel, das den Fehler verursacht.
- `e.left` beginnt mit '+', also muss dort der Fehler sein.
- Korrigiere dort `expval` nach `expeval`.

Nach der Korrektur

```
>>> print(e.left)
Node('+', Node(2, None, None), Node(5, None, None))
>>> print(expeval(e.left))
7
```

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum kompletten Beispiel

```
>>> print (expeval (e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
AttributeError: 'Node' object has no attribute 'right'
```

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum kompletten Beispiel

```
>>> print (expeval (e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
AttributeError: 'Node' object has no attribute 'right'
```

- Keiner der Teilbäume liefert noch einen Fehler.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum kompletten Beispiel

```
>>> print (expeval (e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
AttributeError: 'Node' object has no attribute 'right'
```

- Keiner der Teilbäume liefert noch einen Fehler.
- Problem muss an der Wurzel beim Operator '*' liegen.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zurück zum kompletten Beispiel

```
>>> print (expeval (e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
AttributeError: 'Node' object has no attribute 'rigt'
```

- Keiner der Teilbäume liefert noch einen Fehler.
- Problem muss an der Wurzel beim Operator '*' liegen.
- Korrigiere dort `rigt` nach `right`.

Programm-

entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-

fassung



Zurück zum kompletten Beispiel

```
>>> print (expeval (e))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in expeval
AttributeError: 'Node' object has no attribute 'rigt'
```

- Keiner der Teilbäume liefert noch einen Fehler.
- Problem muss an der Wurzel beim Operator '*' liegen.
- Korrigiere dort `rigt` nach `right`.

Nach der Korrektur

```
>>> print (expeval (e))
42
```

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Unser Programm enthält 3 Fehler, die zu **Exceptions** führen.

Evaluating an Expression Tree

```
def expeval(tree):  
    match tree:  
        case Node('+', left, right):  
            return expeval(left)+expeval(right)  
        case Node('-', left, right):  
            return expeval(left)-expeval(right)  
        case Node('*', left, right):  
            return expeval(left)*expeval(right)  
        case Node('/', left, right):  
            return expeval(left)/expeval(right)
```

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Unser Programm enthält 3 Fehler, die zu **Exceptions** führen.
- Das korrekte Programm:

Evaluating an Expression Tree

```
def expeval(tree):
    match tree:
        case Node('+', left, right):
            return expeval(left)+expeval(right)
        case Node('-', left, right):
            return expeval(left)-expeval(right)
        case Node('*', left, right):
            return expeval(left)*expeval(right)
        case Node('/', left, right):
            return expeval(left)/expeval(right)
        case Node(mark, _, _):
            return mark
```

Programm-

entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Ein logischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der **Erwartung** abweicht, die der Programmier hat.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Ein logischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der **Erwartung** abweicht, die der Programmier hat.
 - **Beispiele:** Statt Addition wird eine Multiplikation durchgeführt, metrische und imperiale Messwerte werden ohne Konversion verglichen.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Ein logischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der **Erwartung** abweicht, die der Programmier hat.
 - **Beispiele:** Statt Addition wird eine Multiplikation durchgeführt, metrische und imperiale Messwerte werden ohne Konversion verglichen.
- Tatsächlich liegt hier erst dann ein Fehler vor, wenn das erwartete Verhalten **(formal) spezifiziert** war. Auch informelle Vorgaben können verletzt werden.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Ein logischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der **Erwartung** abweicht, die der Programmier hat.
 - **Beispiele:** Statt Addition wird eine Multiplikation durchgeführt, metrische und imperiale Messwerte werden ohne Konversion verglichen.
- Tatsächlich liegt hier erst dann ein Fehler vor, wenn das erwartete Verhalten **(formal) spezifiziert** war. Auch informelle Vorgaben können verletzt werden.
- **Best practice:** Beschreibe das erwartete Verhalten durch **Beispiele** / Testfälle.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Ein logischer Fehler liegt vor, wenn das Verhalten/die Ausgabe des Programms von der **Erwartung** abweicht, die der Programmier hat.
 - **Beispiele:** Statt Addition wird eine Multiplikation durchgeführt, metrische und imperiale Messwerte werden ohne Konversion verglichen.
- Tatsächlich liegt hier erst dann ein Fehler vor, wenn das erwartete Verhalten **(formal) spezifiziert** war. Auch informelle Vorgaben können verletzt werden.
- **Best practice:** Beschreibe das erwartete Verhalten durch **Beispiele** / Testfälle.
- Unvollständig, aber besser als nichts!

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung

Logische Fehler in unserem Programm



- Gibt es logische Fehler in unserem Programm?
- Forderung: ganzzahlige Arithmetik, aber Operator „/“ liefert Gleitkommazahl!

Evaluating an Expression Tree

```
def expeval(tree):  
    match tree:  
        case Node('+', left, right):  
            return expeval(left)+expeval(right)  
        case Node('-', left, right):  
            return expeval(left)-expeval(right)  
        case Node('*', left, right):  
            return expeval(left)*expeval(right)  
        case Node('/', left, right):  
            return expeval(left)/expeval(right)  
        case Node (mark, _, _):  
            return mark
```

Programm-

entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Gibt es logische Fehler in unserem Programm?
- Forderung: ganzzahlige Arithmetik, aber Operator „/“ liefert Gleitkommazahl!

Evaluating an Expression Tree

```
def expeval(tree):
    match tree:
        case Node('+', left, right):
            return expeval(left)+expeval(right)
        case Node('-', left, right):
            return expeval(left)-expeval(right)
        case Node('*', left, right):
            return expeval(left)*expeval(right)
        case Node('/', left, right):
            return expeval(left)//expeval(right)
        case Node (mark, _, _):
            return mark
```

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Diese Beispiele hätten alle Fehler identifiziert!

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Diese Beispiele hätten alle Fehler identifiziert!

- Ein Beispiel für einen konstanten Ausdruck.

Programm-
entwicklung

Fehlertypen

Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Diese Beispiele hätten alle Fehler identifiziert!

- Ein Beispiel für einen konstanten Ausdruck.
- Je ein Beispiel pro Operator.

Programm-
entwicklung

Fehlertypen

Syntaktische

Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Diese Beispiele hätten alle Fehler identifiziert!

- Ein Beispiel für einen konstanten Ausdruck.
- Je ein Beispiel pro Operator.
- Für die Division ein Beispiel, wo ganzzahlig dividiert werden muss.
Z.B. Node ('/', leaf(5), leaf(3))

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Diese Beispiele hätten alle Fehler identifiziert!

- Ein Beispiel für einen konstanten Ausdruck.
- Je ein Beispiel pro Operator.
- Für die Division ein Beispiel, wo ganzzahlig dividiert werden muss.
Z.B. Node ('/', leaf(5), leaf(3))

Welche Test sind erforderlich?

- Für jeden booleschen Ausdruck im Programm je ein Test, der den Ausdruck wahr bzw. falsch macht.
- Für jede Anweisung im Programm je ein Test, der zu ihrer Ausführung führt.
- (Kann eine solche Menge von Tests automatisch bestimmt werden?)

Programm-
entwicklung

Fehlertypen
Syntaktische
Fehler

Laufzeitfehler

Logische Fehler

Debuggen

Tests

Ausblick

Zusammen-
fassung



Debuggen

Programm-
entwicklung

Debuggen

Print-Anweisungen

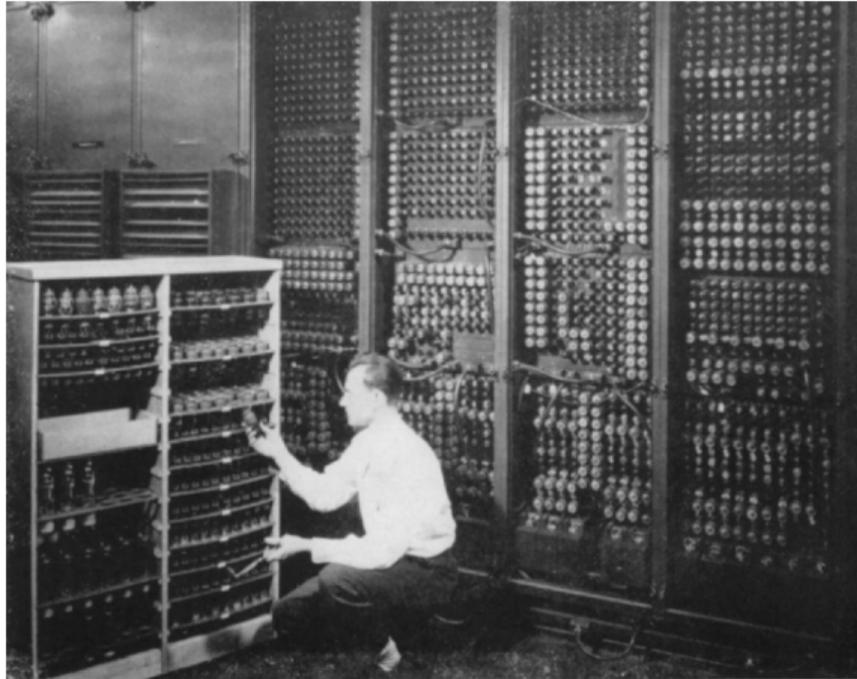
Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Replacing a bad tube meant checking among ENIAC's 19,000 possibilities.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



9/9

0800 Antenn started
 1000 " stopped - antenn ✓ { 1.2700 9.037 847 025
 1300 (032) MP-MC ~~1.50776715~~ 9.037 846 995 correct
 (033) PRO 2 2.130476415
 correct 2.130676415

Relays 6-2 in 033 failed special speed test
 in relay 11,000 test.

Relays changed

1100 Started Cosine Tape (Sine check)
 1525 Started Multi-Adder Test.

1545  Relay #70 Panel F
 (moth) in relay.

First actual case of bug being found.
 1650 Antennant started.
 1700 closed down.

Relay
2145
Relay 3370

Debuggen = Käfer jagen und töten



- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung

Debuggen = Käfer jagen und töten



- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.
- Diese Käfer (oder andere Ursachen für Fehlfunktionen) zu finden heißt *debuggen*, im Deutschen manchmal *entwanzen*.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung

Debuggen = Käfer jagen und töten



- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.
- Diese Käfer (oder andere Ursachen für Fehlfunktionen) zu finden heißt *debuggen*, im Deutschen manchmal *entwanzen*.
- Hat viel von **Detektivarbeit** (wer ist der Schuldige?)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung

Debuggen = Käfer jagen und töten



- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.
- Diese Käfer (oder andere Ursachen für Fehlfunktionen) zu finden heißt *debuggen*, im Deutschen manchmal *entwanzen*.
- Hat viel von **Detektivarbeit** (wer ist der Schuldige?)
- Aber nicht mystifizieren; vieles ist heute systematisiert und automatisierbar.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung

Debuggen = Käfer jagen und töten



- In den frühen Computern haben Motten/Fliegen/Käfer (engl. *Bug*) durch Kurzschlüsse für Fehlfunktionen gesorgt.
- Diese Käfer (oder andere Ursachen für Fehlfunktionen) zu finden heißt *debuggen*, im Deutschen manchmal *entwanzen*.
- Hat viel von **Detektivarbeit** (wer ist der Schuldige?)
- Aber nicht mystifizieren; vieles ist heute systematisiert und automatisierbar.
- Die Verbesserungen heißen **Bugfixes** – und sollten das Problem dann lösen!

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Das Wichtigste

Versuche minimale Eingaben zu finden, die den Fehler hervorrufen!

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Das Wichtigste

Versuche minimale Eingaben zu finden, die den Fehler hervorrufen!

Nachvollziehen der Berechnung bis zum Fehler

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Das Wichtigste

Versuche minimale Eingaben zu finden, die den Fehler hervorrufen!

Nachvollziehen der Berechnung bis zum Fehler

- 1 Kleine Beispiele von Hand oder mit `pythontutor`

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Das Wichtigste

Versuche minimale Eingaben zu finden, die den Fehler hervorrufen!

Nachvollziehen der Berechnung bis zum Fehler

- 1 Kleine Beispiele von Hand oder mit `pythontutor`
- 2 Modifikation des Programms zur Ausgabe von bestimmten Variablenwerten an bestimmten Stellen (Einfügen von `print`-Anweisungen)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Das Wichtigste

Versuche minimale Eingaben zu finden, die den Fehler hervorrufen!

Nachvollziehen der Berechnung bis zum Fehler

- 1 Kleine Beispiele von Hand oder mit `pythontutor`
- 2 Modifikation des Programms zur Ausgabe von bestimmten Variablenwerten an bestimmten Stellen (Einfügen von `print`-Anweisungen)
- 3 Einsatz von Debugging-Werkzeugen:
Post-Mortem-Analyse-Tools und **Debugger**

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- Beobachten von internen Werten (vgl. bei Hardware mit einem Oszilloskop).

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- Beobachten von internen Werten (vgl. bei Hardware mit einem Oszilloskop).
- In vielen Sprachen/Systemen können `print`-Anweisungen eingefügt werden.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- Beobachten von internen Werten (vgl. bei Hardware mit einem Oszilloskop).
- In vielen Sprachen/Systemen können `print`-Anweisungen eingefügt werden.
- Einfachste Möglichkeit das Verhalten eines Programmes zu beobachten.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- Beobachten von internen Werten (vgl. bei Hardware mit einem Oszilloskop).
- In vielen Sprachen/Systemen können `print`-Anweisungen eingefügt werden.
- Einfachste Möglichkeit das Verhalten eines Programmes zu beobachten.
 - **Achtung:** Oft nicht angebracht, weil zusätzliche Ausgaben das Verhalten (speziell das Zeitverhalten) signifikant ändern können!

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- Beobachten von internen Werten (vgl. bei Hardware mit einem Oszilloskop).
- In vielen Sprachen/Systemen können `print`-Anweisungen eingefügt werden.
- Einfachste Möglichkeit das Verhalten eines Programmes zu beobachten.
 - **Achtung:** Oft nicht angebracht, weil zusätzliche Ausgaben das Verhalten (speziell das Zeitverhalten) signifikant ändern können!
- Eine generalisierte Form ist das *Logging*, bei dem `prints` generell im Code integriert sind und mit Schaltern an- und abgestellt werden können.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 *Post-Mortem-Tools*: Analyse des Programmszustands nach einem Fehler
 - Stack Backtrace wie in Python

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 *Post-Mortem-Tools*: Analyse des Programmszustands nach einem Fehler
 - Stack Backtrace wie in Python
 - Früher: Speicherbelegung (Hex-Dump)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 *Post-Mortem-Tools*: Analyse des Programmszustands nach einem Fehler
 - Stack Backtrace wie in Python
 - Früher: Speicherbelegung (Hex-Dump)
 - Heute: Variablenbelegung (global und lokal in der Kellertabelle)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 *Post-Mortem-Tools*: Analyse des Programmszustands nach einem Fehler
 - Stack Backtrace wie in Python
 - Früher: Speicherbelegung (Hex-Dump)
 - Heute: Variablenbelegung (global und lokal in der Kellertabelle)
- 2 *Interaktive Debugger*

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 *Post-Mortem-Tools*: Analyse des Programmszustands nach einem Fehler
 - Stack Backtrace wie in Python
 - Früher: Speicherbelegung (Hex-Dump)
 - Heute: Variablenbelegung (global und lokal in der Kellertabelle)
- 2 *Interaktive Debugger*
 - Setzen von Breakpoints (u.U. konditional)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands
- Einzelschrittausführung (Stepping / Tracing):

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands
- Einzelschrittausführung (Stepping / Tracing):

Step in: Mache einen Schritt, ggfs. in eine Funktion hinein

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands
- Einzelschrittausführung (Stepping / Tracing):

Step in: Mache einen Schritt, ggfs. in eine Funktion hinein

Step over: Mache einen Schritt, führe dabei ggfs. eine Funktion aus

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands
- Einzelschrittausführung (Stepping / Tracing):

Step in: Mache einen Schritt, ggfs. in eine Funktion hinein

Step over: Mache einen Schritt, führe dabei ggfs. eine Funktion aus

Step out: Beende den aktuellen Funktionsaufruf

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands
- Einzelschrittausführung (Stepping / Tracing):

Step in: Mache einen Schritt, ggfs. in eine Funktion hinein

Step over: Mache einen Schritt, führe dabei ggfs. eine Funktion aus

Step out: Beende den aktuellen Funktionsaufruf

Go/Continue: Starte Ausführung bzw. setze fort

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



1 *Post-Mortem-Tools*: Analyse des Programmzustands nach einem Fehler

- Stack Backtrace wie in Python
- Früher: Speicherbelegung (Hex-Dump)
- Heute: Variablenbelegung (global und lokal in der Kellertabelle)

2 *Interaktive Debugger*

- Setzen von Breakpoints (u.U. konditional)
- Inspektion des Programmzustands (Variablenbelegung)
- Ändern des Zustands
- Einzelschrittausführung (Stepping / Tracing):

Step in: Mache einen Schritt, ggfs. in eine Funktion hinein

Step over: Mache einen Schritt, führe dabei ggfs. eine Funktion aus

Step out: Beende den aktuellen Funktionsaufruf

Go/Continue: Starte Ausführung bzw. setze fort

Quit: Beendet alles.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formuliere eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formulare eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
- 2 **Instrumentiere** die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formulare eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
- 2 **Instrumentiere** die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)
- 3 Versuche zu verstehen, was die tiefere Ursache des Fehlers ist.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formulare eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
- 2 **Instrumentiere** die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)
- 3 Versuche zu verstehen, was die tiefere Ursache des Fehlers ist.
- 4 Formulare einen **Bugfix** erst dann, wenn das **Problem verstanden** ist. Einfache Lösungen sind oft nicht hilfreich.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formulare eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
- 2 **Instrumentiere** die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)
- 3 Versuche zu verstehen, was die tiefere Ursache des Fehlers ist.
- 4 Formulare einen **Bugfix** erst dann, wenn das **Problem verstanden** ist. Einfache Lösungen sind oft nicht hilfreich.
- 5 Teste nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formulare eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
- 2 **Instrumentiere** die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)
- 3 Versuche zu verstehen, was die tiefere Ursache des Fehlers ist.
- 4 Formulare einen **Bugfix** erst dann, wenn das **Problem verstanden** ist. Einfache Lösungen sind oft nicht hilfreich.
- 5 Teste nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.
- 6 Weitere Tests laufen lassen (s.u.).

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



- 1 Formuliere eine Hypothese, warum der Fehler auftritt, an welcher Stelle des Programms sich der Fehler manifestiert, welche Variablen betroffen sind!
- 2 **Instrumentiere** die Stelle, sodass die betroffenen Variablen inspiziert werden können (Breakpoints oder `print`-Anweisungen)
- 3 Versuche zu verstehen, was die tiefere Ursache des Fehlers ist.
- 4 Formuliere einen **Bugfix** erst dann, wenn das **Problem verstanden** ist. Einfache Lösungen sind oft nicht hilfreich.
- 5 Teste nach dem Bugfix, ob das Problem tatsächlich beseitigt wurde.
- 6 Weitere Tests laufen lassen (s.u.).
- 7 Wenn es nicht weiter geht: frische Luft und eine Tasse Kaffee hilft!

Programm-
entwicklung

Debuggen

Print-Anweisungen

Debugger

Debugging-
Techniken

Tests

Ausblick

Zusammen-
fassung



Automatische Tests

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren.**

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren**.
- Ein **Testfall** besteht aus einer Eingabe und dem erwarteten Ergebnis. Die Testfälle bilden eine **Testsuite**.

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren**.
- Ein **Testfall** besteht aus einer Eingabe und dem erwarteten Ergebnis. Die Testfälle bilden eine **Testsuite**.
- **Schon vor dem Programmieren** systematisch Testfälle erstellen:

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren**.
- Ein **Testfall** besteht aus einer Eingabe und dem erwarteten Ergebnis. Die Testfälle bilden eine **Testsuite**.
- **Schon vor dem Programmieren** systematisch Testfälle erstellen:
 - Basisfälle, Grenzfälle (z.B. erstes bzw. letztes Element einer Datenstruktur).

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren**.
- Ein **Testfall** besteht aus einer Eingabe und dem erwarteten Ergebnis. Die Testfälle bilden eine **Testsuite**.
- **Schon vor dem Programmieren** systematisch Testfälle erstellen:
 - Basisfälle, Grenzfälle (z.B. erstes bzw. letztes Element einer Datenstruktur).
 - Jede Anweisung soll durch einen Testfall abgedeckt (d.h. ausgeführt) werden.

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren**.
- Ein **Testfall** besteht aus einer Eingabe und dem erwarteten Ergebnis. Die Testfälle bilden eine **Testsuite**.
- **Schon vor dem Programmieren** systematisch Testfälle erstellen:
 - Basisfälle, Grenzfälle (z.B. erstes bzw. letztes Element einer Datenstruktur).
 - Jede Anweisung soll durch einen Testfall abgedeckt (d.h. ausgeführt) werden.
 - Versuche Eingaben zu finden, die die Bedingungen im Programm unabhängig voneinander wahr bzw. falsch machen (soweit möglich).

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Testen eines Programms heißt, **fehlerhaftes Verhalten zu provozieren**.
- Ein **Testfall** besteht aus einer Eingabe und dem erwarteten Ergebnis. Die Testfälle bilden eine **Testsuite**.
- **Schon vor dem Programmieren** systematisch Testfälle erstellen:
 - Basisfälle, Grenzfälle (z.B. erstes bzw. letztes Element einer Datenstruktur).
 - Jede Anweisung soll durch einen Testfall abgedeckt (d.h. ausgeführt) werden.
 - Versuche Eingaben zu finden, die die Bedingungen im Programm unabhängig voneinander wahr bzw. falsch machen (soweit möglich).
- **Beim Programmieren**: Tests, die zur Entdeckung eines Fehlers geführt haben, müssen aufbewahrt werden!

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Viele Testfälle ergeben sich schon aus der Typsignatur!

- `list[T]`: ein Test mit leerer Liste, mehrere Tests mit nicht-leeren Listen verschiedener Länge mit Elementen variiert gemäß `T`
- `tuple[T1, T2, ...]`: Kombiniere Testfälle gemäß `T1`, `T2`, dots
- `Optional[T]`: ein Test mit `None`, Tests mit Werten variiert gemäß `T`
- `T1 | T2 | ...`: Tests mit Werten gemäß `T1`, Tests mit Werten gemäß `T2`, ...
- `Any`: Tests zumindest mit allen Grundtypen (`int`, `float`, `str`, `bool`, ...)
- Datenklasse: Generiere Tests gemäß der Typen der Attribute und kombiniere
- `Literal[V1, V2, ...]`: Die Werte dieses Typs sind genau `V1`, `V2` usw. Also je ein Testfall mit `V1`, `V2`, ...

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung
Unittests
pytest

Ausblick

Zusammen-
fassung



Testgetriebene Entwicklung

- Formuliere zu Beginn mit dem Auftraggeber Testfälle, die nach und nach während der Entwicklung erfüllt werden.
- Der **Fortschritt der Entwicklung** des Systems kann anhand der Anzahl der bestandenen Tests gemessen werden.

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Testgetriebene Entwicklung

- Formuliere zu Beginn mit dem Auftraggeber Testfälle, die nach und nach während der Entwicklung erfüllt werden.
- Der **Fortschritt der Entwicklung** des Systems kann anhand der Anzahl der bestandenen Tests gemessen werden.

Regressionstest

Wiederholung von Tests um sicher zu stellen, dass nach Änderungen der Software keine neuen (oder alten) Fehler eingeschleppt wurden.

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Unittest

- Testfälle für Teile eines Systems (Modul, Funktion, usw.).
- Überprüfen die Funktion der Einzelteile.
- Automatisch ausführen nach Änderung / beim Einchecken → Regressionstests!

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Unittest

- Testfälle für Teile eines Systems (Modul, Funktion, usw.).
- Überprüfen die Funktion der Einzelteile.
- Automatisch ausführen nach Änderung / beim Einchecken → Regressionstests!

Automatisierung von Tests in Python

- Mehrere Werkzeuge zur Automatisierung von Tests.
- `pytest` — Beispiel für ein umfassendes Framework
- Installation durch `pip3 install pytest`.

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



- Operiert auf dem Modullevel.
- Für jede FUT (**function under test**) werden **Testfunktionen** geschrieben.
- Der Name einer Testfunktion beginnt mit “test_...”.
- Typischerweise in Dateien namens “test_xyz.py”.
- Die Testfunktion enthält Aufrufe der FUT, wobei die erwarteten Rückgabewerte als **Assertions** formuliert sind.
- Syntax der assert-Anweisung:
assert *Bedingung* [, *String*]
- assert überprüft die *Bedingung*.
 - Wenn sie erfüllt ist, wird die Testfunktion weiter ausgeführt.
 - Anderenfalls Abbruch mit **Exception** und Ausgabe von *String*.

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Beispiel für einen Testfall

```
import pytest
###
def test_expeval_b():
    """Test of expeval that fails."""
    exp = Node('*', Node('+', leaf(3), leaf(5)),
                leaf(6))
    assert expeval(exp) == 42
```

- Ausführung mit pytest `expeval.py`

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung

Verwendung von pytest (3)



Die Ausgabe in obigem Beispiel:

```
===== test session starts =====
expeval.py::test_expeval_b FAILED

===== FAILURES =====
----- test_expeval_b -----

    def test_expeval_b():
        """Test of expeval that fails."""
        exp = Node('*', Node('+', leaf(3), leaf(5)),
                    leaf(6))
>       assert expeval(exp) == 42
E       assert 48 == 42
E         + where 48 = expeval(Node('*', Node('+', leaf(3), leaf(5)), leaf(6)))

expeval.py:50: AssertionError
===== 1 failed, 1 passed in 0.02 seconds =====
```

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung

Verwendung von pytest (4)



Testsuite wie oben besprochen

```
import pytest
###
def test_expeval_1():
    """Testing expeval from example."""
    e = Node('*', Node('+', leaf(2), leaf(5)), leaf(6))
    assert expeval (e.left.left) == 2
    assert expeval (e.left) == 7
    assert expeval (e) == 42

def test_expeval_2():
    """Testing logical bug in expeeval"""
    assert expeval (Node ('/', leaf(2), leaf(3))) == 0
```

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Ausgabe vor Bugfix des logischen Fehlers

```
[Peters-MacBook-Pro:python thiemann$ pytest trees.py  
===== test session starts =====  
platform darwin -- Python 3.7.0, pytest-4.0.1, py-1.7.0, pluggy-0.8.0  
rootdir: /Users/thiemann/svn/teaching/info2018/slides/python, inifile:  
collected 2 items  
  
trees.py .F [100%]  
  
===== FAILURES =====  
----- test_expreval_2 -----  
  
def test_expreval_2():  
> assert expreval (Node ('/', leaf(2), leaf(3))) == 0  
E     AssertionError: assert 0.6666666666666666 == 0  
E       + where 0.6666666666666666 = expreval(<trees.Node object at 0x10f4ed978>)  
E       +   where <trees.Node object at 0x10f4ed978> = Node('/', <trees.Node obje  
ct at 0x10f4ede48>, <trees.Node object at 0x10f4ed898>)  
E       +     where <trees.Node object at 0x10f4ede48> = leaf(2)  
E       +     and <trees.Node object at 0x10f4ed898> = leaf(3)  
  
trees.py:35: AssertionError  
===== 1 failed, 1 passed in 0.07 seconds =====
```

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Ausgabe nach Bugfix

```
[Peters-MacBook-Pro:python thiemann$ pytest trees.py ]
===== test session starts =====
platform darwin -- Python 3.7.0, pytest-4.0.1, py-1.7.0, pluggy-0.8.0
rootdir: /Users/thiemann/svn/teaching/info2018/slides/python, inifile:
collected 2 items

trees.py ..                                     [100%]

===== 2 passed in 0.02 seconds =====
```

Programm-
entwicklung

Debuggen

Tests

Testgetriebene
Entwicklung

Unittests

pytest

Ausblick

Zusammen-
fassung



Ausblick

Programm-
entwicklung

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Können wir (von Menschen erschaffener) Software für AKWs, Flugzeuge, Autos, usw. vertrauen?
 - Aktive Forschungsrichtungen innerhalb der Informatik
 - Verbesserung der Testmethoden — *keine* Garantie für Korrektheit!
 - Maschinelle Beweise (d.h. für alle Fälle gültig) der Korrektheit.
 - Kein Schutz gegen Fehler in der Spezifikation gegen die geprüft wird!
 - Auch das Beweissystem kann Fehler besitzen (aber Selbstanwendung).
- In jedem Fall *reduzieren wir die Fehlerwahrscheinlichkeit!*
- Heute wird auch über *probabilistische Korrektheit* nachgedacht und geforscht.

Programm-
entwicklung

Debuggen

Tests

Ausblick

Zusammen-
fassung



Zusammenfassung

Programm-
entwicklung

Debuggen

Tests

Ausblick

Zusammen-
fassung



- Fehlerfreie Programmentwicklung gibt es nicht.
- Wir unterscheiden zwischen syntaktischen Fehlern, Laufzeitfehlern und logischen Fehlern.
- Fehlersuche: **Debuggen**
- **Checkliste Ursachenforschung**
- Der Debuggingprozess: Eingabe minimieren, Testfall erstellen, Werte beobachten, Hypothese entwickeln
- Fehler verstehen und beseitigen: **Bugfix**.
- **Testgetriebene Entwicklung**, Erstellung sinnvoller Testfälle
- Automatische Tests erhöhen die Qualität von Software!
- **pytest** ist ein Werkzeug zur Automatisierung von Regressionstests.