

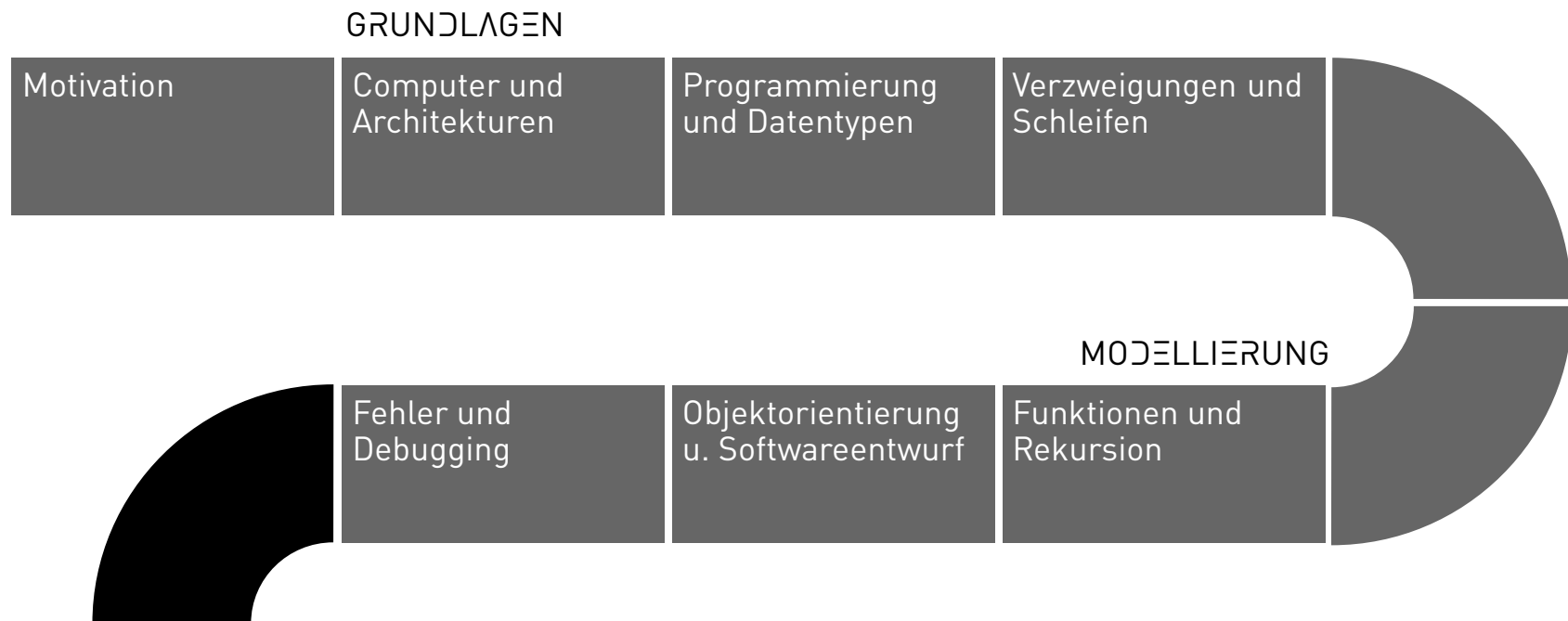
programmierung und datenbanken

Joern Ploennigs

Unit-Tests

MIDJOURNEY: WAVES TESTING A BOAT, REF. HOKUSAI

ABLAUF



UMGANG MIT PROGRAMMFEHLERN - STRATEGIEN

5 Hauptstrategien:

1. Fehlervermeidung
2. Fehlererkennung
3. Fehlerbehebung
4. Fehlerbehandlung
5. Fehlerausschluss

Jede Strategie hat ihre eigenen Methoden und Werkzeuge

1. FEHLERVERMEIDUNG

- Grundprinzip: Beginnt mit Erfahrung darüber, wo Fehler entstehen können
- Strategien für sicheren Code:
 - *Code Reviews* → Jede Codezeile wird von einem anderen Programmierer geprüft
 - *Test-Driven Development* → Man schreibt den Test vor dem eigentlichen Code
 - *Full Test Coverage* → Jede Codezeile sollte mindestens einen Test haben
- Achtung: Vorbeugung kann viel Vorarbeit bedeuten

2. FEHLERERKENNUNG

Verschiedene Fehlertypen erfordern verschiedene Werkzeuge

Syntaktische Fehler:

- Automatisch in der IDE erkannt

Statische Fehler:

- Mittels *Lint-Tools* (statische Code-Analyse)
- Regelbasierte Verfahren
- Moderne IDEs nutzen intern Lint-Tools

Dynamische Fehler:

- Prüfen durch *automatisch ausgeführte Unit-Tests*
- Nach jeder Code-Änderung
- Testen Funktionen/Methoden systematisch

2. FEHLERERKENNUNG – UNIT-TESTS



Definition: Unit-Test

Unit-Tests sind zusätzlich geschriebener Code, der nur zum Testen eines Moduls (Funktion, Klassen, Module) geschrieben wurde.

- Wichtige Eigenschaften:
 - Oft ist Test-Code umfangreicher als der zu testende Code
 - *Ziel*: Prüfen ob Modul gewünschte Funktion korrekt ausführt
 - *White-Box-Test*: Modul mit bestimmten Eingaben aufrufen
 - Erwartete Ausgaben oder Exceptions prüfen
 - Sollte *automatisierte Ausgeführt* werden nach jeder Code-Änderung

2. FEHLERERKENNUNG – UNIT-TEST TYPEN

Funktionstests

Korrekte Erfüllung der Funktion
Beispiel: Division-Modul

```
zaehler = 10
nenner = 2
ergebnis =
division(zaehler,
nenner)
assert ergebnis==5 #
Erwartet: 5
```

Grenzwerttests

Korrekter Umgang mit
Grenzwerten

```
zaehler = 10
nenner = 0
ergebnis =
division(zaehler,
nenner)
assert ergebnis==None #
Erwartet: None
```

Datentyptests

Korrekter Umgang mit
unerwarteten Eingaben

```
zaehler = 'keine_zahl'
nenner = 0
try:
    ergebnis =
division(zaehler,
nenner)
    assert False # We
let the test fail
except TypeError as e:
    # Erwartet:
    TypeError
```

3. FEHLERBEHEBUNG - GRUNDLAGEN



Definition: Debugging

Debugging ist das systematische Suchen und Beheben von Fehlern.

- Methoden:
 - Logging-Nachrichten zur Verfolgung des Programmflusses
 - Debugger für detaillierte Analyse
- Debugger-Funktionen:
 - Schrittweise Ausführung von Codezeilen
 - Überwachen von Variablenwerten
 - Dynamisches Einfügen von Code

3. FEHLERBEHEBUNG – IN PYTHON

- Das systematische Debugging ist eine Kernkompetenz jeden Programmierers
- Python Debugging-Tools
 - Standard-Debugger: pdb (Python-Modul)
 - IDE-eigene Debugger (oft benutzerfreundlicher)
- Mehr Details dazu in der Übung!

4. FEHLERBEHANDLUNG - AUF ALLES VORBEREITET SEIN

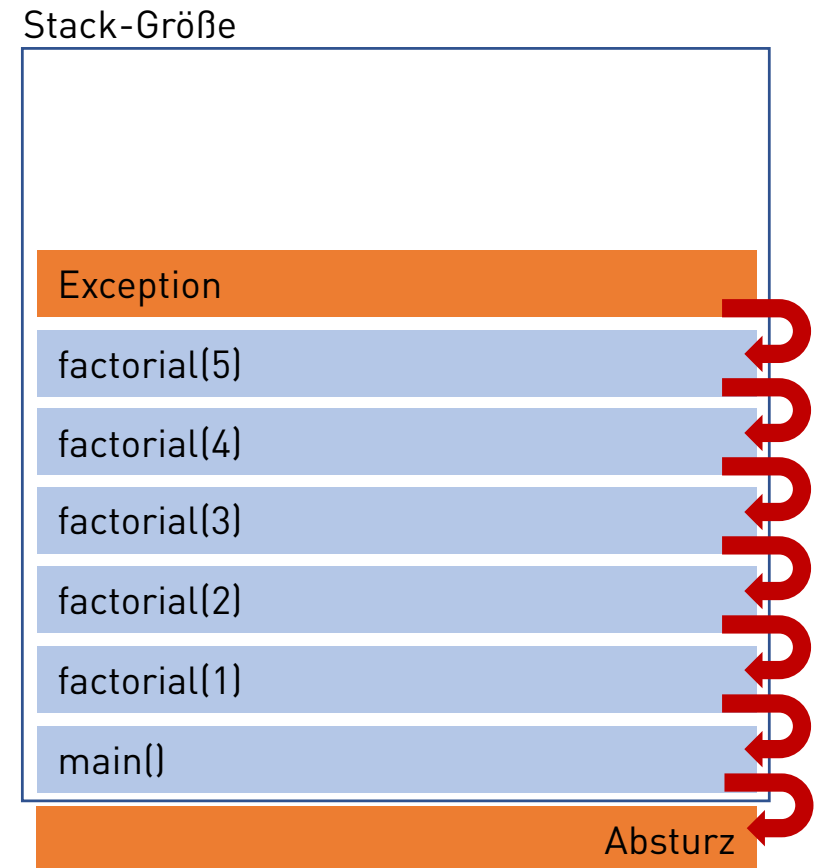
- Realität: Allen Bemühungen zum Trotz werden in realen Programmen Fehler zur Laufzeit auftreten
- Ziel:
 - Fehler abfangen und behandeln
 - Programm läuft weiter oder beendet kontrolliert
- Mechanismus:
 - Dynamischer Fehler → *Exception* wird erzeugt
 - Python-Werkzeuge: `try`, `except`, `raise`

4. FEHLERBEHANDLUNG - IN PYTHON

- **try:** Enthält ein (vermutlich fehleranfälliges) Stück Programmcode
- **except:** Tritt im try-Block ein bestimmter Fehler (Exception) auf, wird dieser beendet und der passende except-Block aufgerufen.
- **raise:** Wird in einem except-Block aufgerufen um eine Exception weiterzugeben
- **finally:** Wird nach dem Ende des try-Blocks aufgerufen, egal ob es einen Fehler gab oder nicht
- **else:** optional definiert nach allen except-Blöcken, wird ausgeführt falls im try-Block keine Exception auftritt

SEMANTISCHE FEHLER - EXCEPTIONS

- *Exception*-Mechanismus:
 - Unterbrechen normalen Programmverlauf
 - Kommunizieren Fehler mit Fehlermeldung
 - Verhindern unkontrolliertes Abstürzen
 - Werden im *Stack nach oben weitergegeben*
 - Bis sie abgefangen oder Programm abstürzt
- Ziel: Programm in lauffähigen Zustand halten



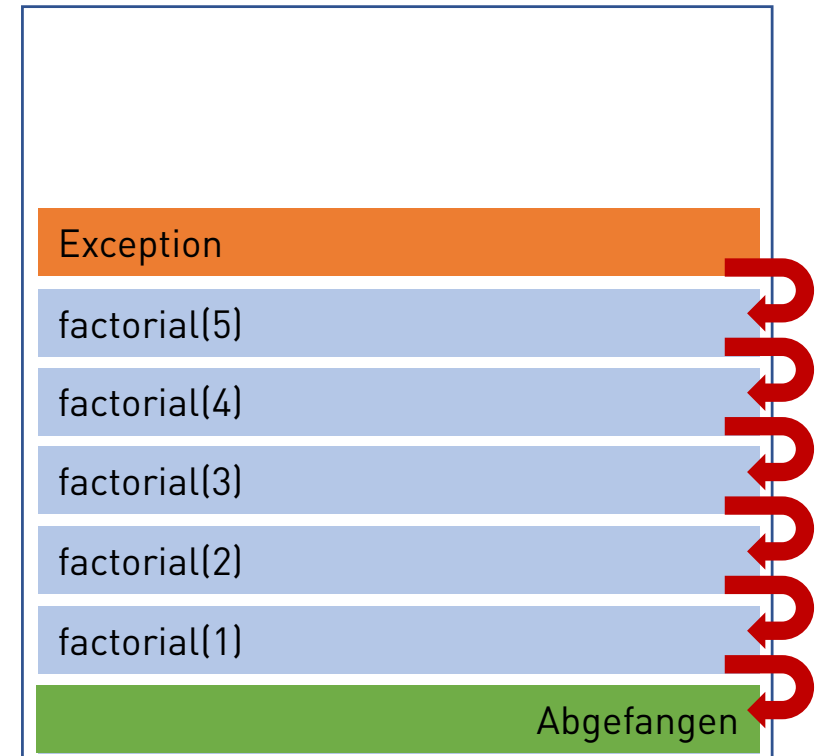
SEMANTISCHE FEHLER – EXCEPTIONS ABFANGEN

Bekannte Exceptions sollten immer abgefangen werden

Python `try-except`-Block:

```
try:
    # Block mit erwarteter Exception
    ergebnis = zaehler / nenner
except:
    # Fehlerbehandlung
    print("Division durch 0")
    ergebnis = None
```

Stack-Größe



SEMANTISCHE FEHLER - EXCEPTION-EBENEN

Exceptions werden auf mehreren Ebenen genutzt:

- Betriebssystem - Vordefinierte System-Exceptions
- Programmiersprache - Eingebaute Language-Exceptions
- Eigener Code - Selbst definierte Exceptions

Einsatzgebiete (wo Fehler wahrscheinlich sind):

- Kommunikation mit externen Quellen
- Lesen von Dateien
- Nutzereingaben
- Netzwerkverbindungen

4. FEHLERBEHANDLUNG - BEISPIEL: DIVISION DURCH 0

```
zaehler = 12
nenner = 0

try:
    ergebnis = zaehler / nenner
except:
    print("Exception")
```

Frage: Welche Exceptions können hier geworfen werden?

4. FEHLERBEHANDLUNG - BEISPIEL: DIVISION DURCH 0

```
zaehler = 12
nenner = 0

try:
    ergebnis = zaehler / nenner
except:
    print("Exception")
```

Frage: Welche Exceptions können hier geworfen werden?

Mögliche Exceptions:

- `ZeroDivisionError` - Division durch 0
- `TypeError` - Falscher Datentyp
- `NameError` - Variable nicht definiert

4. FEHLERBEHANDLUNG - EXCEPTIONS PROPAGIEREN

- Wichtiges Konzept in komplexerer Software
- Wir können Fehler zwar lokal behandeln, wollen aber dennoch die "oberen Ebenen" des Programmes darüber informieren.
- Die `raise`-Anweisung "wirft" dafür aus einem `except`-Block eine Exception weiter nach oben.

4. FEHLERBEHANDLUNG - BEISPIEL: EXCEPTION WEITERREICHEN

```
def count_up_and_down(x):  
    if x <= 0:  
        raise ValueError("No negative values allowed")  
  
    # Verknüpft vorwärts- und rückwärts-Liste  
    return list(range(x)) + list(reversed(list(range(x-1))))  
  
try:  
    print(count_up_and_down(-6))  
except ValueError:  
    print("That value was invalid.")
```

5. FEHLERAUSSCHLUSS - KORREKTHEIT VON PROGRAMMEN MATHEMATISCH BEWEISEN

- Gewisser Programmcode kann mittels *mathematischen Beweises* validiert werden
- *Hoare-Kalkül*: Programm Zeile für Zeile mit mathematischer Logik formalisieren
- Von vorne nach hinten durch das gesamte Programm
- *Problem*: Für die meisten Anwendungen zu zeitaufwendig und komplex
- *Einsatz*: Nur bei kritischen Systemen (Medizintechnik, Luftfahrt, etc.)

fragen?

