Aufnahme der Hybrid-Vorlesung

- **Hybrid-Vorlesung mit Aufnahme**
  - Die Aufnahme ist anschließend in Stud.IP verfügbar
  - Nutzen Sie die Gelegenheit zur Live-Veranstaltung!
- Wir nehmen auf
  - Folien, Dozent, Live-Audio sowie BBB-Audio
  - Ihre Stimme beim Fragen und Sprechen
  - **Durch aktive Teilnahme erklären Sie sich einverstanden!**
- Fragen: Live, im Chat, Sprechen in der BBB-Sitzung

Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

Programmiersprachen und Übersetzer

11 - Das Objektorientierte Programmierparadigma

Christian Dietrich

Sommersemester 2024

Wiederholung: Die semantische Lücke

[Problem]

Hochsprache

⇑

Semantische Lücke

⇓

CPU-Steuersignale

[Ausführung]

Mit unserem Übersetzer haben wir die semantische Lücke überwunden:

$$\text{Source} \rightarrow \text{AST} \rightarrow \text{IR} \rightarrow \text{ASM} \rightarrow \text{ELF}$$

- Beschreibung des Problems in Hochsprache
- Schrittweises Absenken der Abstraktion
- Schichten versprechen wohldefinierte Semantiken.

Wiederholung: Die semantische Lücke

[Problem]

Hochsprache
⇑
Semantische Lücke
⇓
CPU-Steuersignale

[Ausführung]

Mit unserem Übersetzer haben wir die semantische Lücke überwunden:

$$\text{Source} \rightarrow \text{AST} \rightarrow \text{IR} \rightarrow \text{ASM} \rightarrow \text{ELF}$$

- Beschreibung des Problems in Hochsprache
- Schrittweises Absenken der Abstraktion
- Schichten versprechen wohldefinierte Semantiken.

Aber: Warum türmen wir überhaupt diese semantische Lücke auf?

- Lernkurve: Abstraktionen müssen gelernt werden
- Fehleranfällig: Übersetzer können Bugs haben
- Abstraktionen erzeugen oft einen Overhead

Welchen Nutzen hat die semantische Lücke?

Komplexität und Nichtdeterminismus

- Große Systeme haben zwei Dimensionen, die das Verständnis erschweren
  - Komplexität: Anzahl der interagierenden Elemente?
  - Nichtdeterminismus: Wie vorhersagbar sind die Interaktionen?

Komplexität und Nichtdeterminismus

- Große Systeme haben zwei Dimensionen, die das Verständnis erschweren
  - Komplexität: Anzahl der interagierenden Elemente?
  - Nichtdeterminismus: Wie vorhersagbar sind die Interaktionen?

- **Beispiel**: Das gesamte Universum
  - Viele Elemente (Atome), die zufällig interagieren (Quantenmechanik)
  - Für das Funktionieren des Universums ist das egal
  - Für unser Verständnis vom Universum ein Problem

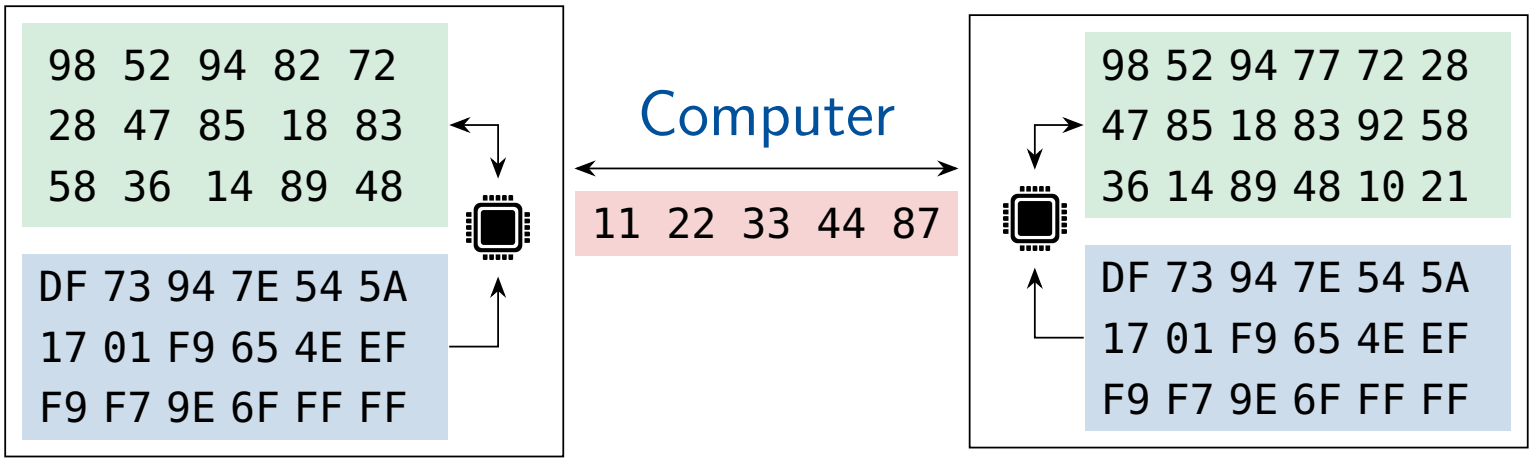
Komplexität und Nichtdeterminismus

- Große Systeme haben zwei Dimensionen, die das Verständnis erschweren
  - Komplexität: Anzahl der interagierenden Elemente?
  - Nichtdeterminismus: Wie vorhersagbar sind die Interaktionen?

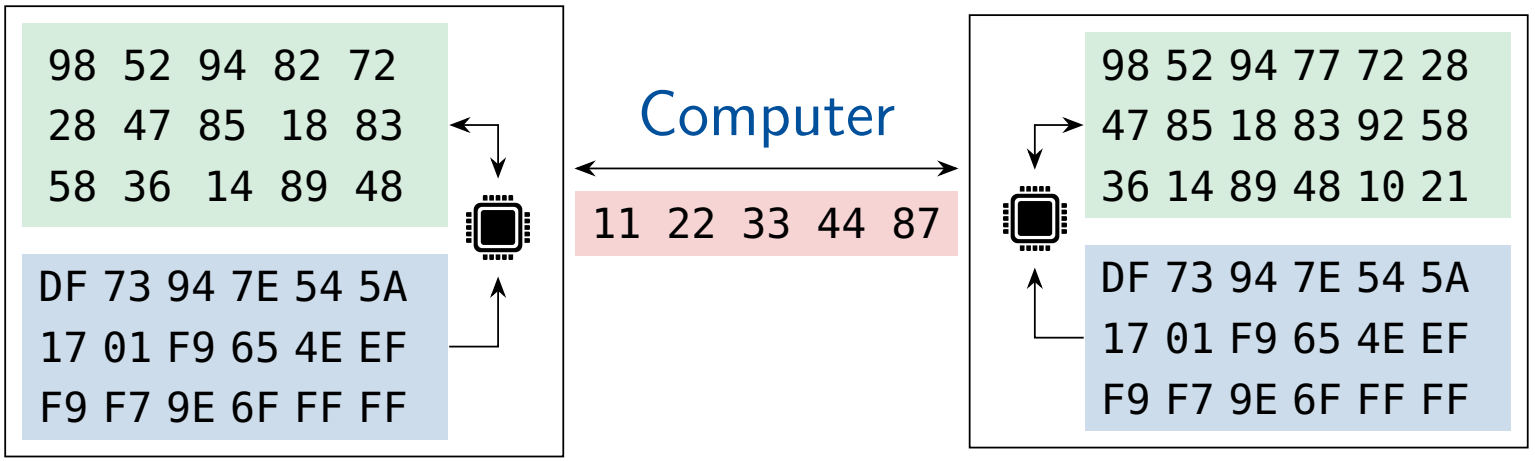
- **Beispiel**: Das gesamte Universum
  - Viele Elemente (Atome), die zufällig interagieren (Quantenmechanik)
  - Für das Funktionieren des Universums ist das egal
  - Für unser Verständnis vom Universum ein Problem

- Ein großes Programm als ein großes System
  - Viele Elemente (Daten) interagieren miteinander (abhängige Operationen).
  - Für die Maschine egal. Operationen werden stumpf nacheinander ausgeführt.
  - Problem Menschen: Da sie es nicht verstehen, können sie es nicht bauen.

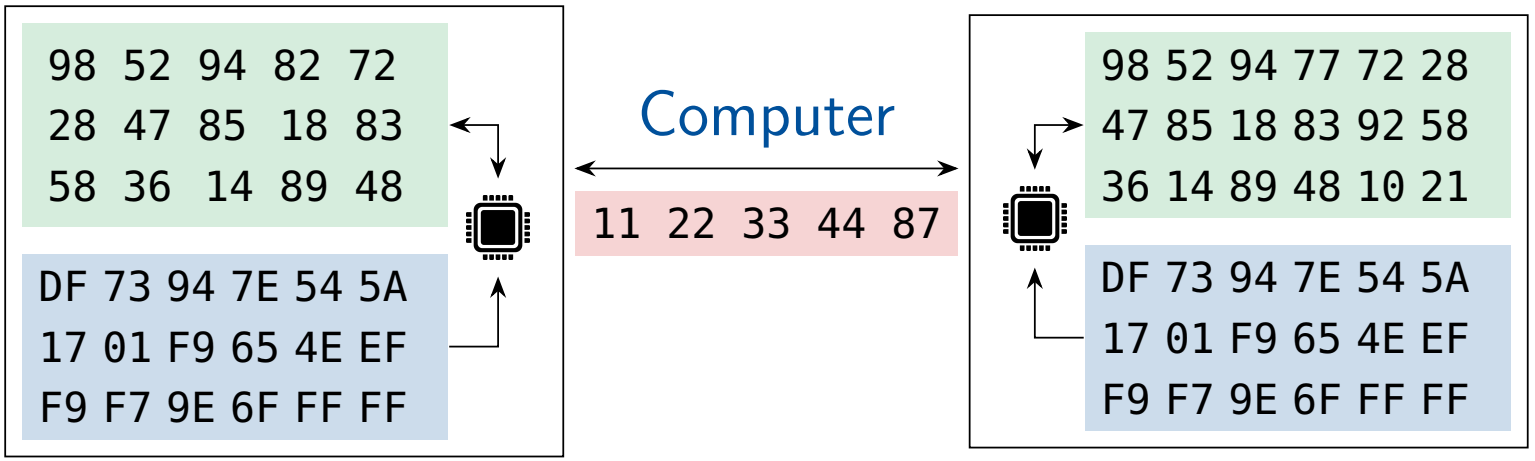
Mensch vs. Maschine

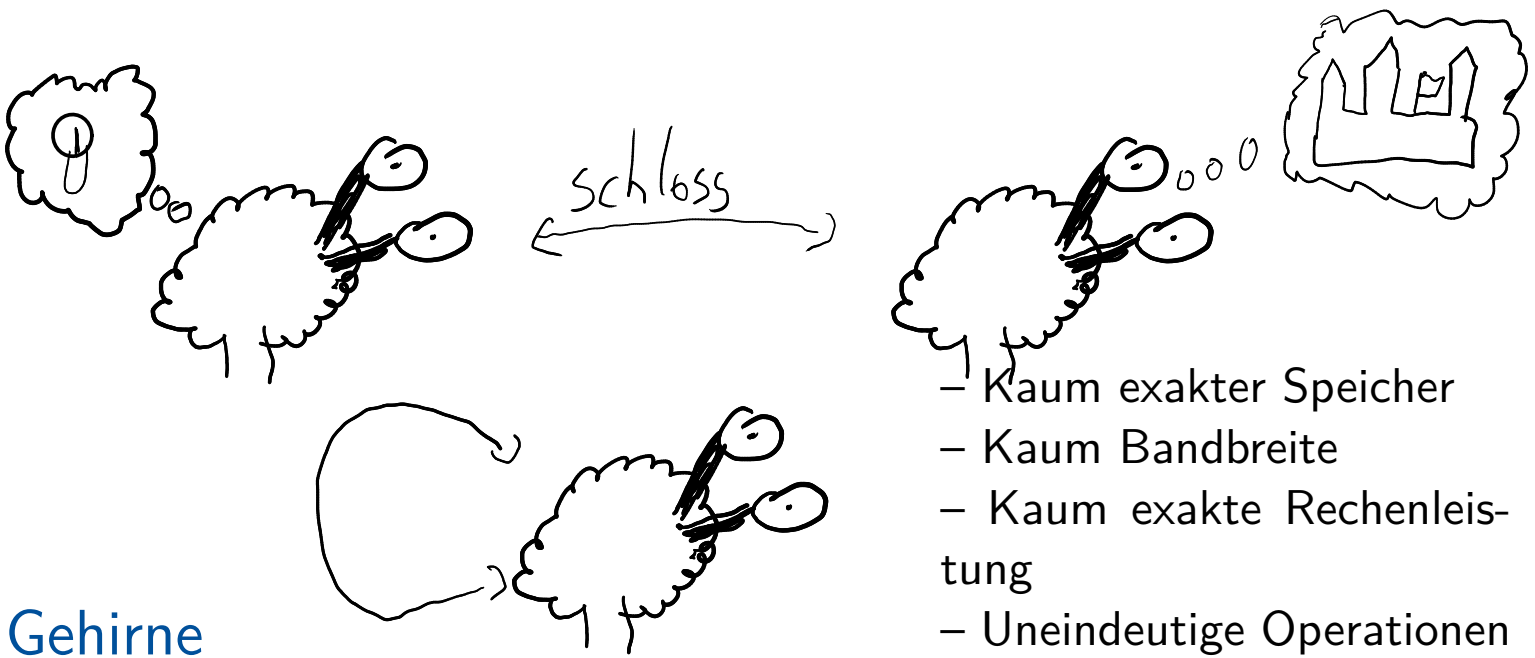


Mensch vs. Maschine



Mensch vs. Maschine

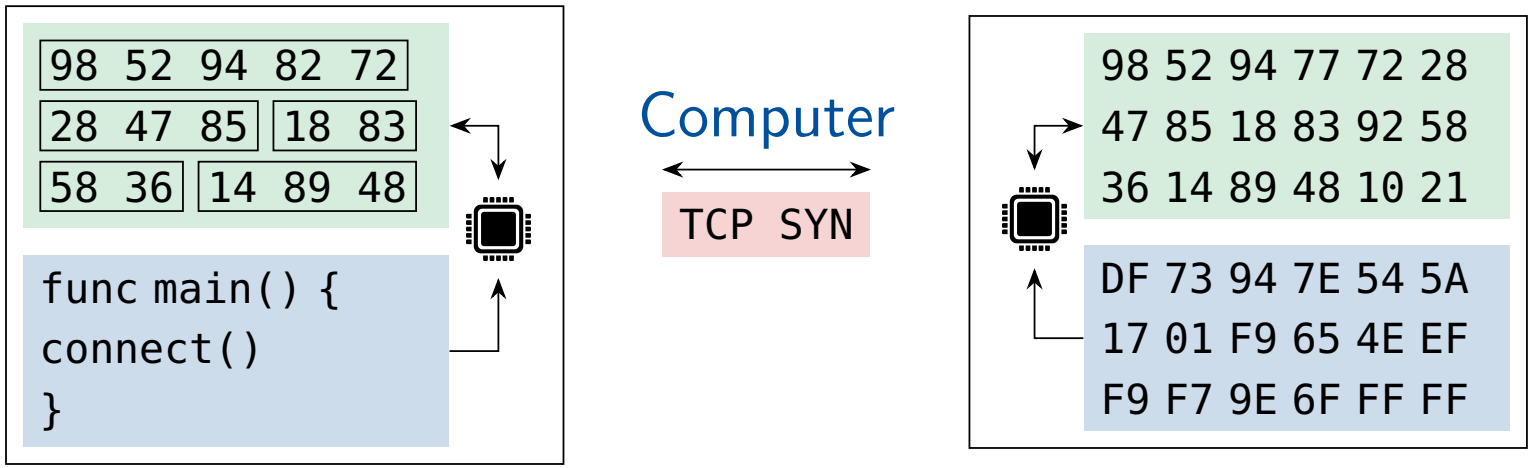




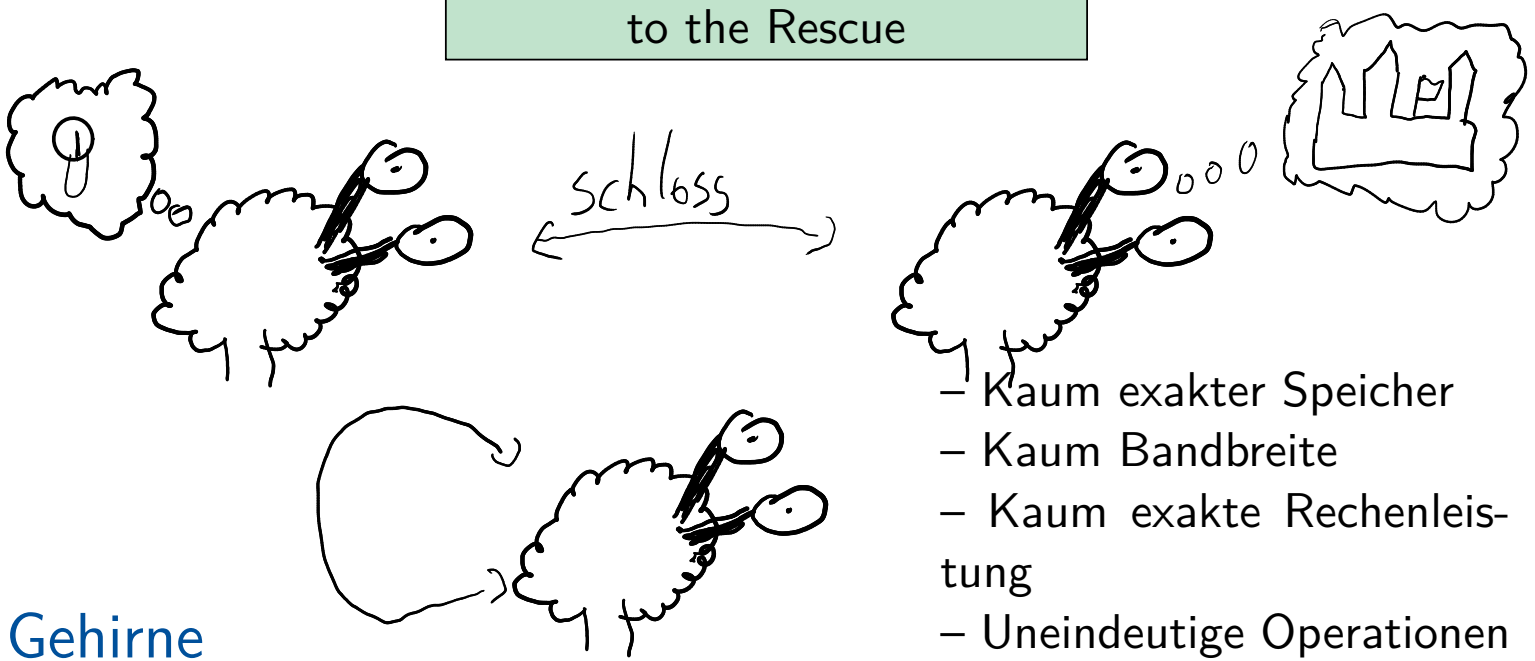
Gehirne

– Kaum exakter Speicher
– Kaum Bandbreite
– Kaum exakte Rechenleistung
– Uneindeutige Operationen

Mensch vs. Maschine



Abstraktionen und Strukturen to the Rescue



Gehirne

– Kaum exakter Speicher
– Kaum Bandbreite
– Kaum exakte Rechenleistung
– Uneindeutige Operationen

Konzepte, Paradigmen, Prinzipien

Damit wir Menschen **komplexe Systeme** bauen, verstehen und kommunizieren können, brauchen wir Abstraktionen und Strukturen.

- Sprachen bieten Abstraktionen zum Denken und Kommunizieren
  - Konzepte: Welche Abstraktionen kann es überhaupt geben?

    Beispiele: Records, Zustand, Prozeduren, Closures, Threads, Nachrichten

  - Paradigmen: Welche Abstraktionen nutze ich in Kombination?

    Beispiel: Imperativ = Records + Zustand + Sequenzierung + Prozeduren

  - Prinzipien: Wie nutze ich die Abstraktionen (ohne Knieschuss)?

    Beispiel: Liskovsches Substitutionsprinzip

Konzepte, Paradigmen, Prinzipien

Damit wir Menschen **komplexe Systeme** bauen, verstehen und kommunizieren können, brauchen wir Abstraktionen und Strukturen.

- Sprachen bieten Abstraktionen zum Denken und Kommunizieren
  - Konzepte: Welche Abstraktionen kann es überhaupt geben?

    Beispiele: Records, Zustand, Prozeduren, Closures, Threads, Nachrichten

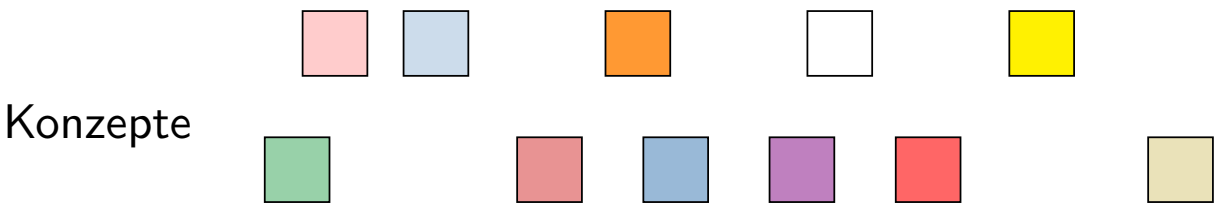
  - Paradigmen: Welche Abstraktionen nutze ich in Kombination?

    Beispiel: Imperativ = Records + Zustand + Sequenzierung + Prozeduren

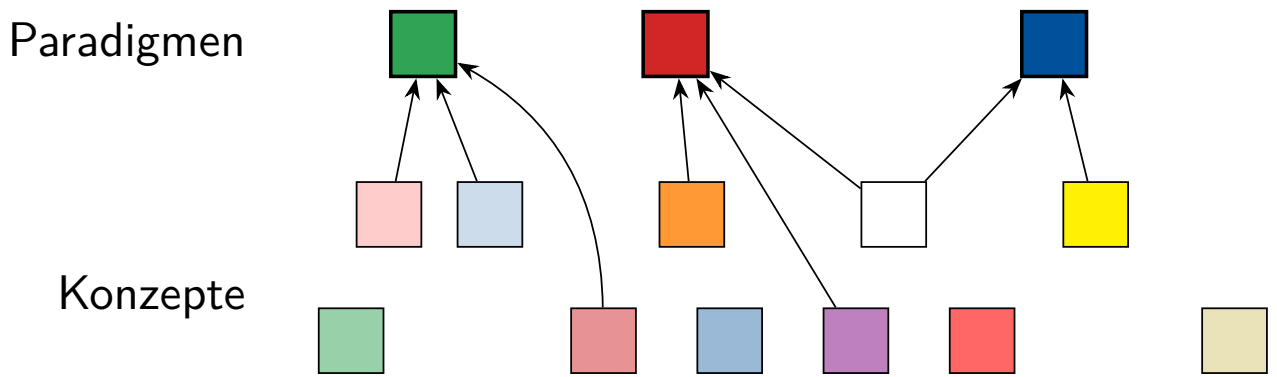
  - Prinzipien: Wie nutze ich die Abstraktionen (ohne Knieschuss)?

    Beispiel: Liskovsches Substitutionsprinzip
    $\Rightarrow$ Software Engineering

Hierarchie der Begrifflichkeiten

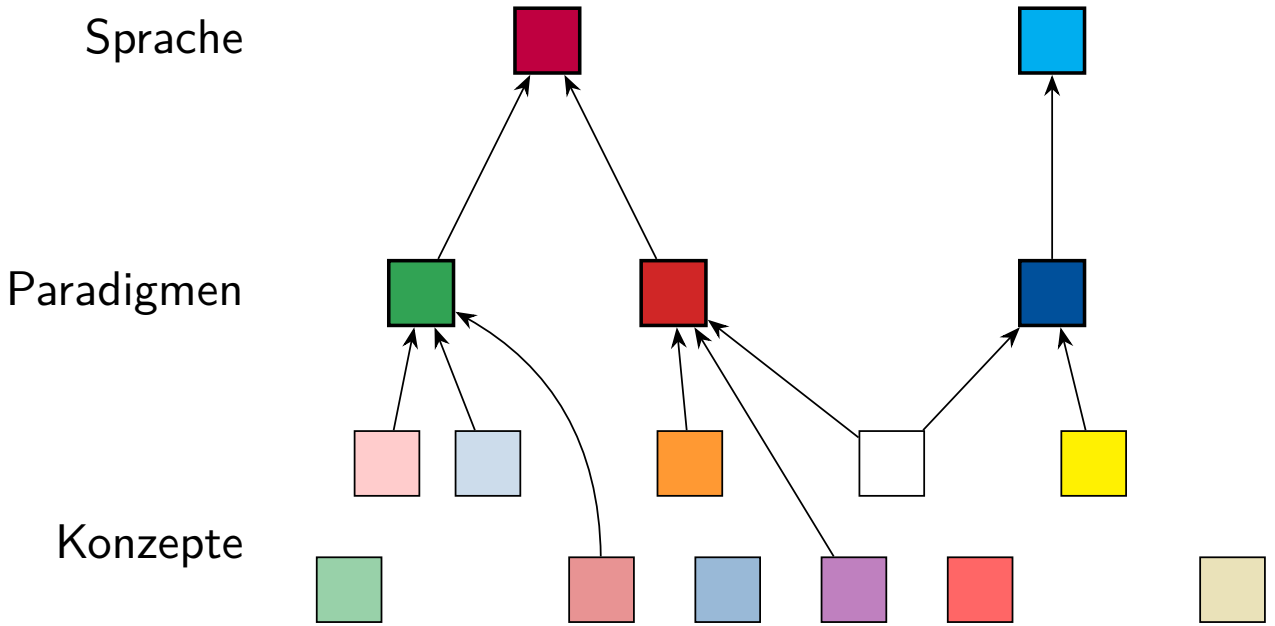


Hierarchie der Begrifflichkeiten



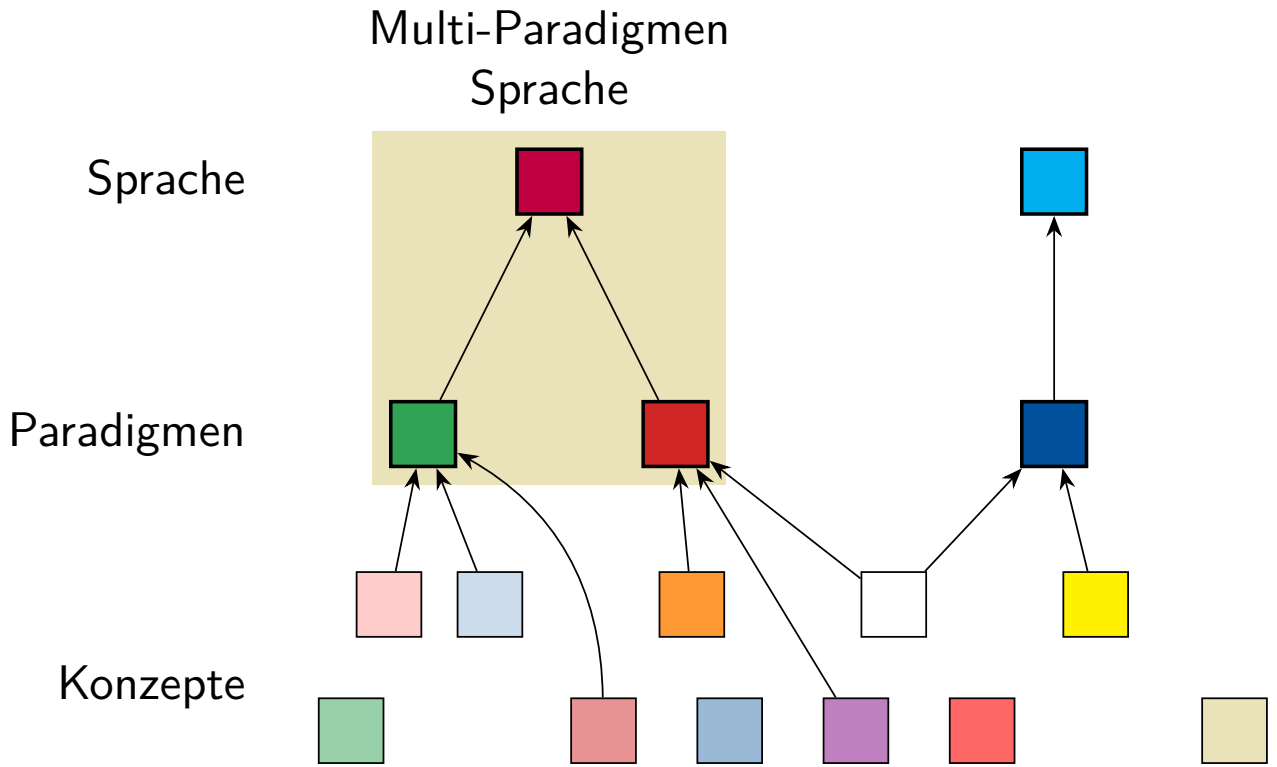
- Paradigmen bündeln Konzepte zu einem konsistenten Programmiermodell.

Hierarchie der Begrifflichkeiten



- Paradigmen bündeln Konzepte zu einem konsistenten Programmiermodell.
- Es gibt **viel mehr** Sprachen als Paradigmen.

Hierarchie der Begrifflichkeiten

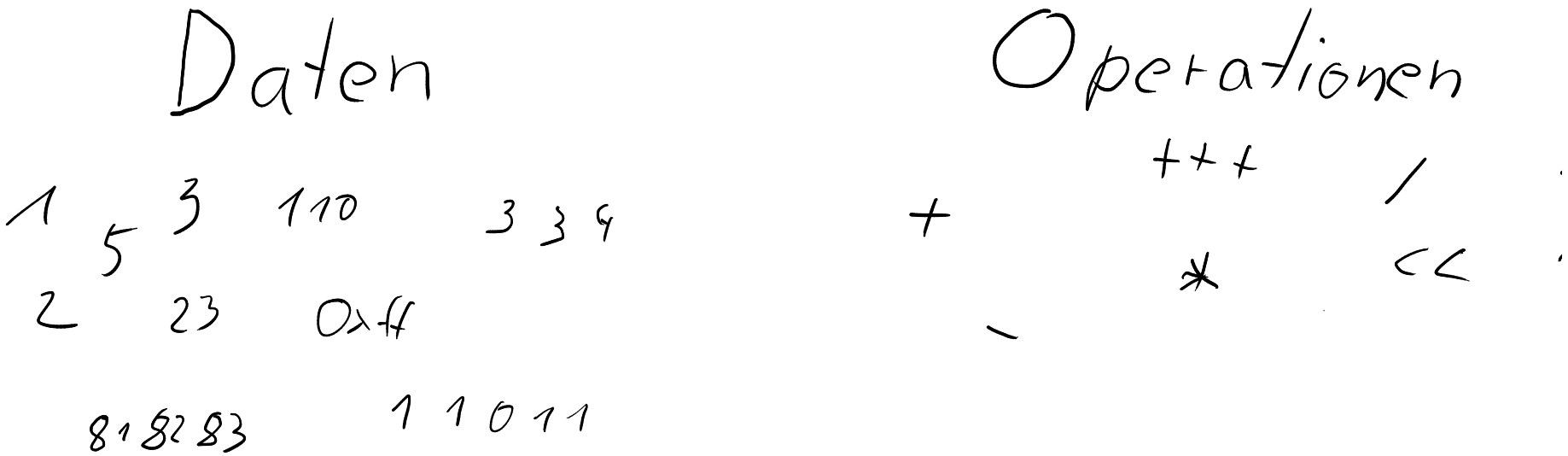


- Paradigmen bündeln Konzepte zu einem konsistenten Programmiermodell.
- Es gibt **viel mehr** Sprachen als Paradigmen.
- Die meisten Sprachen bedienen mehr als ein Paradigma.

  Oft: Ein Paradigma für Grobstruktur + Ein Paradigma für die Feinstruktur

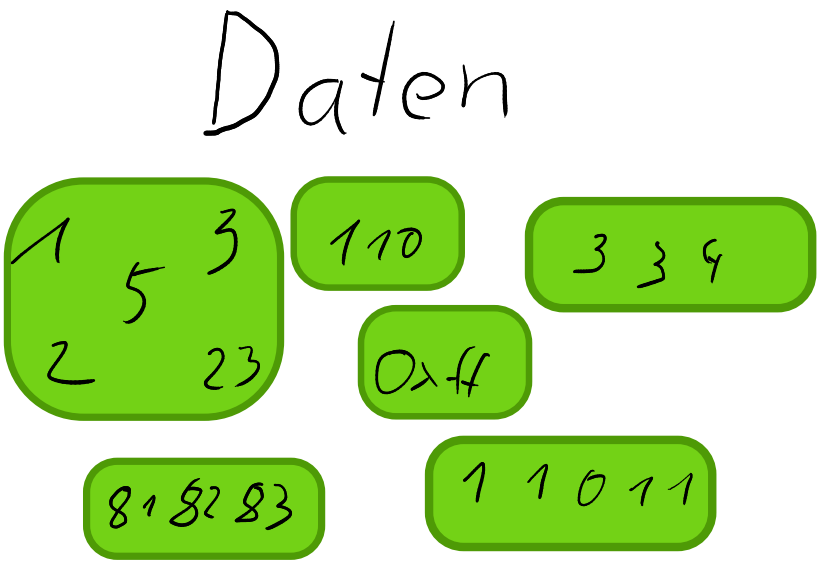
Das Imperative Paradigma

In the Beginning,...



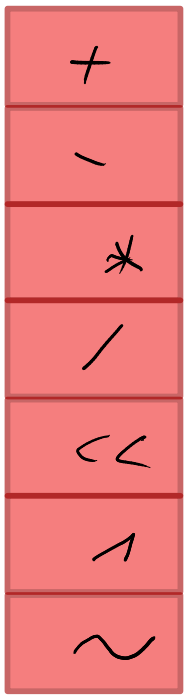
- ...all data was unstructured and all operations were floating.

In the Beginning,...

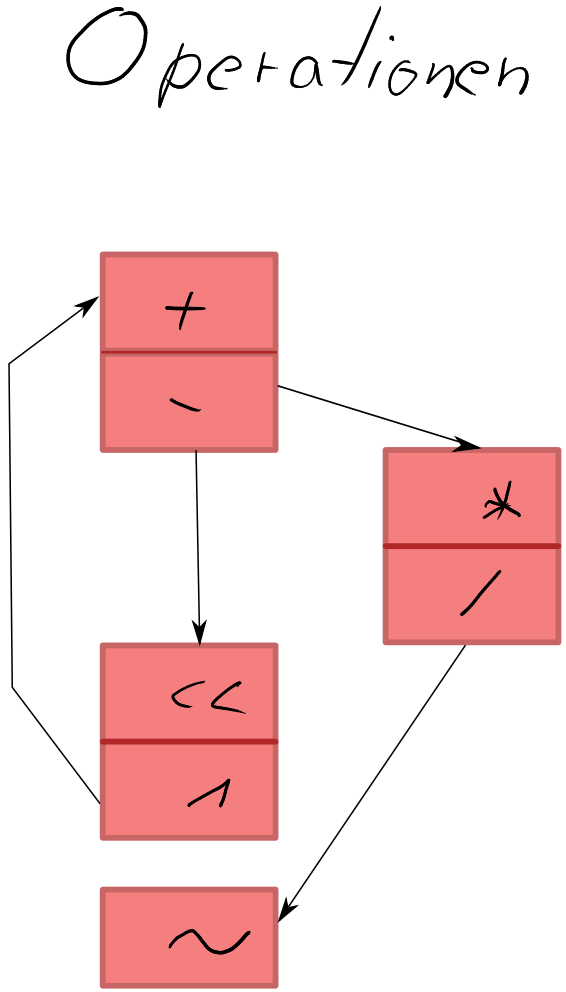


- ...all data was unstructured and all operations were floating.
  - += Records: Daten werden zu Datenstrukturen zusammengefasst.
  - += Sequenzierung: Befehle nacheinander ausführen, Ergebnisse weitergegeben
- Beispiel: Reverse Polish Notation
  - Von links nach rechts ausführen, impliziter Stack für die Berechnung
  - `1 3 + 5 4 - * == (1 + 3) * (5 - 4)`

Operationen

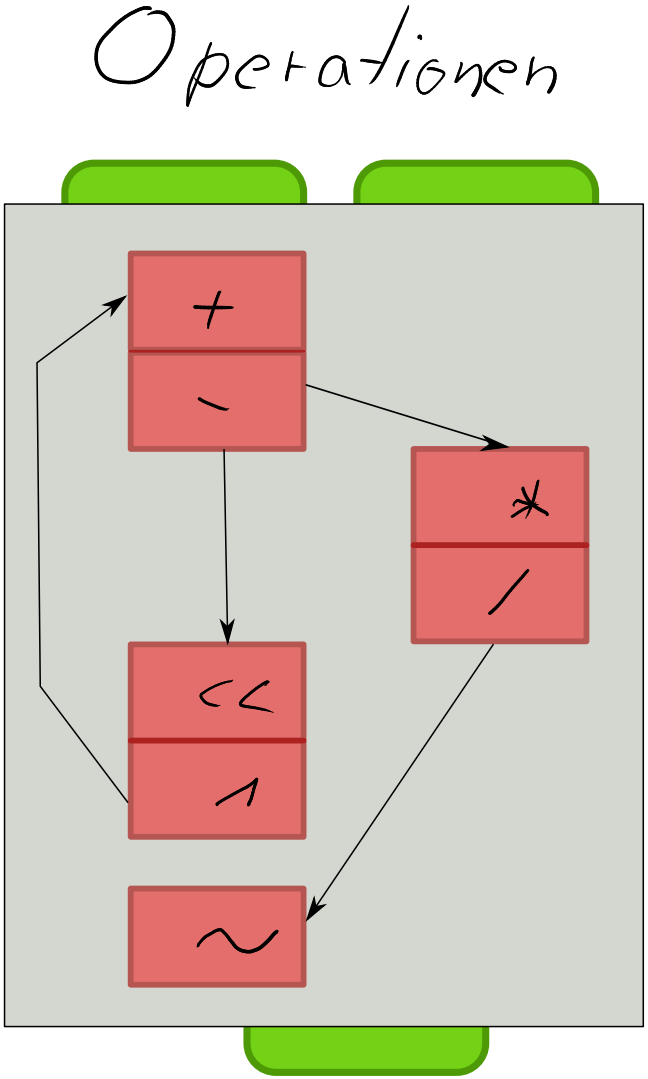


Kontrollfluss und Prozeduren



- Komplexe Abläufe als Kontrollfluss
  - += Selektion: Endlich können wir eine Operation auch mal auslassen.
  - += Iteration: Endlich können wir eine Operation mehrfach ausführen ohne sie doppelt zu notieren.

Kontrollfluss und Prozeduren



- Komplexe Abläufe als Kontrollfluss
  - += Selektion: Endlich können wir eine Operation auch mal auslassen.
  - += Iteration: Endlich können wir eine Operation mehrfach ausführen ohne sie doppelt zu notieren.
- **Wiederverwendung** durch Prozeduren
  - Operationen werden ein Verbund und bekommen einen Namen.
  - Parameter und Rückgabewerte
  - += Invokation: Starte die Ausführung des Operationsverbundes.

(Beinahe) Synonyme: Funktionen, Prozedur, Routinen

Entwurfsprinzip: Prozedurale Abstraktion

Technische Universität Braunschweig

- Prozeduren sind eine **technische Einrichtung** der Sprache
  - Gibt uns das grundlegende Prinzip der Code-Wiederverwendung
  - Die Sprache schreibt **nicht** vor, wie diese zu Verwenden sind.

```
def step3():
    step1()
    doIntermediateStep()
    step2()
```

Entwurfsprinzip: Prozedurale Abstraktion

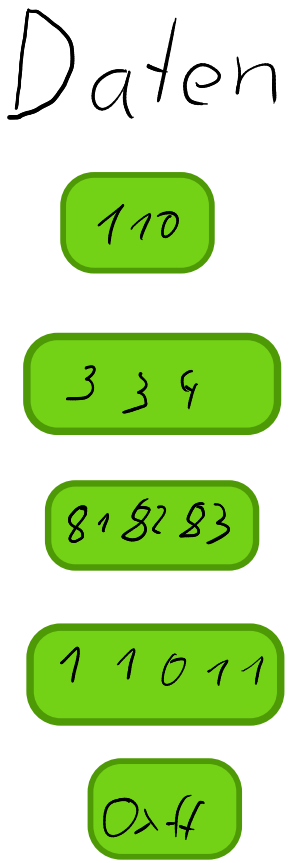
- Prozeduren sind eine **technische Einrichtung** der Sprache
  - Gibt uns das grundlegende Prinzip der Code-Wiederverwendung
  - Die Sprache schreibt **nicht** vor, wie diese zu Verwenden sind.

```
def step3():
    step1()
    doIntermediateStep()
    step2()
```

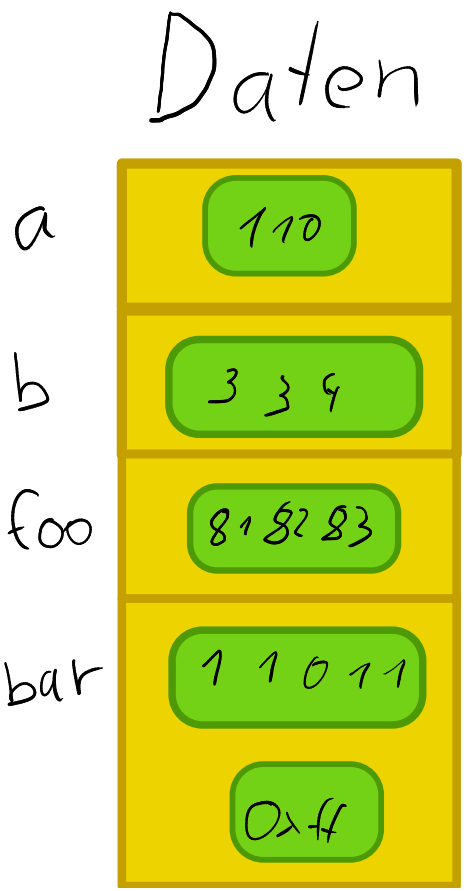
- Prozedurale Abstraktion: Zerlegung des Problems in Teilprobleme
  - Aufteilung des Codes anhand **problemspezifischer Grenzen**
  - Eine Funktionalität → Eine Prozedur
  - Implementierung der Funktionalität wird hinter dem Funktionsnamen versteckt und kann problemlos ausgetauscht werden (z.B. Bugfix).

```
char *strstr(const char *haystack, const char* needle);
```

Benannter und veränderlicher Zustand

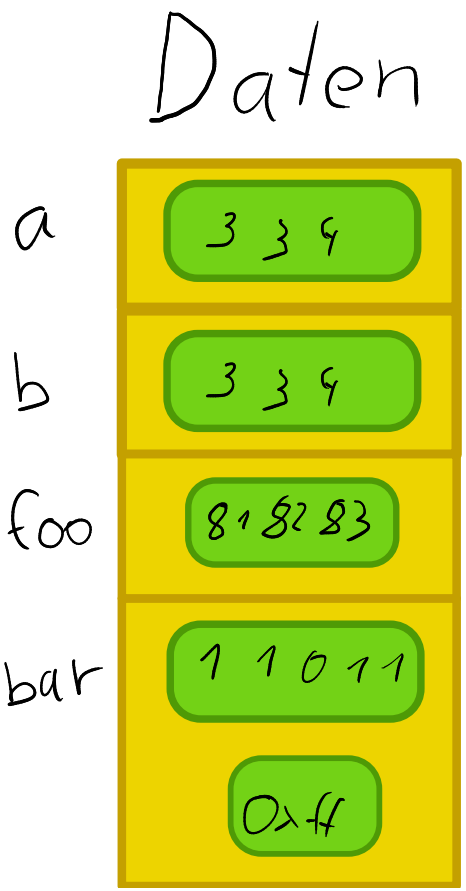


Benannter und veränderlicher Zustand



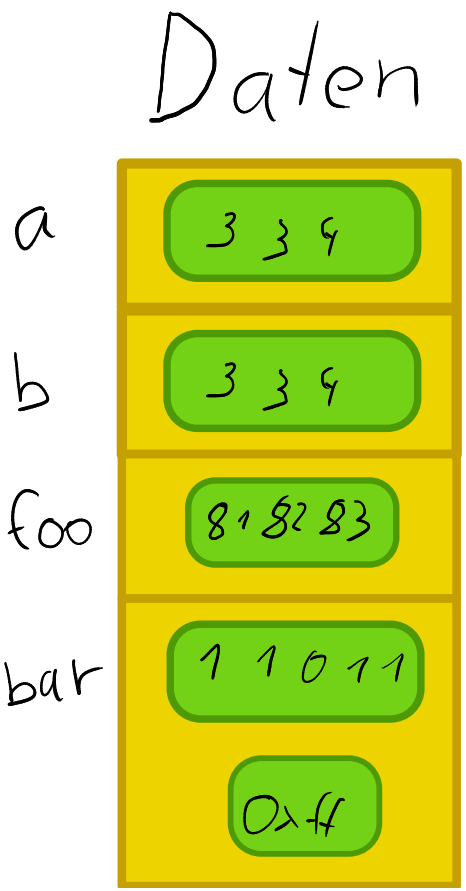
- Eingaben und (Zwischen-)Ergebnisse speichern
  - Speicherzellen mit symbolischem Namen
  - Objekte werden an diese Namen gebunden
  - Späterer Zugriff über das Symbol

Benannter und veränderlicher Zustand



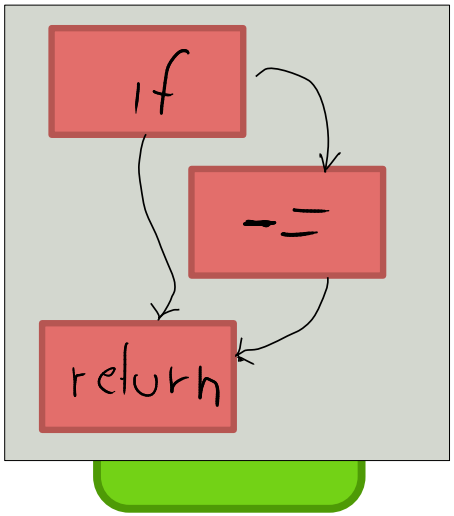
- Eingaben und (Zwischen-)Ergebnisse speichern
  - Speicherzellen mit symbolischem Namen
  - Objekte werden an diese Namen gebunden
  - Späterer Zugriff über das Symbol
- Veränderlicher Zustand ist ein eigenes Konzept
  - += Variablen und Zuweisung
  - Gebundene Namen können neu gebunden werden
  - Flexibler Datenfluss und Schleifen mit Auswirkung

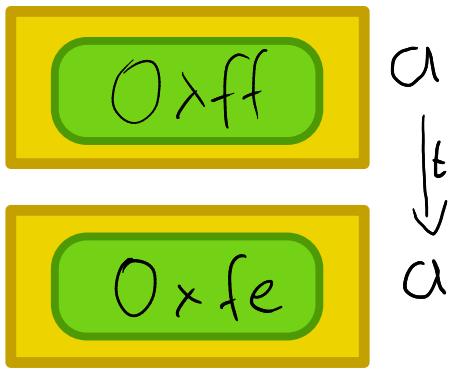
Benannter und veränderlicher Zustand



- Eingaben und (Zwischen-)Ergebnisse speichern
  - Speicherzellen mit symbolischem Namen
  - Objekte werden an diese Namen gebunden
  - Späterer Zugriff über das Symbol
- Veränderlicher Zustand ist ein eigenes Konzept
  - += Variablen und Zuweisung
  - Gebundene Namen können neu gebunden werden
  - Flexibler Datenfluss und Schleifen mit Auswirkung
- Variablen sind Quelle unendlich vieler **Bugs**
  - Variablen sind ein Seitenkanal zu den Parametern
  - Funktionsverhalten kann überraschend werden
  - Zustand kann inkonsistent werden

Das imperatives Programmierparadigma

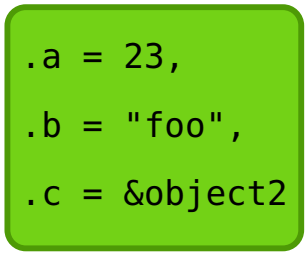




- Das imperative Programmierparadigma umfasst mehrere Konzepte
  - Operationen: Sequenzierung, Kontrollstrukturen, Prozeduren
  - Daten: strukturierte Daten, benannter und veränderlicher Zustand
  - Beispiele für rein imperative Sprachen: Fortran, COBOL, PL/I, C, Pascal
- Imperatives Programmieren ist oft das Paradigma „im Kleinen“
  - Starke Kontrolle des Programmierers über die Ausführung (−, +)
  - Java, Rust, C++ bieten alle das imperative Paradigma an.

Das objektorientierte Programmierparadigma

Probleme imperativen Programmierens

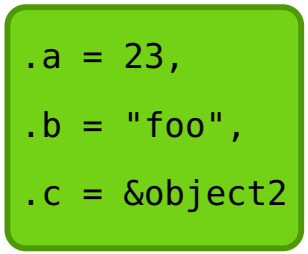



```
incrementCounterOfQueue(queue_t)
getNameOfQueue(queue_t)
popObjectFromQueue(queue_t)
```

- Imperativ: **Betonung** der Operationen bei **Vernachlässigung** der Daten
  - Codeduplikation, da Funktionen strikt an ihre Datentypen gekoppelt sind
  - Keine Kontrolle, da programmweiter Zugriff auf Datenstrukturen möglich
  - Leicht unübersichtlich, da kaum bis keine Hierarchie der Funktionen

  $\Rightarrow$ Erfordert Disziplin und geistige Kapazität von den Entwicklern

Probleme imperativen Programmierens




```
incrementCounterOfQueue(queue_t)
getNameOfQueue(queue_t)
popObjectFromQueue(queue_t)
```

- Imperativ: **Betonung** der Operationen bei **Vernachlässigung** der Daten
  - Codeduplikation, da Funktionen strikt an ihre Datentypen gekoppelt sind
  - Keine Kontrolle, da programmweiter Zugriff auf Datenstrukturen möglich
  - Leicht unübersichtlich, da kaum bis keine Hierarchie der Funktionen

  $\Rightarrow$ Erfordert Disziplin und geistige Kapazität von den Entwicklern
- Besonders problematisch bei **großen Softwareprojekten**
  - Wartung von dupliziertem Code macht N-fachen Aufwand.
  - Viele Entwickler arbeiten an der gleichen Quellcodebasis.
  - Prozedurale Abstraktion bietet nur eine Ebene von Abstraktion.

„Objekte“ als zentraler Begriff

Technische Universität Braunschweig

Menschen denken eher in Objekten (Nomen) als in Aktionen (Verben)

Prozedural

Fahrrad waschen
Apfel waschen

Objektorientiert

Fahrrad waschen
fahren

„Objekte“ als zentraler Begriff

Technische Universität Braunschweig

Menschen denken eher in Objekten (Nomen) als in Aktionen (Verben)

Prozedural

Fahrrad waschen
Apfel waschen

Objektorientiert

Fahrrad waschen
fahren

- Entwurf entlang von Objekten spiegelt menschliche Erfahrungswelten
  - Eine Fabrik besteht aus: Hallen, Arbeitern, Maschinen, einem Gelände.
  - All diese Entitäten haben (veränderliche) Eigenschaften.
  - Entitäten interagieren miteinander anstatt verarbeitet zu werden:
    Arbeiter montiert Reifen $\Rightarrow$ montieren(reifen, by=arbeiter)

„Objekte“ als zentraler Begriff

Technische Universität Braunschweig

Menschen denken eher in Objekten (Nomen) als in Aktionen (Verben)

Prozedural

Fahrrad waschen
Apfel waschen

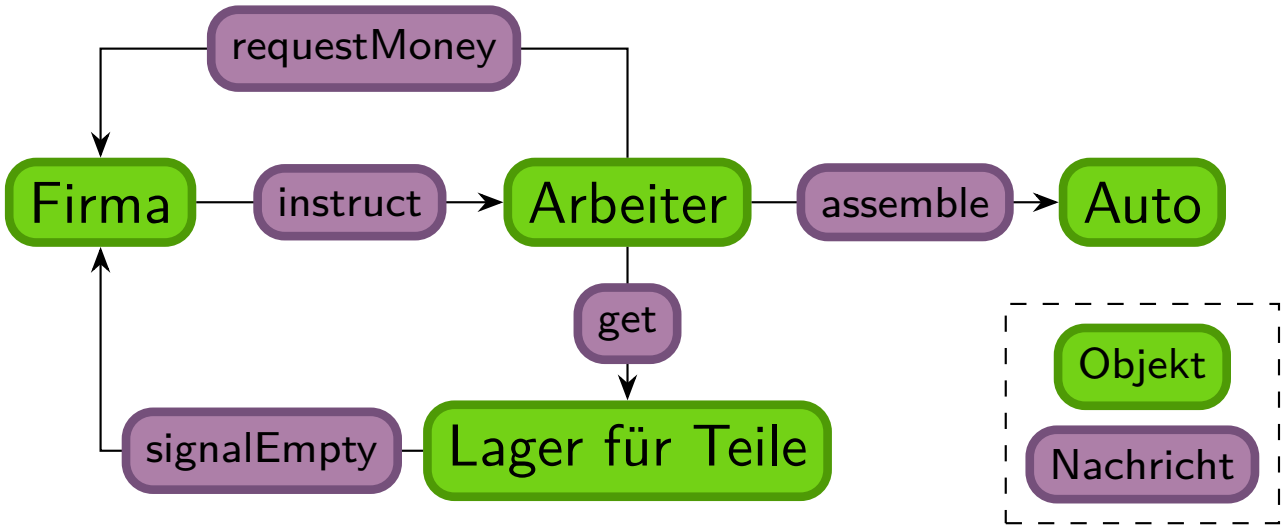
Objektorientiert

Fahrrad waschen
fahren

- Entwurf entlang von Objekten spiegelt menschliche Erfahrungswelten
  - Eine Fabrik besteht aus: Hallen, Arbeitern, Maschinen, einem Gelände.
  - All diese Entitäten haben (veränderliche) Eigenschaften.
  - Entitäten interagieren miteinander anstatt verarbeitet zu werden:
    Arbeiter montiert Reifen $\Rightarrow$ montieren(reifen, by=arbeiter)

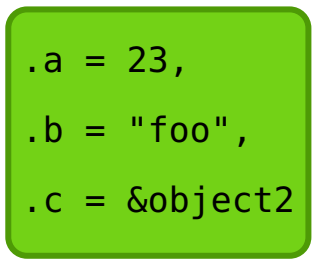
Diese (objektorientierten) Objekte sind eine Erweiterung unserer bisherigen Objekte aus Vorlesung ”06 - Objekte”.

Nachrichten und Objekte

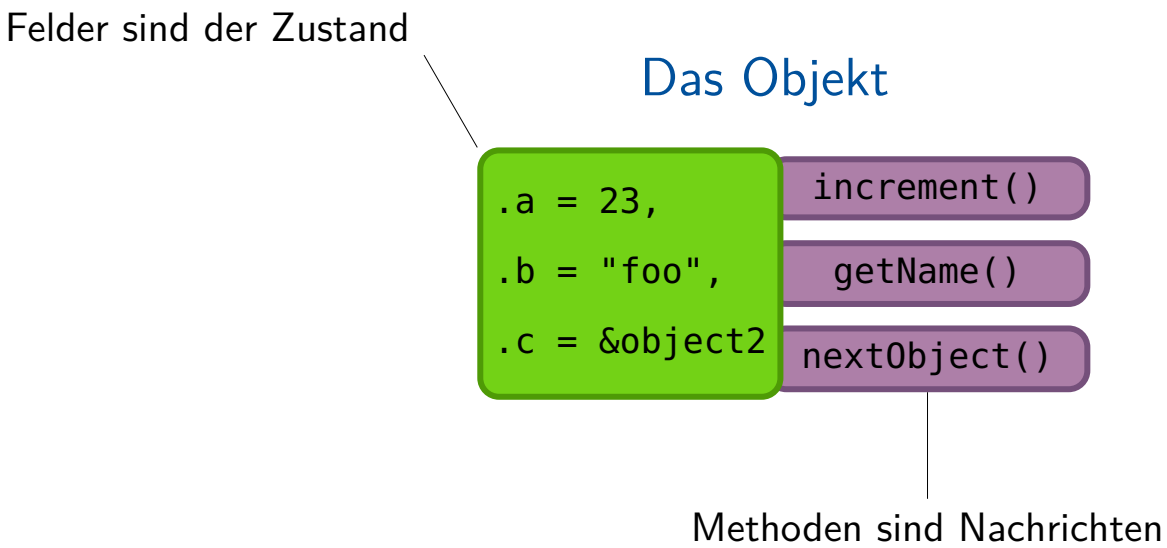


Definition nach Alan Kay (Erfinder von Smalltalk)

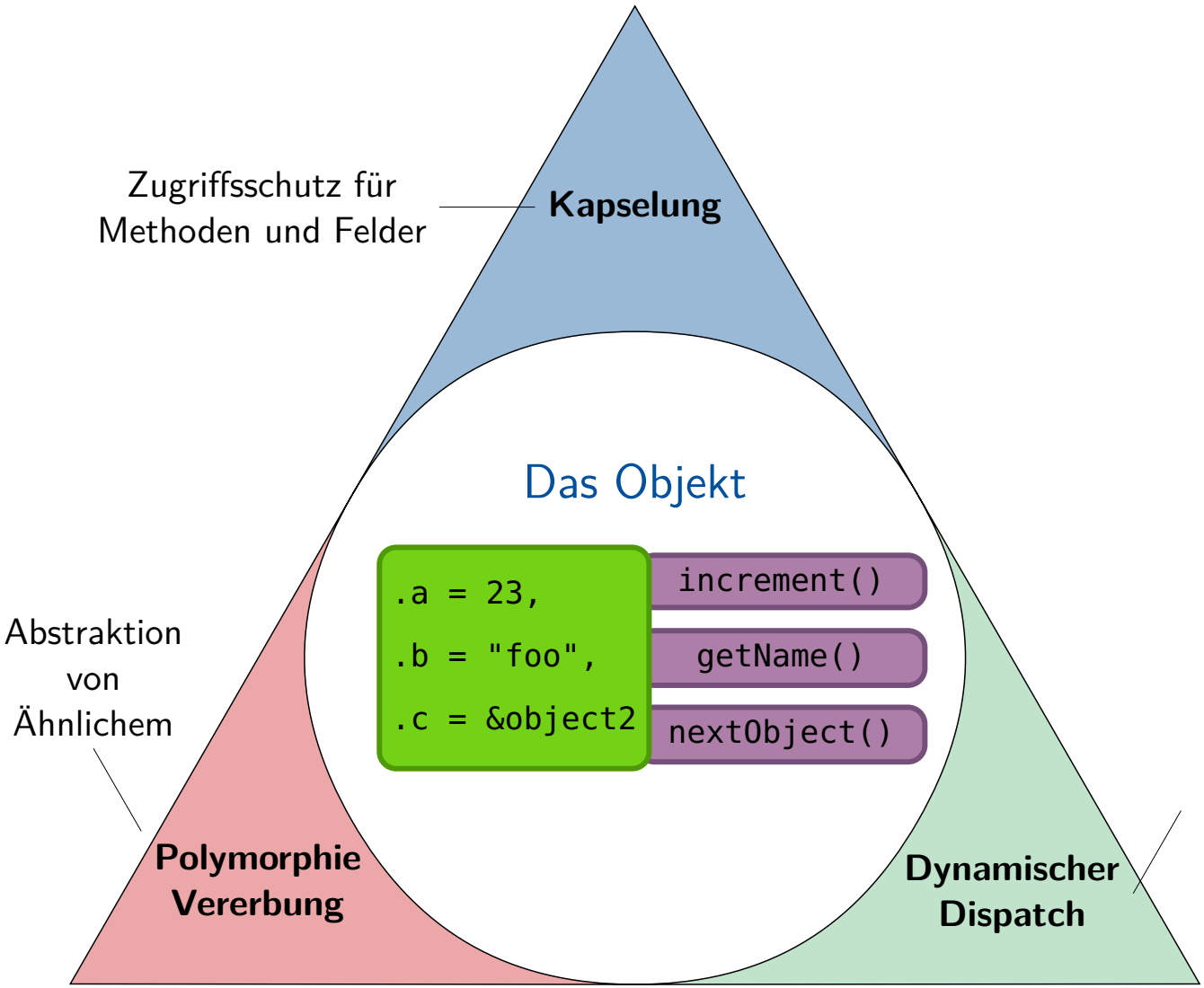
1. Messaging: Objekte kommunizieren durch den Versand von Nachrichten.
2. Persistenz: Objekte haben privaten Zustand, auch wenn sie nicht aktiv sind.
3. Kapselung: Informationen werden im Objekt verborgen.
4. Späte Bindung: Ausgeführte Operation wird möglichst spät ausgewählt.




```
incrementCounterOfQueue(queue_t)
getNameOfQueue(queue_t)
popObjectFromQueue(queue_t)
```


Felder sind der Zustand
Das Objekt
.a = 23,
.b = "foo",
.c = &object2
increment()
getName()
nextObject()
Methoden sind Nachrichten

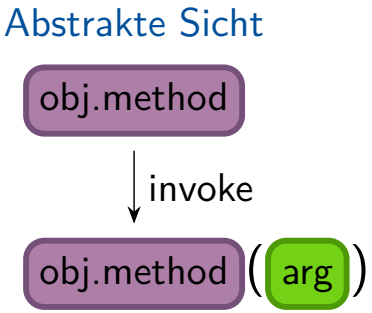
Konzepte für Objektorientierung



Spätes Binden
Auswahl der Prozedur
beim Nachrichtenempfang

Die Methode als die zentrale Erweiterung

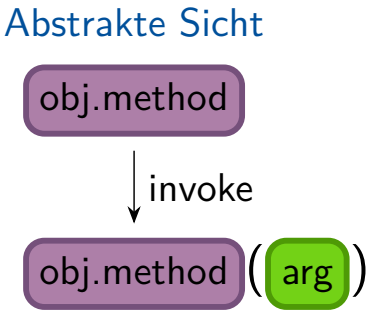
- Methoden haben einen Ausführungskontext mit dem Objekt
  - Objekt, auf dem die Methode aufgerufen wird, wird sichtbar.
  - Objektattribute sind in der Methode zugreifbar.
  - Methode ist eine Closure! (Siehe „03 - Namen“)
- Closure: Eine Prozedur, die einen Ausführungskontext mitbringt



  - Prozeduren werden **immer** in einem Kontext ausgeführt.
  - Bei C sind nur die Argumente und die globalen Variablen enthalten.
  - Bei Closures sind zusätzliche Objekte an Namen gebunden (z.B. `this`).

Die Methode als die zentrale Erweiterung

- Methoden haben einen Ausführungskontext mit dem Objekt
  - Objekt, auf dem die Methode aufgerufen wird, wird sichtbar.
  - Objektattribute sind in der Methode zugreifbar.
  - Methode ist eine Closure! (Siehe „03 - Namen“)
- Closure: Eine Prozedur, die einen Ausführungskontext mitbringt

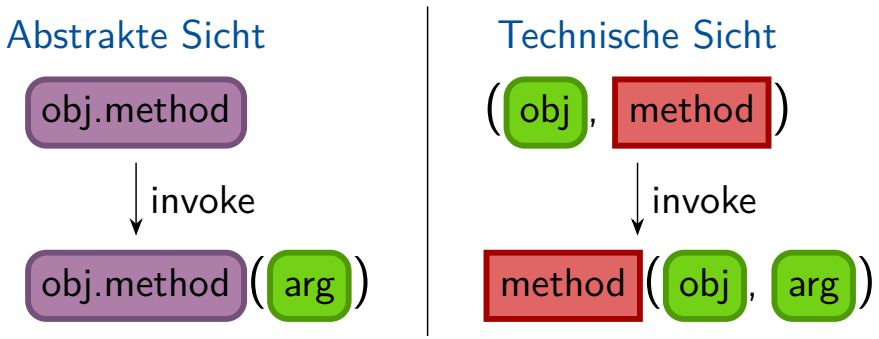


  - Prozeduren werden **immer** in einem Kontext ausgeführt.
  - Bei C sind nur die Argumente und die globalen Variablen enthalten.
  - Bei Closures sind zusätzliche Objekte an Namen gebunden (z.B. `this`).

Die Methode als die zentrale Erweiterung

- Methoden haben einen Ausführungskontext mit dem Objekt
  - Objekt, auf dem die Methode aufgerufen wird, wird sichtbar.
  - Objektattribute sind in der Methode zugreifbar.
  - Methode ist eine Closure! (Siehe „03 - Namen“)

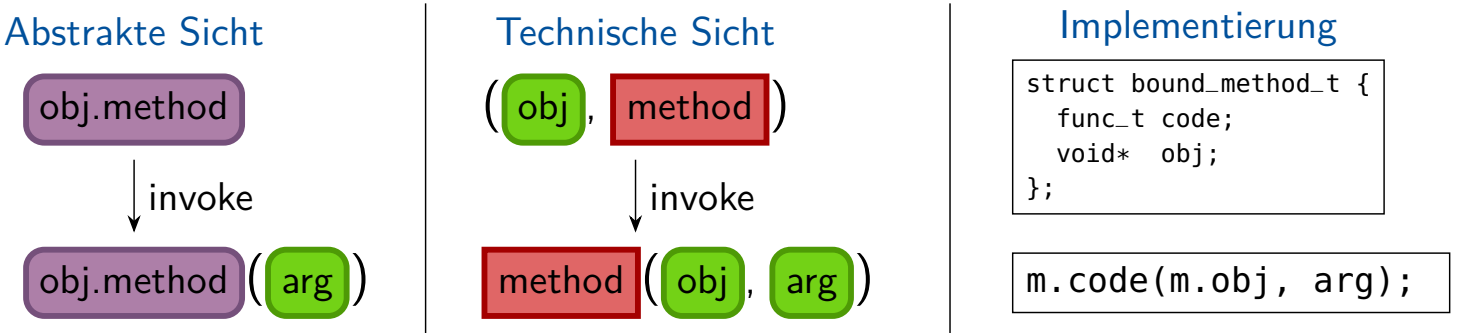
- Closure: Eine Prozedur, die einen Ausführungskontext mitbringt



  - Prozeduren werden **immer** in einem Kontext ausgeführt.
  - Bei C sind nur die Argumente und die globalen Variablen enthalten.
  - Bei Closures sind zusätzliche Objekte an Namen gebunden (z.B. `this`).

Die Methode als die zentrale Erweiterung

- Methoden haben einen Ausführungskontext mit dem Objekt
  - Objekt, auf dem die Methode aufgerufen wird, wird sichtbar.
  - Objektattribute sind in der Methode zugreifbar.
  - Methode ist eine Closure! (Siehe „03 - Namen“)
- Closure: Eine Prozedur, die einen Ausführungskontext mitbringt



  - Prozeduren werden **immer** in einem Kontext ausgeführt.
  - Bei C sind nur die Argumente und die globalen Variablen enthalten.
  - Bei Closures sind zusätzliche Objekte an Namen gebunden (z.B. `this`).

Prototypen

Objektorientierung ohne Klassen

Technische Universität Braunschweig

Da Sie alle OO auf Basis von Klassen zur Genüge kennen, werden wir eine andere Art der OO Programmierung betrachten.

- Javascript kennt **keine Klassen** und ist **dennoch** objektorientiert
  - Javascript-Objekte haben Attribute (✓) und gebundene Methoden (✓)
  - Methoden werden erst beim Aufruf aufgelöst (✓)
  - Informationen können mittels lexikalischem Scoping verborgen (✓) werden
- JavaScript verwendet prototypenbasierte Vererbung
  - Flexibler, erweiterbarer, und mächtiger als klassenbasierte Vererbung
  - Hohe Dynamik, die besonders für eine Skriptsprache geeignet ist
  - Wenige Sprachmechanismen, aber ein gewisser Zoo an syntaktischem Zucker

JavaScript: Die Prototypenkette

Technische Universität Braunschweig

```
var obj = { a: 23 };
obj.zz = 42;
[obj.a, obj["a"], obj["zz"]];
// => [23, 23, 42]
```

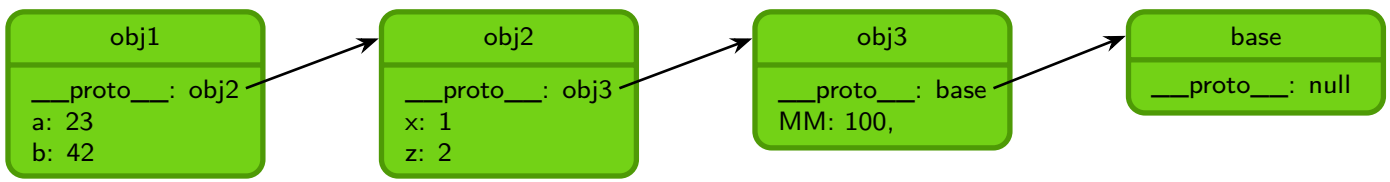
- JavaScript-Objekte sind *„bags“ of properties*
  - Zugriff über `[]` und den Punktoperator
  - Recordtyp und Abbildungstyp fallen zusammen
  - Properties können zu einem Objekt hinzugefügt werden

JavaScript: Die Prototypenkette

```
var obj = { a: 23 };
obj.zz = 42;
[obj.a, obj["a"], obj["zz"]];
// => [23, 23, 42]
```

- JavaScript-Objekte sind *„bags“ of properties*
  - Zugriff über `[]` und den Punktoperator
  - Recordtyp und Abbildungstyp fallen zusammen
  - Properties können zu einem Objekt hinzugefügt werden

- `__proto__`: Fallback für die Suche nach Attributen
  - Situation: Ein Objekt hat das geforderte Attribut nicht.
  - Die Suche wird beim Prototypen, der in `__proto__` referenziert ist, fortgesetzt.
  - Es entsteht die Prototypenkette, die bei `{}` endet.

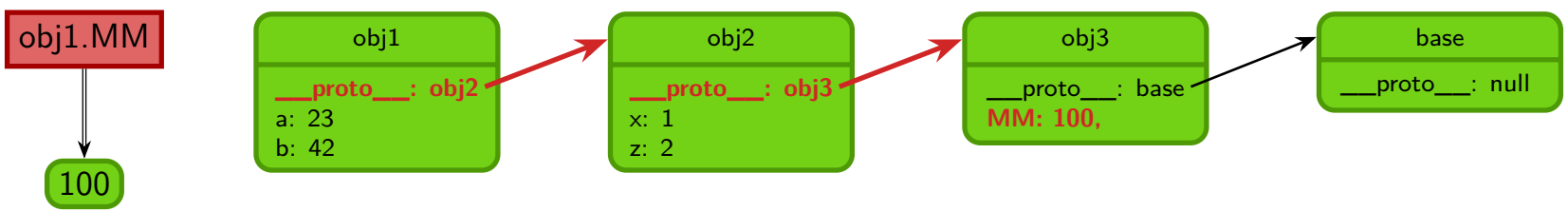


JavaScript: Die Prototypenkette

```
var obj = { a: 23 };
obj.zz = 42;
[obj.a, obj["a"], obj["zz"]];
// => [23, 23, 42]
```

- JavaScript-Objekte sind *„bags“ of properties*
  - Zugriff über `[]` und den Punktoperator
  - Recordtyp und Abbildungstyp fallen zusammen
  - Properties können zu einem Objekt hinzugefügt werden

- `__proto__`: Fallback für die Suche nach Attributen
  - Situation: Ein Objekt hat das geforderte Attribut nicht.
  - Die Suche wird beim Prototypen, der in `__proto__` referenziert ist, fortgesetzt.
  - Es entsteht die Prototypenkette, die bei `{}` endet.



JavaScript: Attribut-Lookup und `this`

Technische Universität Braunschweig

- Prototypenkette implementiert dynamischen und hierarchischen Lookup
  - Objekt in der Kettenmitte erweitern $\rightarrow$ Alle Kinder werden erweitert
  - Speichereffizient, aber mit höheren Lookup-Kosten verbunden

JavaScript: Attribut-Lookup und `this`

- Prototypenkette implementiert dynamischen und hierarchischen Lookup
  - Objekt in der Kettenmitte erweitern → Alle Kinder werden erweitert
  - Speichereffizient, aber mit höheren Lookup-Kosten verbunden
- JavaScript bindet den Namen `this` an das aktuelle Objekt
  - Attribute können auf Funktionen verweisen
  - Objekt-Attribut als Funktion aufrufen ⇒ `this` wird an das Objekt gebunden

```javascript
var obj = {
  prop: 37,
  f: function() {
    return this.prop;
  }
};
var ff = obj.f;
```

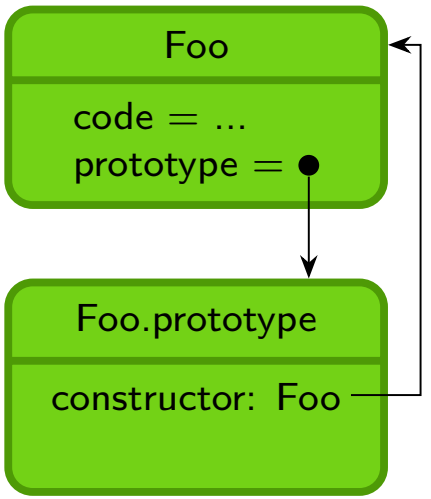
JavaScript

- Aufruf der Funktion als Attribut
  `obj.f()` ⇒ `37`
- Aufruf der „blanken“ Funktion
  `ff()` ⇒ `undefined`

JavaScript: Funktionen als Konstruktoren

- **Alle** Funktionen können als Konstruktoren verwendet werden
  - Das Schlüsselwort `new` ruft Funktion im Kontext eines neuen Objekts auf
  - Jede Funktion hat ein `.prototype`-Attribut, das als `__proto__` gesetzt wird.

```
function Foo(name) {
  this.name = name;
}
```



JavaScript: Funktionen als Konstruktoren

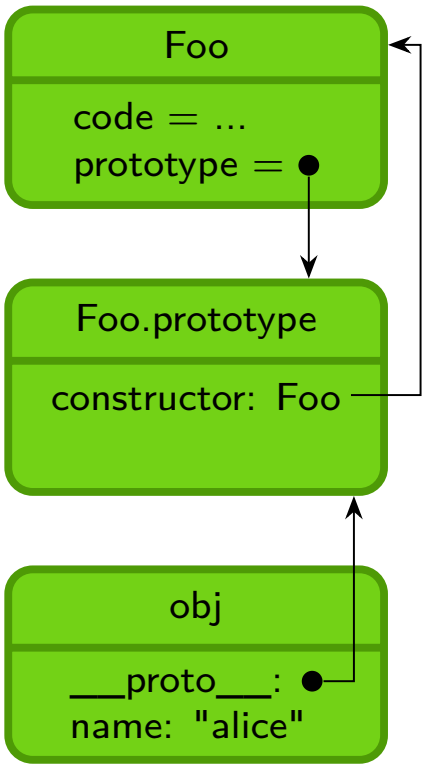
- **Alle** Funktionen können als Konstruktoren verwendet werden
  - Das Schlüsselwort `new` ruft Funktion im Kontext eines neuen Objekts auf
  - Jede Funktion hat ein `.prototype`-Attribut, das als `__proto__` gesetzt wird.

```
function Foo(name) {
  this.name = name;
}

var obj = new Foo("alice");
```

dies ist äquivalent zu

```
var obj = {
  __proto__: Foo.prototype
}
// Aufruf mit this == obj
obj.constructor("alice");
```



JavaScript: Funktionen als Konstruktoren

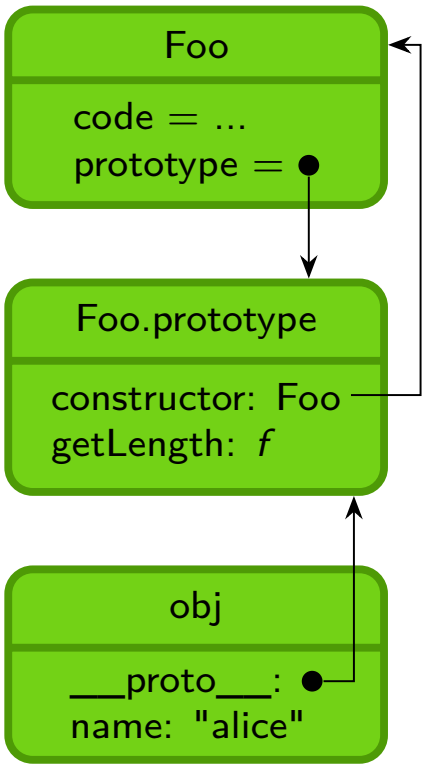
- **Alle** Funktionen können als Konstruktoren verwendet werden
  - Das Schlüsselwort `new` ruft Funktion im Kontext eines neuen Objekts auf
  - Jede Funktion hat ein `.prototype`-Attribut, das als `__proto__` gesetzt wird.

```
function Foo(name) {
  this.name = name;
}

var obj = new Foo("alice");

Foo.prototype.getLength = \
  function() {
    return this.name.length;
  };

obj.getLength() // => 5
```



JavaScript: Funktionen als Konstruktoren

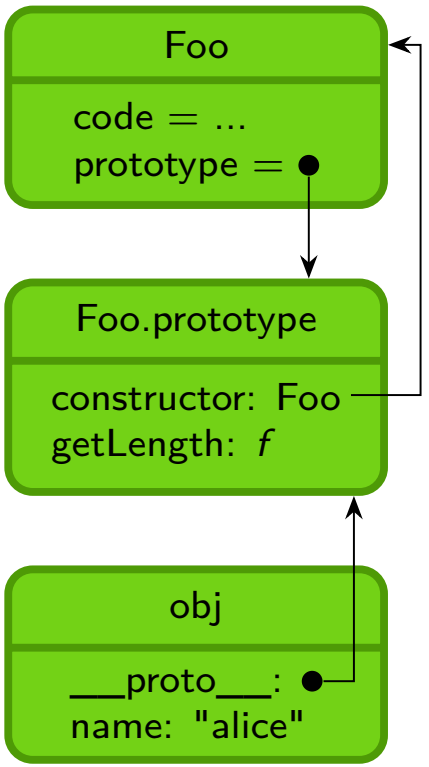
- **Alle** Funktionen können als Konstruktoren verwendet werden
  - Das Schlüsselwort `new` ruft Funktion im Kontext eines neuen Objekts auf
  - Jede Funktion hat ein `.prototype`-Attribut, das als `__proto__` gesetzt wird.

```
function Foo(name) {
  this.name = name;
}

var obj = new Foo("alice");

Foo.prototype.getLength = \
  function() {
    return this.name.length;
  };

obj.getLength() // => 5
```



- Modernes JavaScript bietet `class` und `extends` als syntaktischen Zucker

Prototypenbasierte Objektorientierung

Technische Universität Braunschweig

- Objekte erben alle Eigenschaft ihrer Prototypen
  - JavaScript: Dynamischer Lookup entlang der Prototypenkette
  - Konstruktion der Prototypen-Hierarchie mittels new
  - Sieht aus wie klonen, ist aber effizienter.
- Alle Objekte sind gleichberechtigt und können Prototypen werden
  - Keine Unterscheidung zwischen Objekten und Klassen
  - JavaScript: `obj.__proto__` zeigt wieder auf ein Objekt
  - Es entsteht ein Baum von Objekten, die Prototypen füreinander sind.
  - Die Wurzel des Baums ist das leere Objekt: `{}`
- Veränderung eines Prototypen ändert alle erbenden Objekte
  - JavaScript: Eingebaute Objekte können erweitert werden (discouraged)
  - `Array.prototype.forEach = function(...) { ... }`
  - Alle Arrays können jetzt `forEach`: `[2,3,4].forEach(...)`

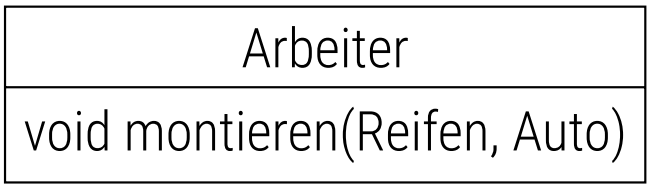
Kritik am objektorientierten Entwurf

Konzentration auf Objekte

- Objektorientierte Entwurfsmethoden fokussieren Objekte
  - Verb/Nomen-Analyse von Use-Case-Beschreibungen

    Alle Nomen werden Klassen/Objekte. Alle Verben werden Methoden.

  - Arbeiter (Nomen) montiert (verb) Reifen (Nomen) an Auto (Nomen).



Konzentration auf Objekte

- Objektorientierte Entwurfsmethoden fokussieren Objekte
  - Verb/Nomen-Analyse von Use-Case-Beschreibungen

    Alle Nomen werden Klassen/Objekte. Alle Verben werden Methoden.

  - Arbeiter (Nomen) montiert (verb) Reifen (Nomen) an Auto (Nomen).

Arbeiter
void montieren(Reifen, Auto)

- **Kritik**: Objektorientierung vernachlässigt Verben

  *Object Oriented Programming puts the Nouns first and foremost. Why would you go to such lengths to put one part of speech on **a pedestal**? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think.*

  (Steve Yegge, Execution in the Kingdom of Nouns)

Konzentration auf Objekte

- Objektorientierte Entwurfsmethoden fokussieren Objekte
  - Verb/Nomen-Analyse von Use-Case-Beschreibungen

    Alle Nomen werden … en Methoden.

  - Arbeiter (Nomen) m… (Nomen).

- **Kritik**: Objektorientie…

  *Object Oriented Pr… …d foremost. Why would you go to such len… …**pedestal**? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think.*

  (Steve Yegge, Execution in the Kingdom of Nouns)

Konzeptionell: Verben kommen nie allein!

- Nicht jede Funktionalität kann einem Objekt zugeordnet werden.
  - Zu welchem Objekt gehört der Algorithmus **größter gemeinsamer Teiler**?
  - Methode von `AlgorithmManager`-Klasse? Unterschiedliche Instanzen?
  - Gehört sie zur Klasse `int`? Hat `BigInt` eine eigene `gcd()` Methode?

⇒ **„Lösung“**: Nicht-instantiierte Klassen die nur statische Methoden beinhalten

```
void java.lang.Math.sin(double a);
```

Konzeptionell: Verben kommen nie allein!

- Nicht jede Funktionalität kann einem Objekt zugeordnet werden.
  - Zu welchem Objekt gehört der Algorithmus **größter gemeinsamer Teiler**?
  - Methode von `AlgorithmManager`-Klasse? Unterschiedliche Instanzen?
  - Gehört sie zur Klasse `int`? Hat `BigInt` eine eigene `gcd()` Methode?

  ⇒ **„Lösung“**: Nicht-instantiierte Klassen die nur statische Methoden beinhalten

```
void java.lang.Math.sin(double a);
```

- Wir können keine blanken Funktion herumreichen
  - Callback-Funktionen müssen immer von einem Objekt „bewacht“ werden

  ⇒ Leere Objekte, die nur eine Methode „`execute()/run()/do()`“ haben

```
void java.lang.Runnable.run();
```

  - Designpatterns, um diese Schwäche zu umschiffen: Command, Strategy, Proxy

Konzeptionell: Daten kommen nie alleine!

Technische Universität Braunschweig

- Von manchen Klassen kann es nur eine Instanz geben
  - Manche Entitäten sollten in einem Programm nur einmal existieren.
  - Beispiel: Managerobjekt, das **alleinige** Kontrolle über einen Aspekt hat
  - Designpattern: Singleton (nur möglich durch statische Attribute)

Konzeptionell: Daten kommen nie alleine!

- Von manchen Klassen kann es nur eine Instanz geben
  - Manche Entitäten sollten in einem Programm nur einmal existieren.
  - Beispiel: Managerobjekt, das **alleinige** Kontrolle über einen Aspekt hat
  - Designpattern: Singleton (nur möglich durch statische Attribute)

- Strikte Objektorientierung kennt keinen globalen Kontext
  - An welchem Objekt würde eine globale Variable hängen?
  - Globale Variablen erlauben globale Konfiguration des Programmverhaltens

  ⇒ Durchschleifen von Kontext-Objekten mit Programmoptionen

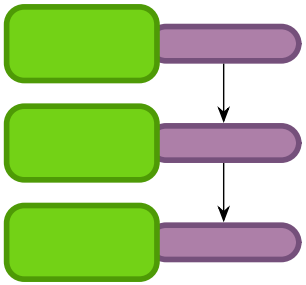
```
void app.grep.Searcher.search(ProgramContext);
void app.grep.Searcher.iterateDirectory(ProgramContext, Direct
void app.grep.Searcher.processFile(ProgramContext, String);
void app.grep.Searcher.searchLine(ProgramContext, String);
```

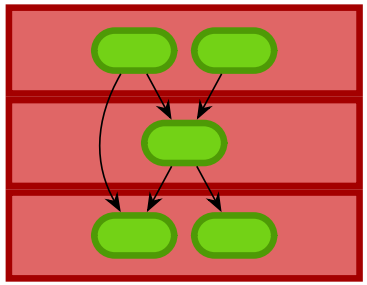
  - Designpatterns, die dieses Problem beheben: Registry, ServiceLocator

(Beides Singletons)

„Objektorientierung ist ineffizient!“

- OO-Entwurfsprinzipien verleiten, Probleme zu feinschichtig zu zerlegen

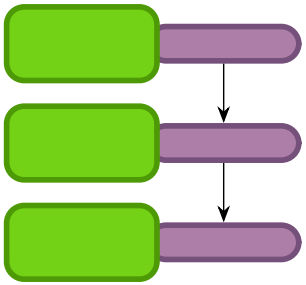


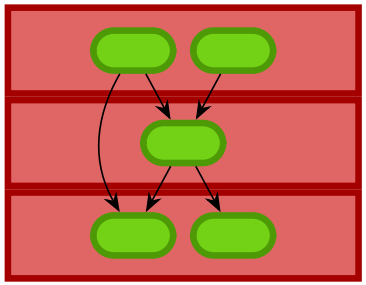


  - Verteilung der Verantwortlichkeit auf viele Objekte
  - Unübersichtlicher Kontrollfluss zwischen den einzelnen Methoden
  - Viele Methodenaufrufe zwischen den Schichten

„Objektorientierung ist ineffizient!“

- OO-Entwurfsprinzipien verleiten, Probleme zu feinschichtig zu zerlegen





  - Verteilung der Verantwortlichkeit auf viele Objekte
  - Unübersichtlicher Kontrollfluss zwischen den einzelnen Methoden
  - Viele Methodenaufrufe zwischen den Schichten

- Hohe Kosten für den einzelnen Methodenaufrufe
  - Späte Bindung: Dynamic Dispatch wählt Methode zur Laufzeit aus
  - this: Der implizite durchgeschleifte Parameter ist nicht umsonst.

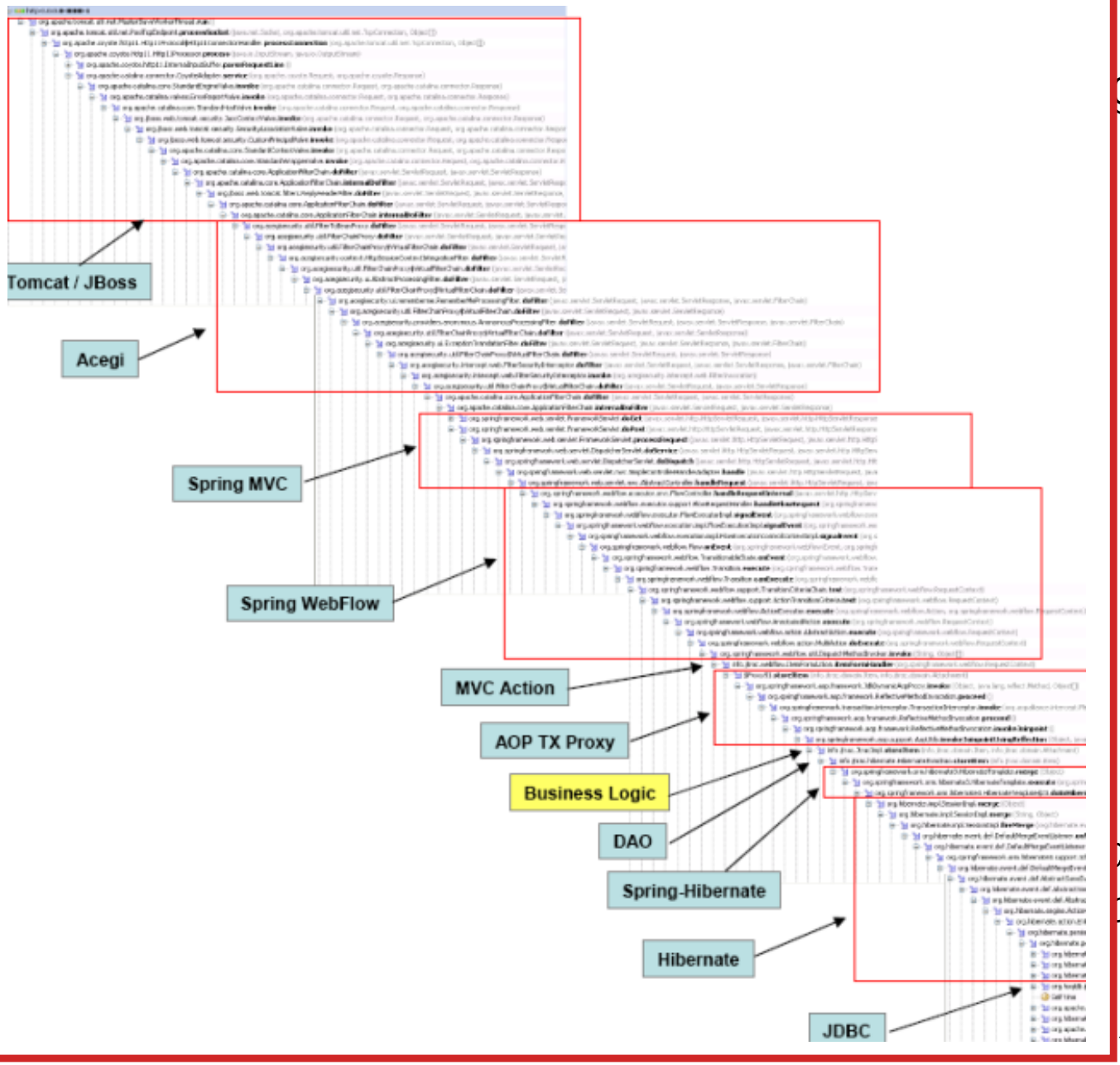
The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.

(Joe Armstrong, Erlang Erfinder)

„Objektorientierung ist ineffizient!“

- OO-Entw…gen
  - Vertei…
  - Unübe…
  - Viele M…
- Hohe Ko…
  - Späte…
  - this: D…

The pr…cit environ-
ment t…ou got was
a goril…g Erfinder)



Zusammenfassung

- Programmierparadigmen sind Sammlungen von Sprachkonzepten
  - Paradigmen machen Sprachen nicht mächtiger,
    aber den Code besser verständlich.
  - Paradigmen werden durch Software-Engineering-Prinzipien diszipliniert.
- Das imperative Programmierparadigma
  - Sequentielles Abarbeiten von Operationen, die den Zustand verändern
  - Strikte Vorgabe, wie eine Aufgabe durch zu führen ist
  - Prozedurale Abstraktion zerlegt das Problem entlang von Aufgaben
- Das objektorientierte Programmierparadigma
  - Interagierende Objekte sollen die reale Welt abbilden
  - Objekte senden Nachrichten bzw. rufen sich ihre Methoden gegenseitig auf
  - Prototypen erlauben klassenlose Objektorientierung
- Objektorientierung hat ebenfalls Probleme
  - Funktionen werden immer von Objekten dominiert.
  - Globaler Kontext nur über statische Methoden/Attribute
  - Designpatterns müssen konzeptionelle Schwächen ausgleichen.