

13 Objektorientiertes Programmieren

Objektorientiertes Programmieren (OOP) basiert auf

Objekten: zusammengesetzten Datenstrukturen mit

- gekapseltem Zustand
- Operationen auf dem Zustand, den **Methoden**

Vererbung: (*inheritance*) ein Konzept zur Erweiterung von Zustand und Funktionalität von Objekten

Message passing: ein Konzept zum Finden der Komponente eines Objekts, die für einen Methodenaufruf zuständig ist

13.1 Einfaches Bankkonto als Objekt

Erinnerung: Konto mit gekapseltem Zustand und separater withdraw Operation

; Geld abheben und anzeigen, ob das möglich war

```
(: account-withdraw (account real -> boolean))
```

; Effekt: (account-withdraw a n) ändert den Kontostand von a

```
(define account-withdraw
```

```
  (lambda (a n)
```

```
    (if (>= (account-balance a) n)
```

```
      (begin
```

```
        (set-account-balance! a (- (account-balance a) n))
```

```
        #t)
```

```
      #f)))
```

- Beobachtung:

- Das account Objekt muss als Parameter mitgegeben werden.
- Andere Prozeduren können auch account Objekte bearbeiten.

- Objektorientierter Ansatz:

Zustand account mit den zugehörigen Prozeduren (withdraw) zusammenpacken

Spezifikation: Bankkonto als Objekt

- Ein Kontoobjekt ist eine Prozedur mit Vertrag

```
(define-contract account (real -> (mixed real (one-of #f))))
```

Die Prozedur hebt einen Geldbetrag ab und liefert #f, falls das Geld nicht verfügbar ist. Ansonsten liefert sie den Kontostand nach dem Abheben.

Effekt: Der Kontostand wird geändert.

- Argument 0 liefert den aktuellen Kontostand.

; Konto aus Geldbetrag erzeugen

```
(: make-account (real -> account))
```

```
(define make-account
```

```
  (lambda (initial-balance)
```

```
    (let ((balance (make-ref initial-balance)))
```

```
      ...)))
```

Implementierung: Bankkonto als Objekt

```
; Konto aus Geldbetrag erzeugen
(: make-account (real -> account))
; make-account : real -> (real -> (mixed real (one-of #f)))
(define make-account
  (lambda (initial-balance)
    (let ((balance (make-ref initial-balance)))
      ;; das account Objekt ist die Abhebe-prozedur!
      (lambda (amount)
        (if (<= amount (get-ref balance))
            (begin
              (set-ref! balance (- (get-ref balance) amount))
              (get-ref balance))
            #f)))))
```

Beispiel: Bankkonto als Objekt

```
> (define acc1 (make-account 1000))  
> acc1  
#<procedure>  
> (acc1 0) ; Kontostand abfragen  
1000  
> (acc1 200) ; abheben  
800  
> (acc1 1000) ; mehr abheben  
#f  
> (acc1 0) ; Kontostand  
0
```

Beispiel mit Bankkonto/1

Speicher_____

Kommandosequenz_____

```
(define acc1 (make-account 1000))  
(acc1 0)  
(acc1 200)  
(acc1 1000)
```

Beispiel mit Bankkonto/2

Speicher_____

Kommandosequenz_____

```
(define acc1 ((lambda (initial-balance) ...) 1000))  
(acc1 0)  
(acc1 200)  
(acc1 1000)
```

Beispiel mit Bankkonto/3

Speicher_____

Kommandosequenz_____

```
(define acc1 (let ((balance (make-ref 1000))) ...))  
(acc1 0)  
(acc1 200)  
(acc1 1000)
```


Beispiel mit Bankkonto/4

Speicher

L1000 |-> 1000

Kommandosequenz

```
(define acc1 (lambda (amount)
  (if (<= amount (get-ref (ref L1000)))
      (begin
        (set-ref! (ref L1000) (- (get-ref (ref L1000) amount)))
        (get-ref (ref L1000)))
      #f)))

(acc1 0)
(acc1 200)
(acc1 1000)
```

Beispiel mit Bankkonto/5

Speicher

L1000 |-> 1000

Kommandosequenz

```
((lambda (amount)
  (if (<= amount (get-ref (ref L1000)))
      (begin
        (set-ref! (ref L1000) (- (get-ref (ref L1000)) amount))
        (get-ref (ref L1000)))
      #f)) 0)
((lambda (account) ...) 200)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/6

Speicher

L1000 |-> 1000

Kommandosequenz

```
(if (<= 0 1000)
  (begin
    (set-ref! (ref L1000) (- (get-ref (ref L1000)) 0))
    (get-ref (ref L1000)))
  #f)
((lambda (account) ...) 200)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/7

Speicher

L1000 |-> 1000

Kommandosequenz

```
(begin
  (set-ref! (ref L1000) (- (get-ref (ref L1000)) 0))
  (get-ref (ref L1000)))
((lambda (account) ...) 200)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/8

Speicher

L1000 | -> 1000

Kommandosequenz

```
(begin
  (set-ref! (ref L1000) (- 1000 0))
  (get-ref (ref L1000)))
((lambda (account) ...) 200)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/9

Speicher

L1000 | -> 1000

Kommandosequenz

```
(begin
  (set-ref! (ref L1000) 1000)
  (get-ref (ref L1000)))
((lambda (account) ...) 200)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/10

Speicher

L1000 | -> 1000

Kommandosequenz

```
(begin
  (get-ref (ref L1000)))
((lambda (account) ...) 200)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/11

Speicher

L1000 | -> 1000

Kommandosequenz

```
(begin  
  1000)
```

```
((lambda (account) ...) 200)
```

```
((lambda (account) ...) 1000)
```


Beispiel mit Bankkonto/12

Speicher

L1000 | -> 1000

Kommandosequenz

1000

((lambda (account) ...) 200)

((lambda (account) ...) 1000)

Beispiel mit Bankkonto/13

Speicher

L1000 |-> 1000

Kommandosequenz

1000

```
(if (<= 200 (get-ref (ref L1000)))  
  (begin  
    (set-ref! (ref L1000) (- (get-ref (ref L1000)) 200))  
    (get-ref (ref L1000)))  
  #f)  
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/14

Speicher

L1000 |-> 1000

Kommandosequenz

1000

```
(if (<= 200 1000)
  (begin
    (set-ref! (ref L1000) (- (get-ref (ref L1000)) 200))
    (get-ref (ref L1000)))
  #f)
((lambda (account) ...) 1000)
```

Beispiel mit Bankkonto/15

Speicher

L1000 | -> 1000

Kommandosequenz

1000

(begin

 (set-ref! (ref L1000) (- (get-ref (ref L1000)) 200))

 (get-ref (ref L1000)))

((lambda (account) ...) 1000)

Beispiel mit Bankkonto/16

Speicher

L1000 |-> 1000

Kommandosequenz

1000

(begin

(set-ref! (ref L1000) 800)

(get-ref (ref L1000)))

((lambda (account) ...) 1000)

Beispiel mit Bankkonto/17

Speicher

L1000 | -> 800

Kommandosequenz

1000

(begin

(get-ref (ref L1000)))

((lambda (account) ...) 1000)

Beispiel mit Bankkonto/18

Speicher

L1000 | -> 800

Kommandosequenz

1000

800

`((lambda (account) ...) 1000)`

Beispiel mit Bankkonto/19

Speicher

L1000 | -> 800

Kommandosequenz

1000

800

#f

13.2 Bankkonto mit mehreren Operationen

```
(define make-account
  (lambda (initial-balance)
    (let ((balance (make-ref initial-balance)))
      ;; Abheben
      (lambda (amount)
        (if (<= amount (get-ref balance))
            (begin
              (set-ref! balance (- (get-ref balance) amount))
              (get-ref balance))
            #f))))))
```

- Das einfache Bankkonto erlaubt nur eine Operation, das Abheben.
- Weitere Operationen (z.B. Kontostand) müssen Zugriff auf balance haben.
- Wo müssen diese Operationen eingefügt werden?

Bankkonto mit Kontostand (unvollständig)

```
(define make-account
  (lambda (initial-balance)
    (let ((balance (make-ref initial-balance)))
      ...
      ;; Kontostand
      (lambda ()
        (get-ref balance))
      ...
      ;; Abheben
      (lambda (amount)
        (if (<= amount (get-ref balance))
            (begin
              (set-ref! balance (- (get-ref balance) amount))
              (get-ref balance))
            #f))))))
```

- balance ist nur im Rumpf von (let ((balance ...)) ...) sichtbar

Message Passing

- Auswahl zwischen den Operationen für Kontostand und Abheben notwendig
- Implementiert durch separate Prozedur (*message dispatcher*)
- Eingabe: Nachricht (*message*) mit dem Namen der gewünschten Operation
- Ausgabe: die ausgewählte Operation
- Fürs Bankkonto:
 - Nachrichten sind die Strings "balance" und "withdraw"
 - Vertrag der Prozedur ist

```
((one-of "balance") -> ( -> real))
and
((one-of "withdraw") -> (real -> (mixed real (one-of #f))))
```

Bankkonto mit Message Passing

```
(define make-account
  (lambda (initial-balance)
    (let ((balance (make-ref initial-balance)))
      (lambda (message)
        (cond
          ;; Kontostand
          ((string=? message "balance")
           (lambda ()
             (get-ref balance)))
          ;; Abheben
          ((string=? message "withdraw")
           (lambda (amount)
             (if (<= amount (get-ref balance))
                 (begin
                  (set-ref! balance (- (get-ref balance) amount))
                  (get-ref balance))
                 #f))))))))))
```

Verwendung: Bankkonto mit MP

```
> (define acc (make-account 5000))
> acc ; (lambda (message) ...)
#<procedure>
> (acc "withdraw") ; (lambda (amount) ...)
#<procedure>
> ((acc "withdraw") 10)
4990
> (acc "balance") ; (lambda () balance)
#<procedure>
> ((acc "balance"))
4990
```

13.3 Versenden von Nachrichten

- Die Aufrufe der Operationen sind sperrig:
 - `((acc "balance"))`
 - `((acc "withdraw") 77)`
- Abhilfe: Definiere Prozedur `send`, so dass
 - `(send acc "balance")`
 - `(send acc "withdraw" 77)`
- Beobachtung: Die Aufrufe von `send` brauchen unterschiedlich viele Parameter

Definition von send

```
; verschicken einer Nachricht an ein Objekt  
; send : object string value* -> value  
(define send  
  (lambda (obj message . args)  
    (apply (obj message) args)))
```

Neues:

- (lambda (obj message . args) ...)
 - Prozedur, die mindestens zwei Parameter akzeptiert
 - beliebig viele weitere Parameter werden in der Liste args zusammengefasst
 - Sonderfall: (lambda args *<body>*) erwartet beliebig viele Argumente
- (apply f args)
 - wendet f auf die Argumentliste args an
 - f muss genausoviele Parameter erwarten, wie args Elemente hat

Verwendung: Bankkonto mit MP und send

```
> (define acc (make-account 5000))
> acc ; (lambda (message) ...)
#<procedure>
> (acc "withdraw") ; (lambda (amount) ...)
#<procedure>
> (send acc "withdraw" 10)
4990
> (acc "balance") ; (lambda () balance)
#<procedure>
> (send acc "balance")
4990
```


13.4 Vererbung

- Codierung von Objekten: siehe acc
- Die Prozedur `make-account` erzeugt Objekte
- Sie kann als Klasse angesehen werden:
 - Klasse als Objektgenerator
 - Argumente der Klasse spezifizieren den Objektzustand
 - Prozedur im message dispatcher spezifizieren die Operationen (Methoden)
- Methodenaufruf durch message passing
- Jetzt: Erweiterungsmechanismus „Vererbung“

Personen

Eine Person besitzt drei Operationen

1. `get-name`: Sie kann ihren Namen angeben.
2. `say`: Sie kann sprechen (indem sie den Text ausdrückt).
3. `slap`: Sie soll Schläge einstecken, auf die sie jeweils durch "huh?" reagiert; bei jedem dritten Schlag kommt "ouch!".

D.h. der Zustand eines Personenobjekts enthält zumindest den Namen.

Person mit MP (1. Näherung)

```
; Person konstruieren
; make-person : string -> (message -> method)
(define make-person
  (lambda (name)      ;; nicht veränderliche Eigenschaft
    (lambda (message)
      (cond
        ((string=? message "get-name")
         ;; Namen liefern
         ;; -> string
         (lambda ()
          name))
        ((string=? message "say")
         ;; Text ausdrucken
         ;; list(string) -> unspecified
         (lambda (text)
          (write-list-newline text))))))))
```

Hilfsprozedur write-list-newline

```
; Drucke Liste von Strings
(: write-list-newline ((list string) -> unspecified))
(define write-list-newline
  (lambda (text)
    (begin
      (for-each (lambda (s) (write-string s)) text)
      (write-newline))))
```

Dabei ist

- `(: write-string (string -> unspecified))` druckt einen String aus
- `(: write-newline (-> unspecified))` druckt einen Zeilenvorschub
- `(: for-each ((%a -> %b) (list %a) -> unspecified))`
(for-each f xs) wendet f von links nach rechts auf alle Elemente der Liste xs an

Probelauf

```
> (define sarah (make-person "Sarah"))
```

```
> (send sarah "get-name")
```

```
"Sarah"
```

```
> (send sarah "say" (list "I'm" " " "so" " " "clever"))
```

```
I'm so clever
```

Die slap Operation

- Benötigt eine weitere Variable für die Anzahl der bisher eingesteckten Schläge
- Die gesprochene Reaktion soll nicht über `write-list-newline` erfolgen, sondern unter Verwendung der eigenen Operation `say`
- Problem dabei: wie wird `say` aufgerufen?

Person mit slap (2. Näherung)

```
(define make-person
  (lambda (name)
    (let ((slaps (make-ref 0)))          ; Anzahl der Schläge
      (lambda (message)
        (cond
          ((string=? message "get-name") ...)
          ((string=? message "say") ...)
          ((string=? message "slap")
           (lambda ()
             (begin
               (set-ref! slaps (+ (get-ref slaps) 1))
               (if (< (get-ref slaps) 3)
                   (send ... "say" (list "huh?"))
                   (begin
                     (set-ref! slaps 0)
                     (send ... "say" (list "ouch!")))))))))))))))
```

Self

- Der Empfänger von `say` muss das Objekt selbst sein
- Das Objekt selbst wird durch `(lambda (message) ...)` repräsentiert

⇒ Diese Prozedur muss mit `letrec` lokal rekursiv definiert werden!

- Traditionelle Name für das Objekt selbst:
 - `self` (in Smalltalk, hier)
 - `this` (in C++, Java, usw)

Person mit slap (endgültig)

```
(define make-person
  (lambda (name)
    (let ((slaps (make-ref 0)))          ; Anzahl der Schläge
      (letrec ((self
        (lambda (message)
          (cond
            ((string=? message "get-name") ...)
            ((string=? message "say") ...)
            ((string=? message "slap")
              (lambda ()
                (begin
                  (set-ref! slaps (+ (get-ref slaps) 1))
                  (if (< (get-ref slaps) 3)
                    (send self "say" (list "huh?"))
                    (begin
                      (set-ref! slaps 0)
                      (send self "say" (list "ouch!"))))))))))))
          self))))))
```

Person in Aktion

```
> (define sarah (make-person "Sarah"))  
> (send sarah "slap")  
huh?  
> (send sarah "slap")  
huh?  
> (send sarah "slap")  
ouch!  
> (send sarah "slap")  
huh?
```

Sänger

- Ein Sänger ist eine Person mit zusätzlichen Fähigkeiten
 - Es gibt alle Methoden von Person
 - zusätzlich die Methode `sing`, die einen Text singt.

Ansatz:

```
; Sänger konstruieren
; make-singer : string -> (message -> method)
(define make-singer
  (lambda (name)
    (let ((person (make-person name)))
      ...)))
```

Sänger mit sing (1. Näherung)

```
; Sänger konstruieren
; make-singer : string -> (message -> method)
(define make-singer
  (lambda (name)
    (let ((person (make-person name)))
      (letrec ((self
        (lambda (message)
          (cond
            ((string=? message "sing")
             ;; Text singen
             ;; list(string) -> unspecified
             ...)
            ...)))
        self)))))
```

- Die Nachricht sing wird verarbeitet
- Was passiert mit get-name, say und slap?

Sänger mit sing (2. Näherung)

```
; Sänger konstruieren
; make-singer : string -> (message -> method)
(define make-singer
  (lambda (name)
    (let ((person (make-person name)))
      (letrec ((self
        (lambda (message)
          (cond
            ((string=? message "sing")
             ;; Text singen
             ;; list(string) -> unspecified
             ...)
            (else (person message))))))
        self))))
```

- get-name, say und slap werden an person *delegiert*
- sing wird unter Rückgriff auf say implementiert

Sänger mit sing (endgültig)

```
; Sänger konstruieren
; make-singer : string -> (message -> method)
(define make-singer
  (lambda (name)
    (let ((person (make-person name)))
      (letrec ((self
        (lambda (message)
          (cond
            ((string=? message "sing")
             ;; Text singen
             ;; list(string) -> unspecified
             (lambda (text)
              (send self "say" (make-pair "tra-la-la " text))))
            (else (person message))))))
        self))))))
```

Sänger(in) in Aktion

```
> (define sarah (make-singer "Sarah"))
```

```
> (send sarah "say" (list "hi"))
```

```
hi
```

```
> (send sarah "sing" (list "hi"))
```

```
tra-la-la hi
```

Einfache Vererbung (Single Inheritance)

- Konstruktion eines Objekts, das alle Eigenschaften (Methoden und Zustand) eines anderen Objekts hat und noch weitere dazu
- Bsp: Sänger hat alle Eigenschaften von Person und besitzt Methode `sing`
- `make-singer` und `make-person` spielen die Rolle von *Klassen*
- Person ist *Oberklasse* von Sänger (*superclass*)
- Sänger ist *Unterklasse* von Person (*subclass*)
- Ein Objekt, das von einer Klasse erzeugt wurde, heißt *Instanz* der Klasse
Bsp: `sarah`
- Lokale Variable einer Instanz heißen *Instanzvariable*
Bsp: `slaps`

Einführen von einfacher Vererbung

- Klassenhierarchie
 - Eine Klasse kann mehrere Unterklassen besitzen
 - Jede Unterklasse kann selbst wieder Unterklassen besitzen
- Die Aufteilung der Eigenschaften zwischen Ober- und Unterklasse wird beim Programmentwurf festgelegt
- Ausgehend von einer Menge von benötigten Klassen werden Oberklassen definiert, die gemeinsame Eigenschaften der Klassen zusammenfassen.
- Nicht übertreiben: Klassenhierarchie sollte nicht zu feingranular sein

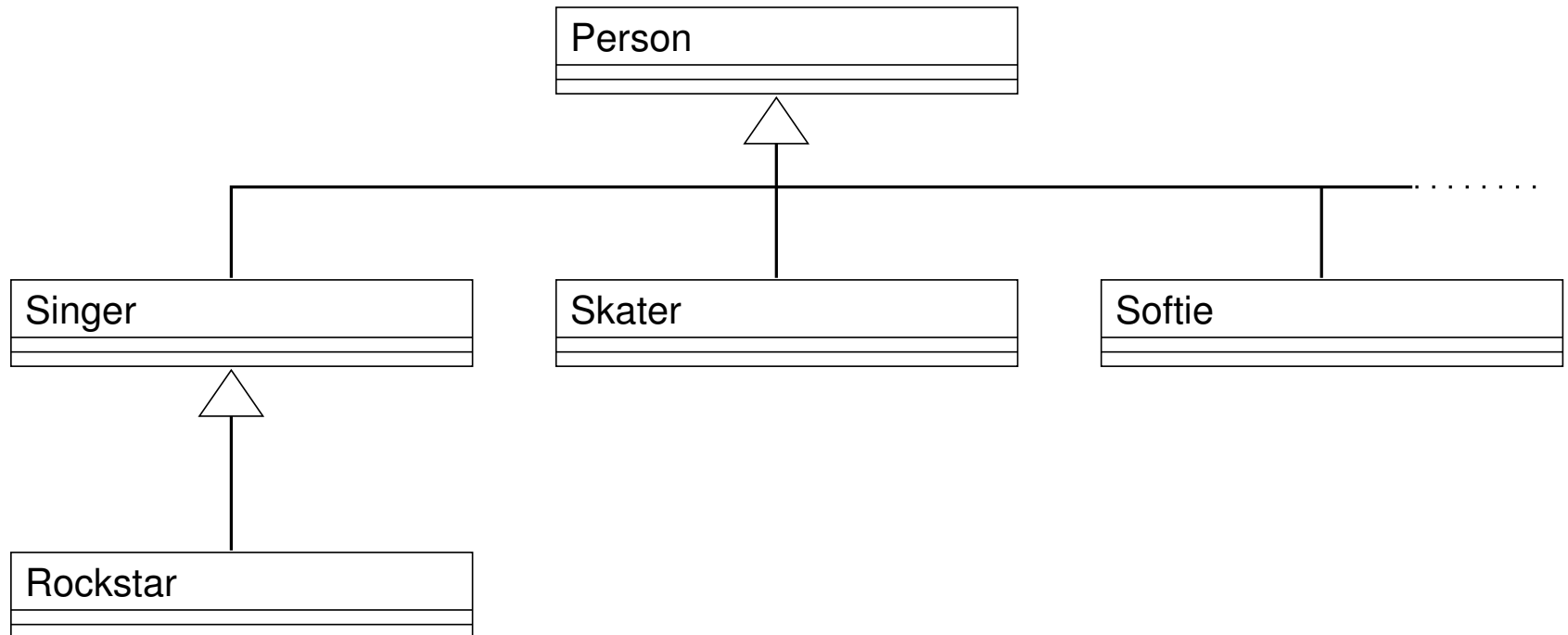


Mantra #15 (Oberklassen)

Fasse Gemeinsamkeiten von Klassen in Oberklassen zusammen.

Klassendiagramm

- Graphische Darstellung von Vererbungshierarchien



13.5 Überschreiben von Methoden

- Im Zuge der Vererbung können Methoden überschrieben werden (*method override*)
- Klasse definiert neue Implementierung einer Methode einer Oberklasse
- **Vorsicht:** die Funktion dieser Methode kann beliebig geändert werden!
- Beispiel: Ein Rockstar ist ein Sänger, der
 - an alles, was er sagt, noch ", dude" anhängt und
 - auf Schläge anders als ein normaler Mensch reagiert.

Beispiel: Rockstar

```
; Rockstar erzeugen
; make-rockstar : string -> (message -> method)
(define make-rockstar
  (lambda (name)
    (let ((singer (make-singer name)))
      (letrec ((self
        (lambda (message)
          (cond
            ((string=? message "say")
             ;; Text sprechen
             ;; list(string) -> unspecified
             (lambda (text)
               (send singer "say" (append text (list ", dude"))))))
            ((string=? message "slap")
             ;; Schlag einstecken
             ;; -> unspecified
             (lambda ()
               (send self "say" (list "pain just makes me stronger"))))
            (else (singer message))))))
        self))))
```

Rockstar in Aktion

```
> (define marilyn (make-rockstar "Marilyn"))  
> (send marilyn "say" (list "hello"))  
hello, dude  
> (send marilyn "slap")  
pain just makes me stronger, dude  
> (send marilyn "sing" (list "happy birthday smurfs"))  
tra-la-la happy birthday smurfs
```

- Unerwartet: die sing Methode hängt **kein** ", dude" an!
- Warum?
- Wie kann das repariert werden?

Mache self zum Parameter jeder Methode!

```
(define make-person
  (lambda (name)
    (let ((slaps (make-ref 0)))
      (lambda (message)
        (cond
          ((string=? message "get-name")
           ;; person -> string
           (lambda (self)
            name))
          ((string=? message "say")
           ;; person list(string) -> unspecified
           (lambda (self text)
            (write-list-newline text)))
          ((string=? message "slap")
           ;; person -> unspecified
           (lambda (self)
            (begin
              (set-ref! slaps (+ (get-ref slaps) 1))
              (if (< (get-ref slaps) 3)
                  (send self "say" (list "huh?"))
                  (begin
                     (send self "say" (list "ouch!"))
                     (set-ref! slaps 0))))))))))))))
```

Erweiterung von send

Alte Implementierung

```
; verschicken einer Nachricht an ein Objekt  
; send : object string value* -> value  
(define send  
  (lambda (obj message . args)  
    (apply (obj message) args)))
```

Neue Implementierung

```
; verschicken einer Nachricht an ein Objekt  
; send : object string value* -> value  
(define send  
  (lambda (obj message . args)  
    (apply (obj message) obj args)))
```

- Verwendet erweitertes apply (mit mehr als zwei Argumenten)

Erweiterung von apply

- (apply f a1 ...an args)
 - wendet f auf a1 ... an sowie weitere Parameter aus Argumentliste args an
 - f muss (+ n (length args)) Parameter erwarten

Beispiel:

```
> (apply / 120 (list))  
0.0083  
> (apply / 120 (list 1))  
120  
> (apply / 120 (list 1 2))  
60  
> (apply / 120 (list 1 2 3))  
20  
> (apply / 120 (list 1 2 3 4))  
5  
> (apply / 120 (list 1 2 3 4 5))  
1
```


Korrigierter Sänger

```
(define make-singer
  (lambda (name)
    (let ((person (make-person name)))
      (lambda (message)
        (cond
          ((string=? message "sing")
           ;; Text singen
           ;; singer list(string) -> unspecified
           (lambda (self text)
              (send self "say" (make-pair "tra-la-la " text))))
          (else (person message)))))))
```

Korrigierter Rockstar

```
(define make-rockstar
  (lambda (name)
    (let ((singer (make-singer name)))
      (lambda (message)
        (cond
          ((string=? message "say")
           ;; Text sagen
           ;; rockstar list(string) -> unspecified
           (lambda (self text)
             (send singer "say" (append text (list ", dude"))))))
          ((string=? message "slap")
           ;; Schlag einstecken
           ;; rockstar -> unspecified
           (lambda (self)
             (send self "say" (list "pain just makes me stronger"))))
          (else (singer message)))))))
```

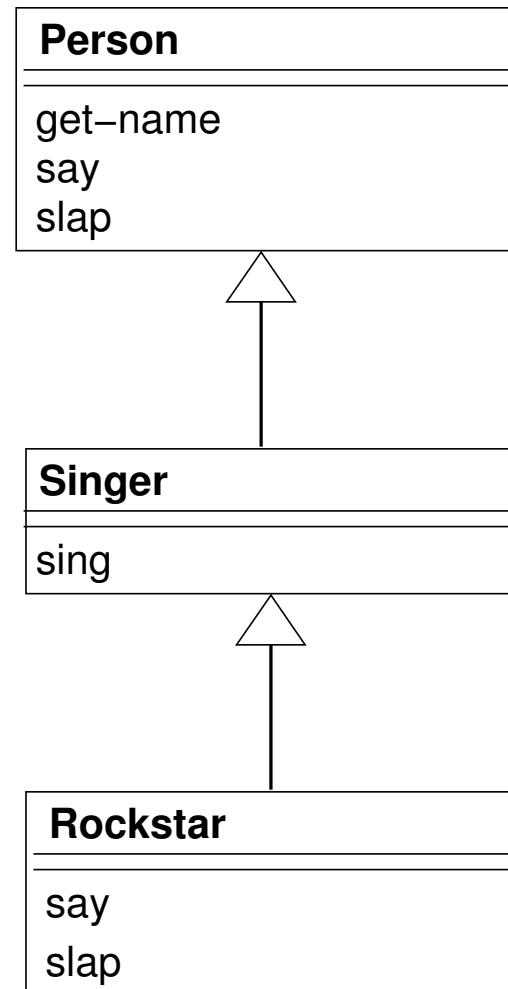
Korrigierter Rockstar in Aktion

```
> (define marilyn (make-rockstar "Marilyn"))  
> (send marilyn "say" (list "hello"))  
hello, dude  
> (send marilyn "slap")  
pain just makes me stronger, dude  
> (send marilyn "sing" (list "happy birthday smurfs"))  
tra-la-la happy birthday smurfs, dude
```

Analyse der Methodenaufrufe

- say wird überschrieben und ruft singer.say auf
- slap wird überschrieben und ruft self.say (rockstar.say) auf
- sing wird an singer.sing weiter gereicht;
dort wird jetzt self.say (rockstar.say) aufgerufen

Analyse der Methodenaufrufe



Zusammenfassung (Vererbung und Überschreibung)

- Überschreiben einer Methode kann Funktionalität auf unvorhersehbare Weise ändern
 - Ein Aufruf (`send self mname ...`) in einer Klasse A kann zum Aufruf von `mname` in einer beliebigen Ober- oder **Unterklasse** B von A führen.
 - Der Code einer **Unterklasse** B liegt zum Zeitpunkt des Erstellens von A meistens noch nicht vor.
- ⇒ Der Effekt des Aufrufs einer überschriebenen Methode ist für den Programmierer nicht vorhersehbar.
- ⇒ Bei Verwendung von Überschreiben kann die Funktionsweise eines OO-Programms nur durch Studium der **gesamten Klassenhierarchie** verstanden werden.



Mantra #16 (Überschreiben von Methoden)

Vermeide das Überschreiben von Methoden.

13.6 Mehrfachvererbung (Multiple Inheritance)

- Bisher:
 - einfache Vererbung
 - ein Objekt kann von genau einem Objekt erben
- Mögliche Erweiterung
 - Mehrfachvererbung
 - ein Objekt kan von mehreren Objekten erben
- MV nur von wenigen Sprachen unterstützt (z.B. Eiffel, C++)
- Effiziente Implementierung von MV nicht einfach
- MV wird von manchen prinzipiell abgelehnt

Beispiel: ein Poet

- Ein Poet ist ein eigenständiges Objekt (keine Person)
- Ein Poet kann
 - sprechen "say" und
 - einen auswendig gelernten Text rezitieren "recite"

Implementierung des Poet

```
; Dichter konstruieren
; make-poet : string -> (message -> method)
(define make-poet
  (lambda (name)
    (lambda (message)
      (cond
        ((string=? message "say")
         ;; poet list(string) -> unspecified
         (lambda (self text)
           (write-list-newline (append text (list " and the sky is blue"))))))
        ((string=? message "recite")
         ;; poet -> unspecified
         (lambda (self)
           (write-list-newline (list "the sky is blue")))))))))
```

Ein Poet in Aktion

```
> (define james (make-poet "James"))
```

```
> (send james "say" (list "hi"))
```

```
hi and the sky is blue
```

```
> (send james "recite")
```

```
the sky is blue
```

Erst Rockstar, dann Poet

- James ist eigentlich Rockstar
- Aber seine Texte können auch als Gedichte durchgehen
- Modellierung davon:

```
; james, der rockstar-poet
; james : message -> method
(define james
  (let* ((name "James")
        (rockstar (make-rockstar name))
        (poet (make-poet name)))
    (lambda (message)
      (if ...
          ...
          ...))))
```

Erweiterung des Dispatches

- Der Dispatch muss zwischen rockstar und poet als Empfänger unterscheiden.
- Falls rockstar die Nachricht versteht, dann soll rockstar sie verarbeiten.
- Falls rockstar die Nachricht nicht versteht, dann soll sie an poet weitergereicht werden.

⇒ Erweitere den Dispatch-Code um Signalisierung, ob Nachricht verstanden

Erweiterter Dispatch-Code für person und poet

```
(define make-person
  (lambda (name)
    (let ((slaps (make-ref 0)))
      (lambda (message)
        (cond
          ((string=? message "get-name") ...)
          ((string=? message "say") ...)
          ((string=? message "slap") ...)
          (else #f)))))))
```

```
(define make-poet
  (lambda (name)
    (lambda (message)
      (cond
        ((string=? message "say") ...)
        ((string=? message "recite") ...)
        (else #f)))))
```

Implementierung: Erst Rockstar, dann Poet

- James, der Rockstar, der auch Poet ist

```
; james, der rockstar-poet
; james : message -> method
(define james
  (let* ((name "James")
         (rockstar (make-rockstar name))
         (poet (make-poet name)))
    (lambda (message)
      (let ((rockstar-method (rockstar message)))
        (if (equal? rockstar-method #f)
            (poet message)
            rockstar-method))))))
```

Rockstar-Poet in Aktion

```
> (send james "say" (list "honey"))
```

honey, dude

```
> (send james "recite")
```

the sky is blue

```
> (send james "slap")
```

pain just makes me stronger, dude

```
> (send james "sing" (list "something"))
```

tra-la-la something, dude

Alternative: Erst Poet, dann Rockstar

```
; henry, der poetische rockstar
; henry : message -> method
(define henry
  (let* ((name "Henry")
         (rockstar (make-rockstar name))
         (poet (make-poet name)))
    (lambda (message)
      (let ((poet-method (poet message)))
        (if (equal? poet-method #f)
            (rockstar message)
            poet-method))))))
```


Poeten-Rockstar in Aktion

```
> (send henry "say" (list "honey"))
```

honey and the sky is blue

```
> (send henry "recite")
```

the sky is blue

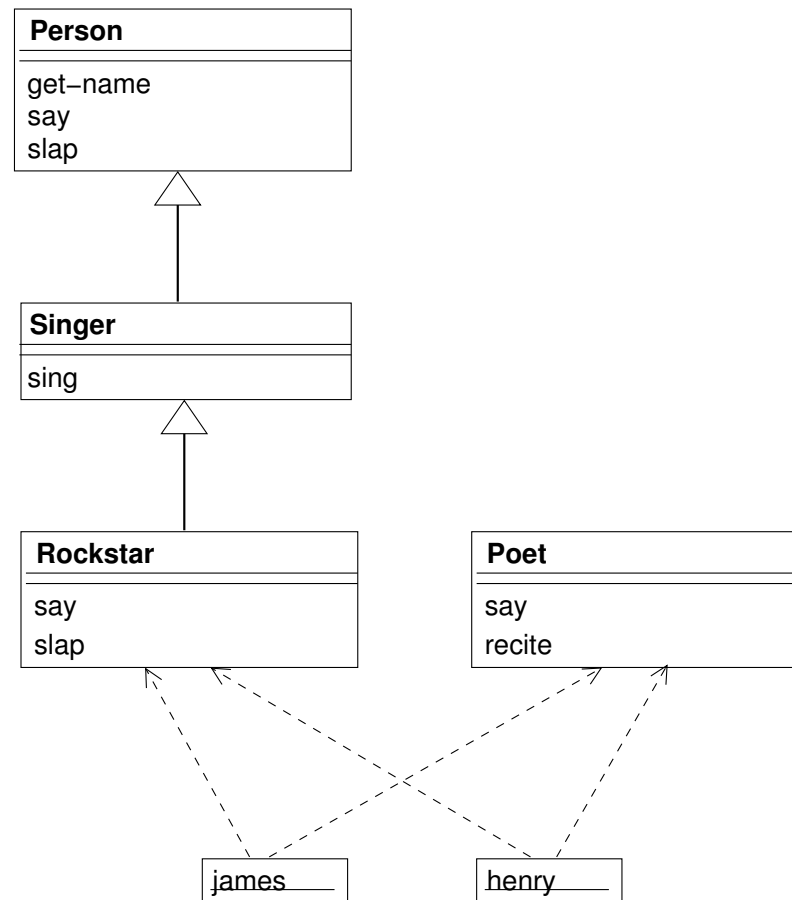
```
> (send henry "slap")
```

pain just makes me stronger and the sky is blue

```
> (send henry "sing" (list "loo"))
```

tra-la-la loo and the sky is blue

Mehrfachvererbung im Klassendiagramm



Zusammenfassung: Mehrfachvererbung

- Ein Objekt kann von mehreren Objekten erben
- Strategie für Methodenauswahl wichtig
Für Oberklassen, die die gleiche Nachricht verstehen, muss eine Dispatch-Reihenfolge vereinbart werden
Bsp: zuerst Rockstar oder zuerst Poet?
- Zustandskomponenten?
- Zusammenwirken von Methoden noch komplexer

13.7 Abstraktion über Klassen

- Ein Rockstar ist ein „cooler“ Sänger
- „Coolheit“ äußert sich in einer Veränderung der say-Methode
- Warum ist „Coolheit“ auf Sänger beschränkt?
- Warum können nicht auch Poeten „cool“ sein?
- Ansatz: Abstrahiere über die Oberklasse singer

Rockstar wieder besucht

```
(define make-rockstar
  (lambda (name)
    (let ((singer (make-singer name)))
      (lambda (message)
        (cond
          ((string=? message "say")
           ;; Text sagen
           ;; rockstar list(string) -> unspecified
           (lambda (self text)
              (send singer "say" (append text (list ", dude"))))))
          ...))))))
```

- Die Klasse Sänger wird durch ihre Konstruktorfunktion `make-singer` vertreten
- `singer` ist nur eine Variable, die umbenannt werden kann

Ein Klassengenerator

```
; zu einer Klasse Coolness hinzufügen
; make-make-cool-someone :
;   (string -> (message -> method)) -> (string -> (message -> method))
(define make-make-cool-someone
  (lambda (make-super)
    (lambda (name)
      (let ((super (make-super name)))
        (lambda (message)
          (cond
            ((string=? message "say")
             ;; Text sagen
             ;; rockstar list(string) -> unspecified
             (lambda (self text)
               (send super "say" (append text (list ", dude"))))))
            ...
            (else (super message))))))))))
```

Alternative Konstruktion des Rockstar

```
; make-rockstar : string -> (message -> method)
(define make-rockstar
  (make-make-cool-someone make-singer))
```

- funktioniert wie bisher ...

```
; make-cool-poet : string -> (message -> method)
(define make-cool-poet
  (make-make-cool-someone make-poet))
```

Beispiel:

```
> (define charles (make-cool-poet "Charles"))
> (send charles "say" (list "hello"))
hello, dude and the sky is blue
```

Zusammenfassung: Mixins

- Ein **Mixin** ist eine Klasse, die von ihrer Oberklasse abstrahiert ist
- Ein **Mixin** ist eine Funktion, die eine Klasse erweitert
- Die Eigenschaften werden in der Reihenfolge „probiert“, in der die Mixins angewendet werden



Mantra #17 (Mixins)

Kapsel isolierte Eigenschaften von Klassen in Mixins

13.8 Kontext: Objektorientierte Programmiersprachen

Hier	OOPS
Einblick in die Implementierung	Durch Compiler/System festgelegt
Definition und Verwendung durch Standardformen	Spezielle Syntax für Methoden, Methodenaufrufe, Klassen, Vererbung usw.
(in vollem Scheme: Syntaxerweiterung möglich)	?
Eigene Erweiterungen ausprobieren	(als Compilerschreiber)
Einfach- oder Mehrfachvererbung	Durch Compiler/System festgelegt
Klassen sind Werte	Klassen sind spezielle Konstrukte
Abstraktion über Klassen möglich	Abstraktion über Klassen nicht möglich