Paradigmen
2. Objektorientiertes Programmieren

das *topmodische* **Paradigma**
eine Weiterentwicklung des imperativen Paradigmas

Inhalt
Ausgangspunkt und Ziele
Was ist typisch objektorientiert?
Fallstudien: Smalltalk und Eiffel

Notiz: Ausgangspunkt und Ziele

Ausgangspunkt: *Imperative Programmierung*
Seiteneffekte erschweren Verständnis und Verifikation
• *global variables considered harmful (Parnas)*
• pointers considered harmful
Datenstrukturen sind nur schwer zu erweitern

Ziel
Modularität
• Variablen und Zustandsveränderungen in Moduln *kapseln*
Integrität des Speichers
• automatische Speicherbereinigung, Verzicht auf Zeigerarithmetik
Vererbung
• erweiterbare Datenstrukturen erlauben
Wiederverwendung

Das objektorientierte Ausführungsmodell

Objekte
• *Objekte* sind abstrakte Variablen mit den dazugehörigen *Methoden*

Nachrichtenaustausch
• Objekte schicken Nachrichten an *Empfängerobjekte* (Methodenaufrufe)
und bekommen ein Ergebnis zurückgeliefert

Unterschied zur imperativen von-Neumann-Maschine
datenorientierte Programmstruktur
datenorientiertes Ausführungsmodell
dynamisches Speichermodell
Nebenläufigkeit und *Verteilung* ist möglich

objektorientierte Konzepte

Darstellung nach M. Abadí und L. Cardelli
A Theory of Objects, Springer New York 1996 Kapitel 1-4

Inhalt
Was ist Objektorientierung?
Klassenbasierte Sprachen
fortgeschrittene klassenbasierte Eigenschaften
Objektbasierte (klassenlose) Sprachen

Klassen

```
class cell is
  var contents: Integer := 0;
  method get(): Integer is
    return self.contents;
  end;
  method set(n: Integer) is
    self.contents := n;
  end;
end
```


Notiz: Klassen

Speichermodell (Verbund mit Prozeduren, *record*)
einbettend (*naiv*): jedes Objekt enthält den Code seiner Methoden
deligierend: jedes Objekt verweist auf die Methodentabelle seiner Klasse

in klassenbasierten Sprachen macht das semantisch keinen Unterschied

Objekte

Instanziierung

```
var mycell: InctanceType(cell) := new cell;
var yourcell: InctanceType(cell) := new cell;
```

Benutzung und Verändern

```
mycell.set(2*mycell.get());
```

Referenzsemantik

```
yourcell.set(*mycell.get());
yourcell.set(0);
```


Notiz: Objekte

Instanziierung
Objekte sind Verweise auf Verbunde, die auf der Halde angelegt werden

Verweise werden versteckt, aber nicht ganz (siehe unten)

Benutzung und Verändern
durch Methodenaufrufe

Referenzsemantik
Zuweisungen ersetzen *Zeiger*, erzeugen *Aliase*

Vererbung

```
subclass reCell of cell is
  var backup: Integer := 0;
  override set(n: Integer) is
    self.backup := self.contents;
    super.set(n);
  method restore() is
    self.contents := self.backup;
  end;
end
```


Notiz: Vererbung

Unterklassen
- erben Attribute und Methoden der Oberklasse
- können Attribute und Methoden hinzufügen
- können geerbte Methoden überschreiben
- können Methoden der Oberklasse aufrufen (`super`)

Speichermodell

Attribute einbetten

Methoden einbetten oder deligieren

Vererbung ersetzt die Typkonstruktion *disjunkte Vereinigung*

Notiz: mehrfache Vererbung

jede Klasse kann mehrere Oberklassen haben

Probleme
- duplizierte Attribute und Methoden?
- widersprüchliche Attribute und Methoden?
- Was bedeutet `super`?

Alternative

einfache Vererbung von *Implementierungen*

mehrfache Vererbung von *Schnittstellen*

Typeinschluss

```
var myCell: InctanceType(cell) := new cell;
var myReCell: InctanceType(reCell) := new reCell;
procedure f:(x: InctanceType(cell)) is ... end;

myCell := myReCell;
f(myReCell);
```

subsumption

instanceType(*c'*) <: instanceType(*c*) :? *c'* ist Unterklasse von *c*

a : *A* und *A* <: *B* ? *a* : *B*

Typ einer Variablen *b* : *B*

statischer Basistyp: *B*

"wahrer" dynamischer Typ: irgendein *A* <: *B*

dynamisches Binden (*dynamic dispatch*)

```
procedure g:(x: InctanceType(cell)) is
  x.set(3);
end;
g(myReCell);
```

Welche Methode wird in `g` aufgerufen?
- statisches Binden: die Methode des Basistyps `cell`
- dynamisches Binden: die Methode des synamischen Typs `reCell`

Typabfragen

was ist der wahre Typ eines Objekts?

```
typecase
  when rc: instanceType(reCell) do rc.restore();
  when cc: instanceType(cell) do c.set(3);
end;
```

Nachteile
- wenn `class` eine weitere Unterklasse bekommt, muss das `typecase` erweitert werden

Spezialisierung des Typs von `self`

```
class c is
  var x: Integer := 0;
  method m(): InstanceType(c) is ... self ... end;
end
subclass c' of c is
  var y: Integer := 0;
end
```

welchen Ergebnistyp hat eine Nachricht an ein *c'*-Objekt?

```
class c is
  var x: Integer := 0;
  method m(): Self is ... self ... end;
end
```


Notiz: fortgeschrittene klassenbasierte Eigenschaften

einfache klassenbasierte Konzepte

basieren auf der Annahme
- Vererbung (Übernahme von Implementierungen)
- Unterklassen-Beziehung
- Untertyp-Beziehung

fallen alle zusammen

Das ist sehr ökonomisch

Manchmal schränkt es aber die Wiederverwendbarkeit von Code ein

modernere Sprachen

trennen diese drei Konzepte
- insbesondere Untertyp- und Unterklassen-Beziehung

Objekttypen (Schnittstellen)

Implementierung und Schnittstelle (*interface*, Typ) wird getrennt

```
ObjectType Cell is
  var contents: Integer;
  method get(): Integer;
  method set(n: Integer);
end
ObjectType ReCell is
  var contents: Integer;
  var backup: Integer;
  method get(): Integer;
  method set(n: Integer);
  method restore();
end
```


Untertyp-Beziehung auf Objekttypen

strukturierte Untertyprelation

O' <: *O* wenn *O'* mindestens die Komponenten von *O* enthält

Also: ReCell <: Cell

mehrfache Untertypbeziehung

```
ObjectType ReInteger is
  var contents: Integer;
  var backup: Integer;
  method restore();
end
```

Dann gilt auch: ReCell <: ReInteger

c' ist Unterklasse von *c* ? ObjectType(*c'*) <: ObjectType(*c*)

Objektbasierte (klassenlose) Sprachen

Objekte ohne Klassen

Objekttypen definieren Schnittstellen (Signaturen)
als Verbunde mit Variablen und Methoden (Funktionen und Prozeduren)
Objekte implementieren die Komponenten eines Objekttyps
Vererben durch *Klonen* (Kopieren der Komponenten eines Objekts)
Überschreiben durch Neuzuweisung von Methoden

Der Weg zur objektorientierten Glückseligkeit (I)

laut *Bertrand Meyer* **(1988)** *Object-oriented Software Construction*

1. Objekte
- Variablen werden mit ihren Methoden gekapselt
- Daten legen die Programmstruktur fest (nicht die Funktionen, *data-driven design*)

2. Datenabstraktion (*information hiding*)
- Objekte sind abstrakte Variablen; ihre Darstellung bleibt verborgen
- sie dürfen nur mit Zugriffsmethoden überschrieben werden

3. automatische Speicherverwaltung (*garbage collection*)
- nicht mehr benutzte Objekte werden automatisch wieder freigegeben

4. Klassen
- jeder zusammengesetzte Typ ist ein Modul und jeder Modul definiert einen Typ
- von jeder Klasse können beliebig viele Objekte angelegt werden (*Instantiierung*)

Der Weg zur objektorientierten Glückseligkeit (II)

5. Vererbung
- Klassen können *Erweiterungen* / *Beschränkungen* anderer Klassen sein (*Unterklassen*)

6. Polymorphie, Überschreiben und dynamisches Binden
- Methoden lassen sich auf Objekte mehrerer Klassen anwenden
- Methoden können in Unterklassen verschieden realisiert sein
- Methoden werden anhand des dynamischen Typs des Empfängerobjekts bestimmt

7. mehrfache und wiederholte Vererbung
- eine Klasse kann von *mehreren* Klassen erben (*multiple inheritance*)
- eine Klasse kann *mehrmals* von einer Klasse erben (*repeated inheritance*)

8. *Fehlt sonst noch was zur objektorientierten Glückseligkeit* **?**

Vorteile und Einsatzfelder

Vorteile

natürliche Modellierung zustandsbehafteter Systeme
Erweiterbarkeit erlaubt evolutionäre Programmentwicklung
Modularisierung und Vererbung erleichtern Wiederverwendung

Einsatzfelder

alle (sagen manche)
- interaktive Systeme
- nebenläufige und verteilte Systeme

Nachteile

das Maschinenmodell ist nicht so fassbar
Implementierungen sind nicht so effizient wie in imperativen Sprachen
- lohnt sich der Preis?

Smalltalk (1980)

Entwerfer

Xerox Palo Alto (1972–80)

Zweck

eine Sprache und ein interaktives graphisches Betriebssystem

Entwurfsziele

eine *reine* objektorientierte Sprache
alle Werte sind Objekte (auch *Klassen* selber)
alle Operationen sind Methoden

Datenstrukturen in Smalltalk

Werte

alle Werte sind *Objekte*
primitive Klassen enthalten Grundwerte und die üblichen Operationen
Beispiel
`Integer` die Klasse mit den Objekten ... `-2`, `-1`, `0`, `+1`, `+2` ...

`+ - * //` ty pische Methoden
`even gcd`

Bedeutung von Methodenanwendungen
`n+m`
die Methode `+` des Empfänger-Objektes `n` wird auf sich selbst und das Objekt `m` angewendet und liefert ein Objekt, die Summe

Ausdrücke in Smalltalk

Ausdrücke

einstellig *E M*
zweistellig *E ? E1*
mehrstellig *E M1: E1 ... Mn: En*

Beispiel (zusammengesetzte Objekte)
`Array`die Klasse der Felder (dynamisch)

`at:`Selektion `a at:i`
`at: put:`selektives Überschreiben `a at:i put: x`

Notiz: Typen in Smalltalk

Typisierung

dynamisch, Überprüfung während der Laufzeit

Konsequenzen

Heterogenität
- ein zusammengesetztes Objekt kann Objekte *verschiedener* Unterklassen enthalten

Erweiterbarkeit
- es kann sogar Objekte aus Klassen enthalten, die bei seiner Konstruktion noch *gar nicht existierten*

Variablen in Smalltalk

Variablen

sind immer *Referenzen auf Objekte* (beliebigen Typs)
auch Argumente und Ergebnisse von Operationen sind Referenzen auf Objekte

Beispielsweise die Zuweisung
```
n <- m
```

Vorsicht bei zusammengesetzten Objekten mit selektivem Überschreiben!
```
b <- a
a at:i put: x
```
verändert das eine Objekt, auf das sowohl `a` als auch `b` zeigen
- Referenzsemantik

Kontrollfluss in Smalltalk

Kontrollfluss

Kontrollstrukturen sind nicht *eingebaut*
sondern als Methoden von *Bibliotheks-Klassen*, vordefiniert

Abstraktion in Smalltalk

Blöcke (Funktionsabstraktionen)

parameterlos
`counter <- [ n <- n+1 ]`Definition
`counter value`Aufruf
Durch Senden der Blockmethode `value` wird der Block ausgewertet und `n` um `1` erhöht
Blöcke sind Objekte der Klasse `BlockContext`, die die Methode `value` zur Verfügung stellt

parameterisiert
`succ <- [ i: i <- i+1 ]`Definition
`succ i`Aufruf

Befehle in Smalltalk

bedingte Anweisungen

`ifTrue: ifFalse` ist eine vordefinierte Methode der Klasse `Boolean`
```
n > 0 ifTrue: [ n /m] ifFalse: [ 0]
```

Schleifen

`whileTrue:` ist eine vordefinierte Methode der Klasse `Boolean`
```
m <- 0.
[ n > 0 ] whileTrue: [ n <- n // 2. m <- m+1 ]
```

Zählschleifen

werden mit der Methode `do:` realisiert.
`do:` ist für alle zusammengesetzten Objekte definiert und erwartet einen Block mit einem Parameter
```
total <- 0.
monthsize do: [ : size | total <- total+size ]
```


Klassen in Smalltalk

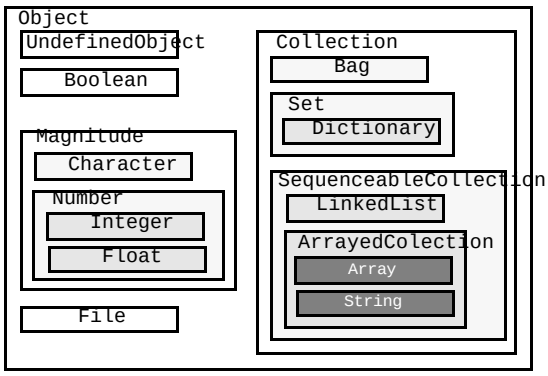
Klassen-Definitionen

Eine Klassen-Definition enthält
- den Namen der Klasse
- den Namen ihrer Oberklasse
- den Namen aller verborgenen Variablen
- den Namen und die Definition aller Methoden
- Der Smalltalk -Interpreter behandelt diese Komponenten separat
  — es gibt keine “Syntax” für Klassen

Beispiel: Punkte

Name `Point` Oberklasse: keine Variablen: `x` und `y` (Zahlen)
Operationen
```
placeat: xnew and: ynew
  x <- xnew. y <- ynew
xcoord
  ^ x
ycoord
  ^ y
moveby: dx and: dy
  x <- x + dx. y <- y + dy
distance: other
  ^ ( (x - other xcoord) squared
    + (y - other ycoord) squared ) sqrt
draw
...
```

vordefinierte Klassen in Smalltalk (vereinfacht)



Vererbung in Smalltalk

Unterklassen

`Circle`ist Unterklasse von `Point` mit der neuen verborgenen Variablen `r` (Radius)

und der neuen Methode `radius`

```
radius
    ^ rKreis-Radius selektieren
```

Vererbung

alle Operationen von`Point` können auf`Circle` angewendet werden

Beispiel (c ist ein Kreis)

`c xcoord`	Koordinate des Kreises
`c distance: p`	Abstand von einem Punkt
`p distance: c`	Abstand eines Punktes von c
`c.moveby: 5 and: 0`	Kreis verschieben

Überschreiben in Smalltalk

Überschreiben

Methoden von`Point` können für`Circle` *überschrieben* werden

```
placeat: xnew and: ynew radius: rnew
x <- xnew. y <- ynew r <- rnew

draw
   ...
```

`c placeat: 0 and: 0 radius:` 2schafft einen Kreis

`c draw`zeichnet einen Kreis

Zusammenfassung: Smalltalk

Prinzipientreue

Typvollständigkeit *ja*

Qualifikation: *ja*

Abstraktion: *ja*

Korrespondenzprinzip*nicht anwendbar*

Besonderheiten

ausdrucksorientiert

dynamische Typisierung

interpretiert

Entwurfsschwächen

dynamische Typisierung schafft Flexibilität,

verhindert aber statische Typüberprüfung

Notiz: Eiffel (Bertrand Meyer ab 1985)

Motto

methodisch korrekte Objektorientierung

design by contract

"thought is inseparable from its expression"

Entwurfsziele

wiederverwendbar, erweiterbar, verläßlich, effizient, offen, portabel

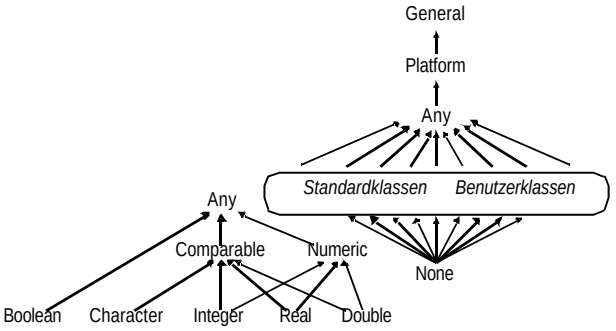
Eigenschaften

rein, sicher, statisch typisiert

Besonderheit

Zusicherungen

Klassenhierarchie



Klassendefinition

```
class POINT
feature {ANY}
  xcoord, ycoord: REAL;
  placeat (xnew, ynew: REAL) is
    do xcoord := xnew;
       ycoord := ynew
    end;
  moveby (xshift, yshift: REAL) is
    do xcoord := xcoord + xshift;
       ycoord := ycoord + yshift
    end;
  distance (other: like Current) : REAL is
    do Result := sqrt (  (xcoord - other.xcoord) ^ 2
       + (ycoord - other.ycoord) ^ 2 )
    end;
  draw is ...
end -- class POINT
```


Notiz: Eigenschaften von Klassen

keine explizite Schnittstelle

kein expliziter Import

qualifizierter Export

exportierte Variablen können syntaktisch nicht überschrieben werden

keine globalen Variablen

Schreibweise verbirgt die Realisierung einer Eigenschaft

Variable, Konstante, Funktion?

Kopier- oder Zeigersemantik (**`expanded`**)

Vererbung

Beispiel: Kreise

```
class CIRCLEinherit POINT
            undefine placeat
            redefine placeat, draw
            end;
feature {CIRCLE}
  radius: REAL;
feature {ANY}
  placeat (xnew, ynew, rnew : REAL) is
    do xcoord := xnew;
       ycoord := ynew;
       radius := rnew
    end;
  draw
    Zeichnen eines Kreises auf dem Bildschirm
end -- class CIRCLE
```


Notiz: Eigenschaften

explizites Überschreiben von Methoden (`draw`)
"Undefinieren" für Redefinition
von Methoden mit anderer Signatur (`placeat`)
Überladen kann ausgeschlossen werden (**`final`**)
(für Methoden oder ganze Klassen)

Mehrfachvererbung

```
class WINDOW inherit
  SCREEN_OBJECT,
  TEXT
    redefine close,
  TREE
    rename
      make_leaf as make_window,
      parent as parent_window,
      sibling as sub_window,
       ...
    export {WINDOW} make_window
    undefine insert_top
  end
feature
  close ...
end
```


Notiz: Kontrolle der Vererbung

- umbenennen
- reexoprtieren
- an andere (weniger) Klienten
- löschen
- überladen

Referenz- und Wertsemantik

```
class C
feature {ANY}
  x,y: COMPLEX
end

expanded class COMPLEX
  inherit NUMERIC
feature {ANY}
  re, im: Real;
end -- COMPLEX

class D
feature {ANY}
  c: expanded C
end
```


zurückgestellte Klassen

```
deferred class NUMERIC
feature {ANY}
  infix "+"  (other: NUMERIC): NUMERIC deferred
  infix "-"  (other: NUMERIC): NUMERIC deferred
  infix "*"  (other: NUMERIC): NUMERIC deferred
  infix "/"  (other: NUMERIC): NUMERIC deferred
  prefix "+" (other: NUMERIC): NUMERIC deferred
  prefix "-" (other: NUMERIC): NUMERIC deferred
end -- NUMERIC
```


Notiz: : deferred

die Implementierung zurückgestellter Klassen ist unvollständig
Beispiel: vordefinierte Klasse`NUMERIC`:
einige Attribute oder Methoden können aber definiert sein

Unterschied zu Schnittstellen von Java
eine zurückgestellte Klasse kann
- Atribute und
- (einige) Methoden

definieren

die Verwendung ist ber ähnlich (wie im Beispiel

generische Klassen

Definition

```
class TREE[T] ...
class MAP[S,T] ...
```

Instanziierung

```
t: TREE[ARRAY[INTEGER]];
m: MAP[INTEGER, REAL];
```

eingeschränkte Parameter

```
class SORTED_LIST[COMPARABLE -> T] ...
```

Entwurf durch Vertrag

Zusicherungen

erlauben die Angaben von Eigenschaften des Programms
mit booleschen Ausdrücken

sie sind spezielle Kommentare an spezifizierten Sellen

sie dienen als Kommentar und für Laufzeitüberprüfungen

Zusicherungen an eine Klasse gehören zum Vertrag
zwischen Klassenautor und Klienten