3 ST-Klassen und Methoden

Im vorigen Abschnitt wurden die wichtigsten sprachlichen Ausdrucksmittel der OOP-Sprache ST eingeführt und nebenbei verschiedene Objektarten dargestellt. In diesem Abschnitt wollen wir uns den eigentlichen Klassen, ihre Definition, der Definition ihrer Methoden und der Vererbung zuwenden.

Ein OO-Programm ist letztlich ein Objekt einer Klasse, welches durch eine Nachricht aktiviert wird.

3.1 Das Klassenkonzept

Zur Erinnerung, eine Klasse ist aufgebaut aus:

(1)	**Instanzvariablen**	Haben bei verschiedenen Objekten verschiedene Werte.
(2)	**Instanzmethoden**	Werden vom Objekt zur eigenen Datenmanipulation herangezogen.
(3)	**Klassenvariablen**	Haben immer den gleichen Wert.
(4)	**Klassenmethoden**	Werden von der Klasse ausgeführt.

Klasse:	# *Name*
(1) Instanzvariablen	**(2) Instanzmethoden**
(3) Klassenvariablen	**(4) Klassenmethoden**

Klassenphilosophie:

- Jedes Objekt ist einer Klasse zugeordnet.
- Objekte können Nachrichten empfangen. Alle Objekte einer Klasse antworten auf die gleichen Nachrichten, benutzen die gleichen Instanzmethoden, um diese auszuführen.
- Alle Objekte einer Klassen haben die gleichen Instanzvariablen. Sie füllen den Datenspeicher mit konkreten Werten. Diese können nur durch Instanzmethoden verändert werden und sind keinem anderen Objekt der Klasse oder anderen Klassen zugänglich.

⇒ Klassen geben die äußere Form ihrer Objekte vor.

- Jede Klasse selbst ist ein Objekt.
- Klassen können Nachrichten empfangen. Sie benutzen Klassenmethoden, um diese auszuführen.
- Klassenvariablen haben einen konkreten Wert, diese können nur durch Klassenmethoden verändert werden.

⇒ Klassen besitzen alle Eigenschaften eines Objektes, sie sind Objekte der Klasse **Metaclass.**

3.1.1 Klassendefiniton

NameOfSuperclass* subclass: #*NameOfClass
instanceVariableNames: '*instVarName1 instVarName2*'
classVariableNames: '*ClassVarName1 ClassVarName2*'
poolDictionaries: '*PoolDicName1 PoolDicName2*'

NameOfSuperclass
Klassenname der Oberklasse, der die zu definierende Klasse untergeordnet werden soll,
NameOfClass
Klassennamen sind *ST-Namen* und beginnen mit einem Großbuchstaben.
instVarName1 instVarName2
Instanzvarialennamen sind *ST-Namen*, beginnen mit einem Kleinbuchstaben und werden vererbt.
ClassVarName1 ClassVarName2
Instanzvarialennamen sind *ST-Namen*, beginnen mit einem Großbuchstaben und werden vererbt.
PoolDicName1 PoolDicName2
Poolverzeichnisse sind Objekte der Klasse **Dictionary**. Ihre Namen sind *ST-Namen* und beginnen mit einem Großbuchstaben. Sie ordnen speziellen Symbolen Objekte zu, welche in den Methoden der Klasse benötigt werden. Da Poolverzeichnisse nicht vererbt werden, sind diese in den Unterklassen, um Methoden der Oberklasse benutzen zu können, stets wieder aufzunehmen.

In ST werden die Klassen in sogenannte **Applikationen** unterteilt. Dabei werden üblicherweise Klassen mit ähnlichen Aufgaben bzw. Klassen, die zu einem Projekt gehören, zu einer Applikation zusammengefasst. Applikationen sind selbst Klassen, ihre Namen sind *ST-Namen* und beginnen mit einem Großbuchstaben. Applikationen sind oft von anderen abhängig, d.h. Klassen anderer Applikationen müssen zur Verfügung stehen. Diese Abhängigkeit muss mit angegeben werden, i.R. ist es die Applikation **Kernel**.

Mit einem Schalter **private / public** kann festgelegt werden, ob eine Klasse nur in der Applikation sichtbar ist, in der sie definiert wurde oder nicht.

Vererbung:

NameOfSuperclass* subclass: #*NameOfClass
Die Definition von Klassen als Unterklasse anderer Klassen ist in OOP-Sprachen ein grundlegendes Konzept. In ST ist jede Klasse Unterklasse genau einer anderen Klasse (Einfachvererbung). Auf diese Weise entsteht eine Hierarchie von Klassen. Tatsächlich ist **Object** in ST die einzige für den Programmierer interessante Klasse, die nicht Unterklasse einer anderen Klasse ist. Die meisten Klassen werden direkt als Unterklasse der Klasse **Object** angelegt.

nil subclass: #Object
instanceVariableNames: ''

classVariableNames: 'Dependents'
poolDictionaries: 'SystemPrimitiveErrors SystemExceptions'

3.1.2 Methodendefinition

messagePattern
"comment"
| temporaries |
statements

messagePattern
Nachricht, die die Methode verarbeitet, einschließlich notwendiger formaler Parameter.
comment
Kommentar, oft einzige Informationsquelle zur Methode.
| temporaries | statements
Ausdrucksfolge

Mit einem Schalter **instance / class** kann festgelegt werden, ob eine Methode als Instanzmethode oder als Klassenmethode abgespeichert werden soll.
Mit einem Schalter **private / public** kann festgelegt werden, ob die Methode nur durch ihre eigene Objekte zugänglich sein soll oder öffentlich zur Verfügung steht.

Die Methoden werden analog entsprechend ihrer Funktion in **Kategorien** unterteilt. Es werden Methoden mit ähnlichen Aufgaben zu einer Kategorie zusammengefasst. Diese dienen als Ordnungsmerkmal und haben keinen Einfluss auf ihre Funktion. Kategoriennamen sind *ST-Namen* und beginnen mit einem Großbuchstaben.

3.1.3 System Browser

Klassen und Methoden werden in unterschiedlichen **System Browser** definiert und können dort recherchiert werden. In jedem der Fenster sind sowohl der Quellcode der Klassen, Applikationen und Kategorien als auch der Methoden (sofern sie nicht in Assembler geschrieben wurden) und ihre Abhängigkeiten ersichtlich. Oft ist es die einzige Informationsquelle. Die Methodenquellen sollten nicht nur als Information verstanden werden, sondern auch als Beispiele für die Programmierpraxis. Sie sind effektiv und kompakt angelegt.
Es gibt im wesentlichen zwei Typen von Fenster der **System Browser**. Ihr Aufruf erfolgt aus dem **System Transcript**.

System Transcript
- Smalltalk tools / Browse Applications ⇒ *Applications Browser*

Browse Senders..., **Browse Implementors...**, **Browse Variable Refs...** und **Browse Category...** weisen das Vorkommen bestimmter Methoden und Variablen in den Klassen des Systems aus (**class>>method**).

Browse Applications, **Browse Classes**, **Browse Hierarchy...**, **Browse Application...** und **Browse Class...** geben Informationen zu speziellen Applikationen, Klassen, Kategorien, Methoden und ihre Abhängigkeit.

3.2 Eigene Klassen

3.2.1 Applications Browser

Alle Applikationen, die dem System bekannt sind, werden im **Applications Browser** zusammengefasst. Dort lassen sich auch Applikationen erzeugen, löschen und zugehörige Klassen, Kategorien und Methoden bearbeiten. Unterstützt wird eine derartige Bearbeitung durch die Angabe des Quellcodes bzw. durch Template (Schablonen), in denen direkt aktualisiert werden kann. Ähnliche Aktionen sind auch in den anderen Fenstern möglich, sollen aber hier nicht besprochen werden.

System Transcript
- Smalltalk tools / Browse Applications ⇒ *Applications Browser*

Erzeugen einer neuen Applikation:
Applications Browser
- Applications / Applications / Create **Biologie**
- Abhängigkeit **Kernel**
- Speichern: File / Save

Biologie erscheint in der Applikationsliste in runden Klammern und ist als Unterklasse der Klasse **Application** eingetragen:

```
Application subclass: #Biologie
   instanceVariableNames: ''
   classVariableNames: ''
   poolDictionaries: ''
```

Zu jeder Applikation kann man Unterapplikationen erstellen ⇒ Applikationshierarchie.

Löschen einer Applikation:
Applications Browser
- Markieren der zu löschenden Applikation.
- Applications / Applications / Delete

Die markierte Applikation verschwindet aus der Applikationsliste und mit ihr alle in ihr erzeugten Klassen und deren Methoden.

Erstellen einer Klasse zu einer Applikation:
Applications Browser
- Markieren der Applikation, in der die Klasse erstellt werden soll.
- Klassentemplate wird angezeigt und ist zu aktualisieren.
- Speichern: File / Save

```
NameOfSuperclass subclass: #NameOfClass
   instanceVariableNames: 'instVarName1 instVarName2'
   classVariableNames: 'ClassVarName1 ClassVarName2'
   poolDictionaries: ''
```

⇒

```
Object subclass: #LifeForm
   instanceVariableNames: 'age color'
   classVariableNames: ''
   poolDictionaries: ''
```

Mit einem Schalter **private / public** kann **vor der Definition** der Klasse festgelegt werden, ob eine Klasse nur in der Applikation sichtbar ist, in der sie definiert wurde oder nicht.

Löschen einer Klasse zu einer Applikation:
Applications Browser

- Markieren der Applikation und in ihr die zu löschenden Klasse.
- Classes / Delete Classes

Die markierte Klasse verschwindet aus der Klassenliste, mit ihr alle in ihr definierten Kategorien und Methoden. Klassen lassen sich nicht löschen, falls noch globale Variable mit Objekten der Klasse als Wert existieren.

Erstellen einer Kategorie zu einer Klasse:
Applications Browser

- Markieren der Applikation und der Klasse, in der eine neue Kategorie erstellt werden soll.
- Categories / Add Category **Alter**
- Speichern: File / Save

Alter erscheint in der Kategorienliste, zunächst in runden Klammern, da zu **Alter** noch keine Methoden existieren.

Löschen einer Kategorie zu einer Klasse:
Applications Browser

- Markieren der Kategorie, die gelöscht werden soll.
- Categories / Remove Categories

Die markierte Kategorie verschwindet aus der Kategorienliste, mit ihr alle definierten Methoden.

Erstellen einer Methode zu einer Klasse:
Applications Browser

- Markieren der Applikation, der Klasse und der Kategorie, in der eine neue Methode erstellt werden soll.
- Methods / New Method Template
- Methodentemplate wird angezeigt und ist zu aktualisieren.
- Speichern: File / Save

```
messagePattern
   "comment"
   | temporaries |
   statements
```

⇒

```
age
   "Antwortet mit dem Alter"

   ^age isNil
      ifTrue: [0]
      ifFalse: [age]
```

age erscheint in der Methodenliste unter der Kategorie **Alter**.

Mit einem Schalter **instance / class** kann vor der eigentlichen Definition festgelegt werden, ob eine Methode als Instanzmethode oder als Klassenmethode abgespeichert werden soll.
Mit einem Schalter **private / public** kann vor der eigentlichen Definition festgelegt werden, ob die Methode nur durch ihre eigene Objekte zugänglich sein soll oder öffentlich zur Verfügung steht.

Löschen einer Methode zu einer Klasse:
Applications Browser
- Markieren der Methode, die gelöscht werden soll.
- Methods / Delete Methods

Die markierte Methode verschwindet aus der Methodenliste.

3.2.2 Beispiel

Es sollen verschiedene Lebewesen dargestellt werden. Als Eigenschaften sind das Alter und die Farbe zu erfassen. Jedes Lebewesen soll bei entsprechenden Nachrichten sein Alter und seine Farbe bestimmen, sein Alter und seine Farbe verändern können und am Geburtstag um ein Jahr älter werden.
Dazu führen wir die Klasse **LifeForm** der Applikation **Biologie** mit den Instanzvariablen **age** und **color** ein. Die Instanzmethoden zum Verständnis der Nachrichten seien **age**, **age:**, **birthday** der Kategorie **Alter** und **color**, **color:** der Kategorie **Farbe**.

```
Object subclass: #LifeForm
   instanceVariableNames: 'color age '
   classVariableNames: ''
   poolDictionaries: ''

LifeForm comment: 'Klasse aller Lebewesen.'
```

```
LifeForm publicMethods

age
   "Antwortet mit dem Alter"
   ^age isNil
      ifTrue: [0]
      ifFalse: [age]

age: anInteger
   "Aendert sein Alter und teilt dem Empfaenger der Nachricht sein altes
Alter mit."
   | previousAge |
   previousAge := age isNil
      ifTrue: [0]
      ifFalse: [age].
   age := anInteger.
   ^previousAge

birthday
   "Es ist Geburtstag. Der Empfaenger wird ein Jahr aelter."
   self age: self age +1

color
   "Liefert die Farbe eines Lebewesens."
   ^color isNil
      ifTrue:[#none]
      ifFalse:[color]

color: aSymbol
   "Teile dem Lebewesen mit, welche Farbe es annehmen soll."
   ^color := aSymbol


|a|
a := LifeForm new.
a color: #blue; age: 3; birthday; birthday.
^Array with: a color with: a age                ⇒              (#blue 5)
```

3.3 Unterklassen und Vererbung

Um verschiedene Arten von Lebewesen jetzt genauer zu beschreiben, definieren wir verschiedene Unterklassen von **LifeForm**. Diese erben die gesamte Beschreibung der Instanzeigenschaften und Instanzmethoden aus der Oberklasse und werden um spezifische Eigenschaften und Methoden erweitert.
Für die Erzeugung von Unterklassen der Klasse **LifeForm** eignet sich besonders der **Class Browser** dieser Klasse. Dort lassen sich Unterklassen erzeugen, löschen und zugehörige Kategorien und Methoden bearbeiten. Ähnliche Aktionen sind auch in den anderen Fenstern möglich.

System Transcript

- Smalltalk tools / Browse Class... **LifeForm** ⇒ *Class Browser*

Es sollen Fische durch die Unterklasse **Fish** der Klasse **LifeForm** in der Applikation **Biologie** dargestellt werden, die auf die Nachricht **canSwim** der Kategorie **Bewegung** mit **true** antworten.

```
LifeForm subclass: #Fish
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

Fish comment: 'Fische. Sie koennen schwimmen.'

Fish publicMethods

canSwim
    "Jeder Fisch kann schwimmen."
    ^true
```

Objekte der Klasse **Fish** erben alle Variablen und Methoden der Klasse **LifeForm**:

```
|a|
a := Fish new.
a color: #blue; age: 3; birthday; birthday.
^Array with: a color with: a age                ⇒        (#blue 5)
```

Objekte der Klasse **Fish** verstehen die Instanzmethode **canSwim**, Objekte der Klasse **LifeForm** nicht:

```
|a|
a := Fish new.
^a canSwim                ⇒        true

|a|
a := LifeForm new.
^a canSwim        ⇒        LifeForm does not understand canSwim
```

Es sollen Pflanzen durch die Unterklasse **Plant** der Klasse **LifeForm** in der Applikation **Biologie** dargestellt werden. Pflanzen sind in der Regel grün und ortsgebunden. Die Instanzmethode **location, location:** der Kategorie **Bewegung** sollen den Standort senden bzw. festlegen.

```
LifeForm subclass: #Plant
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

Plant comment: 'Pflanzen. Sie verfuegen ueber einen festen Standort.'
```

```
Plant publicMethods

color
   "Eine Pflanze ist, falls nicht anders festgelegt, gruen"
   ^color isNil
      ifTrue: [#green]
      ifFalse: [color]
```

Alle Pflanzen sind grün:

```
|a|
a := Plant new.
^a color                                  ⇒          #green
```

Oder anders gefragt:

```
Plant new color == #green                 ⇒          true
```

Sie können aber auch andere Farben annehmen:

```
|a|
a := Plant new.
a color: #blue.
^a color                                  ⇒          #blue
```

Da nicht nur Pflanzen einen Standort haben, sondern alle Lebewesen, sollte man den Standort in einer Instanzvariablen **location** der Klasse **LifeForm** abspeichern und auch dort die notwendigen Instanzmethoden **location**, **location:** zur Verfügung stellen, die den Standort angeben bzw. verändern sollen. Damit werden die neue Instanzvariable und die entsprechenden Instanzmethoden allen von **LifeForm** abgeleiteten Klassen zugänglich gemacht.
Diese Änderungen in der Klasse **LifeForm** können direkt im **Class Browser** durchgeführt werden und wirken sofort nach deren Speicherung.

```
Object subclass: #LifeForm
   instanceVariableNames: 'color age location '
   classVariableNames: ''
   poolDictionaries: ''

LifeForm publicMethods

location
   "Antwortet mit dem Standort."
   ^location

location: aPoint
   "Legt den Standort fest.
   Antwortet mit dem neuen Standort"
   ^location := aPoint
```

Eine Pflanze hat zunächst keinen Standort, um dies zu ändern muss man ihn erst definieren.

```
|a|
a := Plant new.
^a location                                   ⇒              nil

|a|
a := Plant new.
^a location: 1 @ 1                            ⇒              1 @ 1
```

Die Instanzmethode **location:** gestattet jeder Zeit eine Standortänderung, auch für Pflanzen. Dies sollte durch eine eigene Instanzmethode der Klasse **Plant** unterbunden werden. Diese Methode überlagert somit die Methode gleichen Namens der Klasse **LifeForm**. Bei der Methodendefinition findet die Systemvariable **super** Verwendung, um auf die gleichnamige Methode der Klasse **LifeForm** zurückzugreifen:

```
Plant publicMethods

location: aPoint
   "Legt den Standort fuer Pflanzen fest.
   Dieser kann sich nicht aendern"
   location isNil
      ifTrue: [super location: aPoint].
   ^location
```

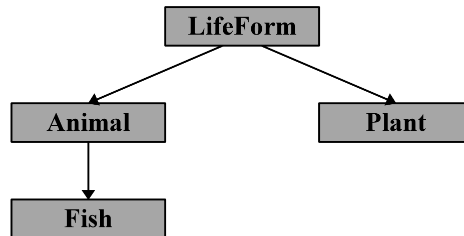
Pflanzen können sich nicht fortbewegen, jedoch Fische:

```
|a|
a := Plant new.
a location: 1 @ 1.
^a location: 2 @ 2                            ⇒              1 @ 1

|a|
a := Fish new.
a location: 1 @ 1.
^a location: 2 @ 2                            ⇒              2 @ 2
```

3.4 Klassenhierarchie und abstrakte Klassen

Um jetzt das Beispiel zu erweitern, sollten noch andere Lebewesen hinzugezogen werden. Damit bietet sich eine Unterteilung der Lebewesen in Pflanzen und Tiere an, wobei Fische dann eine Unterklasse der Tiere darstellen würden. Diese Überlegung zieht eine Veränderung der jetzigen Hierarchie nach.



Die neuen Klassenbeziehungen lassen sich im **Class Browser** der Klasse **LifeForm** durch Einführen der neuen Klasse **Animal** und Änderung der Ober- und Unterklassenbeziehung der Klasse **Fish** leicht durchführen. Alle anderen Eintragungen bleiben dabei erhalten.

```
Object subclass: #LifeForm
    instanceVariableNames: 'color age location '
    classVariableNames: ''
    poolDictionaries: ''

LifeForm subclass: #Plant
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

LifeForm subclass: #Animal
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

Animal subclass: #Fish
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
```

Wie wir feststellen können, wird es bei dem Ausbau der Lebewesenhierarchie kein Objekt der Klasse **LifeForm** geben. Solche Klassen nennt man **abstrakte Klassen**. Sie bilden die Wurzel einer Hierarchie und stellen allen Unterklassen allgemeine Methoden und Variablen zur Verfügung. Betrachten wir die Möglichkeit, daß jedes Objekt die Frage ob es ein Fisch ist, ob es eine Pflanze ist usw., beantworten können soll. Zunächst definiert man eine Methode in der abstrakten Klasse, die auf eine derartige Fragen mit **false** antwortet. In jeder Unterklasse wird die entsprechende Frage mit **true** überlagert:

```
LifeForm publicMethods

isAnimal
    "Antwortet mit false."
    ^false
```

```
isPlant
    "Antwortet mit false."
    ^false

Plant publicMethods

isPlant
    "Antwortet mit true."
    ^true

Animal publicMethods

isAnimal
    "Antwortet mit true."
    ^true
```

Während eine Pflanze mit der eigenen Instanzmethode **isPlant** antwortet, wird beim Fisch die Instanzmethode **isPlant** der abstrakten Klasse **LifeForm** aufgerufen:

```
|a|
a := Plant new.
^a isPlant                              ⇒                    true

|a|
a := Fish new.
^a isPlant                              ⇒                    false
```

Diese Vorgehensweise lässt sich nicht auf die Instanzmethode **canSwim** erweitern, denn es gibt außer Fischen durchaus noch andere Tiere, die schwimmen können. Möchte man diese Eigenschaft für alle Lebewesen in das Modell aufnehmen, so geht dies nur über eine neue Instanzvariable in der abstrakten Klasse **LifeForm**.

3.5 Klassenvariablen und Klassenmethoden

Eine Pflanze hat zunächst keinen Standort, um dies zu ändern, muss man ihn erst definieren.

```
|a|
a := Plant new.
^a location                             ⇒                    nil

|a|
a := Plant new.
^a location: 1 @ 1                      ⇒                    1 @ 1
```

Wird die zweite Nachricht nicht gesendet, so bleibt die Anfangsbelegung undefiniert und die erste Nachricht liefert auch weiterhin **nil**.

Anfangsbelegungen der Instanzvariablen kann man unterschiedlich planen.
1. Möglichkeit: Anfangsbelegungen in den Methoden berücksichtigen:

```
age
"Antwortet mit dem Alter"
^age isNil
   ifTrue: [0]                        ←    Anfangsbelegung
   ifFalse: [age]
```

2. Möglichkeit: Defaultwerte als Klassenvariable einführen und diese mittels Klassenmethode beim Erzeugen eines Objekts setzen:
Dazu ist eine Erweiterung der Klasse **LifeForm** notwendig. Die **Klassenvariable DefaultLocation** wird eingeführt und durch die **Klassenmethode new** gesetzt:

```
Object subclass: #LifeForm
   instanceVariableNames: 'color age location '
   classVariableNames: 'DefaultLocation '      ←    Klassenvariable
   poolDictionaries: ''

LifeForm class publicMethods

new                                            ←    Klassenmethode
   "Legt den Ausgangsstandort fest."
   DefaultLocation := 0 @ 0. ^super new
```

Die Instanzmethode **location** nutzt dann die Klassenvariable **DefaultLocation** als Anfangsbelegung:

```
LifeForm publicMethods

   location
      "Antwortet mit dem Standort."
      ^location isNil
         ifTrue: [DefaultLocation]
         ifFalse: [location]
```

Jedes Lebewesen hat jetzt stets einen Standort:

```
|a|
a := LifeForm new.
^a location                        ⇒    0 @ 0
```

Diesen kann ein Lebewesen ändern:

```
|a|
a := LifeForm new.
a location: 1 @ 2.
^a location                        ⇒    1 @ 2
```

Auch Pflanzen können ihren Standort festlegen:

```
|a|
```

```
a:= Plant new.
^a location: 1 @ 1                                  ⇒            1 @ 1
```

Pflanzen können aber ihren Standort nicht wechseln:

```
|a|
a:= Plant new.
a location: 1 @ 1.
^a location: 2 @ 2                                  ⇒            1 @ 1
```

Fische können ihren Standort wechseln:

```
|a|
a:= Fish new.
a location: 1 @ 1.
^a location: 2 @ 2                                  ⇒            2 @ 2
```

⇒ In **Klassenvariablen** werden Gemeinsamkeiten der Objekte einer Klasse zusammengefaßt.
Klassenmethoden manipulieren ihre Werte.

Schließlich soll noch den Fall betrachtet werden, dass ein Objekt seiner eigenen Klasse eine Nachricht sendet.

LifeForm publicMethods

```
createDescendant
    "Erzeuge einen eigenen Nachkommen"
    ^self class new
```

self class bestimmt die eigene Klasse, **new** erzeugt dann ein Objekt dieser Klasse.

Erzeugen eines Nachkommen einer Pflanze:

```
|a b|
a := Plant new.
b := a createDescendant.
^b class                                            ⇒            Plant
```

■ **2. Aufgabe**

3.6 Abhängigkeiten

3.6.1 Partnerbeziehung

Zwei Lebewesen sind Partner. Partner sind gleichberechtigt. Sie informieren sich gegenseitig über Änderungen ihrer Daten und reagieren auf Änderungen des anderen Partners.

Bei solchen Methoden müssen indirekte Rekursionen abgefangen werden. Zur Wiederholung:
Direkte Rekursionen: Methoden, die sich selbst aufrufen.
Indirekte Rekursionen: Methoden, die sich wechselseitig aufrufen.

Beispiel: Es ist sicher sinnvoll, eine Partnerbeziehung für Tiere aufzubauen. Dazu ist **partner** als Instanzvariable einzuführen, um einen Partner registrieren zu können:

```
LifeForm subclass: #Animal
    instanceVariableNames: 'partner '
    classVariableNames: ''
    poolDictionaries: ''
```

Mit einer Instanzmethode **partner** der Kategorie **Partner** gibt ein Objekt seinen Partner bekannt und mit einer Instanzmethode **partner:** der Kategorie **Partner** legt ein Objekt seinen Partner fest. Dies ist nur möglich, wenn er selbst und sein zukünftiger Partner noch keine Partner haben:

```
Animal publicMethods

partner
    "Gibt seinen Partner bekannt."
    ^partner

partner: anAnimal
    "Der Empfaenger und anAnimal sind Partner. Falls dies nicht moeglich ist,
    Fehlermeldung."
    ((anAnimal partner notNil and: [anAnimal partner ~~ self])
        or: [self partner notNil and: [self partner ~~ anAnimal]])
            ifTrue: [^self error: 'Has already a partner'].
    self basicPartner: anAnimal
```

Die hier verwendetet Instanzmethode ~~ der Klasse **Object** ist als Verneinung der Identität wie folgt vereinbart:

```
~~ anObject
   "Answer a Boolean which is false when the receiver and anObject
    represent the identical object, and true otherwise."
    ^(self == anObject) not
```

Bei der Instanzmethode **partner:** ist eine Instanzmethode **basicPartner:** aufgerufen wurden, welche die Partnerwahl vollzieht. Diese Instanzmethode wird nur von einer eigenen Instanzmethode aufgerufen und sollte nicht von außen aktiviert werden. Deshalb wird sie als **private** Instanzmethode vereinbart:

```
Animal privateMethods

basicPartner: anAnimal
    "Private - anAnimal ist der neue Partner des Empfaengers und
umgekehrt."
```

```
        partner := anAnimal.
        anAnimal partner isNil ifTrue: [ anAnimal basicPartner: self]
```

Zunächst hat ein Tier keinen Partner:

```
    |a|
    a := Animal new.
    ^a partner                                  ⇒                nil
```

Zwei Tiere werden zu Partnern:

```
    |a b|
    a := Animal new.
    b := Animal new.
    a partner: b.
    ^Array with: a partner with: b partner      ⇒    (an Animal an Animal)
```

Drei Tiere können keine Partner werden:

```
    |a b c|
    a := Animal new.
    b := Animal new.
    c := Animal new.
    a partner: b.
    a partner: c.                               ⇒    Has already a partner
    ^Array with: a partner with: b partner with: c partner
```

Ein Partner soll nun seine Bewegungen dem anderen mitteilen und dieser soll darauf reagieren. Dazu wird eine Instanzmethode **moveTo:** der Kategorie **Bewegung** eingeführt, welche sowohl den Standort ändert, als auch die Mitteilung an den Partner übernimmt:

```
    Animal publicMethods

    moveTo: aPoint
        "Der Empfaenger veraendert seinen Standort und teilt dies seinem Partner
    mit."
        self moveTo: aPoint informedPartner: false
```

Hier wird wiederum eine **private** Instanzmethode aktiviert, welche so angelegt ist, dass mittels einem Boolean-Objekt Rekursionen ausgeschlossen werden:

```
    Animal privateMethods

    moveTo: aPoint informedPartner: aBoolean
        "Private - Der Empfaenger veraendert seinen Standort in aPoint. aBoolean
    ist
        genau dann wahr, wenn der Partner ueber die Veraenderung informiert
    worden
        ist."
        | oldLocation partnerLocation |
        oldLocation := self location.
```

```
    self location: aPoint.
    (aBoolean and: [self partner notNil])
        ifFalse:
        [partnerLocation := self partner location + aPoint - oldLocation.
        self partner moveTo: partnerLocation informedPartner: aBoolean not]
```

Zwei Tiere sind Partner und bewegen sich gemeinsam:

```
|a b|
a := Animal new.
b := Animal new.
a partner: b.
b location: 1 @ 1.
a moveTo: 3 @ 3.
^Array with: a location with: b location          ⇒          (3 @ 3  4 @ 4)
```

3.6.2 Der Dependency-Mechanimus

Sollen mehrere Objekte voneinander abhängig werden, so kann man den in ST bereits vorhandenen **Dependency-Mechanimus** nutzen. Dieser erspart weitere Instanzvariablen und stellt zahlreiche Instanzmethoden der Klasse **Object** zur Verfügung. Die wichtigsten sind:

dependents
Diese Methode gibt alle Abhängigen eines Objekts aus.

addDependent: *anObjekt*
anObjekt wird vom Empfänger der Nachricht abhängig und kann in Zukunft automatisch über Veränderungen des Empfängers unterrichtet werden.

removeDependent: *anObjekt*
Eine Abhängigkeit wird, falls vorhanden, aufgelöst.

changed: *aParameter*
Führt ein Objekt Anweisungen aus, von denen seine Abhängigen unterrichtet werden sollen, so sendet dieses Objekt diese Nachricht mit *aParameter* als Wert der Änderung.

update: *aParameter*
Anschließend werden die Abhängigen durch die Nachricht **update:** über die Veränderung unterrichtet und ihre Reaktion darauf festgelegt. Der Parameter *aParameter* stimmt mit dem von **changed:** überein. Diese Instanzmethode in der Klasse **Object** besteht darin, nichts zu tun und muss für den konkreten Fall aktualisiert werden.

Ein Leittier soll eine Herde von Tieren über seine Ortsveränderungen informieren. die Herde soll mit dem Leittier mitziehen. Dazu ist zunächst ein Tier als Leittier auszuzeichnen, indem man ihm mehrere Tiere unterordnet.

```
Animal publicMethods
```

```
beLeaderOf: aCollectionOfAnimals
    "Der Empfaenger wird Leittier der Herde aCollectionOfAnimals.
    Diese werden als seine Abhaengigen erklaert."

    aCollectionOfAnimals do: [ :animal | self addDependent: animal ]
```

Ein Leittier hat zwei abhängige Tiere in seiner Herde:

```
| Leittier Tier1 Tier2 Herde |
Leittier := Animal new.
Tier1 := Animal new.
Tier2 := Animal new.
Herde := OrderedCollection new.
Herde add: Tier1; add: Tier2.
Leittier beLeaderOf: Herde.
^Leittier dependents          ⇒          OrderedCollection(an Animal an Animal )
```

Das Leittier sendet bei einer Standortänderung lediglich eine **changed:**-Nachricht, woraufhin alle Mitglieder der Herde mittels einer dadurch ausgelösten **update:**-Nachricht über die Standortveränderung unterrichtet werden. Dazu sind nun die folgenden Instanzmethoden notwendig:

```
Animal publicMethods

moveAll: aPoint
    "Der Empfaenger veraendert seinen Standort. Alle abhaengigen
    Objekte werden ueber diese Veraenderung unterrichtet."
    | oldPoint |
    oldPoint := self location. self location: aPoint.
    self changed: aPoint-oldPoint

update: aParameter
    "Bei einer Standortveraenderung enthaelt aParameter die
    Standortaenderung des Senders."
    self moveAll: self location + aParameter
```

Das Leittier ändert seinen Standort und die Herde zieht mit:

```
| Leittier Tier1 Tier2 Herde |
Leittier := Animal new. Leittier location: 4 @ 4.
Tier1 := Animal new. Tier1 location: 1 @ 2.
Tier2 := Animal new. Tier2 location: 5 @ 3.
Herde := OrderedCollection new.
Herde add: Tier1; add: Tier2.
Leittier beLeaderOf: Herde.

Leittier moveAll: 3 @ 5.
^Array with: Leittier location with: Tier1 location with: Tier2 location
                                                  ⇒       (3 @ 5 0 @ 3 4 @ 4)
```

■ **3. Aufgabe**