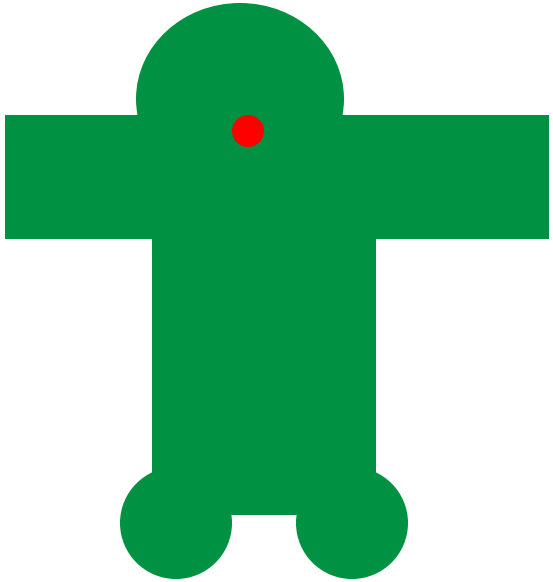
PM1/PT Ruby: Objektrekursion

Aufgabenstellung

- Eine Figur ist entweder ein Kreis, ein Rechteck oder eine Überlagerung von zwei Figuren.

- Aufgabe: Schreiben Sie ein Programm, das für Punkte im kartesischen Koordinaten-system berechnet, ob der Punkt innerhalb einer Figur liegt.



Figur ist eine abstrakte Klasse

- ***Kreis***, ***Rechteck*** und ***Ueberlagert*** sind Figuren, von denen Instanzen erzeugt werden können.

- Eine ***Figur*** ist entweder ***Kreis***, ***Rechteck*** oder ***Ueberlagert***. Von Figur lassen sich keine sinnvollen Instanzen erzeugen ➔ ***Figur*** ist eine abstrakte Klasse.

- Jede Figur soll beantworten können, ob ein Punkt in ihr liegt. Für Figuren können wir keine Vorschrift (Implementierung) für die Methode ***in?(ein_punkt)*** angeben ➔ ***in?*** ist eine abstrakte Methode.

```
class Object
  def abstract()
    raise AbstractMethodError,
    "Subklassen muessen abstrakte Methoden
    implementieren"
  end
end

class AbstractMethodError < StandardError
end

module Figur
  def in?(ein_punkt)
    abstract
  end
end
```

Lösung für Kreise

- Kreise werden mit Mittelpunkt und Radius erzeugt.
- Kreise sind Figuren, daher müssen sie die Methode ***in?*** implementieren und ***Figur*** inkludieren.
- Ein Punkt liegt in einem Kreis, wenn der Abstand des Punktes zum Mittelpunkt kleiner dem Radius ist.
- Die Berechnung des Abstands zwischen zwei Punkten delegieren wir an die Klasse Punkt (Prinzip der Kohäsion).

```
require "Figur"

class Kreis
  include Figur
  def initialize(mittelpunkt, radius)
    @mittelpunkt = mittelpunkt
    @radius = radius
  end

 # in?(Punkt) -> Boolean
  def in?(punkt)
    return @mittelpunkt.abstand(punkt) <
    @radius
  end
end
```

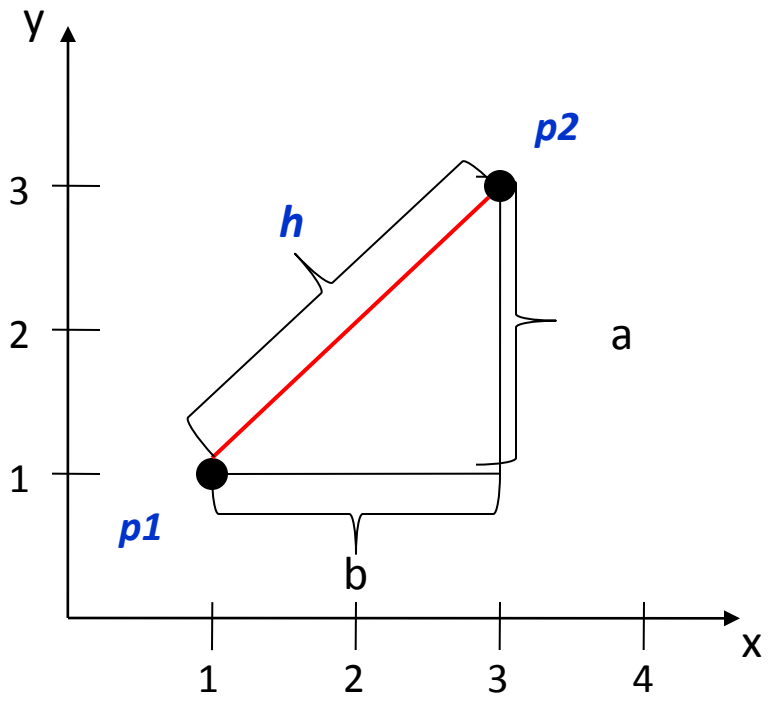
Die Klasse Punkt

- Ein Punkt in einem kartesischen Koordinatensystem ist bestimmt durch x und y Koordinaten.
- Ein Punkt kann den Abstand zu einem anderen Punkt bestimmen.

```
class Punkt
  def initialize(x,y)
    @x = x
    @y = y
  end
  def x
    return @x
  end
  def y
    return @y
  end
  def abstand(ein_punkt)
    a = ein_punkt.y - @y
    b = ein_punkt.x - @x
    return Math.sqrt(a**2 + b**2)
  end
end
```

Methode *abstand* in der Klasse *Punkt*

- Domänenwissen: Abstand ***h*** zwischen zwei Punkten.***p1*** und ***p2***



Satz von Pythagoras:

$$h^2 = a^2 + b^2$$

$$h = \sqrt{a^2 + b^2}$$

$$a = p2.y - p1.y$$

$$b = p2.x - p1.x$$

Implementieren der Methode *abstand* in der Klasse *Punkt*

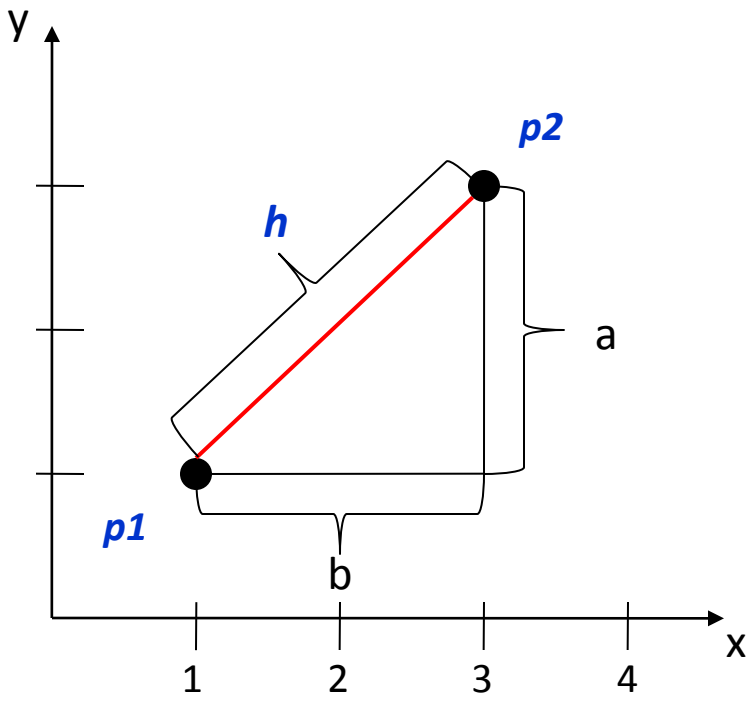
- Satz von Pythagoras:

$$h^2 = a^2 + b^2$$

$$h = \sqrt{a^2 + b^2}$$

$$a = p2.y - p1.y$$

$$b = p2.x - p1.x$$

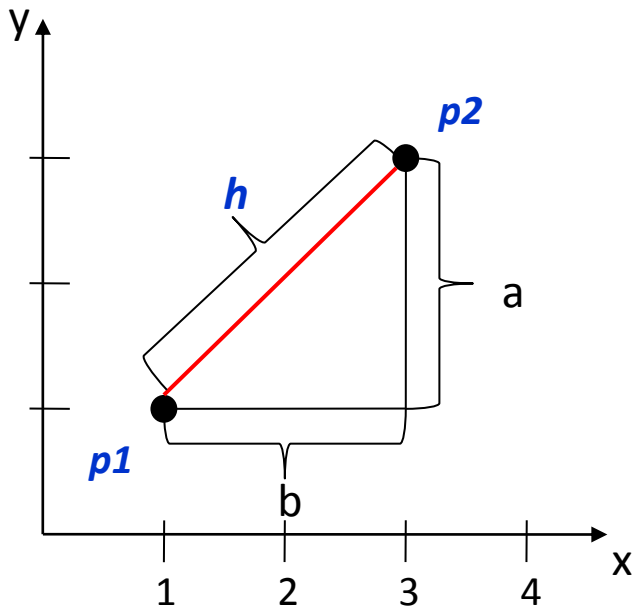


- Wir übersetzen den Satz nun in die Methode ***abstand*** der Klasse ***Punkt.***
- Berechne die Seitenlängen ***a*** und ***b*** aus den Punkten ***p1*** und ***p2***.
- Berechne die Länge der Hypotenuse ***h*** und gebe das Ergebnis zurück

Implementieren der Methode *abstand* in der Klasse *Punkt*

- Berechne die Seitenlängen ***a*** und ***b*** aus den Punkten ***p1*** und ***p2***.
- Berechne die Länge der Hypotenuse ***h*** und gebe das Ergebnis zurück
- ***p1*** ist der Punkt, an den die Methode Abstand geschickt wurde.
- ***p2*** ist der Parameter, der beim Methodenaufruf übergeben wurde.

```
class Punkt
  def abstand(ein_punkt)
    a = ein_punkt.y - @y
    b = ein_punkt.x - @x
    return Math.sqrt(a**2 + b**2)
  end
end
```



Lösung für Rechtecke

- Ein Rechteck wird durch Angabe der linken oberen Ecke und der Breite und Höhe erzeugt.
- Rechtecke sind Figuren, daher müssen sie die Methode ***in?*** implementieren und ***Figur*** inkludieren.
- Ein Punkt liegt in einem Rechteck, wenn die x-Koordinate im Intervall
  (@lo.x..@lo.x+breite)
  und die y-Koordinate im Intervall
  (@lo.y-@hoehe .. @lo.y) liegt

```
class Rechteck
  include Figur

  def initialize(links_oben, breite,
    hoehe)
   @lo = links_oben
   @breite = breite
   @hoehe = hoehe
  end

  def in?(ein_punkt)
    ix = (@lo.x .. @lo.x+@breite)
    iy = (@lo.y-@hoehe .. @lo.y)
    return ix.include?(ein_punkt.x) &&
      iy.include?(ein_punkt.y)
  end
end
```

Lösung für Überlagert

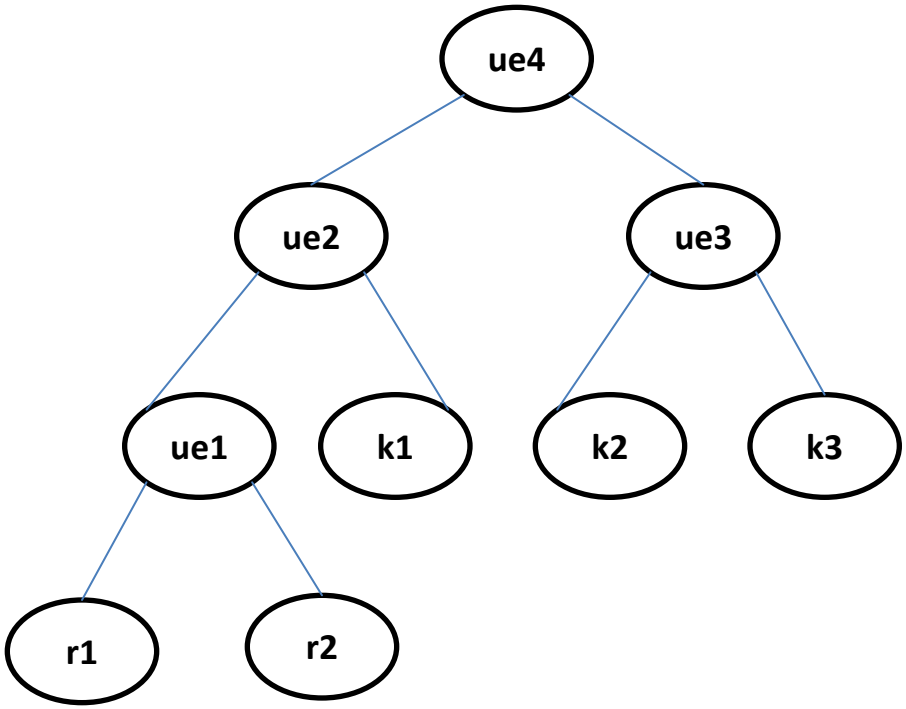
- Eine überlagerte Figur entsteht durch Überlagern einer unteren Figur durch eine obere Figur.
- Die obere und untere Figur sind Figuren, können daher auch wieder überlagerte Figuren sein.
- ***Ueberlagert*** ist eine Figur und muss die Methode in? implementieren.
- Ein Punkt liegt in einer überlagerten Figur, wenn er in der oberen Figur oder der unteren Figur liegt.
- Da wir die Lösung der Frage ***in?*** durch einen **Aufruf der gleichen Methode** auf Objekten eines gemeinsamen Typs (Figur) zurückführen, sprechen wir auch von **Objektrekursion**.

```
class Ueberlagert
  include Figur

  def initialize(oben, unten)
    @obere_figur = oben
    @untere_figur = unten
  end

  # in?(Punkt) -> Boolean
  def in?(ein_punkt)
    return
  @obere_figur.in?(ein_punkt) ||
     @untere_figur.in?(ein_punkt)
  end
end
```

Ueberlagert ist eine rekursive Objektstruktur

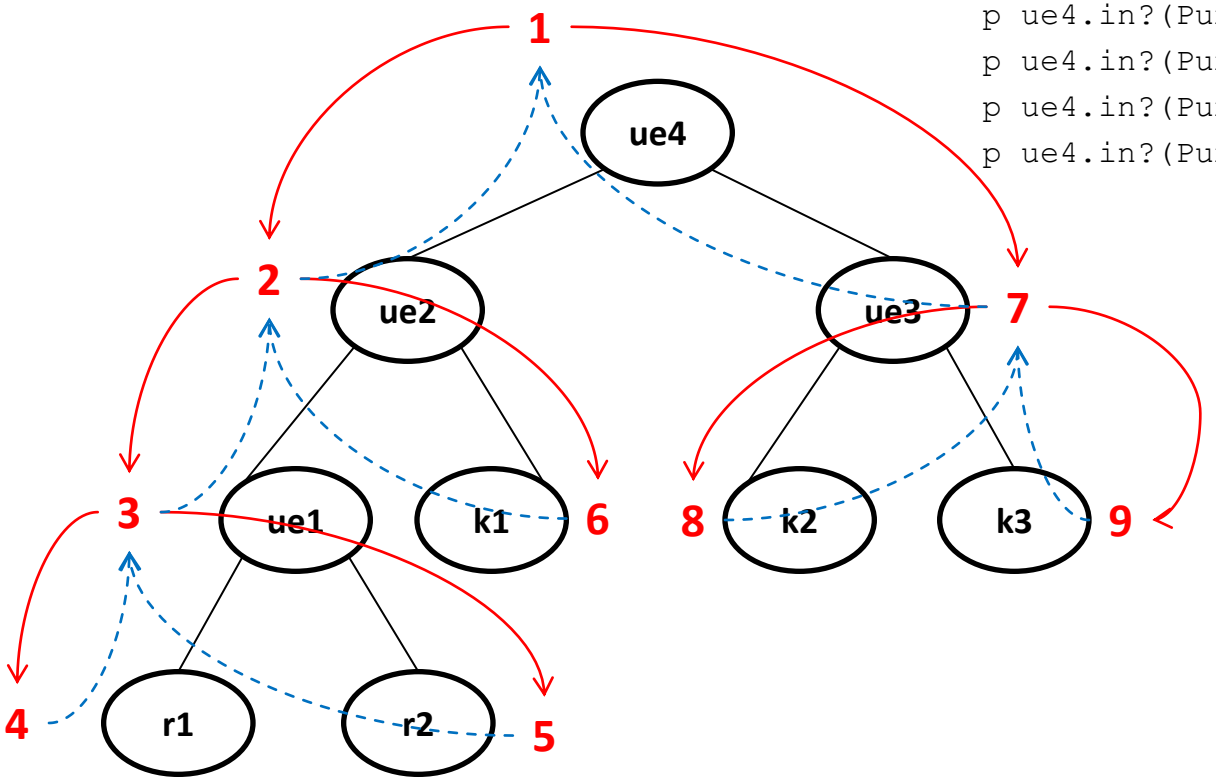
- ***Ueberlagert*** ist eine rekursive Struktur, da ***@obere_figur*** und ***@untere_figur*** Referenzen auf ***Figur*** enthalten ist.
- ***@obere_figur*** und ***@untere_figur*** kann selbst wieder ein ***Ueberlagert*** Objekt sein, das dann wiederum Referenzen auf ***Ueberlagert*** Objekte enthalten kann, usw.
- Da ***Ueberlagert*** selbst eine ***Figur*** ist, enthält die Definition eine Selbstreferenz.
- Definitionen mit Selbstreferenzen definieren immer rekursive Strukturen.

Rekursive Objektstrukturen erzeugen Bäume, deren Knoten den gleichen Typ haben



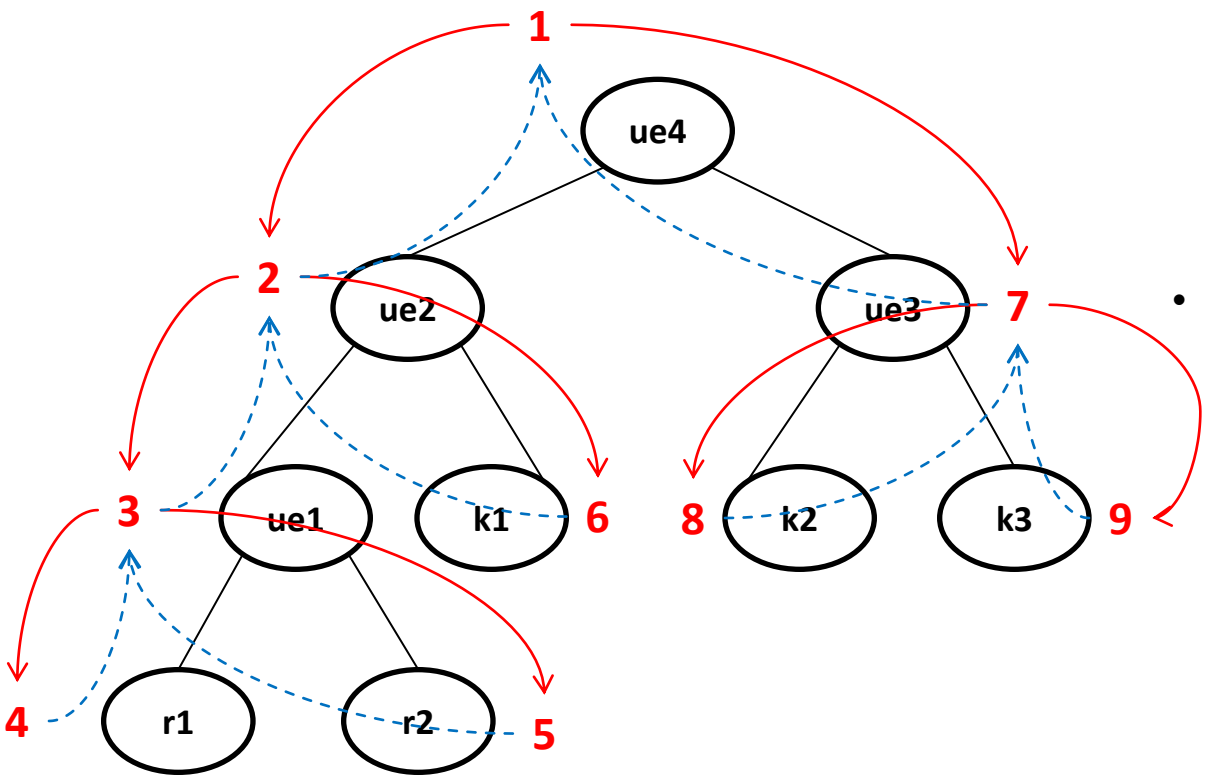

```
k1 = Kreis.new(Punkt.new(2,1),0.5)
k2 = Kreis.new(Punkt.new(3,1),0.5)
k3 = Kreis.new(Punkt.new(2.2,4.5),1)
r1 = Rechteck.new(Punkt.new(2,3.5),1,2.5)
r2 = Rechteck.new(Punkt.new(1,4),3,1)

ue1 = Ueberlagert.new(r1,r2)
ue2 = Ueberlagert.new(ue1,k1)
ue3 = Ueberlagert.new(k2,k3)
ue4 = Ueberlagert.new(ue2,ue3)
```

Methoden in rekursive Objektstrukturen traversieren den Baum und wenden die Methode auf jeden Knoten an



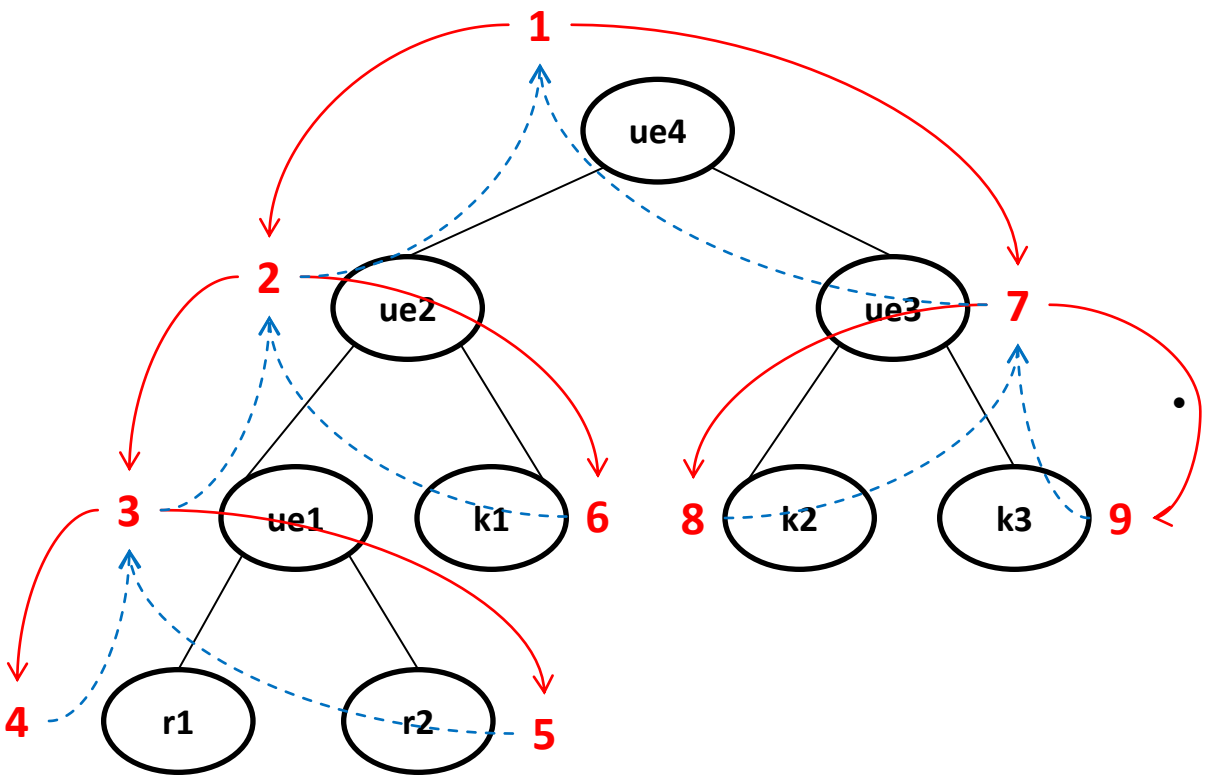

```
p ue4.in?(Punkt.new(2,1))
p ue4.in?(Punkt.new(3.5,3.5))
p ue4.in?(Punkt.new(2,3.5))
p ue4.in?(Punkt.new(0,0))
```

Methoden in rekursive Objektstrukturen traversieren den Baum und wenden die Methode auf jeden Knoten an



- Die roten Pfeile zeigen den rekursiven Aufruf der Methode. Man bezeichnet den Aufruf einer Methode auch als rekursiven Abstieg.
- Der Methode wird solange rekursiv aufgerufen, wie rekursiv definierte innere Knoten im Baum vorhanden sind.

Methoden in rekursive Objektstrukturen traversieren den Baum und wenden die Methode auf jeden Knoten an



- Werden die Blätter erreicht (Kreise, Rechtecke), dann endet der rekursive Abstieg. Die Methoden liefern jetzt die Ergebnisse beim **rekursiven Aufstieg,** gekennzeichnet durch die gestrichelten blauen Pfeile.

- Jede Rekursion benötigt eine **Terminierungsbedingung**. In rekursiven Objektstrukturen sind die Blätter die Terminierungsbedingung, da sie keinen rekursiven Aufruf der Methode ***in?*** enthalten.

Übungen

- **Ü-15-b-0:** Debuggen Sie den Aufruf der Methode ***in?*** auf dem Objekt ***ue4*** und versuchen Sie den rekursiven Abstieg / wie Aufstieg anhand des Callstacks nach zu vollziehen.

- **Ü-15-b-1**: Schreiben Sie eine Methode ***anzahl?(typ)***, die in einer Figur die Anzahl der Objekte von einem beliebigen Typ T zählt. Welche Klassen des Figurenbeispiel benötigen alle eine solche Methode? Begründen Sie Ihre Antwort.

- **Ü-15-b-2**: Muss ***anzahl?(typ)*** eine abstrakte Methode sein oder gibt es eine Alternative? Erläutern Sie Ihre Antwort. Welche Methodenkategorie haben Sie, wenn Sie keine abstrakte Methode verwenden?

- **Ü-15-b-3**: Schreiben Sie eine Methode ***max_figuren_typ(),*** die für eine Figur die Klasse bestimmt, für die die Anzahl der Instanzen in der Figur maximal ist.

  Schreiben Sie ***max_figuren_typ()*** als Templatemethode, die eine Methode ***sammle_typen(typen_hash)*** verwendet.

  Der ***typen_hash*** verwaltet zu jeder Klasse die Anzahl der gezählten Instanzen und wird in der Methode ***sammle_typen*** modifiziert (analog der ***agiere*** Methode im Simulationsbeispiel).

Übungen

- **Ü-15-b-4**: Schreiben Sie einen Iterator (Methode ***each***) für Figuren, der die Elemente in Figur auflistet. Wie verhält sich die Methode für einen Kreis oder ein Rechteck?

- **Ü-15-b-5**: Reimplementieren Sie **Ü-15-b-1** und **Ü-15-b-3** mit geeigneten Methoden von ***Enumerable***.

- **Ü-15-b-6:** Schreiben Sie ein ***to_s*** für Figuren. Ist Ihre Definition von ***to_s*** eine rekursive Definition?

- **Ü-16-b-7**: Testen Sie Ihr ***to_s*** mit der folgenden Definition einer überlagerten Figur.

```
ue5 = Ueberlagert.new(k1,k2)
ue6 = Ueberlagert.new(ue5,r2)
ue7 = Ueberlagert.new(ue5,ue6)
```

Erstellen Sie ein Skizze von der Figur als Baumrepräsentation. Terminiert die Methode ***to_s***?

Übungen

- **Ü-16-b-8:** Schreiben Sie eine Methode ***finde_figur(alte_figur)***, die in einer Figur die Figur findet, die zu einer Figur identisch ist. Verwenden Sie dazu eine Methode von Enumerable.

- **Ü-16-b-9:** Schreiben Sie eine Methode ***finde_gleiche_figur(alte_figur)***, die in einer Figur die Figur findet, die zu einer Figur gleich ist. Welche Methode müssen Sie zuvor für Figuren implementieren.

- **Ü-16-b-10:** Schreiben Sie eine Methode ***figur_tauschen(alte_figur,neue_figur)***, die in einer Figur die vorhandene ***alte_figur*** gegen die ***neue_figur*** tauscht. Es ist nicht möglich die Figur selbst zu tauschen, sondern nur enthaltene Figuren.

  Es ist nicht erlaubt Reader und Writer für enthaltene Figuren zu verwenden.

  Das Enthaltensein soll mit Identität geprüft werden.

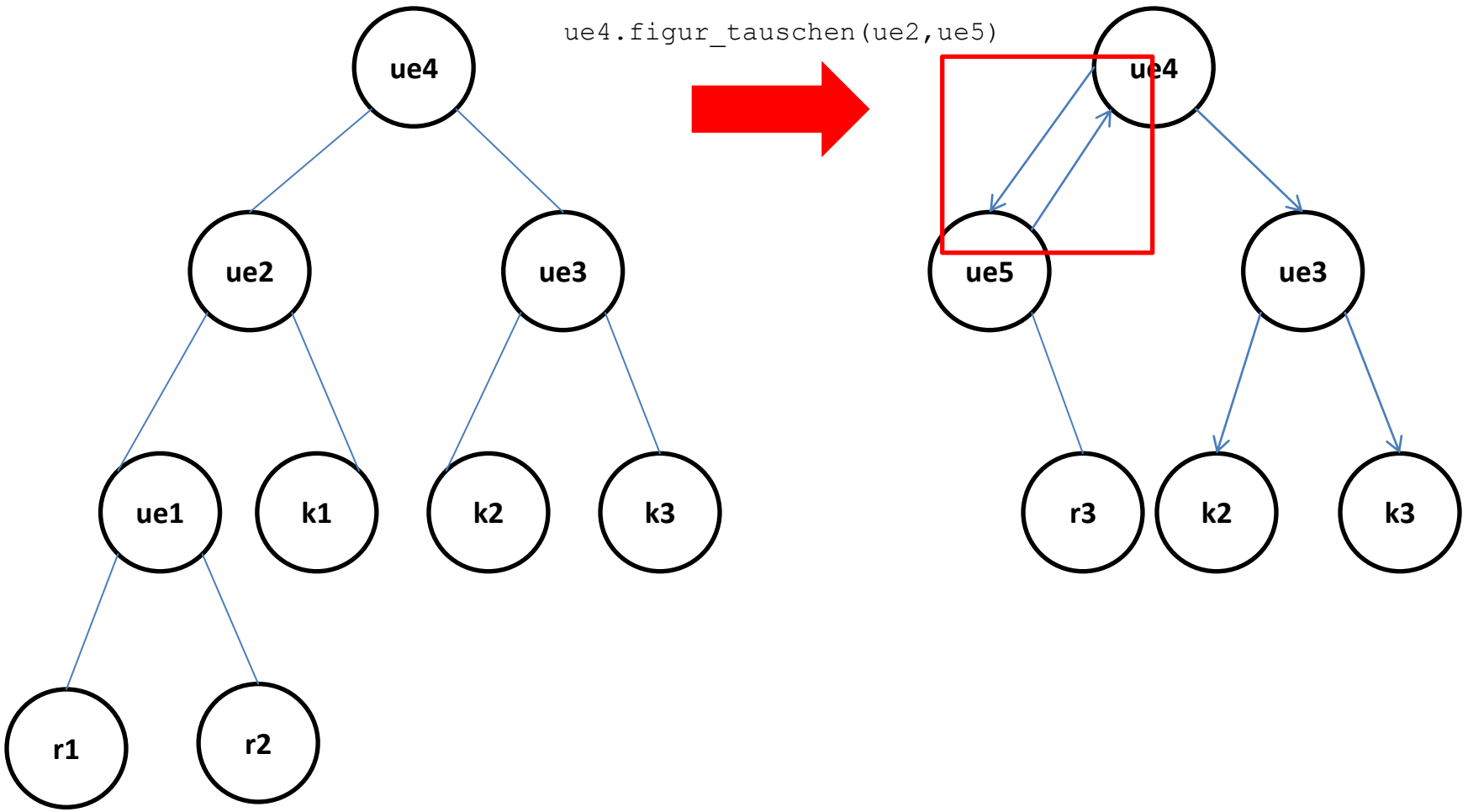
Übungen

- **Ü-16-b-11:** Testen Sie Ihre Implementierung aus **Ü-16-b-10** mit den folgenden Figuren.

```
k1 = Kreis.new(Punkt.new(2,1),0.5)
k2 = Kreis.new(Punkt.new(3,1),0.5)
k3 = Kreis.new(Punkt.new(2.2,4.5),1)
r1 = Rechteck.new(Punkt.new(2,3.5),1,2.5)
r2 = Rechteck.new(Punkt.new(1,4),3,1)
r3 = Rechteck.new(Punkt.new(3,5),2,1)

ue1 = Ueberlagert.new(r1,r2)
ue2 = Ueberlagert.new(ue1,k1)
ue3 = Ueberlagert.new(k2,k3)
ue4 = Ueberlagert.new(ue2,ue3)
ue5 = Ueberlagert.new(ue4,r3)

ue4.figur_tauschen(ue2,ue5)
```

- **Ü-16-b-12:** Geben Sie das Ergebnis mit Ihrer Implementierung von ***to_s*** auf der Konsole aus. Was stellen Sie fest? Veranschaulichen Sie für die Begründung das Ergebnis des Tausches mit Hilfe der Baumnotation.

Das Ergebnis des Tausches



Übungen

- **Ü-15-b-12**: **Ü-15-b-10 11** hat gezeigt, dass das Tauschen Zyklen erzeugt, wenn die zu tauschende Figur Rückreferenzen auf enthaltene Figuren enthält.

  Wir müssen daher vor dem Tausch prüfen, ob wir mit dem Einfügen einer neuen Figur eine zyklische Struktur erzeugen.

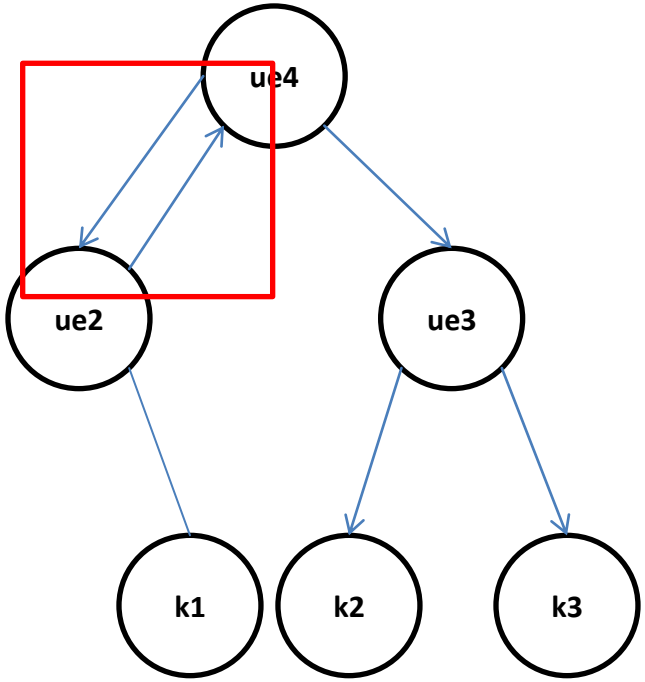
  Wir erzeugen immer dann eine zyklische Struktur, wenn die einzufügende Figur, Figuren enthält, die auf eine Figur zeigen, die Vorgänger der neuen Figur ist.

- Die Lösung geht in drei Schritten:
  1. Wir schreiben eine Methode, die alle Vorgänger zu der enthaltenen Figur ***alte_figur*** berechnet.
  2. Wir schreiben eine Methode, die alle Knoten der einzufügenden Figur ***neue_figur*** berechnet.
  3. Wir prüfen, ob einer der Knoten der einzufügenden Figur in der Liste der Vorgänger enthalten ist. Wenn ja, weisen wir den Tausch ab, indem wir einen ***CyclicInsertionError*** erzeugen. Sonst tauschen wir die Figur und geben eine Referenz auf ***self*** zurück.

Übungen

- **Ü-15-b-13**: Testen Sie Ihre verbesserte Implementierung mit folgenden Figuren.

```
k1 = Kreis.new(Punkt.new(2,1),0.5)
k2 = Kreis.new(Punkt.new(3,1),0.5)
k3 = Kreis.new(Punkt.new(2.2,4.5),1)
r1 = Rechteck.new(Punkt.new(2,3.5),1,2.5)
r2 = Rechteck.new(Punkt.new(1,4),3,1)
r3 = Rechteck.new(Punkt.new(3,5),2,1)

ue1 = Ueberlagert.new(r1,r2)
ue2 = Ueberlagert.new(ue1,k1)
ue3 = Ueberlagert.new(k2,k3)
ue4 = Ueberlagert.new(ue2,ue3)

ue2.figur_tauschen(ue1,ue4)
```



Übungen

- **Ü-15-b-13**: Testen Sie Ihre verbesserte Implementierung mit folgenden Figuren.

```
k1 = Kreis.new(Punkt.new(2,1),0.5)
k2 = Kreis.new(Punkt.new(3,1),0.5)
k3 = Kreis.new(Punkt.new(2.2,4.5),1)
r1 = Rechteck.new(Punkt.new(2,3.5),1,2.5)
r2 = Rechteck.new(Punkt.new(1,4),3,1)
r3 = Rechteck.new(Punkt.new(3,5),2,1)

ue1 = Ueberlagert.new(r1,r2)
ue2 = Ueberlagert.new(ue1,k1)
ue3 = Ueberlagert.new(k2,k3)
ue4 = Ueberlagert.new(ue2,ue3)

ue2.figur_tauschen(ue1,ue4)
```

- Das **Problem**: Wir erzeugen einen Zyklus durch Tauschen in einer enthaltenen Figur ***ue2***. ***ue4*** ist in der übergeordneten Figur Vorgänger von ***ue1***, aber ***ue2*** weiß nichts über die übergeordnete Figur.

- Um dieses Problem zu lösen, müssen wir für jede Figur eine **Referenz auf die übergeordnete Figur** einführen, die die übergeordnete Figur bei der Erzeugung setzt.

- Dann schreiben wir eine Methode ***top***, die die alleroberste Figur des Baumes liefert und führen das Tauschen auf ***top*** durch.

- Beim Tausch müssen wir daran denken, dass wir die Referenz von ***alte_figur*** auf die übergeordnete Figur löschen (auf ***nil*** setzen)

Übungen

- **Ü-15-b-14**: Überlegen Sie ob mit der letzten Verbesserung die Möglichkeit des Erzeugens von Zyklen für alle Fälle unterbunden werden kann?

- **Ü-15-b-15**: Wie könnte eine Lösung aussehen, die das Erzeugen von Zyklen für alle Fälle unterbindet.