PM1/PT - Vererbung

Konzepte und Techniken

- Vererbung
- Ersetzbarkeit
- Subtyping
- Polymorphie
- Überschreiben von Methoden
- dynamische Methodensuche

Beispiel DoME (Database Of Multimedia Entertainment)

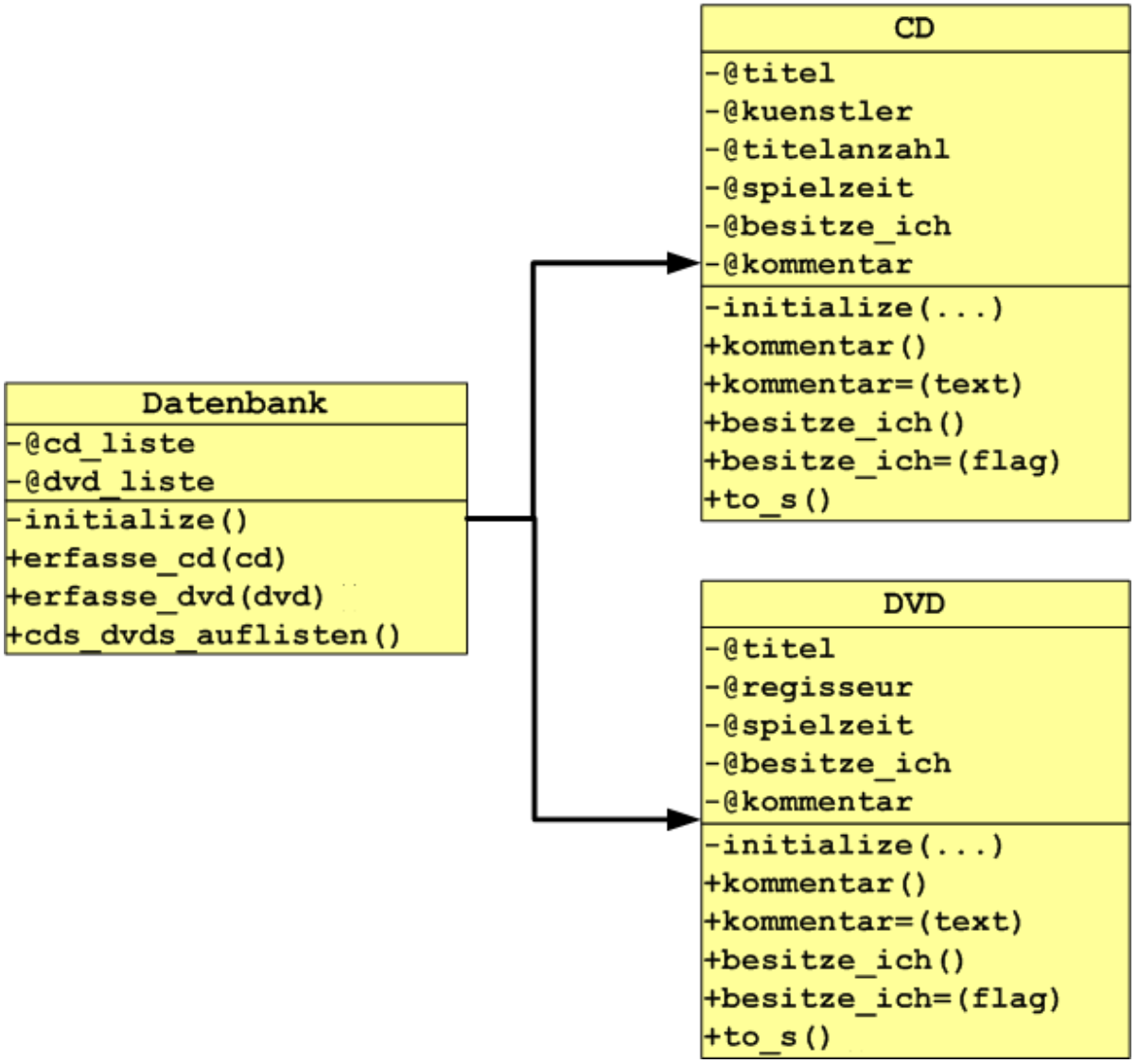
- Eine Anwendung, in der Informationen über CD‘s und DVD‘s gespeichert werden.

- Funktionen:
  - Informationen über CDs und DVDs erfassen
  - Informationen permanent speichern, um sie später weiter zu verwenden
  - Suche nach Künstlern, Regisseuren (Vereinfachung hier nur Film DVDs)
  - Ausgabe von Listen aller CDs oder DVDs in der Datenbank
  - Löschen von CDs und DVDs aus der Datenbank

- Details von CDs
  - Titel des Albums
  - Künstler (Name der Band oder des Sängers)
  - Anzahl der Titel auf dem Album
  - Gesamtspielzeit
  - Markierung besitze_ich
  - Kommentar

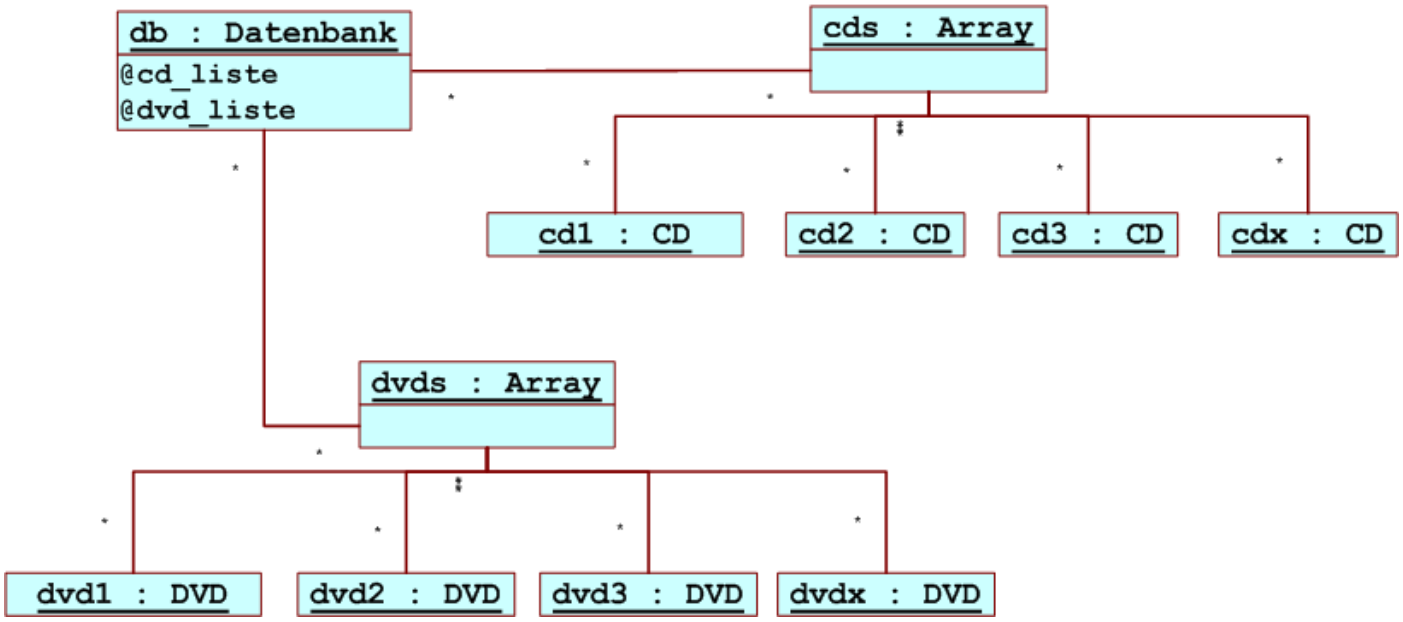
- Details von DVDs
  - Titel der DVD
  - Name des Regisseurs
  - Gesamtspielzeit (Spielzeit des Hauptfilms)
  - Markierung besitze_ich
  - Kommentar

Klassen in DoME



Objekte in DoME

- DoME hat ein Datenbankobjekt, das beliebig viele CDs und DVDs enthalten kann.
- Diese sind innerhalb der Datenbank als Sammlung abgelegt.



Quelltext von Datenbank

```ruby
# Diese Version speichert noch keine
# Daten
# und hat noch keine Suchfunktion

class Datenbank
  def initialize()
    @cd_liste = []
    @dvd_liste = []
  end
  def erfasse_cd(cd)
    @cd_liste << cd
  end
  def erfasse_dvd(dvd)
    @dvd_liste << dvd
  end
  def cds_dvds_auflisten()
    @cd_liste.each {|cd| puts cd }
    puts
    @dvd_liste.each{ |dvd| puts dvd }
    puts
  end
end
```

- Öffnen Sie das ***Projekt v12-DoME1***

Quelltext von CD und DVD

```ruby
class CD
  attr_reader :kommentar,:besitze_ich
  attr_writer :kommentar,:besitze_ich
  # titel,kuenstler,kommentar: String
  # anzahl: Integer
  # spielzeit: Integer
  def initialize(titel,kuenstler,anzahl,
                 spielzeit,besitz,kommentar)
    @titel = titel
    @kuenstler = kuenstler
    @titelanzahl = anzahl
    @spielzeit = spielzeit
    @besitze_ich = besitz
    @kommentar = kommentar
  end
  def to_s()
    "CD: #@titel (#@spielzeit Min)
    #{besitze_ich ? '*' :''}
          #@kuenstler
          Titelanzahl: #@titelanzahl
          #@kommentar"
  end
end
```

```ruby
class DVD
  attr_reader :kommentar,:besitze_ich
  attr_writer :kommentar,:besitze_ich
  # titel, regisseur,kommentar: Strings
  # spielzeit: Integer
  def
    initialize(titel,regisseur,spielzeit,b
    esitz,kommentar)
    @titel = titel
    @regisseur = regisseur
    @spielzeit = spielzeit
    @besitze_ich = besitz
    @kommentar = kommentar
  end

  def to_s()
    "DVD: #@titel (#@spielzeit Min)
    #{besitze_ich ? '*' :''}
          #@regisseur
          #@kommentar"
  end
end
```

Probleme mit der jetzigen Implementierung

Code-Duplizierung

- CD und DVD sind bis auf einige Attribute nahezu identisch. In beiden Klassen haben wir die gleichen Reader und Writer.
- Die Klasse Datenbank macht für CDs und DVDs alles doppelt. Zwei Listenvariablen, zwei Methoden zum Erfassen, zweimal nahezu identischer Quelltext für die Ausgabe.
- Führen wir in die Datenbank neue Medien ein (Bücher, Photos, Videoclips), dann müssten wir in der Datenbank für jedes Medium die erwähnten Methoden vervielfachen.

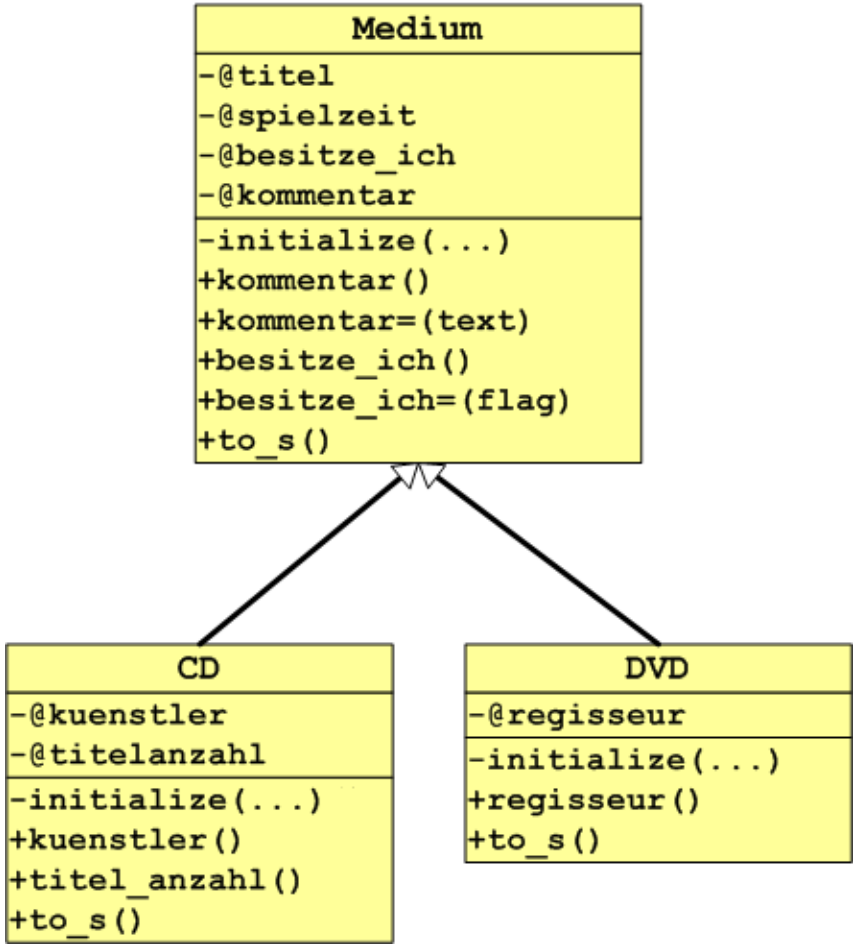
Lösungsansatz

- Die Gemeinsamkeiten der Medien (Klasse CD und DVD) in einer Klasse zusammenfassen und diese Gemeinsamkeiten auf die Klassen CD und DVD zu vererben.

➔ ***v12-DoME2***

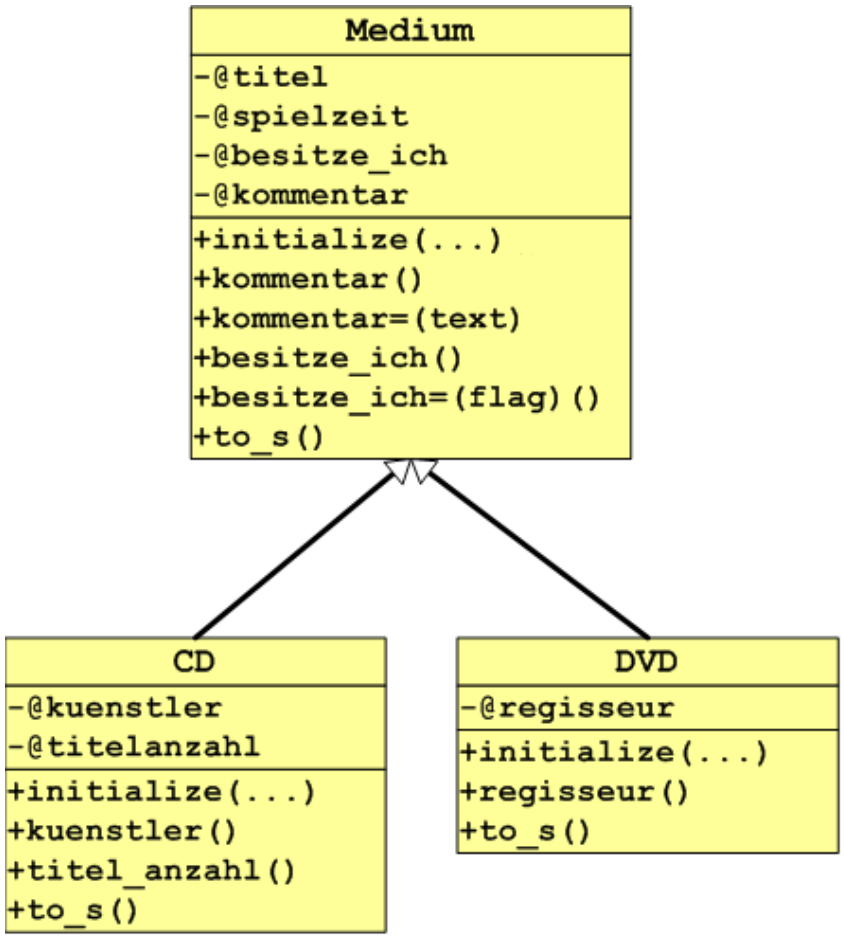
Vererbung

- Vererbung erlaubt es uns, eine Klasse als Erweiterung einer anderen zu definieren.
- Die Gemeinsamkeiten von ***CD*** und ***DVD*** werden in der Klasse ***Medium*** zusammengefasst. Dann definieren wir, dass ***CD*** ein ***Medium*** ist und ***DVD*** ein ***Medium*** ist.
- In den Klassen ***CD*** und ***DVD*** haben wir nur noch die spezifischen, die beiden Klassen unterscheidenden Merkmale.
- Nach dieser Restrukturierung ergibt sich das nebenstehende Klassendiagramm.



Vererbung

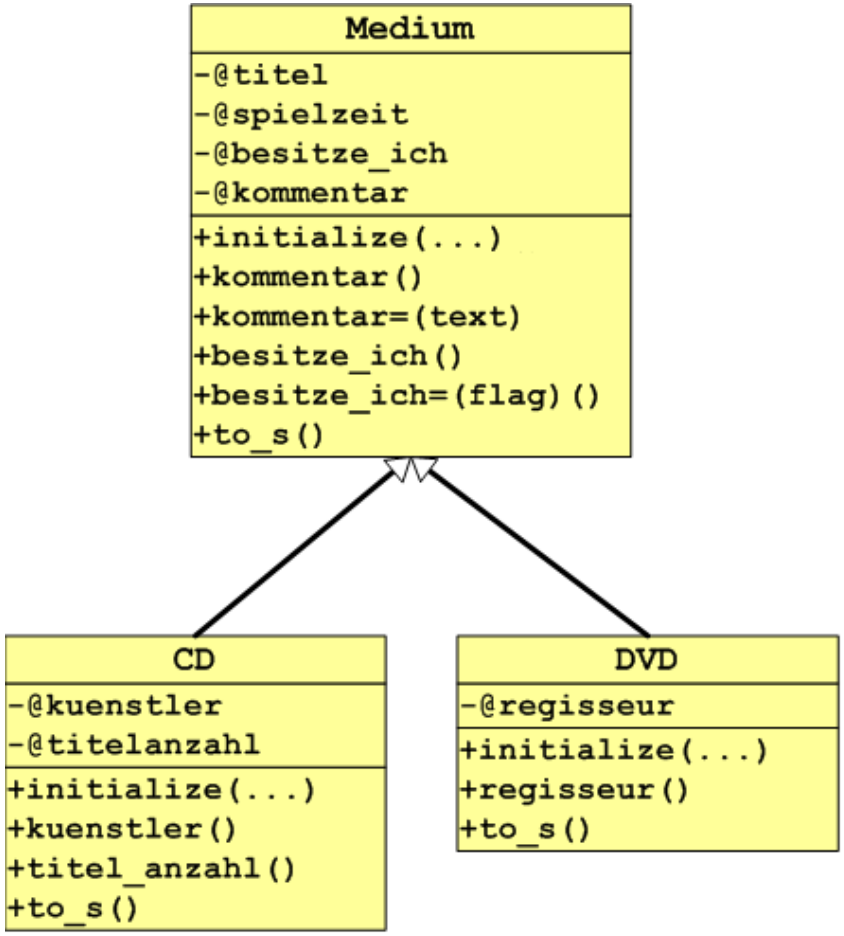
- Die Klasse ***Medium*** fasst die Gemeinsamkeiten der Klassen ***CD*** und ***DVD*** zusammen.
- Die Klassen ***CD*** und ***DVD*** **erben** von der Klasse Medium alle Instanzvariablen und Objektmethoden.
- Die Klassen ***CD*** und ***DVD*** **erweitern** die Klasse ***Medium*** um eigene Instanz-variablen und –methoden.
- Die Vererbungsbeziehung wird auch als eine **ist-ein** Beziehung bezeichnet, da eine Subklasse als Spezialisierung der Super-klasse angesehen werden kann.



Superklasse - Subklasse

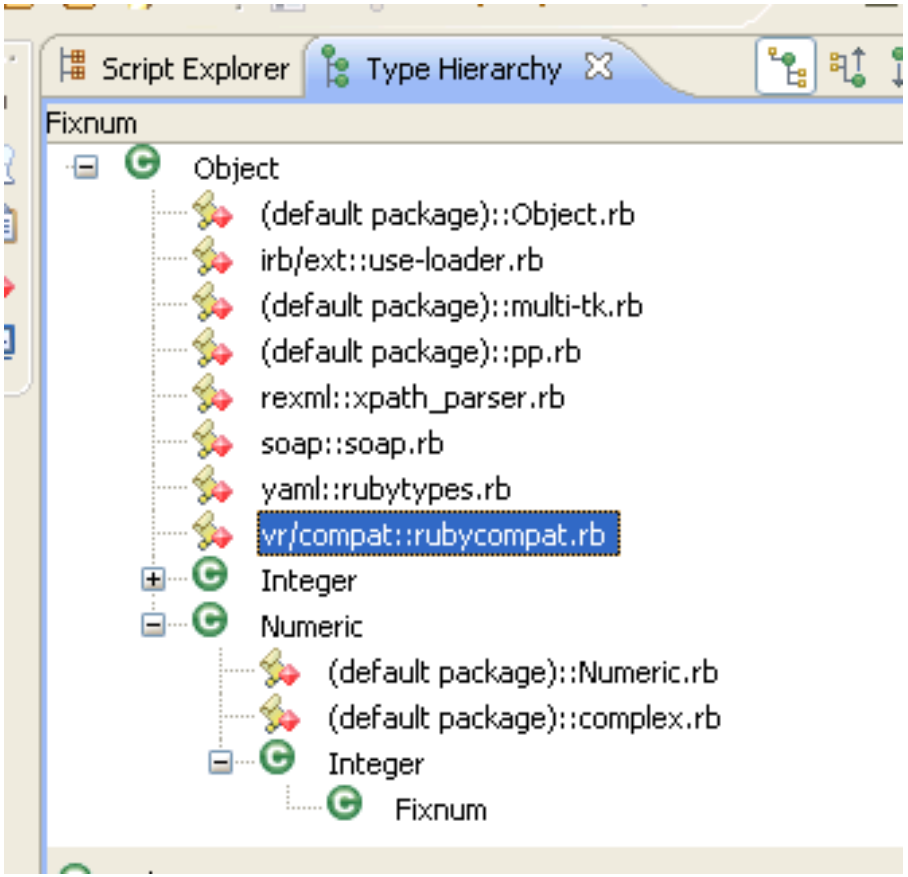
- Die Klasse ***Medium*** ist die **Superklasse** von ***CD*** und ***DVD***. Eine Superklasse ist eine Klasse, die von anderen Klassen erweitert wird.

- ***CD*** und ***DVD*** sind **Subklassen** der Klasse ***Medium.*** Ein Subklasse ist ein Klasse, die eine andere Klasse erweitert bzw. von ihr erbt. Sie erbt alle Instanzvariablen und Objektmethoden.



Vererbungshierarchien

- Klassen, die über Subklassenbeziehungen miteinander verknüpft sind, bilden eine Vererbungshierarchie.
- Ein Beispiel für eine Vererbungshierarchie ist die Ruby Klassenhierarchie. Diese lässt sich in Eclipse für jede Klasse in der Ansicht ***Type Hierarchy*** anzeigen.
- Ein Beispiel aus dem **Alltag** ist die Vererbungshierarchie, die Sie als Klassifikation der Tierwelt kennen.



Vererbungsbeziehungen in Ruby

- Eine Subklassen-Superklassenbeziehung wird durch das < Zeichen ausgedrückt.
- Die Klasse vor dem < Zeichen (***CD***) ist die Subklasse, die Klasse danach (***Medium***) die Superklasse.
- In ***CD*** initialisieren wir jetzt nur noch die Instanzvariablen ***@kuenstler*** und ***@titel-anzahl***. Die anderen Variablen werden in der Klasse ***Medium*** gespeichert und initialisiert. (Wie, das sehen wir gleich.)

```
class CD < Medium

  # titel, kuenstler,kommentar sind
    Strings
  # anzahl ein Integer
  # spielzeit ein Float
  def initialize(titel,kuenstler,anzahl
           ,spielzeit,besitz,kommentar)
    ...
    @kuenstler = kuenstler
    @titelanzahl = anzahl
  end
end
```

Vererbung und Zugriffsrechte

- Die Zugriffsmodifikatoren ***public*** und ***private*** regeln nicht nur die Zugriffsrechte für die Außenwelt, sondern auch entlang einer Vererbungshierarchie.

- Ein Objekt eine Subklasse hat (in Ruby) Zugriff auf die Instanzvariablen und die privaten Methoden der Superklasse.

- Wir könnten also das ***to_s*** und das ***initialize*** von ***CD*** unverändert lassen.

```
class CD < Medium
  # schlechte Lösung
  # Verletzung der Zuständigkeiten
  def
    initialize(titel,kuenstler,anzahl,spielze
    it,besitz,kommentar)
    @titel = titel
    @kuenstler = kuenstler
    @titelanzahl = anzahl
    @spielzeit = spielzeit
    @besitze_ich = besitz
    @kommentar = kommentar
  end
  # schlechte Lösung
  # Verletzung der Zuständigkeiten
  def to_s()
    "CD: #@titel (#@spielzeit Min)
    #{besitze_ich ? '*' :''}
          #@kuenstler
          Titelanzahl: #@titelanzahl
          #@kommentar"
  end
end
```

Vererbung und Zugriffsrechte

```
puts clapton = CD.new("Unplugged","Eric Clapton",13,160,true,"einfach cool")
puts nirvana = CD.new("Nevermind","Nirvana",12,120,true,"probates Antistress Mittel")
puts germain = CD.new("TOURIST","StGermain",9,80,true,"Chill out")
```

```
CD: Unplugged (160 Min) *
        Eric Clapton
        Titelanzahl: 13
        einfach cool
CD: Nevermind (120 Min) *
        Nirvana
        Titelanzahl: 12
        probates Antistress Mittel
CD: TOURIST (80 Min) *
        StGermain
        Titelanzahl: 9
        Chill out
```

Bewertung der ersten Lösung

- Wir könnten also das ***to_s*** und das ***initialize*** von ***CD*** unverändert lassen.
- Die ist eine **schlechte Lösung**, da sie eine Verletzung der Zuständigkeiten darstellt und wiederum zu Code-Duplizierung führt-
- Zuständig für Initialisierung der Instanz-variablen der Klasse ***Medium*** ist die Klasse ***Medium***.
- Zuständig für die Umwandlung der Instanz-Variablen von ***Medium*** in Zeichenketten ist ***Medium***.

```
class CD < Medium
  # schlechte Lösung
  # Verletzung der Zuständigkeiten
  def
    initialize(titel,kuenstler,anzahl,spielze
    it,besitz,kommentar)
    @titel = titel
    @kuenstler = kuenstler
    @titelanzahl = anzahl
    @spielzeit = spielzeit
    @besitze_ich = besitz
    @kommentar = kommentar
  end
  # schlechte Lösung
  # Verletzung der Zuständigkeiten
  def to_s()
    "CD: #@titel (#@spielzeit Min)
    #{besitze_ich ? '*' :''}
         #@kuenstler
         Titelanzahl: #@titelanzahl
         #@kommentar"
  end
end
```

Vererbung und Initialisierung

- In der guten Lösung übernimmt die Klasse Medium die Initialisierung ihrer Instanz-variablen.
- Die Frage ist nun, wie Subklassen das ***initialize*** der Superklasse „aktivieren“ (aufrufen) können?

```
class Medium
  attr_reader :kommentar,:besitze_ich
  attr_writer :kommentar,:besitze_ich

  def initialize(titel,spielzeit,besitz,
                                    kommentar)
    @titel = titel
    @spielzeit = spielzeit
    @besitze_ich = besitz
    @kommentar = kommentar
  end
end
```

Vererbung und Initialisierung

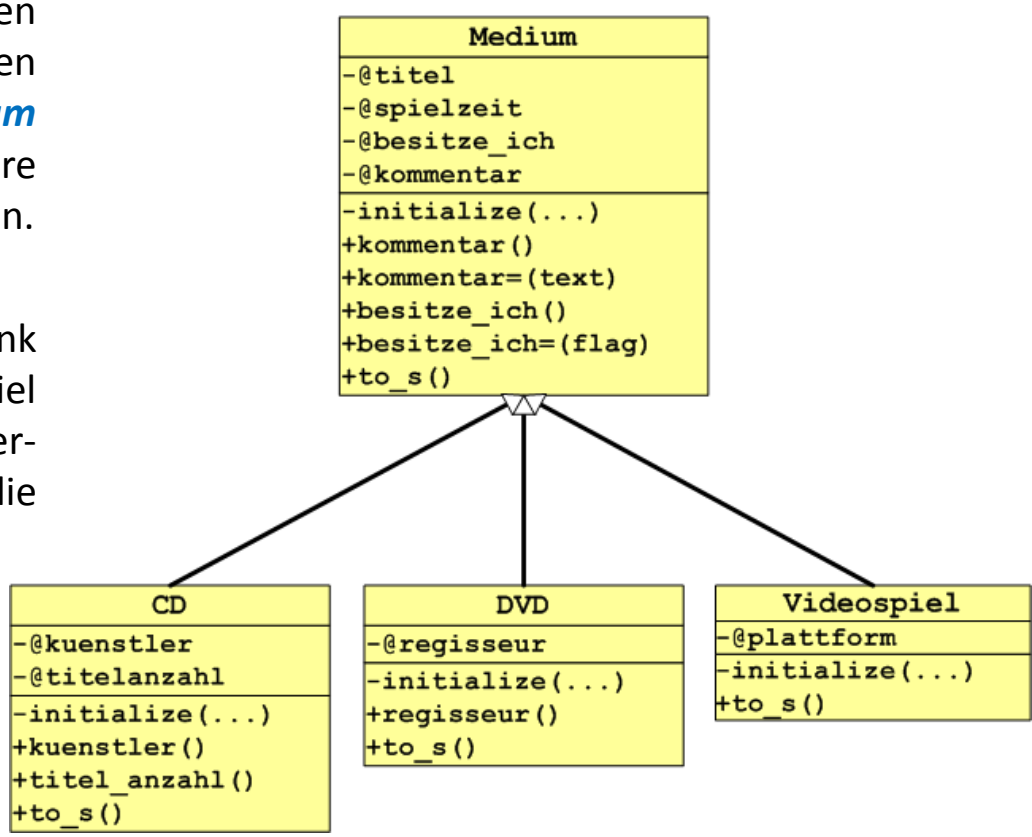
- Subklassen müssen den Initialisierungsanteil ihrer Superklasse über die Methode ***super*** aufrufen.
- ***super*** in der ***initialize*** Methode von ***CD*** ruft das ***initialize*** der Superklasse ***Medium*** auf.
- Im ***initialize*** dürfen der Methode ***super*** Parameter übergeben werden
- Hier werden für das ***initialize*** der Superklasse die Werte für die Instanzvariablen der Superklasse übergeben.
- **Ü-11-1**: Überprüfen Sie das Verhalten von ***super*** im Debugger. Setzen Sie einen Haltepunkt und inspizieren Sie mit Step-Into.

```
class CD < Medium
  # titel, kuenstler,kommentar: Strings
  # anzahl: Integer
  # spielzeit: Float
  def initialize(titel,kuenstler,anzahl,
          spielzeit,besitz,kommentar)

      super(titel ,spielzeit, besitz,
                                  kommentar)

      @kuenstler = kuenstler
      @titelanzahl = anzahl
  end
end
```

Weitere Medien hinzufügen

- Mit der Einführung der Vererbungshierarchie wird das Hinzufügen neuer Medien sehr einfach. Alle neue Medien können die Variablen und Methoden von ***Medium*** wieder verwenden und sich auf ihre spezifischen Erweiterungen konzentrieren.

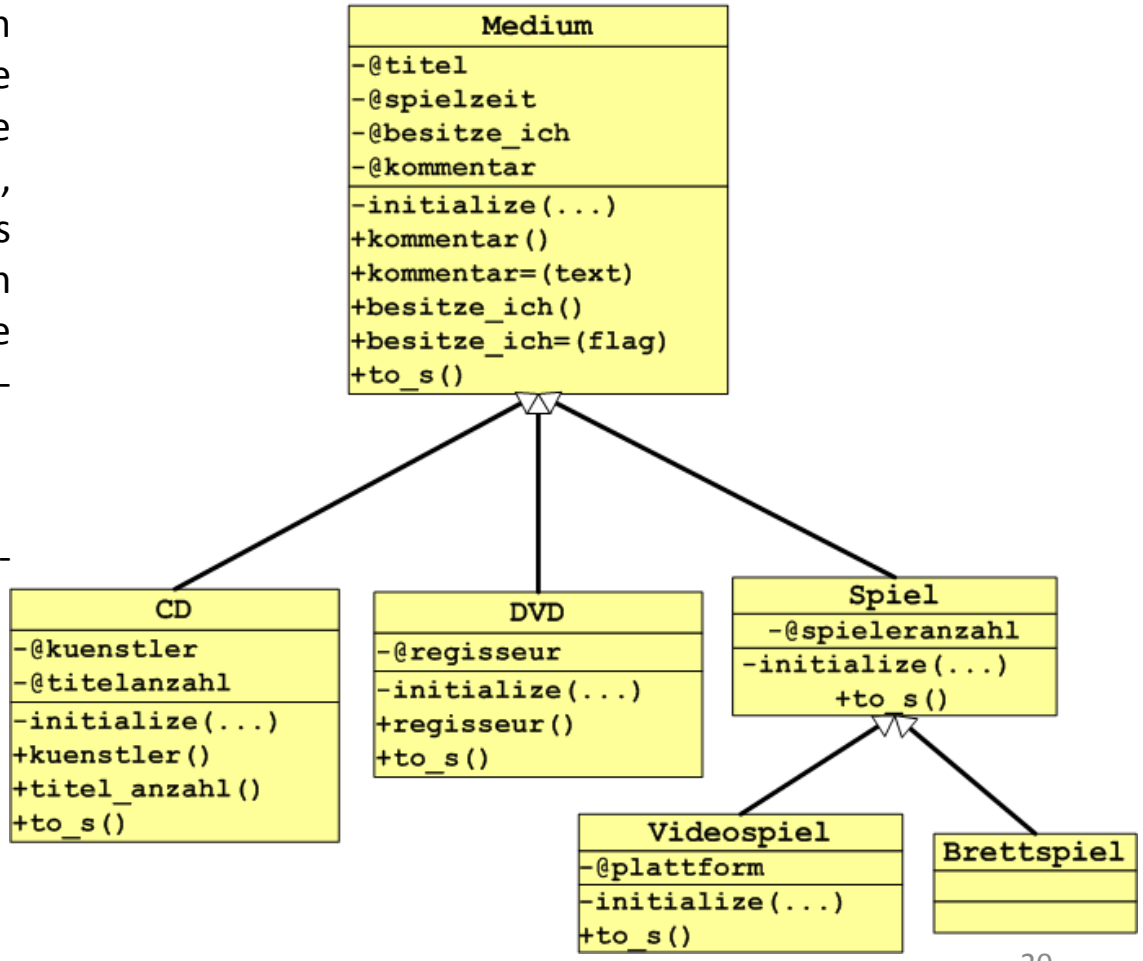
- **Ü-11-2**: Fügen Sie bitte in die Datenbank das Medium ***Videospiel*** ein! Ein Videospiel hat die zusätzlichen Eigenschaften Spieleranzahl und Plattform. Es ergibt sich die rechtsstehende Vererbungshierarchie.



Weitere Medien hinzufügen

- **Ü11-3**: Jetzt sollen neben Videospielen auch Brettspiele in die Datenbank eingefügt werden! Beide Spiele haben eine Spieleranzahl, Brettspiel aber keine Plattform. Es sollen die Gemeinsamkeiten von Spielen in einer separaten Klasse extrahiert werden um Code-Duplizierungen zu vermeiden.

  Wie sieht dann die Vererbungs-hierarchie aus?



Weitere Medien hinzufügen

- Auch hier haben wir wieder die Gemeinsamkeiten zweier Klassen heraus faktorisiert und in einer Superklasse Spiel zusammengefasst.

- **Ü11-4**: Wir übersetzen die Klassenhierarchie für Medium in Ruby Quelltext und schreiben für alle neuen Klassen die Methode ***initialize***.

Klasse Spiel

- Objekte der Klasse ***Spiel*** haben neben den Attributen ***titel, spielzeit, besitze_ich*** und ***kommentar*** (aus ***Medium***), noch das Attribut ***spieleranzahl.***

- Die Initialisierung der ersten vier Attribute delegieren wir an die Superklasse mit ***super,*** die Initialisierung der ***spieleranzahl*** nehmen wir in Spiel vor.

```
require "Medium"
class Spiel < Medium

  def initialize(titel,spieleranzahl,
          spielzeit,besitz,kommentar)

    super(titel,spielzeit,besitz,
          kommentar)

    @spieleranzahl = spieleranzahl
  end
end
```

Klasse Videospiel

- Objekte der Klasse ***Videospiel*** haben neben den Attributen ***titel, spielzeit, besitze_ich*** und ***kommentar*** (aus ***Medium***) und dem Attribut ***spieleranzahl*** (aus ***Spiel***), noch das Attribut ***plattform.***

- Die Initialisierung der ersten fünf Attribute delegiert ***Videospiel*** an die Superklasse ***Spiel*** mit ***super,*** die Initialisierung der ***plattform*** übernimmt ***Videospiel***.

- ***Spiel*** wiederum delegiert die Initialisierung der ersten vier Attribute an die Superklasse ***Medium***.

- **Ü11-5**: Wir setzen in der ersten Zeile des ***initialize*** von ***Videospiel*** einen Breakpoint und inspizieren die Initialisierung entlang der Vererbungshierarchie im Debugger.

```
require "Spiel"

class Videospiel < Spiel
  def initialize(titel,spieleranzahl,
          plattform,spielzeit,
          besitz,kommentar)

    super(titel,spieleranzahl,spielzeit,
          besitz,kommentar)

    @plattform = plattform
  end
end
```

Sourcecode der Klasse Brettspiel

```ruby
class Medium
  attr_reader :kommentar,:besitze_ich
  attr_writer :kommentar,:besitze_ich

  def
    initialize(titel,spielzeit,besitz,k
    ommentar)
     puts("initialize in Medium
    ausgeführt in #{self.class}")
     @titel = titel
     @spielzeit = spielzeit
     @besitze_ich = besitz
     @kommentar = kommentar
  end
end
```

```ruby
class Spiel < Medium
  def
    initialize(titel,spieleranzahl,spie
    lzeit,besitz,kommentar)
     puts("initialize in Spiel
    ausgeführt in #{self.class}")

    super(titel,spielzeit,besitz,kommen
    tar)
     @spieleranzahl = spieleranzahl
  end
end

class Brettspiel < Spiel
end
```

Klasse Brettspiel

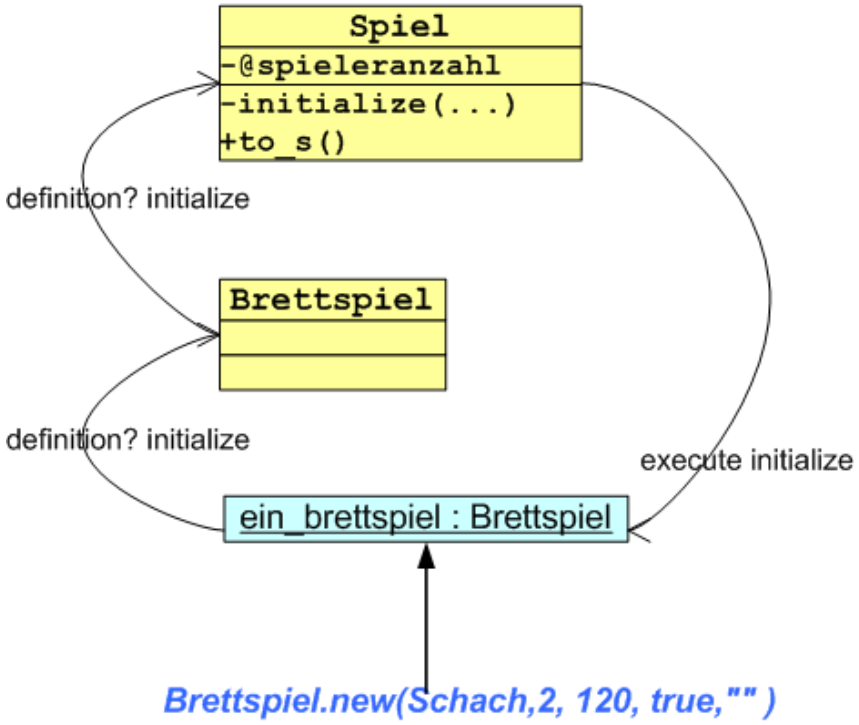
- Objekte der Klasse ***Brettspiel*** haben die gleichen Attribute wie Objekte der Klasse Spiel ***titel, spielzeit, besitze_ich*** und ***kommentar*** (aus ***Medium***) und dem Attribut ***spieleranzahl*** (aus ***Spiel***).

- Da Brettspiel die ***initialize***-Methode von Spiel erbt, benötigen wir für Brettspiel keine eigene ***initialize***-Methode.

- Erzeugen wir Objekte der Klasse ***Brettspiel***, dann wird beim Aufruf von ***initialize*** auf dem noch nicht initialisierten Brettspiel-Objekt ***self*** entlang der Klassenhierarchie nach der ersten Methodendefinition von ***initialize*** gesucht. Diese Methodendefinition wird dann auf ***self*** angewendet.

- Die Suche beginnt in der Klasse von ***self*** und sucht nacheinander die Kette der Superklassen von unten nach oben ab.

Methodensuche in Klassenhierarchien



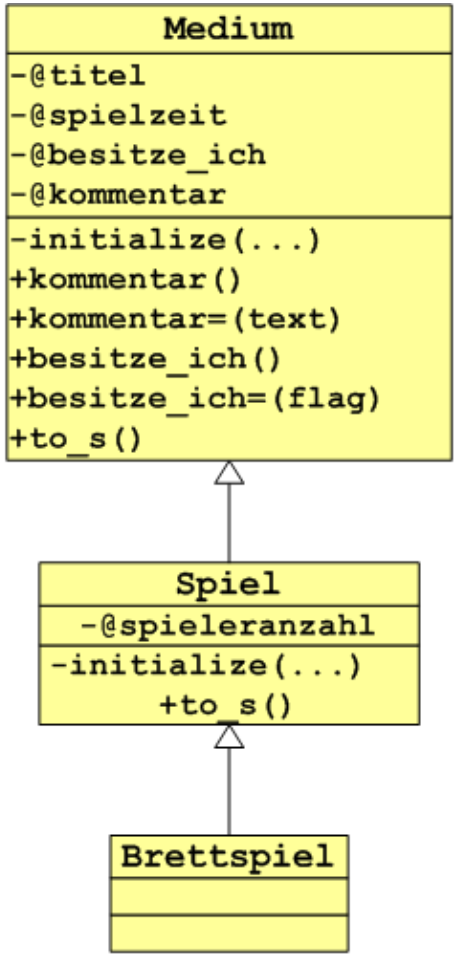
- **definition?** sucht nach einer Methoden-definition (dies ist **keine** Rubymethode!)
- **execute initialize** meint, dass die Methodendefinition im Kontext des Zielobjektes ausgeführt wird (hier ***ein_brettspiel***)
- Der Effekt: die Instanzvariablen werden für das Objekt ***ein_brettspiel*** initialisiert.

Methodensuche mit *super*

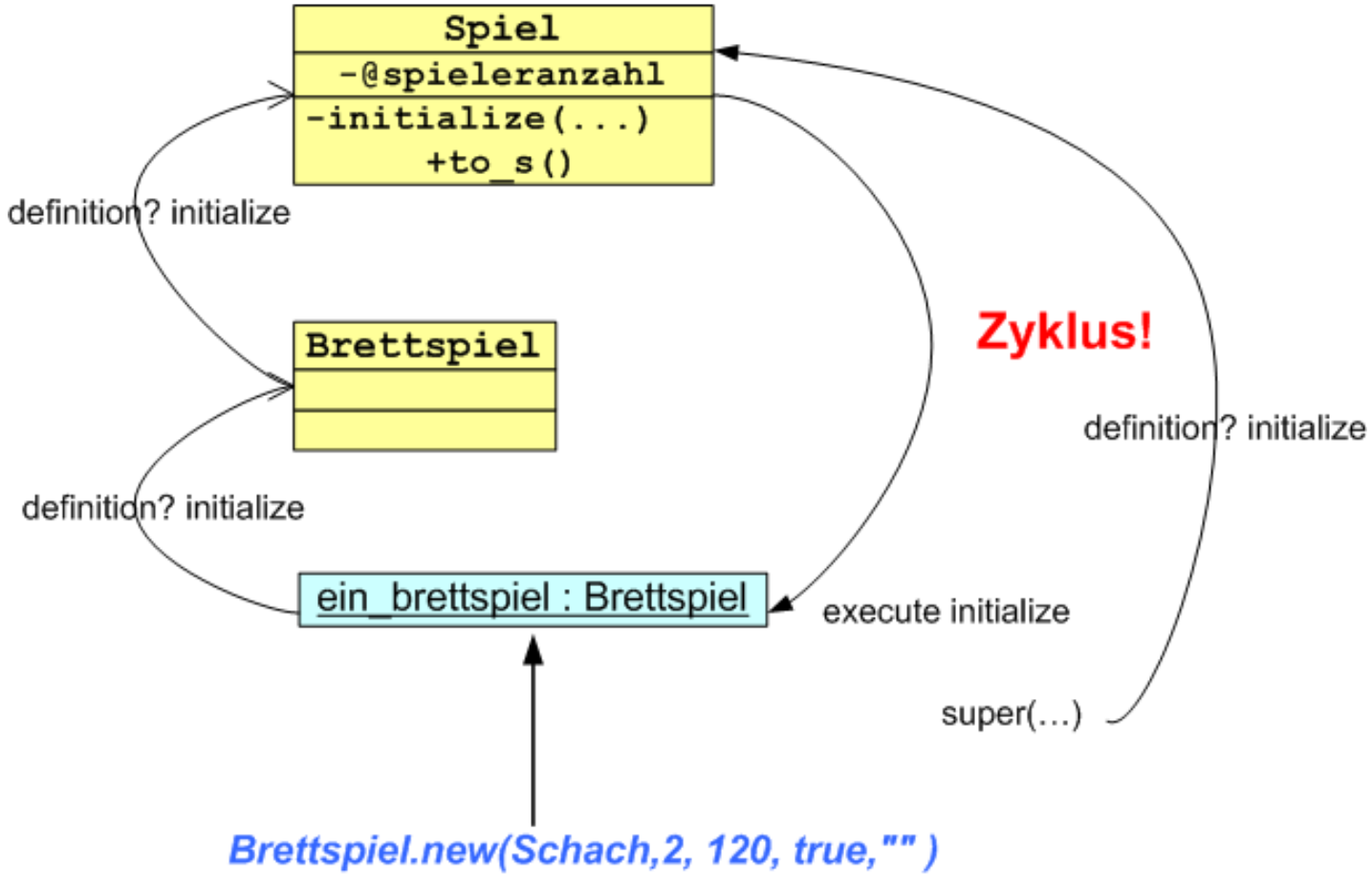
- Die Klasse Spiel ruft in ***initialize*** mit ***super*** das ***initialize*** der Superklasse ***Medium*** auf. Wir wollen uns nun anschauen, wie ***super*** aufgelöst wird, wenn die Methodensuche in einem Objekt der Klasse ***Brettspiel*** beginnt.

- Zwei Varianten stehen zur Auswahl:
  1. ***super*** bezieht sich auf die Superklasse des Objektes in dem wir ***initialize*** aufrufen (für ***Brettspiel*** wäre das die Klasse ***Spiel***)

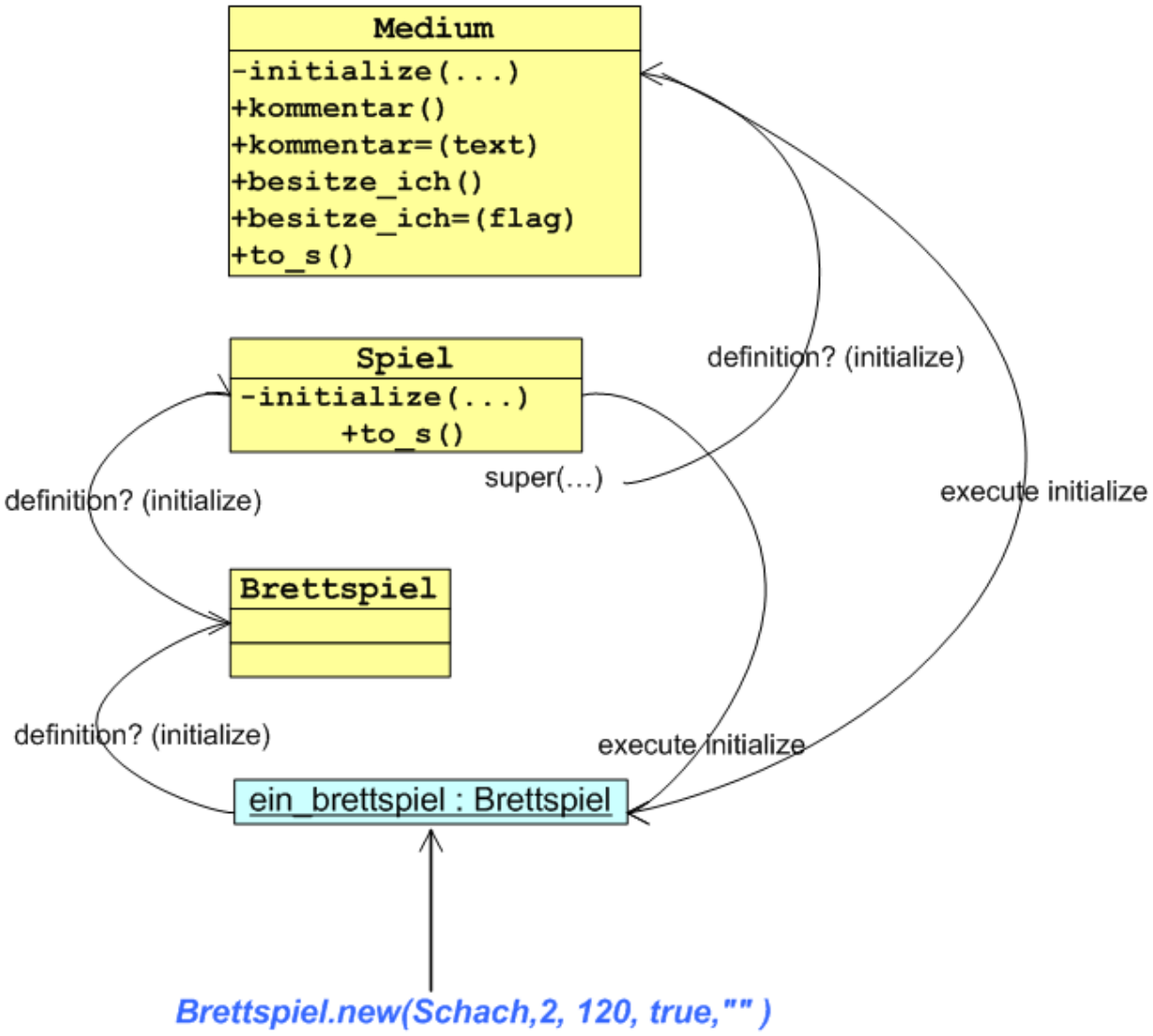
  2. ***super*** bezieht sich auf die Superklasse **der** Klasse, deren Methodendefinition ***super*** aufruft (**Enclosing Class**). Da ***super*** in der ***initialize-***Methode von ***Spiel*** aufgerufen wird, wäre das die Klasse ***Medium.***



Variante 1 würde einen Zyklus erzeugen



Variante 2 ist die richtige Lesart



Nachweis über Ausgaben auf die Konsole

- Wir erweitern die ***initialize*** Methode von ***Spiel*** und ***Medium*** und geben jeweils die **definierende Klasse** und die **Klasse von *self*** aus.

- Dann liefert

```
Brettspiel.new("Schach",2,120,true,"")
```

```
initialize in Spiel ausgeführt in
   Brettspiel
initialize in Medium ausgeführt in
   Brettspiel
```

```
class Medium
  def initialize(titel,spielzeit,besitz,
                       kommentar)
    puts("initialize in Medium ausgeführt in
    #{self.class}")

    @titel = titel
    @spielzeit = spielzeit
    @besitze_ich = besitz
    @kommentar = kommentar
  end
 end
```

```
class Spiel < Medium
  def initialize(titel,spieleranzahl,
           spielzeit,besitz,kommentar)
    puts("initialize in Spiel ausgeführt in
    #{self.class}")

    super(titel,spielzeit,besitz,kommentar)
    @spieleranzahl = spieleranzahl
  end
end
```

Verallgemeinern der Methoden von Datenbank

- Da die Klassen ***CD, … , Brettspiel*** alle Subklassen von ***Medium*** sind, können wir
  - Die Verwaltung auf eine Liste von Medien umstellen,
  - Das Erfassen auf ***Medien*** verallgemeinern.
  - Die Ausgabe auf ***Medien*** umstellen.

```
class Datenbank
  def initialize()
    @medien = []
  end
  def erfasse_medium(medium)
    @medien << medium
  end
  def medien_auflisten()
    @medien.each {|medium| puts medium }
  end
end
```

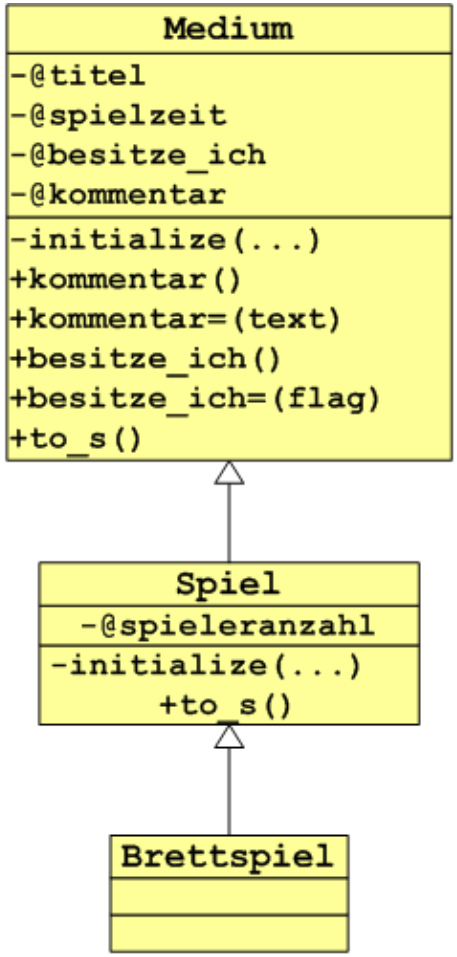
Typen

- Klassen definieren Objekttypen. Parallel zur Klassenhierarchie haben wir also eine Typhierarchie, in der zu jeder Subklasse ein entsprechender Subtyp existiert.
- Wir wollen nun sicherstellen, dass in der Methode ***erfasse_medium*** nur Objekte vom Typ ***Medium*** als Parameter übergeben werden können.
- Dazu müssen wir den Typ eines Objektes prüfen.
- Hierzu einige Begriffe:
  - Wenn wir sagen, dass ein Objekt Instanz einer Klasse ***K*** ist, dann meinen wir, dass das Objekt durch ***K.new*** aus dieser Klasse entstanden ist.
  - Wenn wir sagen, dass ein Objekt ***o*** vom Typ ***T*** ist, dann meinen wir, dass die **Klasse** ***K*** von ***o*** **oder** eine der **Superklassen** von ***K*** den Typ ***T*** definieren.
- Klassenzugehörigkeit prüfen wir mit ***o.instance_of?(K)*** oder mit ***o.class == K***
- Typzugehörigkeit prüfen wir mit ***o.is_a?(T)*** oder ***o.kind_of?(T)***.
- Typprüfung ist immer die allgemeinere Prüfung, da sie Prüfung auf Typen der Superklassen (kurz: Supertypen) erlaubt.

Typprüfung und Klassenzugehörigkeit

- **Ü11-5**: Gegeben die Objekte:
  sp = Spiel.new(…..) und
  bsp = Brettspiel.new(…) , die Parameter sind hier nicht von Interesse. Welche der folgenden Prüfungen liefert ***true***, welche ***false***?

  sp.is_a?(Spiel)
  sp.instance_of?(Spiel)
  sp.instance_of?(Medium)
  sp.kind_of?(Medium)
  bsp.instance_of?(Spiel)
  bsp.class == Spiel?
  bsp.is_a?(Brettspiel)
  bsp.instance_of?(Brettspiel)
  bsp.kind_of?(Medium)
  bsp.instance_of?(Medium)



Typprüfung auf Parametern

- In ***erfasse_medium*** dürfen nur Objekte vom Typ ***Medium*** als Parameter übergeben werden.
- Damit wir die Flexibilität erhalten, später neue Subtypen von ***Medium*** in unsere Datenbank einzufügen, prüfen wir die Parameter auf den Typ ***Medium*** und **nicht** auf die konkreteren Subtypen (z.B. ***Spiel***, ***Brettspiel***).
- Wenn Objekte nicht von geforderten Typ sind, geben wir einen Hinweis auf der Konsole aus und tragen das Objekt nicht in unsere Datenbank ein.

```
class Datenbank
  def initialize()
    @medien = []
  end
  def erfasse_medium(medium)
    if (!medium.is_a?(Medium))
      puts("#{medium.class} nicht
   vom Typ Medium")
      return
    end
    @medien << medium
  end
  def medien_auflisten()
    @medien.each {|medium| puts
   medium }
  end
end
```

Typen und Ersetzbarkeit

- Mit der Lösung für ***erfasse_medium*** können wir nun Objekte aller Subtypen von ***Medium*** in unsere Datenbank eintragen.
- Wir können den **Supertyp** also durch jeden beliebigen **Subtyp ersetzen**.
- Dies heißt auch das **Prinzip der Ersetzbarkeit**, oder nach der Erfinderin Barbara Liskov, das **Liskov‘sche Substitutionsprinzip**.
- Das Prinzip der Ersetzbarkeit ist ein sehr mächtiges Mittel, da wir Programme für allgemeinere Typen entwickeln können, und ohne Änderung beliebig viele konkreter Typen zu späteren Zeitpunkten an den gleichen Stellen benutzen können.
- Dies macht unsere Programme erweiterbar, ohne dass wir an bestehenden Klassen Änderungen vornehmen müssen.
- Genauer lautet das **Liskov‘sche Substitutionsprinzip**:
- Ein Programm ändert sein Verhalten weder syntaktisch noch semantisch nicht, wenn an Stelle eines Objekts eines Typs ein Objekt eines Untertyps eingesetzt wird.

Object

- Alle Klassen erben von ***Object***. Auch Klassen, die nicht explizit von ***Object*** mit **<** ableiten.

- ***Object*** enthält eine Standard-Implementierung aller Methoden, die in einer objekt-orientierten Sprache für den Umgang mit Objekte unerlässlich sind, u.A.:

  - Identitätsprüfung
  - Inhaltsgleichheit
  - Methode zur Berechnung des Hashcodes
  - Methoden zur Typ- und Klassenprüfung
  - Methode zur Darstellung als String
  - Methoden zum Kopieren von Objekten

Das Problem mit der unvollständige Ausgabe

v12-DoME 1

- liefert die Ausgabe einer CD

```
CD: Nevermind (120 Min) *
    Nirvana
    Titelanzahl: 12
    probates Antistress Mittel
```

v12-DoME 2

- liefert die Ausgabe einer CD

```
CD: Nevermind (120 Min) *

    probates Antistress Mittel
```

- Es fehlen die Angaben über die Gruppe und die Titelanzahl.

Unvollständige Ausgabe der spezifischen Eigenschaften

- Schauen wir uns die ***to_s*** Methode von Medium an. Da diese Klasse nur ihre eigenen Instanzvariablen kennt, kann sie die der Subklassen nicht ausgeben.

- Daher liefert die Methode ***medien_auflisten*** in der Datenbank die unvollständigen Ausgaben.

```
class Medium
  def to_s()
    "#{self.class}: #@titel (#@spielzeit
    Min) #{besitze_ich ? '*' :''}\n" +
     "\n    #@kommentar"
  end
end
```

```
CD: Nevermind (120 Min) *

    probates Antistress Mittel
Brettspiel: Schach (120 Min) *

    Verliehen
```

```
nirvana =
    CD.new("Nevermind","Nirvana",12,120,tr
    ue,"probates Antistress Mittel")
schach =
    Brettspiel.new("Schach",2,120,true,"Ve
    rliehen")
db = Datenbank.new()
db.erfasse_medium(nirvana)
db.erfasse_medium(schach)
db.medien_auflisten()
```

Schlechte Lösung:

- Methode ***to_s*** mit den Bestandteilen aus ***Medium*** in die Subklassen verschieben.

- **Ergebnis**: Code-Duplizierung, die wir kurz zuvor durch die Einführung der Vererbungshierarchie beseitigt haben.

Auf dem Weg: Überschreiben von Methoden der Superklasse

- Wir schreiben in den Subklasse eine Methode ***to_s*** für die Darstellung der Instanzvariablen in den Subklassen.
- **Nachteil**: Jetzt wird der allgemeine Anteil in Medium nicht mehr angezeigt.
- **Grund**: Die Methode ***to_s*** in der Subklasse verdeckt die Methode ***to_s*** in der Superklasse. Wir sagen auch die Methode ***to_s*** der Subklasse **überschreibt die Methode** der Superklasse.

```
class Brettspiel < Spiel
end

class Spiel < Medium
  # unvollständig es fehlt der Medium
  # Anteil
  def to_s()
    "    #@spieleranzahl"
  end
end

class CD < Medium
  # unvollständig es fehlt der Medium
  # Anteil
  def to_s()
    "    #@kuenstler\n" +
    "    Titelanzahl: #@titelanzahl"
  end
end
```

Ausgabe bei Überschreiben der Methoden der Superklasse

```
nirvana =
    CD.new("Nevermind","Nirvana",12,120,tr
    ue,"probates Antistress Mittel")
schach =
    Brettspiel.new("Schach",2,120,true,"Ve
    rliehen")
db = Datenbank.new()
db.erfasse_medium(nirvana)
db.erfasse_medium(schach)
db.medien_auflisten()
```

```
Nirvana
Titelanzahl: 12
2
```

Bessere Lösung: Überschreiben und Aktivieren mit *super*

- Methode ***to_s*** in den Subklassen ruft die Methode ***to_s*** der Superklasse auf und setzt die Darstellung aus Bausteinen der Superklasse und den eigenen Bausteinen zusammen.
- Der Aufruf von ***super*** in ***to_s*** ruft ***to_s*** in der Superklasse der definierenden Klasse auf.
- **Nachteil**: Die Einbettung spezifischer Information der Subklassen zwischen der ersten Zeile in ***to_s*** von ***Medium*** und der letzten Zeile, dem Kommentar von Medium ist nicht möglich.

```
class Brettspiel < Spiel
end

class Spiel < Medium
 # vollstaendig aber falsche
 # Reihenfolge
  def to_s()
     super +
     "     #@spieleranzahl"
  end
end

class CD < Medium
 # vollstaendig aber falsche
 # Reihenfolge
  def to_s()
    super +
    "     #@kuenstler\n"+
    "     Titelanzahl: #@titelanzahl"
  end
end
```

Ausgabe bei Überschreiben und Aktivieren mit *super*

```
nirvana =
    CD.new("Nevermind","Nirvana",12,120,true,"probates Antistress Mittel")
schach =
    Brettspiel.new("Schach",2,120,true,"Verliehen")
db = Datenbank.new()
db.erfasse_medium(nirvana)
db.erfasse_medium(schach)
db.medien_auflisten()
```

```
CD: Nevermind (120 Min) *
    probates Antistress Mittel
    Nirvana
    Titelanzahl: 12
Brettspiel: Schach (120 Min) *
    Verliehen
    2
```

Übung

- **Ü11-6:** Zeichnen Sie bitte für die Klassen ***Videospiel*** und ***CD***, den Ablauf der Methodensuche und die Ausführung der gefundenen Methoden auf, analog dem ***initialize*** Beispiel!

Polymorphe Methoden und Methodensuche

- **Methoden** mit gleichem Namen **verhalten sich unterschiedlich**, abhängig davon, auf welchem Objekt diese aufgerufen werden.
- Ausgaben von CDs unterscheiden sich von Ausgaben von Brettspiel.
- Die ***initialize*** und ***to_s*** Methoden verhalten sich unterschiedlich, je nachdem auf welchem Objekt diese aufgerufen werden.
- Zur Laufzeit wird anhand des **Objekttyps** entschieden, welche Methode ausgeführt wird. Der Mechanismus dahinter nennt sich **Methodensuche** oder **dynamisches Binden** von Methoden.
- Die Methodensuche beginnt immer in der Klasse von ***self.***
- Wurde die Methode in der Klasse nicht gefunden, dann wird in der Superklasse gesucht, dann in deren Superklasse solange bis die Methode gefunden wurde, oder die Klasse ***Object*** erreicht ist.
- Enthält ***Object*** diese Methode nicht, dann wird ein ***NoMethodError*** generiert.

Lösung: Verwendung von Hookmethoden

- Wir wollen die ursprüngliche Reihenfolge in der Ausgabe wieder herstellen. D.h. das Ergebnis von ***to_s()*** der Subklassen soll zwischen den allgemeinen Angaben zu einem Medium und dem Kommentar eingebettet werden.

- Subklassen können Methoden der Superklassen aufrufen (***super***).

- Dieser Weg ist eine Einbahnstrasse: (***to_s()*** in ***Medium*** hat keinen Zugriff auf die ***to_s()*** Methoden der Subklassen.

➔ Wir brauchen eine andere Lösung.

```
class Medium

  def to_s()
    "#{self.class}: #@titel (#@spielzeit Min)
    #{besitze_ich ? '*' :''}\n" +
     medium_spezifisch_to_s() +
     "     #@kommentar\n"
  end

  def medium_spezifisch_to_s()
    ""
  end
end
```

Lösung: Verwendung von Hookmethoden

- **Idee**: Wir definieren eine Platzhaltermethode (**Hookmethode**), die die spezifischen Eigenschaften der Subklassen als String liefern soll, und verwenden diese in der Implementierung von ***to_s()*** an der entsprechenden Stellen.

- Die Methode, die die Hookmethode verwendet, heißt **Templatemethode**, da sie den Rahmen definiert, in dem eine Hookmethode ausgeführt wird.

- Die Hookmethode in unserem Beispiel ist ***medium_spezifisch_to_s() .***

- Die Templatemethode in unserem Beispiel ist ***to_s() .***

```
class Medium

  def to_s()
    "#{self.class}: #@titel (#@spielzeit
    Min) #{besitze_ich ? '*' :''}\n" +

    medium_spezifisch_to_s() +

    "     #@kommentar\n"
  end

  def medium_spezifisch_to_s()
    ""
  end
end
```

Lösung: Verwendung von Hookmethoden

- Wir schreiben eine Defaultimplementierung für die Hookmethode in ***Medium*** ***medien_spezifisch_to_s()***, damit wir auch wenn Subklassen diese Methode nicht implementieren, immer ein fehlerfreies Ergebnis bekommen.

- In den Subklassen können wir diese Default-Implementierung überschreiben, damit die spezifischen Information der Subklassen in den endgültigen String integriert werden können.

- ***medien_spezifisch_to_s()*** zeigt die Eigenschaften einer **Hookmethode:** Subklassen können ihre Implementierung in eine Templatemethode der Superklasse (hier ***to_s***) „einhaken“, indem sie Hookmethode überschreiben.

```
class Medium

  def to_s()
    "#{self.class}: #@titel (#@spielzeit
    Min) #{besitze_ich ? '*' :''}\n" +

      medium_spezifisch_to_s() +

     "     #@kommentar\n"
  end

  def medium_spezifisch_to_s()
    ""
  end
end
```

Lösung: Verwendung von Hookmethoden

- In den **Subklassen überschreiben** wir die **Hookmethode** ***medien_spezifisch_to_s()*** und ersetzen dadurch deren Default-Implementierung von in ***Medium***

- Ein **Spezialfall** ist ***Videospiel***: hier wird zusätzlich zur Ersetzung der Hookmethode in ***Medium***, mit ***super*** die Methode ***medien_spezifisch_to_s()*** der Superklasse ***Spiel*** aufgerufen.

```
class CD < Medium
  # Hookmethode kein to_s mehr
  def medium_spezifisch_to_s()
    "    #@kuenstler\n"+
    "    Titelanzahl: #@titelanzahl\n"
  end
end

class Spiel < Medium
  # Hookmethode kein to_s mehr
  def  medium_spezifisch_to_s()
      "    #@spieleranzahl\n"
  end
end


class Videospiel < Spiel
  # Hookmethode mit super kein to_s mehr
  def medium_spezifisch_to_s()
    super +
    "    #@plattform\n"
  end
end
```

Übung

- **Ü11-7**: Zeichnen Sie bitte für die Lösung mit der Hookmethode und die Klassen ***Videospiel*** und ***CD***, den Ablauf der Methodensuche und die Ausführung der gefundenen Methoden von ***to_s*** analog zum Vorgehen im ***initialize*** Beispiel auf!

```
nirvana =
    CD.new("Nevermind","Nirvana",12,120,true,"probates Antistress Mittel")
schach =
    Brettspiel.new("Schach",2,120,true,"Verliehen")
video =
    Videospiel.new("VDSp",4,"XBox",120,false,"no-go")
db = Datenbank.new()
db.erfasse_medium(nirvana)
db.erfasse_medium(schach)
db.erfasse_medium(video)
db.medien_auflisten()
```

Hookmethoden sind interne Methoden

- Die Hookmethode ***medium_spezifisch_to_s()*** kann nur sinnvoll im Kontext der Templatemethode ***to_s*** aufgerufen werden.
- Sie sollte daher nur für die Klasse und deren Subklassen sichtbar sein.
- Da ***private*** Methoden für ***self*** entlang der Klassenhierarchie sichtbar sind, geben wir der Hookmethode in allen Klassen der Medium Hierarchie die Sichtbarkeit ***private.***

```ruby
class Medium
  #Hookmethode
  def medium_spezifisch_to_s()
    ""
  end
  private :medium_spezifisch_to_s
end
```

```ruby
class Spiel < Medium
  def  medium_spezifisch_to_s()
      "    #@spieleranzahl\n"
  end
  private :medium_spezifisch_to_s
end
```

und alle weiteren Klassen analog

Zusammenfassung

- **Vererbung** erlaubt es eine Klasse als Erweiterung einer anderen zu definieren.
- Vererbung hilft uns **Code-Duplizierung** zu vermeiden, indem wir doppelten Code in Superklassen auslagern.
- Vererbung erlaubt es bestehenden **Quelltext** in Subklassen **wieder zu verwenden**.
- Durch Vererbung lassen sich Programme leichter **erweitern**.
- Eine **Superklasse** ist eine Klasse, die von anderen Klassen erweitert wird.
- Eine **Subklasse** erweitert eine andere Klasse und erbt alle Instanzvariablen und Methoden der Superklasse.
- Eine **Vererbungshierachie** ist die Verknüpfung von Klassen über Vererbungsbeziehungen.
- Analog zur Klassenhierarchie bilden die Objekttypen eine Typhierarchie.
- **Liskov'sches Substitutionsprinzip**: Subtypen können an allen Stellen verwendet werden, an denen ein Supertyp erwartet wird, ohne das sich das Verhalten ändert.
- ***Object***: Alle Klassen ohne explizite Vererbungsbeziehung erben von ***Object***.

Zusammenfassung

- Methoden in Subklassen **überschreiben** Methoden in Superklassen gleichen Namens.
- Die Methoden in Superklassen sind dann nur noch durch ***super*** zu erreichen.
- ***super*** ruft die Methode, in der ***super*** aufgerufen wird, mit den gleichen Parametern in der „Enclosing Class“ auf.
- **Polymorphie von Methoden** (übersetzt Vielgestaltigkeit) besagt, dass Methoden sich unterschiedlich verhalten, je nachdem auf welchen Objekt sie aufgerufen werden.
- **Dynamisches Binden** bezeichnet den Vorgang, dass zur Laufzeit auf der Grundlage des Objekttyps die Methodendefinition bestimmt wird.
- Dynamisches Binden basiert auf der **dynamischen Methodensuche**.

Zusammenfassung

- **Hookmethoden** sind Methoden, die in **Templatemethoden** einer Superklasse verwendet werden. Subklassen können Hookmethoden überschreiben und damit das Verhalten der Templatemethode der Superklasse verändern.

- **Templatemethoden** sind Methoden einer Klasse, in der Methoden der Klasse aufgerufen werden. Templatemethoden definieren einen Rahmen für z.B. Hookmethoden.