

8.x Alternative examples of subclasses

Requirements for the example:

- It should be understandable by students and others
- It should be possible to complete it to a program that does something meaningful
- It should be as small as possible
- It must have non-virtual methods in all subclasses and the superclass
- It must have one or more virtual methods defined in the superclass and extends in subclasses
- It will be an advantage if it may have nested classes and virtual classes.
- There should be at least two levels in the subclass hierarchy.
- However, we also need a bigger example, but probably later.

Composition

Classification and composition are means to organize complexity in terms of hierarchies. A classification is represented by a class/subclass hierarchy, as we have seen already.

Composition is related to either part-whole hierarchies on phenomena, and decomposition of methods (procedures and functions) into local methods. We do not consider the last kind of composition further here since it should be well-known from programming languages in general. However, in the following we will cover part-whole hierarchies.

+++?important to be aware that they are two different conceptual means - the one cannot replace the other and vice versa.

The language mechanism to describe part-whole composition is called composite objects. Embedded objects are fixed parts of the composite object, and it has a name that is a constant reference to the object.

In [55] we showed that the combination of embedded objects and nesting as in Beta may be used to inherit/include code from other classes in a form that avoids most of the problems with multiple inheritance. The proposal is thus also an alternative to mixins and traits.

Consider a class Address:

```
class Address:
  street: ref String
  streetNo: var integer
  town: ref String
  country: ref String
  print:<
    inner(print)
    print{street,streetNo,town,Country}
```

An Address may be embedded in a Person object:

```
class Person:
```

```
name: ref String
addr: obj Address
...
```

The keyword **ref** declares an ordinary reference to an object of the class – in the above cases String objects. The keyword **obj** specifies an embedded object of the class – in this case addr is the name of an Address object being embedded in the holding Person object.

+++ eventuelt

Virtual classes

In some cases the concept represented by a class is a concept where the phenomena consist of a sets/structures of phenomena (and their concepts) that are defined in the context of the main concept. We have seen that this is covered by nesting, i.e. defining objects/classes in the context of an object/class. We say that description of these object/classes are nested within the description of the outer object/class.

We also have cases where the nested classes have to extended in subclasses of the enclosing class. This calls for the notion of virtual classes; they are similar to virtual methods. The properties of objects of a virtual class, even if generated as part of activities of the enclosing class, will be the properties of the extended virtual class for the given subclass.

As an example consider the concept of representing expressions as structures of objects representing the various parts of the expression. We call the class representing this concept MetaExpression. For simplicity we just include a subset of what would be in a real meta expression. As the classes have to be extended depending on what we would like to do with expressions, e.g. print or evaluate, we define them as virtual classes:

```
class MetaExpression
  class Expr:< Object

  class Constant(val: var Integer):< Expr
  class BinaryExpr(left, right: ref Expr):< Expr
  class Add:< BinaryExpr
  class Mult:< BinaryExpr
```

Her er en dobbelt virtuel super. Super for Add (og Mult) er BinaryExpr som er virtuel og super for BinaryExpr er Expr som er virtuel

There are two ways of generating an expression, either by a function (makeExpression) or as part of an object of class MetaExpression:

```
exampleExpression: ref MetaExpression
```

```
makeExpression(meta: ref MetaExpression) -> ref meta.Expr:
  const1: ref meta.Expr := meta.Constant(-3)
  const2: ref meta.Expr := meta.Constant(2)
  op1: ref meta.Expr := meta.Mult(const1, const2)
  const3: ref meta.Expr := meta.Constant(5)
  op2: ref meta.Expr := meta.Add(op1, const3)
```

```
return op2
```

```
...
```

```
exampleExpression: ref MetaExpression  
exp: ref exampleExpression.Expr  
exp := makeExpression(exampleExpression)
```

or:

```
exampleExpression: obj MetaExpression  
  const1: ref Expr := Constant(1)  
  const2: ref Expr := Constant(2)  
  op1: ref Expr := Mult(const1, const2)  
  const3: ref Expr := Constant(3)  
  op2: ref Expr := Add(op1, const3)  
  expression: ref Expr := op2
```

```
...
```

```
exampleExpression.expression
```

The above expression may in qBeta be expressed somewhat simpler:

```
expression := Add(Mult(Constant(1),Constant(2)),Constant(3))
```

```
meta: obj MetaExpression
```

```
...
```

```
expr1, expr2: ref meta.Expr
```

```
expr1 :=
```

```
  meta.Add(meta.Mult(  
    meta.Constant(1),meta.Constant(2)),meta.Constant(3))
```

```
expr2 :=
```

```
  meta.Add(meta.Mult(  
    meta.Constant(4),meta.Constant(5)),meta.Constant(6))
```

```
...
```

If we want to print expression we would need a formatting method for each part of the expression. This is done by extending the virtual classes, adding a method format. Note that for the general class Expr this format is defined as a virtual method that is extended in subclasses of Expr:

```
class ExprFormat: MetaExpression
```

```
  class Expr::
```

```
    format -> String:<  
      inner(format)
```

```
  class Constant::
```

```
    format::  
      return val    --??
```

```
  class BinaryExpr::
```

```

    format::
        return "(" + left.format + operatorSymbol
            + right.format + ")";
    operatorSymbol -> String:<
        inner(operatorSymbol)
class Add::
    operatorSymbol::
        return '+'

class Mult::
    operatorSymbol::
        return '*'

meta: ref MetaExpression

meta := MetaExpression
expr1, expr2: ref meta.Expr

expr1 :=
    meta.Add(meta.Mult(
        meta.Constant(1), meta.Constant(2)), meta.Constant(3))
expr2 :=
    meta.Add(meta.Mult(
        meta.Constant(4), meta.Constant(5)), meta.Constant(6))

expr1.format    not possible
expr2.format    not possible

meta := ExprFormat
expr1 :=
    meta.Add(meta.Mult(
        meta.Constant(1), meta.Constant(2)), meta.Constant(3))
expr2 :=
    meta.Add(meta.Mult(
        meta.Constant(4), meta.Constant(5)), meta.Constant(6))
...
expr1.format
expr2.format

class Testing:
    class Meta:< MetaExpression
    metaObj: ref Meta
    expr1, expr2: ref Meta.Expr

    expr1 :=
        metaObj.Add(MetaExpression.Mult(
            MetaExpression.Constant(1), MetaExpression.Constant(2)),
            MetaExpression.Constant(3))
    expr2 :=
        MetaExpression.Add(MetaExpression.Mult(
            MetaExpression.Constant(4), MetaExpression.Constant(5)),

```

```
MetaExpression.Constant(6))
```

```
formatTesting: obj Testing
  Meta:: ExprFormat
  expr1.format
  expr2.format
```

```
evalTesting: obj Testing
  Meta:: ExprEval
  expr1.eval
  expr2.eval
```

For evaluating expressions we need to extend class `Expr` with an `eval` method, and then extend this in the subclasses of `Expr`:

```
class ExprEval: MetaExpression

  class Expr::
    eval -> Integer:<
      inner(eval)

  class Constant::
    eval::
      return val
  class Add::
    eval::
      return left.eval + right.eval
  class Mult::
    eval::
      return left.eval * right.eval
```

Notes:

If just an extension symbol (`::`) is used, then the extension is simply an extension of the virtual class, just as if the extension was a subclass of the virtual class. This also imply that the properties of the virtual class are visible in the extension. `Add` and `Mult` being extensions of `BinaryExpr` therefore implies that `left` and `right` are visible and can be used in `eval`.

```
+++ nævne :: <class> ??
```

shapes

Here are some of the basic elements of 2D geometry: points, lines, line segments and angles:

```
class Point(x,y: var Integer):
  distance(another: var Point) -> Real
    xDiff: var Integer = this.x - another.x
    yDiff: var Integer = this.y - another.y
    return sqrt(xDiff * xDiff + yDiff * yDiff)
class Line(p1,p2: var Point):
  ...
```

```

class LineSegment(p1,p2: var Point):
    ...

class Angle (vertex, p1, p2: var Point):
    side1, side2: var LineSegment
    side1 := LineSegment(vertex, p1)
    side2 := LineSegment(vertex, p2)

class Triangle: Shape2D
    a,b,c: var Point
    class Measures::
        area::
            return abs(
                (a.x * (b.y - c.y) + b.x * (c.y - a.y) +
                 c.x * (a.y - b.y)) / 2.0
            )

class Rectangle: Shape2D
    topLeft: var Point
    height, width: var Real
    class Measures::
        area::
            return height * width
        perimeter::
            return 2 * height + 2 * width

class Circle: Shape2D
    radius: var Real
    class Measures::
        area::
            return PI * radius * radius
        perimeter::
            return 2.0 * PI * radius

class Movable:
    moveTo(to: var Point):<

class Shape:
    center: var Point
    moving: obj Movable
    moveTo::
        center := to

...
origin := Point(0,0)
aShape.moving.moveTo(origin)

class Triangle: Shape2D
    a,b,c: var Point
    area::

```

```

    return abs(
      (a.x * (b.y - c.y) + b.x * (c.y - a.y) +
       c.x * (a.y - b.y)) / 2.0
    )

```

```

class Rectangle: Shape2D
  topLeft: var Point
  height, width: var Real
  perimeter::
    return 2 * height + 2 * width
  area::
    return height * width

```

```

class Circle: Shape2D
  radius: var Real
  perimeter::
    return 2.0 * PI * radius
  area::
    return PI * radius * radius

```

For shapes in 3D the interesting properties are surface and volume, and again, they have in class Shape3D to be defined as virtual methods as these properties depend on the types of shape:

```

class Shape3D:
  surface -> Real:<
    inner
  volume -> Real:<
    inner

class Cylinder: 3dShape
  base: var Circle
  height: var Real
  surface::
    return 2 * base.area + height * base.perimeter
  volume::
    return base.area * height

```

Bank System



```

class Account:
  balance: var Real
  interestRate: var Real
  ...
  deposit(amount: var Real):<      -- iflg Eiffel.Inheritance er denne
    balance := balance + amount    -- virtuel og redefineres i
    inner                          -- saving account; lagt ind her

class SavingsAccount: Account
  bindingPeriod: var Date -- eller Period?
  minimumBalance: var Real -- to avoid monthly fees

```

```

deposit:
    -- in addition add interest to balance
    balance := balance + interest    -- interest to be calculated
+++bmp: sådan forstod jeg eksemplet i Eiffel, men reagerede også
withdraw:
    -- needs to check that it is after bindingPeriod
    -- but here we do not have virtuals/extension
newBindingPeriod(p: var Data, newInterest: var Real):
    bindingPeriod := p
    interest := newInterest
class CreditAccount : Account
    maxCredit: var Real
    overdrawInterest: var Real -- interest for going over maxCredit
    changeCredit(newMax: var Real
                  , newInterest: var Real
                  , newOverdrawInterest: var Real):
        ...
    withdraw: -- must also be specialized
class RetirementAccount: SavingsAccount
    retirementAge: var Integer -- min age to start payments
    lastRetirementAge: var Integer -- latest age to start payments
    -- bindingPeriod actually defines retirementAge,
    -- so rA may not be needed!?
    noOfMonthlyPayOutRates: var integer
    -- en pensionsopsparing udbetales typisk i 10 årlige rater

```

Account:

```

SavingsAccount
CheckingAccount,
CreditAccount
    (limit, non-positive, extraInterestRate,
     paymentDate/period,
PaymentAccount
BusinessAccount

```

Hvis der i Account findes en virtuel metode, som banken gerne vil have udført for alle deres accounts, da ville vi have et eksempel som svarer til Record, Person, ... Det er dog for enkelt bare at summere balance for alle accounts, til det behøver man ikke nogen virtuel.

Nordea:

- GrundKonto
- GrundKonto med kredit
- Lønkonto+ til unge under 18 år
- Kassekredit til unge
- Ungdomskonto 13-17 år
- Juniorkonto 0-12 år

denne her bank har forskellige typer af checking accounts and saving accounts

<https://www.calbanktrust.com/personal/>

Eiffel i denne her

https://www.eiffel.org/doc/eiffel/ET-_Inheritance

bruger Account og SavingsAccount og senere BusinessAccount

University System

```
class Affiliate:
  -- the superclass for employees and students,
  but is Affiliate a good name?
  ++ BMP: nok desværre ikke, men vi kan jo bare bruge Person
class Employee: Affiliate
  titel, salary, unit, computeSalary/updateSalary,
  markVacation,
class Faculty: Employee -- VIP
  research area, publications, addPublication,
  courses, researchGroup
class TAP: Employee
  tasks, ...
class Student: Affiliate
  name: val String   måske i Affiliate
  studId: val integer
  coursesTaken: obj List(Course)
  coursesFailed: obj List(Course)
  coursesActive: obj List(Course)
  passed(C: ref Course):
    ...
  ...

  enrollments: obj Set(Enrollment)
  +++ BMP: dette er ikke noget jeg har fundet på, taget fra nettet
  det er det ord som bruges, og grade er typisk der

  signUpForCourse(c: ref Course, s: var Semester):
    c.enroll(this(Student), s)

class Enrollment(s: ref Student, c: ref Course, s: var Semester):
  grade: var Character -- A-F +++ kan også være A+, mm.

class Course:
  ECTS: val Integer
  teacher: ref Faculty -- måske Set(Faculty)
  required: obj Set(Course)
  students: obj Set(Enrollment)

  enroll(stud: ref Student, sem: var Semester):
    -- Check if required courses have been passed
```

```

er : ref Enrollment
ok: var boolean
ok := true
for each required course rc in required
    for each enrollment er in stud.enrollments
        if not(er.has(rc)) then ok:= false
        if (er.grade = 'F') then ok:= false
if ok then
    students.insert(Enrollment(stud, this(Course), sem)

```

+++ enroll er desværre ikke nogen virtuel metode

+++BMP: med disse subklasser har vi 3 levels, men ingen virtuel!

```

class Bachelor: Student
    requiredECTS var Integer
    project: ref BachelorProject
    dateOfDegree: var Date
    ...
class Master: Student
    requiredECTS var Integer
    project: ref MasterProject
    bachelorDegree: ref Bachelor
    ...
class PhD: Student
    project: ref PhDProject
    masterDegree: ref Master
    ...

```

```

Fra Xiaoping Jia
Student
    Nondegree
    Undergraduate
    Graduate
        Master
        PhD

```

Student, course, bachelor, candidate, PhD, Examines (characters, 'A', .., 'F')

E-bookstore

Xiaoping Jia: OO software development using Java

```

EBookstore, Estore, DigitalStore
Customer
Item
    eBook
    MusicCD
    Software
ShoppingCart
Order

```

```
OrderItem
  I: Item
  quantity: var integer
Payment
Address
```

BlueJ: They start with an example of a social media for displaying Posts.

The hierarchy is as follows:

```
Post
  CommentedPost
  MessagePost
  PhotoPost
  EventPost
```

They also shortly mention the Mammal, Animal, etc example.
And shortly a game example (from chap 8) with Room, TransporterRoom.

In chap. 16 BlueJ has an example of a taxi company

```
TaxiCompany:
Vehicle
  Taxi
  Shuttle
```

We may add limosine as a subclass.

It would be nice if there are subclasses of one or more of Taxi, Shuttle and Limosine.

Vehicle licenseFee

a la dimension/units

```
class Insurance:
  premium: var Real
  noOfDays: var Integer  -- number of days the vehicle
                        -- has been insured this year
  ...
```

```
class Vehicles:
  noOfVehicles: var Integer
  : var Real
  ...
```

```
-- alternativt kan man lage en virtuel klasse:
```

```
class VehicleInsurance:< Insurance
```

```
  kmLimit: var Real
```

```
noOfDays: var Integer -- number of days the vehicle
-- has been insured this year
```

```
class Vehicle(licenseNumber: ref String)
  weight: var Integer
  ins: ref Insurance
  licenseFee:<
    fee: var Real
    inner(licenseFee) -- calculate fee for different Vehicles
    -- add fee to insurance premium
    ins.premium := ins.premium + fee

    noOfVehicles := noOfVehicles + 1
class Car: Vehicle
  ...
motorBikes: obj Vehicles
```

```
class VehicleInsurance::
```

```
...
```

```
class MotorBike: Vehicle
  licenseFee::
    fee := XXXXXXXXXX*ins.noOfDays
```

```
...
```

```
fuelCars: obj Vehicles
  weightLimit: var Integer
  lightCarFee, heavyCarFee: var Real
```

```
...
```

```
class VehicleInsurance::
```

```
...
```

```
class FuelCar: Car
  licenseFee::
    if (weight > weightLimit) :then
      fee := heavyCarFee*ins.noOfDays
    :else
      fee := lightCarFee*ins.noOfDays
```

```
electricCars: obj Vehicles
```

```
class VehicleInsurance::
```

```
...
```

```
class ElectricCar: Car
```

```

    vlicenseFee::
        fee := [redacted]*ins.noOfDays

```

```

taxies: obj Vehicles
[redacted]
[redacted]

```

```

class VehicleInsurance::
    ...
class Taxi: Car
    licenseFee::
        fee := [redacted]*ins.noOfDays

```

Every year the rates are changed, the figures are for2023:

```

taxies.feeRate := 1,42
motorBikes.feeRate := 6,10
fuelCars.weightLimit := 7500
fuelCars.lightCarFee := 8,38
fuelCars.heavyCarFee := 9,84
electricCars.feeRate := 8,38
...

```

```

class VehicleRegister:
    ...
    registrations: obj Set(Registration)
    class Registration:
        theVehicle: ref Vehicles.Vehicle
        theOwner: ref Customer
    ...
    registerTaxi(owner: ref Customer, licenseNo: ref String):
        reg: ref Registration
        reg := Registration
        reg.theVehicle := taxies.Taxi(licenseNo) [redacted]
        reg.theOwner(owner)
        registrations.add(reg)

```

```

number of taxies: taxies.noOfVehicles
number of motor bikes: motorbikes.noOfVehi

```

Instead of a Vehicle Registration System, we may use a Vehicle Rental System, which perhaps have more methods? Da slipper vi at forholde os til hvordan licenseFee beregnes.

Record, Person, Employee, Student, Book

Organization: Company,

Figure, Polygon, Circle, Square, Rectangle, Rombe, ...

Library, Book, Fiction, Non-fiction, ScienceFiction, Novels, Poem, ShortStory, Person, Loan,

Publications, EndNote, Journal, Conference, TechnicalReport, Book,

Kirkerud: Place, MountainTop, House, Town, DetachedHouse, OfficeBuilding

Kirkerud indfører ***syntax diagrams*** (side 26) fra starten

Budd (187-206): Super:CardPile, sub:SuitPile, DeckPile, DiscardPile, TableauPile, ...

It is a quite complex and big example that will take time to understand and probably too big for this chapter. But might be a candidate for a large example!?

Rumbaugh: super:Equipment, sub:Pump, HeatExchanger, Tank, Centrifugal Pump, ..., SphericalTank, ...

Animal, Mammal, Cat, Dog, ... — men hvad skal det bruges til?

let a = 7, and b = 8 in (a < b) <= (a = b) and (a > b)

[a,b] (a < b) <= (a = b) and (a > b) [13,45]

exp

relExp

Settlement may include [hamlets](#), [villages](#), [towns](#) and [cities](#).

```
class Settlement: Graph
  house:< ...
  road: ...
  noOfInhabitants: var integer
  name: var String
class Village: Settlement
  house:: ... ?
```

Adresser, stoppesteder, bus/trick/ruter, ruteplanlægning, ...

Biler, adresser, ...,

Offentlig transport

Gå, cykle, ...,

Travel leg, ...

```
class Country:
  class Route:
    stops: obj list(Links)
  class BusStop: -- Node
    loc: ref Location/Address
  class Link: -- Link / Leg
    start,end: ref BusStop
    distance: var Real

class Graph:
  class Node:<
    firstEdge: ref Edge
    insertEdgeTo(to: ref Node)-> ref Edge:
      ...
    display:<

  class Edge:<
```

```
from, to: ref Node
nextEdge: ref Edge;
deleteMe:<
display:<
    inner(display)
```

```
class RoadAndCityGraph: Graph
```

```
    class Node::<
```

```
        cityName: ref String
```

```
        ...
```

```
        display::
```

```
            ... name of the city
```

```
        firstEdge.roadLength
```

```
    class Edge::<
```

```
        roadLength: var Integer
```

```
        ...
```

```
        display::
```

```
            ... length of the road
```

```
        from.cityName
```