# OBJEKT-ORIENTERING I TEORI OG PRAKSIS

*Af Nicki T. Hansen, Aarhus Universitet 2006.*
*Årskortnummer: 20030605*
*Mail: gwiz665@gmail.com*
*Antal tegn: xxxx*

## INDHOLDSFORTEGNELSE



## INTRODUKTION

Jeg vil i denne opgave kigge nærmere på objekt-orienteret programmering (OOP) i teori og praksis. Jeg vil først forklare de helt basale ideer ved OOP abstrakt, og vise fordelene ved at bruge dette paradigme. Herefter vil jeg, i praksis, implementere, eller i hvert fald skitsere en implementering, af et ordinært spil kort ved brug af OOP.

## TEORI

Det helt basale i OOP, er at man bruger to koncepter kaldet *objekter* og *klasser*. Et objekt er karakteriseret ved tre egenskaber:

- Tilstand
- Opførsel
- Identitet

Et objekts *tilstand* (=”state”) er alle de værdier der bliver gemt i objektet. Et objekts tilstand kan ændres ved at køre funktioner på objektet, der ændrer denne tilstand. Denne tilstand kan have en indflydelse på hvordan objektet opfører sig ved givne funktioner. Tænk f.eks. på et bil-objekt, dets benzintank kan være tom eller ikke-tom, og alt afhængig om det er det ene eller det andet, så kan bil-objektet køre eller ikke.

Et objekts *opførsel* er alle de interne funktioner (også kaldet *metoder*), som objektet understøtter. Et givent objekt kan understøtte visse metode, men altid kun et udsnit af alle mulige. En bil kan f.eks. køre, men ikke flyve – dvs. at et bil-objekt må have en funktion der hedder `kør()`, men ikke en der hedder `flyv()`.

Et objekts tilstand og opførsel er dog ikke nok til at definere et objekt, da der på samme tid kan eksistere to objekter med identisk tilstand og opførsel, som er forskellige. Det er her et konceptet *identitet* kommer ind i billedet. To biler kan være identiske på alle måder, men det er stadig ikke den samme bil.

Definitionen på et objekt er altså, en enhed med en tilstand, en opførsel og en identitet.[1]

En *klasse* kan ses som en form man kan lave objekter ud fra. En klasse er en måde at gruppere lignende objekter i et overskueligt system. Objekter der bliver skabt ud fra en given klasse

[1] En interessant betragtning, er at ud fra denne definition er alle ting i den virkelige verden objekter, hvilket på en måde, også er korrekt.

understøtter de samme interne funktioner og har samme mulige tilstande, og derfor skal en klasse definition, definere to ting:

1. De funktioner der er tilladt i objekter fra denne klasse
2. De mulige tilstande objekter fra klassen kan være i.

Hvis vi kigger på et kort spil, kan man således lave en `Kort` klasse, der indeholder alle mulige tilstande for kort i et givent spil (kulør og nummer) og de relevante metoder der skal bruges på kortet, f.eks. for at få fat i nummer eller kulør.

To vigtige fænomener ved OOP er *indkapsling* og *nedarvning*. Indkapsling er at et givent objekt ikke afslører noget af sin interne algoritmer for omverdenen. Dvs. hvis jeg kalder et kort-objekt om hvilket kort det er, så kender jeg kun de mulige svar der kan komme – jeg ved ikke nødvendigvis hvordan objektet har gemt informationen eller hvordan den laver sit svar. Objektets interne virkemidler er skjult for omverdenen, og vi kan kun stille spørgsmål, i form af metoder, og få svar.

Nedarvning er en måde at forenkle klasser, ved at lade en ”forælder” inkludere den overordnede opførsel og tilstand for en familie af klasser, hvor et ”barn” af den, så vil nedarve opførsel- og tilstandsdefinitionen fra den overordnede klasse. Det smart ved dette, er at hvis der er mange metoder og gemte informationer der går igen i klasser, kan man sætte dem i familie, og forenkle hver klasse, og spare kode. Et eksempel: hvis vi skal implementere et system der kan håndtere en cykel og en bil, kan vi med fordel lave en klasse der hedder `Køretøj` (som kan indeholde basale ting som `hastighed`, `hjul`, `farve`) og lade `Bil` og `Cykel` klasserne nedarve disse. `Bil` klassen kan så selv indeholde mere specifikke informationer og metoder, så som `motor`, `hestekræfter` og `start()`, hvor Cykel klassen kan indeholde `ringeklokke()`, `antalSadler`, osv. Et givent bil-objekt der er skabt ud fra Bil klassen, vil så indeholde den samlede opførsel og tilstand fra Bil klassen og Køretøj klassen.

**PRAKSIS**

Hvis vi i praksis skal implementere et spil kort i Python, må vi først vide konkret hvad et kort spil er, funktionsmæssigt, og hvad vi skal kunne med det.

Et kort spil er 52 forskellige kort, og et kort kan have følgende tilstande: ansigt (hjerter, spar, ruder, klør) og nummer (1-13, hvor 1 er Es og 10-13 er billedkortene Knægt, Dame og Konge). Allerede nu kan vi definere en kort klasse, da vi skal have 52 forskellige kort, som alle deler mulige tilstande og har ens opførsel.

```
class Kort:
  def __init__(self, face, number):
    self.face = face
    self.num = number
```

Den første metode er det man kalder en *konstruktør*. Den bliver kørt hver gang man skaber et nyt objekt ud fra denne klasse. Pga. *scope* kan vi inde i konstruktøren kalde `face` og `number` uden at definere hvad de hører til. Vi ser at de to værdier bliver gemt i hhv. `self.face` og `self.num`. *Self* er en speciel type, som betyder at objektet hvor det er i kan pege på sig selv. Hvis vi så skaber 52 forskellige objekter ud fra denne klasse, vil de alle gemme de passende værdier i sig selv. Jeg definerer nu at face skal være et nummer mellem 0 og 3, som svarer til listen [hjerter, spar, ruder, klør].

Udover konstruktøren skal vi have nogle *accessor* metoder, som er metoder der henter information fra det objekt de hører til, og returner det til hvad der nu kalder det. Der er flere der kan være relevante, men i denne tekst vil jeg kun vise en.

```
  def getFace():
    faceList = (”hjerter”, ”spar”, ”ruder”, ”klør”)
    return faceList[self.face]
```

I dette tilfælde er det ikke nødvendigvis relevant, men i de fleste tilfælde er der også behov for tilsvarende *mutator* metoder, som er metoder der ændrer informationen i det objekt de hører til. Hvis nu man vil ændre et kort fra hjerter til spar, kan man lave følgende metode.

```
  def setFace(faceNumber):
    self.face = faceNumber
```

I mutatoren kan det ses at jeg giver en parameter med, som er som skal udskifte den gamle information, og at metoden ikke returnerer noget, hvilket er typisk for mutatorer.

Udover Kort klassen, skal vi også definere en Deck klasse, som kan holde et helt spil kort. Det gøres således.

```
class Deck:
  def __int__(self):
    self.deckList = []
    for i in range(4):
      for j in range(13):
        self.deckList.append(Kort(i,j))
```

Her defineres et deck til at være en samlet liste af 52 forskellige kort af alle tilstandsmuligheder defineret i Kort klassen.

## KONKLUSION OG AFSLUTTENDE KOMMENTARER

I de foregående få sider har jeg ridset op hvad objekt-orienteret programmering basalt set handler om, og ved brug af praktiske eksempler har jeg vist nogle af fordelene ved at bruge det i forhold til andre paradigmer.

## LITTERATURLISTE

Brookshear, J. Glenn (2005): *Computer science – An overview*, 9th edition, Pearson.
Guzdial, Mark (2005): *Introduction to computing and programming in python*, Pearson.
Horstmann, Cay (2004): *Object-oriented design & patterns*, John Wiley & Sons, Inc.