Introduktion till UML, OOAD & OOP, del 1

Data- och informationsvetenskap: Objektorienterad programmering och modellering för IA

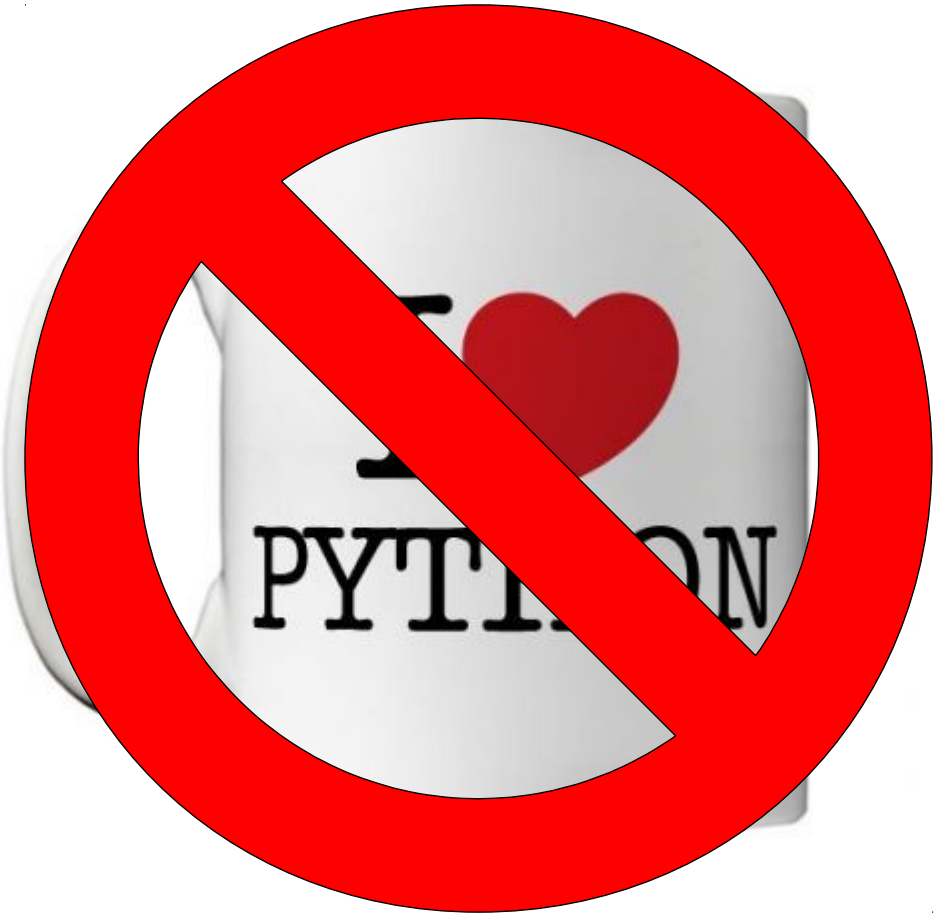
Johan Holmberg 2019-09-04

MALMÖ UNIVERSITET

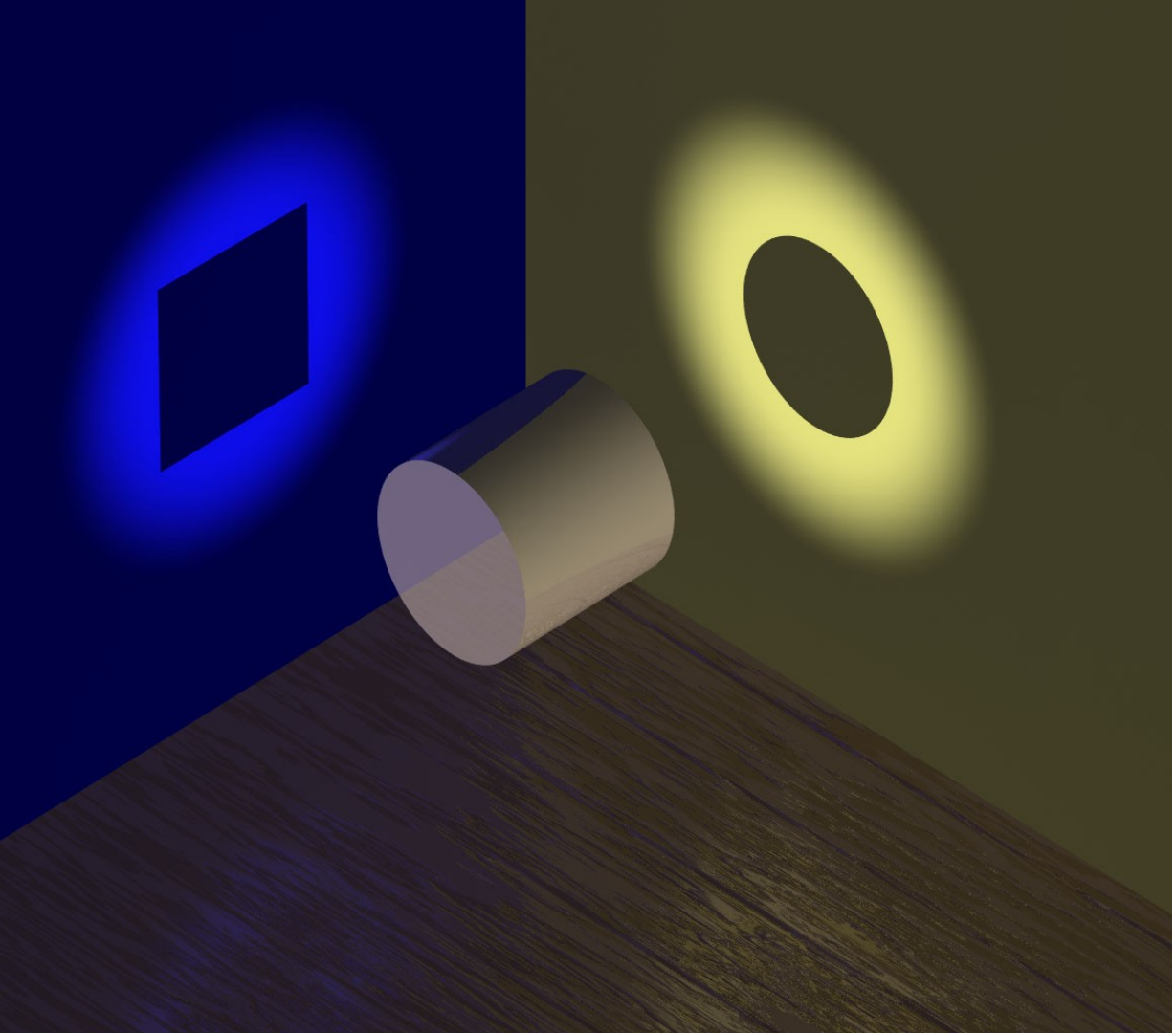
Dagens agenda

- Förra föreläsningen
- Objekt-orienterad utveckling
- Vad är ett objekt?
- Vad är en klass?
- Relationer
- Speciella klasstyper
- Mer UML

MALMÖ UNIVERSITET



MALMÖ UNIVERSITET



Programmerings-paradigmer:

Olika sätt att beskriva världen

MALMÖ UNIVERSITET

Världen består av objekt

Ett fjärde sätt att beskriva världen är att tolka allt som objekt – en stol, en fågel eller en tanke. Dessa objekt har alla olika **egenskaper** och kan **interagera med varandra**.

En programmerare som arbetar med med objekt spenderar mycket tid på att **modellera** och **analysera** innan någon kod skrivs.

Om du som programmerare ser världen som en samling av objekt har du antagit en **objekt-orienterad** världsbild.

MALMÖ UNIVERSITET

Vad är ett objekt?

Filosofen Charles S. Pierce definierade ett objekt som någonting som vi kan tänka eller tala om. I vårt dagliga språk motsvaras detta av våra subjektiv och pronomen.

Inom objekt-orienterad programmering är ett objekt en identifierbar samling data, knuten till funktioner som kan användas på denna data.

Man säger att objektet har **attribut** (data) och **metoder** (funktioner).

Bakgrund

I tidigare kurser har ni arbetat med variabler.

Vi kan exempelvis beskriva en kurs med hjälp av Python på följande vis:

```
university_location = "Malmö"
number_of_teachers = 2

teacher1 = {
"name": "Johan",
"age": 38,
"department": "IT"
}
teacher2 = {
"name": "Anton",
"age": 30,
"department": "DVMT"
}

print(teacher1)
print(teacher2)
print(university_location)
print(number_of_teachers)
```

MALMÖ UNIVERSITET

Ett lite bättre sätt

Med objekt i Python!

Ponera att vi vill få ut åldern från de båda variablerna. Hur gör vi det?

```
print(friend2['age'])
```

```
friend1 = {
"name": "Romina",
"gender": "female",
"age": 38
}

friend2 = {
"name": "Anton",
"gender": "male",
"age": 30
}
```

MALMÖ UNIVERSITET

Ännu bättre: objekt-orienterad programmering

Ett sätt att beskriva objekt **från verkliga världen** i kod är att beskriva dem som **instansierade objekt**!

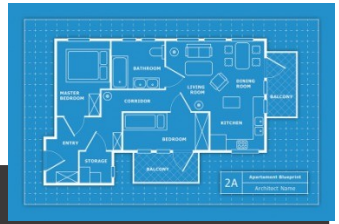
+

```
first_friend = Friend("Romina", "female", 38)
second_friend = Friend("Anton", "male", 30)
```

Klass = Mall

Objekt = Instanser av mallen

MALMÖ UNIVERSITET



```
class Friend(object):
'''Det här är mallen som beskriver en vän'''

def __init__(self, friend_name, friend_gender, friend_phd):
'''Den här metoden anropas när en ny vän skapas'''
self.name = friend_name
self.gender = friend_gender
self.phd = friend_phd

def __str__(self):
'''Den här metoden anropas när en vän skrivs ut'''
return "This is {n} and their gender is {g}.".format(n=self.name, g=self.gender)
```

MALMÖ UNIVERSITET

Objekt-orienterad utveckling

MALMÖ UNIVERSITET

Modellering av världen som objekt

I ett objekt-orienterat mjukvarusystem är allt uppdelat i olika **objekt**. Dessa objekt är oftast modellerade för att återspegla objekt i verkliga världen (tänk *fisk*, *matsalsbord*, *student*), men kan även modelleras efter rent abstrakta konstruktioner (exempelvis *kurs*, *tanke*, *bokningsprocess*).

Att göra denna analys på ett bra sätt är grunden i den objekt-orienterade analysen och designen.

Grundläggande OOP-koncept

Inom objekt-orienterad programmering stöter du på ett grundläggande koncept som du måste behärska. Dessa är:

- Objekt och klasser
- Meddelanden
- Inkapsling
- Komposition
- Arv
- Polymorfism

MALMÖ UNIVERSITET

Objekt och klasser

Objekt-orienterade språk bygger på användandet av **objekt** och **klasser**, vilka förenklat utgör mallar för skapandet av objekt.

Klasser är oftast designade för att utökas med hjälp av **arv**. På detta sätt minskas den totala mängden kod i ett program och koden kan, *teoretiskt sett*, enklare återanvändas i flera projekt.

Meddelanden

Utbyte av information mellan objekt i ett objekt-orienterat system sker med **meddelanden**. I praktiken implementeras detta oftast (men inte alltid) genom att ett objekt anropar en **metod** hos ett annat objekt. Meddelandet kan bestå av noll eller flera **parametrar** eller **argument**.

MALMÖ UNIVERSITET

Inkapsling

En bärande idé inom OOP är den av **inkapsling** av **data** och **funktionalitet**. Detta innebär att datan och funktionaliteten är skyddad från utomstående inblandning. Genom att använda inkapsling kan du som programmerare garantera att ett objekts data är giltig.

För att komma åt denna data och funktionalitet i externa klasser tvingas du som programmerare att använda de fördefinierade vägar in som definierades vid skapandet av klassen i fråga.

Inkapsling i Python

Python saknar många av de språkliga konstruktioner för att kapsla in data och funktionalitet som finns i många andra objekt-orienterade språk. Vi löser detta problem genom att följa **konventioner** och visa **god vilja**.

MALMÖ UNIVERSITET

Komposition och aggregation

Genom att låta en klass använda sig av andra klasser kan mer avancerade objekt modelleras utan att varje klass måste innehålla mängder med kod. Exempelvis kan en klass *Bil* känna till klasserna *Ratt*, *Hjul* och *Motor.* Begreppet **komposition** kan tolkas som att ett objekt *består av* ett eller flera andra, medan begreppet **aggregation** kan tolkas som att ett objekt *har ett* eller flera andra objekt.

Arv

För att inte behöva upprepa kodskrivandet, kan en klass **ärva** funktionalitet och datadefinitioner från en annan klass. Klassen som *ärver* kallas **subklass** och den klass som *ärvs* kallas **superklass**.

Ett exempel på arv skulle kunna vara en superklass *Sittmöbel*, som har ett antal sittplatser och en dyna. Två möjliga subklasser skulle då kunna vara *Stol* med en sittplats och *Soffa* med tre sittplatser.

Polymorfism

En subklass kan välja att **skriva över** funktionaliteten, eller metoderna, hos sin superklass. För en utomstående klass är detta inte synligt, vilket gör att medan gränssnittet för de båda klasserna är likadana, kan funktionaliteten skilja sig åt.

Ett exempel:

Polymorfism, forts.

Vi har en klass *Viking* som har en metod *hälsa*. När vi anropar *Viking.hälsa*, kommer vikingen att svara “góðan morgin”. Vi introducerar sedan två nya klasser, *Dansk* och *Svensk*, som ärver metoden *hälsa*, men definierar om innehållet lite. När vi nu anropar *Dansk.hälsa* kommer dansken att svara “god morgen”, medan *Svensk.hälsa* ger oss “god morgon”.

UML

Vad är UML?

- UML (Unified Modeling Language) är ett modelleringsspråk som används för att beskriva objekt-orienterade mjukvarusystem.

  Språket beskriver ett antal diagramtyper. Dessa diagramtyper kan användas för att beskriva ett mjukvarusystems:

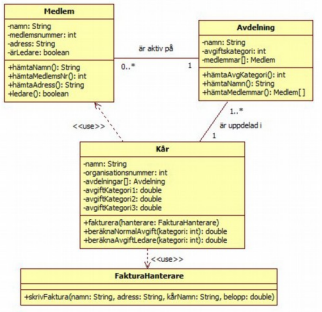
  - Strukturer
  - Beteenden
  - Interaktioner

Vad är UML?

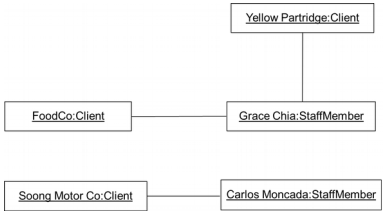
UML är ett standardiserat språk (*ISO/IEC 19501:2005*) – varje detalj i ett diagram har en mening. Försök att hålla koll på dessa för att kunna kommunicera mer exakt med dina kolleger, kunder och andra.

UML har 25 år på nacken, men uppdateras ständigt – den senaste versionen släpptes 2015.

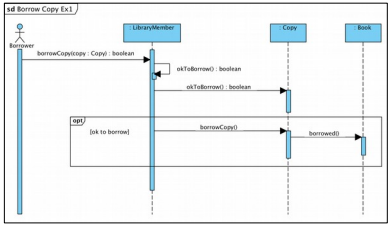
Diagramtyper



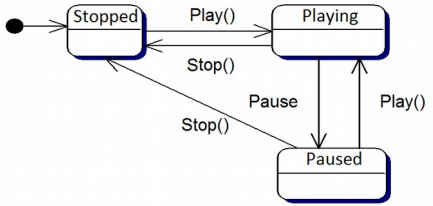
Klassdiagram



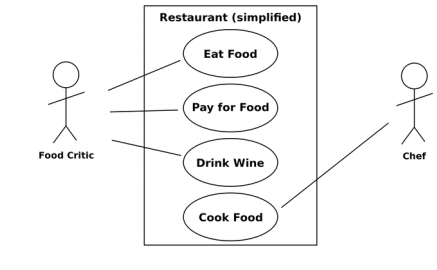
Objektdiagram



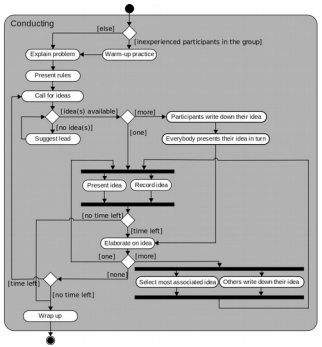
Sekvensdiagram



Tillståndsdiagram



Use case-diagram



Aktivitetsdiagram

MALMÖ UNIVERSITET

Klassdiagram

Ett klassdiagram beskriver strukturen hos ett mjukvarusystem, eller delar av ett sådant system.

Klassdiagrammet visar vilka **klasser** som ingår i systemet, deras **attribut** och **metoder**, och deras **relationer** till varandra.

Bra UML-resurser online

UML Diagrams.org (https://www.uml-diagrams.org/)

- En bra beskrivning av språket UML

Umbrello (https://umbrello.kde.org/)

- Ett bra verktyg för klass- och use case-diagram

Sequence diagram.org (https://sequencediagram.org/)

- Ett bra online-verktyg för att göra sekvensdiagram

Snälla, snälla student!

Du kommer vid något tillfälle lockas att använda **Lucidchart** för att rita dina UML-diagram eller lära dig mer om UML. Undvik dem, de har inte alltid riktig koll på hur UML verkligen ska användas och kommer i slutänden bara att förvirra dig.

MALMÖ UNIVERSITET

Vad är ett objekt?

MALMÖ UNIVERSITET

Definitionen av ett objekt

Ett **objekt** är en **variabel** (en **primitiv**, **datastruktur**, **funktion** eller liknande) som är **sparad i minnet** och är **identifierbar**.

I en del språk, som exempelvis Python, är alla datatyper objekt.

Definitionen av ett objekt, forts.

Objekt håller oftast (men inte alltid!) två eller fler värden. För dig som Python-kunnig kan du tänka dig ett objekt som en dictionary:

```
thisdict =  {
    "brand": "Ford",
    "model": "Mustang",
    "year": 1964
}
```

MALMÖ UNIVERSITET

Ett objekts livslängd

Ett objekt kan **skapas** och **förstöras**. Man säger att objektets **livslängd** avgränsas av dessa båda händelser.

Objekt skapas oftast **explicit** genom att anropa en **konstruktor**, men kan även skapas på andra sätt.

Ett objekts livslängd, forts.

Ett objekt förstörs antingen när det **avrefererats** och samlas upp av minneshanteraren (se nedan) eller när det **avallokeras** med en **destruktor**.

I språk som använder automatisk minneshantering (som exempelvis Python, Java och JavaScript) behöver du som programmerare inte explicit förstöra objekt. I språk som inte gör det, som exempelvis C++, måste du göra detta själv.

Typer

De allra flesta programmeringsspråk använder sig av **typer** för att skilja mellan olika typer av data. Vilka typer som finns skiljer sig åt mellan språken.

I ett språk som Python är alla typer i grunden **objekt**. Genom att skapa **klasser** (som vi gör senare idag) kan vi skapa egna typer och skilja på dessa typer.

Typer, forts.

I många språk, exempelvis Java, används **hård typning**, vilket innebär att en variabel bara kan vara av en specifik typ.

Python (och JavaScript) är exempel på **löst typade** språk. I dessa språk kan variabler växla typ. Vi har fortfarande nytta av typerna, dock. Till detta återkommer vi senare.

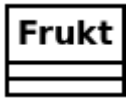
Vad är en klass?

MALMÖ UNIVERSITET

Definitionen av en klass

En **klass** är enkelt uttryckt en bit kod som definierar en samling **attribut** och **metoder**. Klassen kan **instansieras** till **objekt**.

Som tidigare nämnts kan klassen även ses som en **egendefinierad typ**.



← Så här modelleras en klass i UML

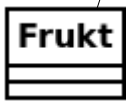
MALMÖ UNIVERSITET

Definitionen av en klass i Python

```
class Frukt(obj):
    pass
```

Berättar att Frukt ärver från object.
I Python 3 behövs den inte, men
vi behåller den i det här exemplet.

Samma namn

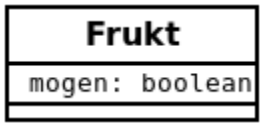


Eftersom vi inte vill att
klassen faktiskt ska göra
något just nu.

MALMÖ UNIVERSITET

Attribut

Attributen definierar de variabler som en klass kan hålla reda på. De kan vara av vilken typ som helst – även andra klasser.



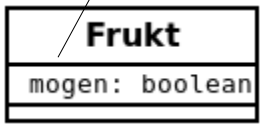
← Så här modelleras ett attribut i UML

Attribut i Python

```
class Frukt(obj):
    mogen = False
```

Samma namn

Attributets fördefinierade värde. Om detta inte skrivs över, sägs värdet vara *statiskt*.



MALMÖ UNIVERSITET

Metoder

Metoderna kapslar in den funktionalitet som är knuten till klassen. Genom att anropa ett objekts metod, kan andra objekt be det aktuella objektet att utföra en handling. Metoder är helt enkelt **funktioner** som är knutna till en speciell klass.

← Så här modelleras en metod i UML

MALMÖ UNIVERSITET

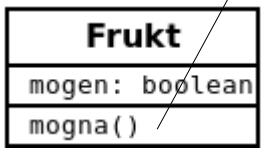
Metoder i Python

```
class Frukt(obj):
    mogen = False

    def mogna(self):
        self.mogen = True
```

Samma namn

Det här värdet måste alltid skickas med till en metod

Här förändrar vi tillståndet på attributet *mogen*.



MALMÖ UNIVERSITET

Metoder

En metod kan ta noll eller flera **parametrar** eller **argument**.
Den kan även ge ett **returvärde**.

I exemplet nedan tar metoden *märk()* emot ett argument *klisterlapp* av typen *Märke* och returnerar ett booleskt värde.

← Så här modelleras en metod i UML

MALMÖ UNIVERSITET

Metoder i Python

```
class Frukt(obj):
    mogen = False

    def mogna(self):
        self.mogen = True

    def märk(self, klisterlapp: Märke):
        return True
```

Här berättar vi att *klisterlapp* ska vara av typen *Märke*.

Samma namn

Här returnerar vi en boolean.

Frukt
mogen: boolean
mogna() märk(klisterlapp:Märke): boolean

MALMÖ UNIVERSITET

Statiska metoder

En del klassmetoder kan användas utan att ett objekt först skapats. Dessa metoder kallas för **statiska** och kan inte påverka objekt av klassen.

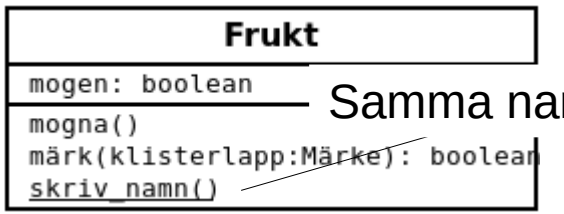
← Så här modelleras en statisk metod i UML

MALMÖ UNIVERSITET

Statiska metoder i Python

```
class Frukt(obj):
    mogen = False

    def mogna(self):
        self.mogen = True

    def märk(self, klisterlapp: Märke):
        return True

    @staticmethod
    def skriv_namn():
        print("Jag är en frukt")
```

Vi markerar metoden som statisk med hjälp av en annotering.



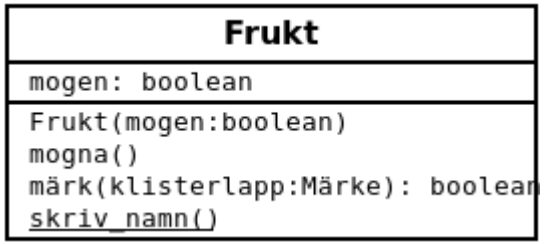
Samma namn

Statiska metoder är inte knutna till objekt, så *self* behövs inte här.

MALMÖ UNIVERSITET

Konstruktorn

Konstruktorn är en särskild typ av metod som används för att **instansiera** en klass till ett objekt. Den kan som alla andra metoder ta emot noll eller flera argument.



← Så här modelleras en konstruktor i UML

MALMÖ UNIVERSITET

Konstruktorn i Python

Här anger vi att inparametern ska vara ett booleskt värde.

```
class Frukt(obj):
    mogen = False

    def __init__(self, mogen: bool):
        self.mogen = mogen

    def mogna(self):
        self.mogen = True

    def märk(self, klisterlapp: Märke):
        return True

    @staticmethod
    def skriv_namn():
        print("Jag är en frukt")
```

Inte samma namn, så här definieras en konstruktor i Python

Så här kommer vi åt ett attribut inifrån klassen själv

Frukt
mogen: boolean
Frukt(mogen:boolean) mogna() märk(klisterlapp:Märke): boolean skriv_namn()

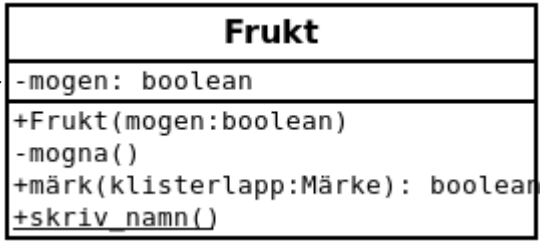
MALMÖ UNIVERSITET

Synlighet

I språk som har språkliga konstruktioner för att hantera inkapsling, kan attribut och metoder vara “osynliga” för andra objekt. Osynliga attribut och metoder sägs vara **privata** och synliga **publika**.

Minustecken betecknar **privat** åtkomst →

Plustecken betecknar **publik** åtkomst →



MALMÖ UNIVERSITET

Synlighet i Python

Python saknar konstruktioner för detta, så vi får förlita oss på konventioner och andra utvecklares goda vilja.

För att markera att ett attribut eller metod är privat, kan vi lägga ett _ framför namnet. Således:

MALMÖ UNIVERSITET

Synlighet i Python

```
class Frukt(obj):
    _mogen = False

    def __init__(self, mogen: bool):
        self.mogen = mogen

    def _mogna(self):
        self._mogen = True

    def märk(self, klisterlapp: Märke):
        return True

    @staticmethod
    def skriv_namn():
        print("Jag är en frukt")
```

Privata

Publika

Frukt
-mogen: boolean
+Frukt(mogen:boolean) -mogna() +märk(klisterlapp:Märke): boolean +skriv_namn()

MALMÖ UNIVERSITET

Klasskonstanter

Klasskonstanter är attribut hos en klass som inte förändras. De är gemensamma för alla objekt av samma klass.

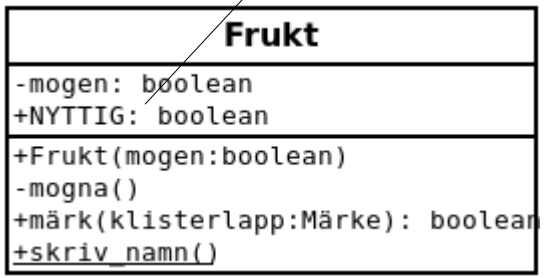
← Så här modelleras en konstant i UML

MALMÖ UNIVERSITET

Klasskonstanter i Python

```
class Frukt(obj):
    _mogen = False
    NYTTIG = True

    def __init__(self, mogen: bool):
        self.mogen = mogen

    def _mogna(self):
        self._mogen = True

    def märk(self, klisterlapp: Märke):
        return True

    @staticmethod
    def skriv_namn():
        print("Jag är en frukt")
```

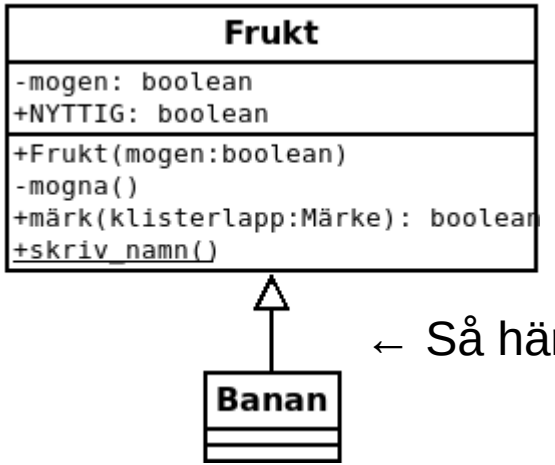
Samma namn



MALMÖ UNIVERSITET

Arv och generalisering

Arvet gör att en klass ärver egenskaper och attribut från en annan klass. Fenomenet kallas **generalisering**.



← Så här modelleras ett arv i UML

MALMÖ UNIVERSITET

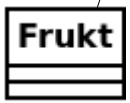
Arv och generalisering i Python

```
class Banan(Frukt):
    pass
```

Samma namn

Berättar att Banan ärver från Frukt

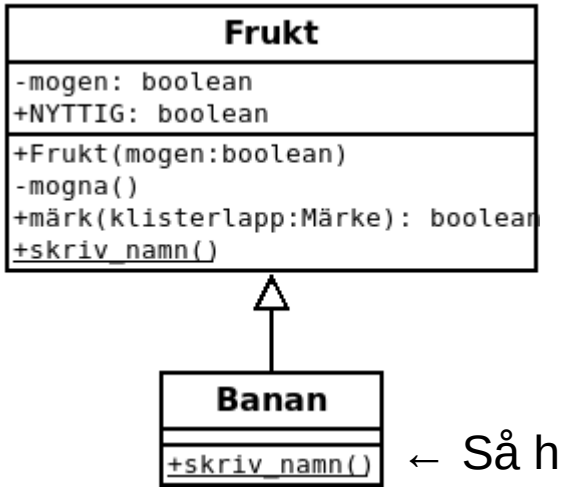
Eftersom vi inte vill att klassen faktiskt ska göra något just nu.



MALMÖ UNIVERSITET

Överlagring

En subklass kan omdefiniera beteendet hos en metod som den ärvt från sin superklass. Detta beteende kallas **överlagring**.



← Så här modelleras en överlagring i UML

MALMÖ UNIVERSITET

Överlagring i Python

```
class Banan(Frukt):
    def skriv_namn():
        print("Jag är en banan")
```

Samma namn

MALMÖ UNIVERSITET

Instansiering av en klass

En klass instansieras genom att anropa konstruktorn med eventuella argument. När detta görs skapas ett objekt i minnet som håller en kopia av alla klassens attribut. Metoderna, däremot, sparas bara i en enda kopia, så objekt i minnet blir sällan särskilt stora.

Instansiering i Python

```
min_frukt = Banan()
```

Anropar Banan-klassens konstruktor

Instansens namn

MALMÖ UNIVERSITET

Extern användning av en klass

Det finns två sätt att använda klassers metoder och utifrån:

- Från instansierade objekt (attribut och objektmetoder)
- Direkt från klassen (statiska metoder, variabler och konstanter)

Extern användning av en klass

Hos instansierade objekt varierar värdet på attributen från objekt till objekt, och resultaten från metoderna kan basera på klassens **interna tillstånd**; de beror på värdena på objektens attribut.

Statiska metoder, variabler och konstanter ger alltid samma svar, oavsett om du kommer åt dem via ett instansobjekt eller direkt från klassen.

Extern användning av en klass i Python

```
min_frukt = Frukt(True)

# Märker fruktinstansen med ett nytt märke
min_frukt.märk(Märke())

# Statisk metod, skriver ut "Jag är en banan"
Banan.skriv_namn()
```

MALMÖ UNIVERSITET

Superklasser och gränssnitt

MALMÖ UNIVERSITET

Superklasser

Superklasser är de klasser från vilka andra klasser ärver metoder och attribut. En klass som överlagrat metoder kan komma åt sin superklass originalmetoder genom att peka på superklassen.

I Python nås superklassen genom att använda funktionen *super()*.

Superklasser i Python

```
class Banan(Frukt):
    def __init__(self, mogen):
        super().__init__(mogen)

    def skriv_namn():
        print("Jag är en banan")
```

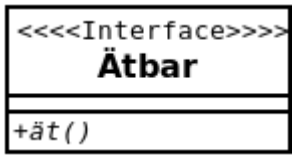
Överlagring

Här anropar vi konstruktorn hos *Frukt*

MALMÖ UNIVERSITET

Gränssnitt

I många objekt-orienterade språk kan du definiera något som kallas **gränssnitt**. De är en slags kontrakt som specificerar en uppsättning metoder som en klass måste implementera för att uppfylla.

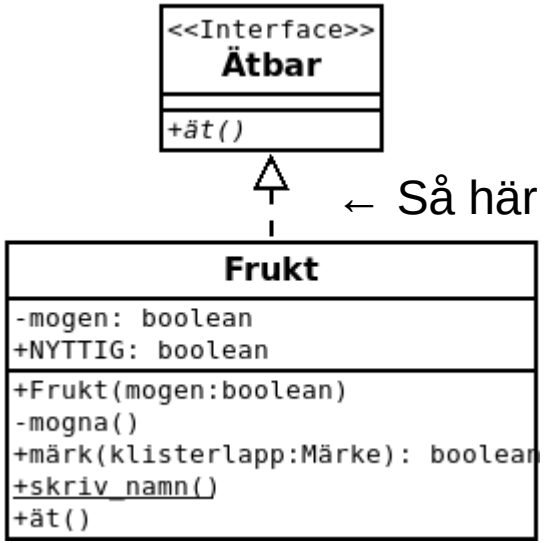


← Så här modelleras ett gränssnitt i UML

MALMÖ UNIVERSITET

Gränssnitt

När din klass uppfyller kraven som ställs av ett gränssnitt sägs klassen **implementera** gränssnittet.



← Så här modelleras implementering i UML

MALMÖ UNIVERSITET

Gränssnitt i Python

Python saknar faktiska gränssnitt, utan använder istället sig av något som kallas **metaklasser** och **annotationer**. Dessa är lite annorlunda, och lämnas utanför kursens innehåll. Är du ändå nyfiken, kan du läsa om dem på:

Interfaces and Annotations in Python3

MALMÖ UNIVERSITET

Duck typing

I löst typade språk, som Python, kan du som programmerare använda något som kallas **duck typing**. Detta innebär att du skiljer på olika typer genom att gissa vilka metoder som ett objekt har.

Huruvida detta är en bra strategi eller inte råder det delade meningar om.

Duck typing

Namnet **duck typing** kommer från idén om att “*if it quacks like a duck and walks like a duck, then it’s a duck*”. Enkelt att förstå.

Ett uppenbart problem uppstår om du jämför två objekt som är lika (exempelvis en gås och en anka). Hur hantera detta? → med typer.

Relationer

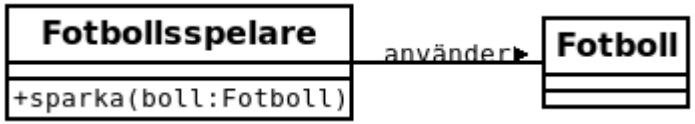
MALMÖ UNIVERSITET

Association

Två klasser som känner till och använder varandra har en **association** mellan varandra. Det finns olika typer av associationer, och de behöver inte vara dubbelriktade.

Association

Ett exempel på en association är en fotbollsspelare och en fotboll. Fotbollsspelaren känner till och använder fotbollen. Fotbollen, å sin sida, känner inte till fotbollsspelaren, men de är ändå associerade.

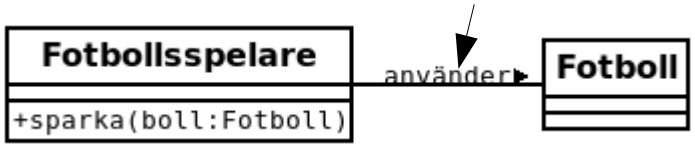


MALMÖ UNIVERSITET

Association i Python

```
class Fotbollsspelare:
    def sparka(boll: Fotboll):
        pass

class Fotboll:
    pass
```

Så här modelleras association i UML



MALMÖ UNIVERSITET

Komposition

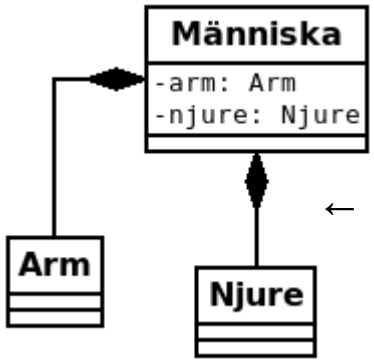
Komposition är en sammansättning där *delarna endast existerar i helheten*. Om instansen som utgör helheten tas bort från systemet tas även *instanserna av delarna bort från systemet*.

Oftast realiseras detta genom att ett objekt skapar andra objekt och håller dess referenser.

Kompositionen är **alltid asymmetrisk**.

Komposition

Ett exempel på detta är en människa, som består av exempelvis en arm och en njure:

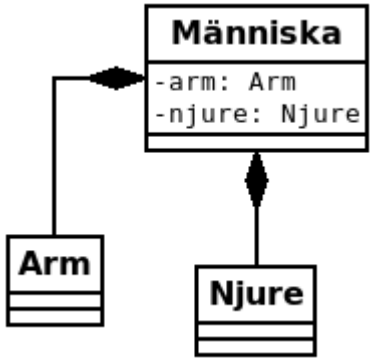


← Så här modelleras komposition i UML

MALMÖ UNIVERSITET

Komposition i Python

```
class Människa:
    def __init(self)__:
        self.arm = Arm()
        self.njure = Njure()

class Arm:
    pass

class Njure:
    pass
```



MALMÖ UNIVERSITET

Aggregation

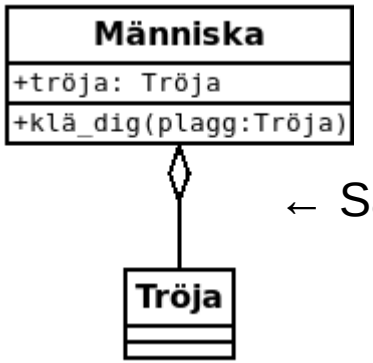
Aggregation är en sammansättning där *delarna kan existera som egna enheter i systemet* men även vara kopplade till en helhet. Om instansen som utgör helheten tas bort från systemet kan *instanserna av delarna fortfarande finnas kvar i systemet*.

Oftast realiseras detta genom att ett tilldelas andra objekt med hjälp av metoder eller konstruktorn.

Aggregationen är **alltid asymmetrisk**.

Aggregation

Ett exempel på detta är en människa som har en tröja och kan klä på sig:

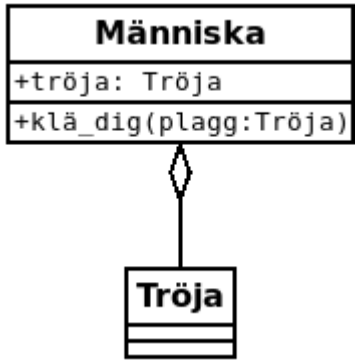


← Så här modelleras aggregation i UML

MALMÖ UNIVERSITET

Aggregation i Python

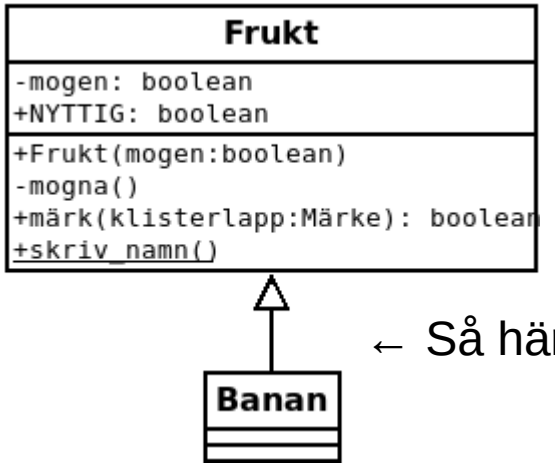
```
class Människa:
    plagg = None

    def klä_dig(self, plagg: Tröja):
        self.plagg = plagg

class Tröja:
    pass
```



MALMÖ UNIVERSITET

Arv och generalisering

Arvet som relation visar att en klass återanvänder alla metoder och attribut som definierats i en annan klass.



← Så här modelleras ett arv i UML

MALMÖ UNIVERSITET

Multiplicitet

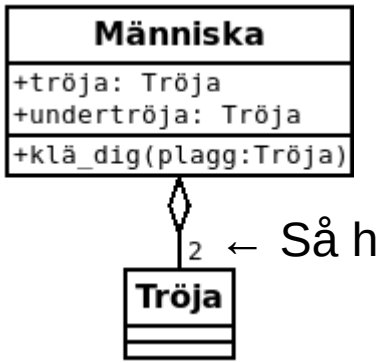
Vi kan ange hur många instanser av ett objekt som en klass känner till genom att skriva ut detta över associationerna. Vi har följande multipliciteter:

- 0, 1, n (dvs ett exakt antal)
- * (dvs ”0 eller fler”)
- + (dvs en eller fler)
- m..n (dvs ett intervall)

MALMÖ UNIVERSITET

Multiplicitet

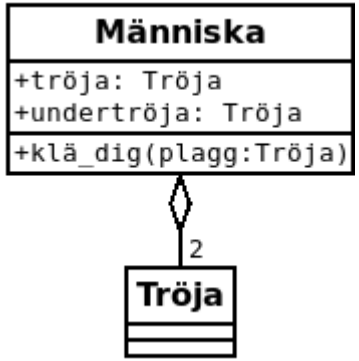
Multipliciteten anges på motsatt sida, sett från det "ägande" objektet:



← Så här modelleras multiplicitet i UML

Aggregation i Python

```
class Människa:
    plagg = None
    undertröja = Tröja()

    def klä_dig(self, plagg: Tröja):
        self.plagg = plagg

class Tröja:
    pass
```



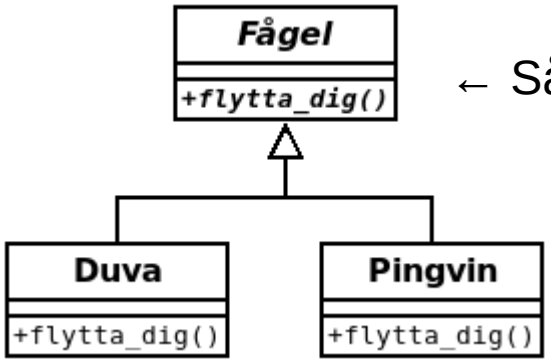
MALMÖ UNIVERSITET

Speciella klasstyper

Abstrakta klasser

I många objekt-orienterade språk kan du definiera något som kallas **abstrakta klasser**. Dessa klasser är inte tänkta att instansieras, utan snarare utökas med subklasser. De metoder som finns hos en abstrakt klass kallas för **abstrakta metoder** och saknar implementering.

Abstrakta klasser



← Så här modelleras en abstrakt klass i UML

Abstrakta klasser i Python

Python erbjuder abstrakta klasser i paketet ABC, tillsammans med annoteringen @*abstractmethod* och *pass*.

MALMÖ UNIVERSITET

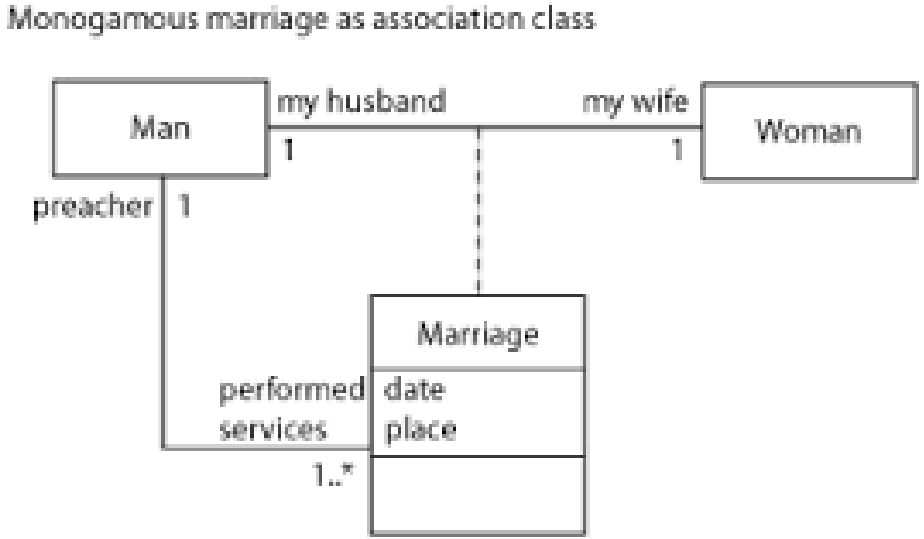
Abstrakta klasser i Python

```
from abc import ABC, abstractmethod

class Fågel(ABC):

    @abstractmethod
    def flytta_dig(self):
        pass

class Duva(Fågel):
    def flytta_dig(self):
        print("Jag flyger")

class Pingvin(Fågel):
    def flytta_dig(self):
        print("Jag simmar")
```

MALMÖ UNIVERSITET

Associationsklasser

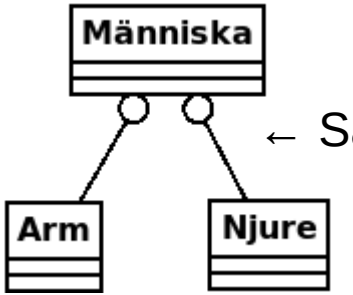
Associationsklasser är klasser som representerar en association mellan två andra klasser. Ett exempel:



MALMÖ UNIVERSITET

Inre klasser

Inre klasser är klasser som enbart är tillgängliga för en specifik klass. Dessa kan användas för att bygga upp mer sofistikerade objekt utan att oroa omvärlden om detta.

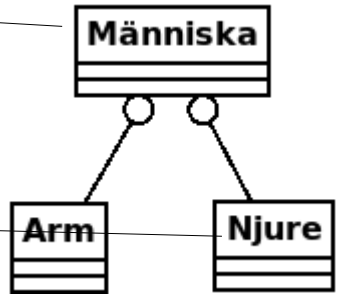


← Så här modelleras inre klasser i UML

MALMÖ UNIVERSITET

Inre klasser i Python

```
class Människa:

    class Arm:
        pass

    class Njure:
        pass

    pass
```



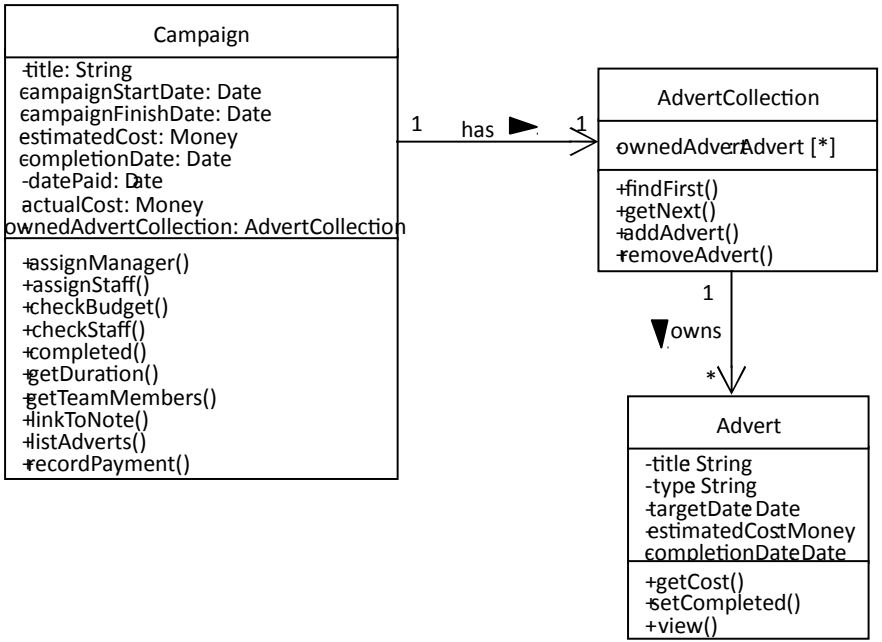
MALMÖ UNIVERSITET

Samlingsklasser

Samlingsklasser används för att skapa en behållare för objekt när ett meddelande ska skickas i ett en-till-många-association.

OO-språk har ofta någon forma av stöd för den här typen av klasser. Detta kan vara gränssnitt som implementeras av en klass eller i form av datastrukturer. En array kan ses som en mycket enkel form av stöd för detta.

Samlingsklasser



MALMÖ UNIVERSITET

Stereotyper

Stereotyper av klasser används för att skilja på olika roller som en klass kan ha i ett system:

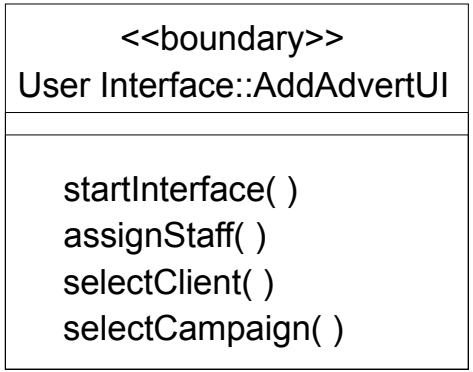
- Boundary: Objekt av den här typen av klass hanterar interaktion mellan systemet och användare (människor eller andra system).
- Control: Objekt av den här typen av klass koordinerar logiken i systemet och anropar andra objekt för att begära information eller för att få tjänster utförda.
- Entity: Objekt av den här typen av klass ansvarar för att lagra data och information samt representerar beteendet för objekt i domänen systemet representerar.

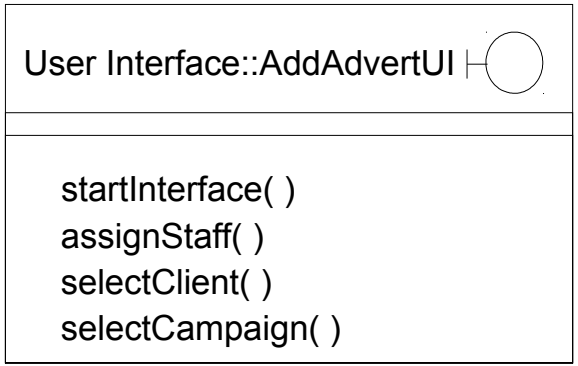
Stereotypnamn skrivs inom << och >>

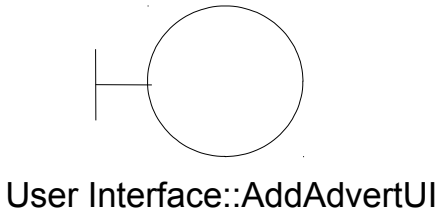
Boundary

Boundary-klasser representerar interaktion med användare och med andra system

Alternativa notationer:





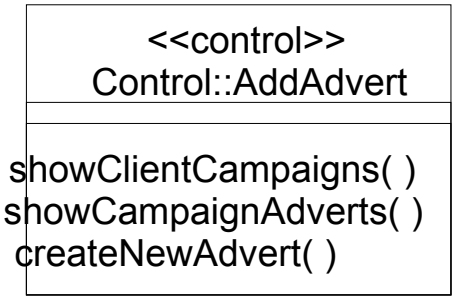


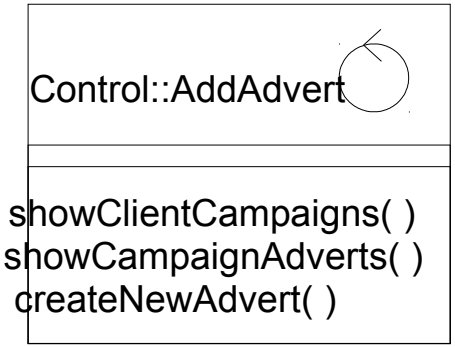
MALMÖ UNIVERSITET

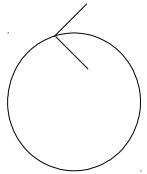
Control

Control-klasser hanterar systemets logik – de binder samman beteenden till en helhet.

Alternativa notationer:





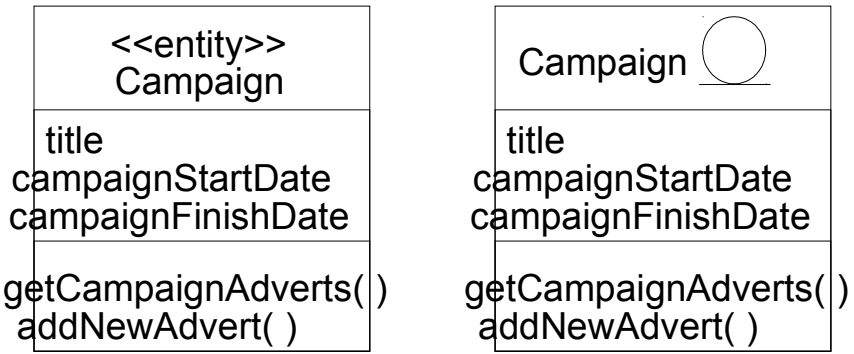


MALMÖ UNIVERSITET

Entity

Entity-klasser representerar data eller information som ska används av systemet och det beteende som är kopplat till denna data/information.

Alternativa notationer:



Mer UML

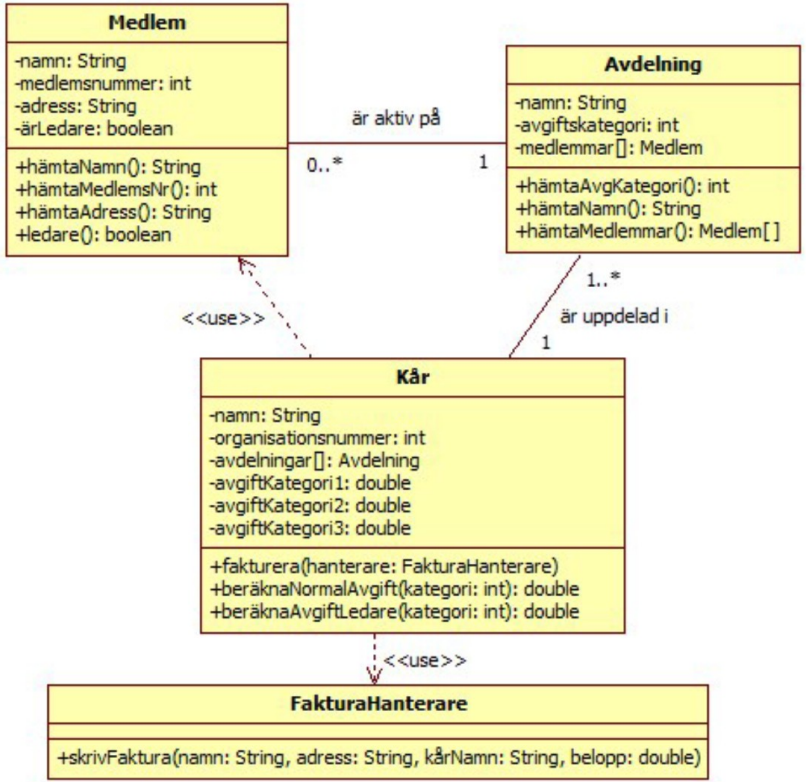
MALMÖ UNIVERSITET

Beroenden

Klass A är beroende av klass B om:

- en instans av A sänder ett meddelande till B
- en instans av A skapar en instans av B
- en instans av A har ett attribut vars värde är en (referens till en) instans eller samling av instanser av klass B
- en instans av klass A får ett meddelande där en parameter är en (referens till en) instans av klass B

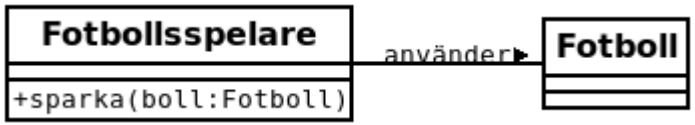
Beroenden

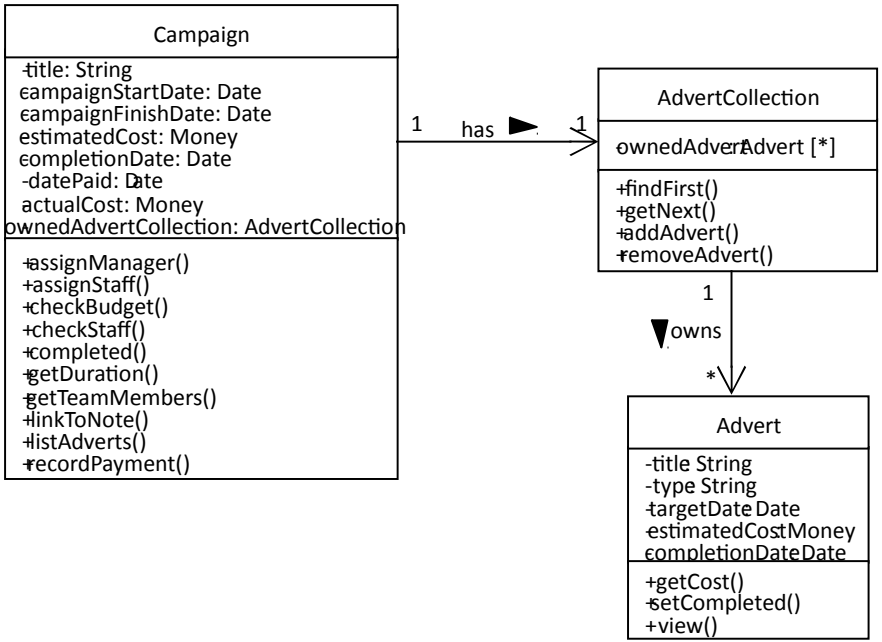


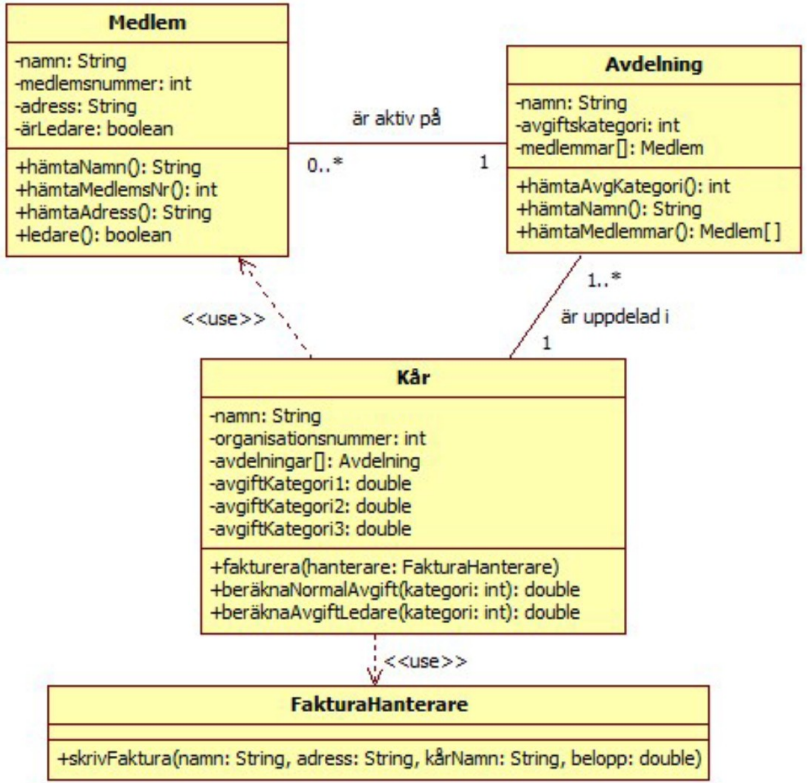
MALMÖ UNIVERSITET

Navigering

Vi kan följa beroenden genom att följa pilar i klass-diagrammen. Några exempel:




Campaign
-title: String
campaignStartDate: Date
campaignFinishDate: Date
estimatedCost: Money
completionDate: Date
-datePaid: Date
actualCost: Money
ownedAdvertCollection: AdvertCollection
+assignManager()
+assignStaff()
+checkBudget()
+checkStaff()
+completed()
+getDuration()
+getTeamMembers()
+linkToNote()
+listAdverts()
+recordPayment()
1
has
1
AdvertCollection
-ownedAdvert: Advert [*]
+findFirst()
+getNext()
+addAdvert()
+removeAdvert()
1
owns
*
Advert
-title String
-type String
-targetDate Date
-estimatedCost Money
-completionDate Date
+getCost()
+setCompleted()
+view()



MALMÖ UNIVERSITET

Deriverade associationer

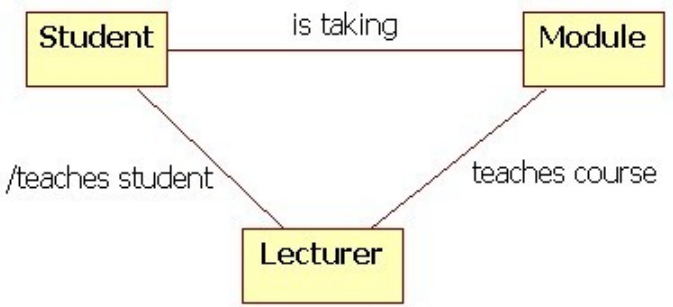
I vissa fall så finns det en association mellan klasser som kanske inte är direkt men som uppstår genom att dessa är associerade via någon annan klass.

Exempel: Om Student är associerad med Module genom att Student ”is taking” Module och Module är associerad med Lectrurer genom ”teaches course” behöver vi då visa en association mellan Student och Lecturer i form av ”teaches student”?

MALMÖ UNIVERSITET

Deriverade associationer

För situationen att en konceptuell association finns har UML en notation som kallas **deriverad association** som innebär att denna association finns om man har implementerat de övriga associationerna.



En deriverad association visas med ett snedstreck framför namnet.

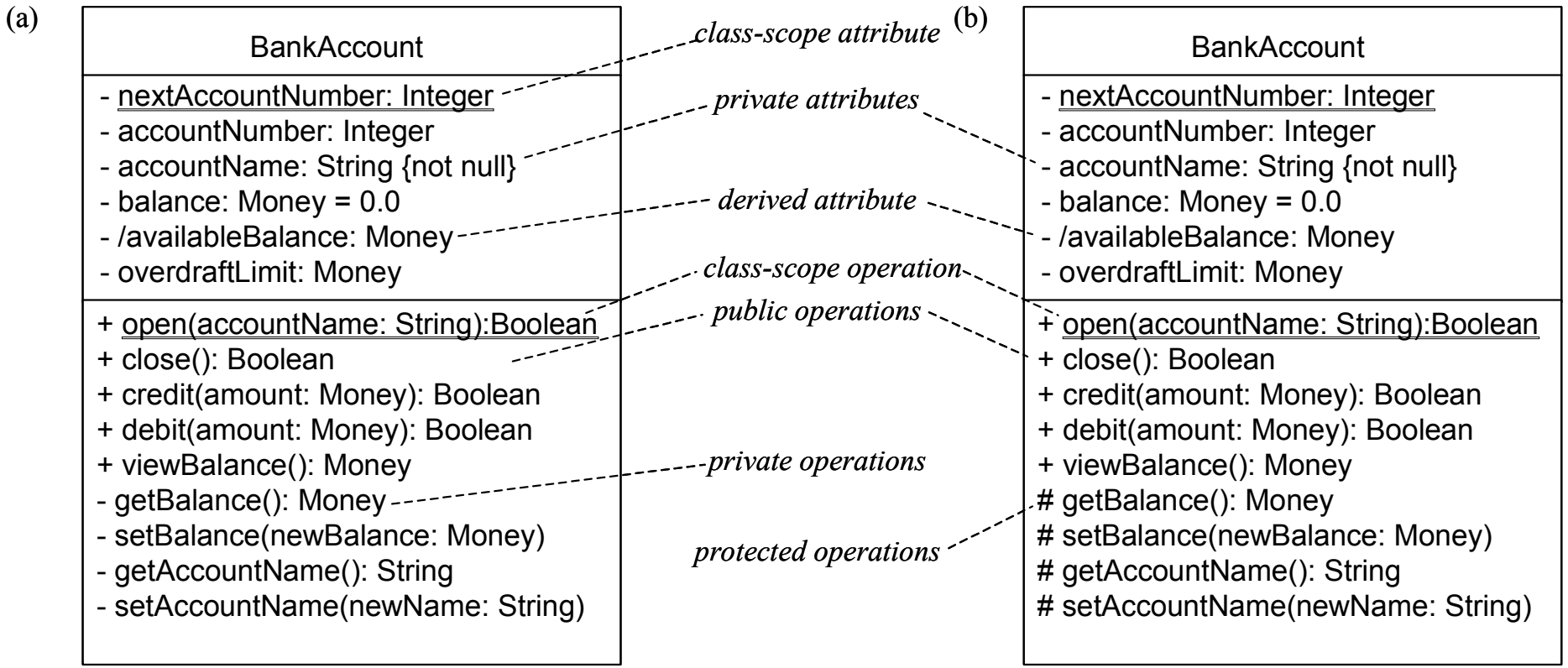
MALMÖ UNIVERSITET

Deriverade attribut

Även attribut kan vad deriverade.

Dessa är attribut som behöver räknas ut givet andra attribut.

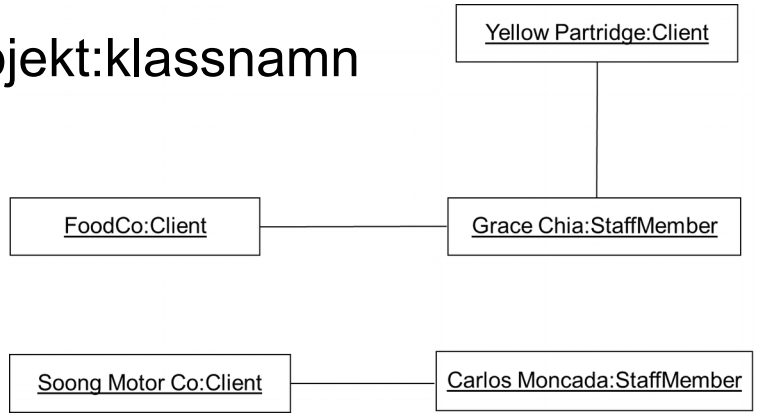
Dessa uträkningar kan ske när en förändring sker i något av de attribut som det deriverade attributet bygger på eller när det deriverade attributet refereras.



MALMÖ UNIVERSITET

Objektdiagram

- Objektdiagram visar systemets objekt vid en given tidpunkt under exekvering.
- Associationen mellan objekt kalls länk.
- Namn på objekt:klassnamn



MALMÖ UNIVERSITET