Föreläsning 9

Objektorienterad design

Chalmers (DAT455) och GU (DIT001)

Jonas Duregård

Hittils om objektorientering

- Använda klasser och ”utifrån-perspektivet” med ett gränssnitt av bestämda konstruerare och metoder
- Att skapa klasser och ”inifrån-perspektivet” med alla implementationsdetaljer
  - Nyckelordet **`class`** för att skapa klasser i Python

```
class Racer:
    def __init__(self):
      …
```

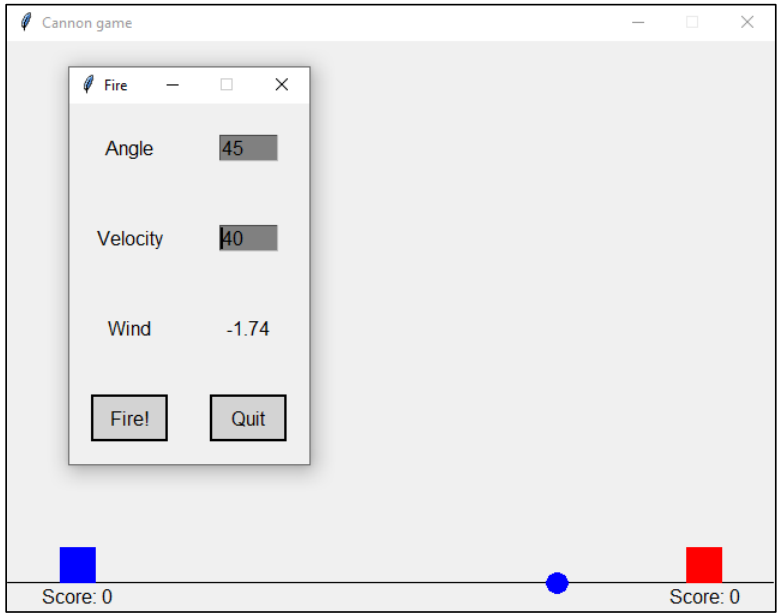
- En del om hur man kan strukturera kod i klasser för att hålla isär olika delar

Dagens ämne

- Lite allmänt om hur man designar ett program med klasser
- Två viktiga begrepp i objektorienterad design:
  - Polymorfism
  - Arv
- För varje begrepp kommer jag visa något exempel, samt hur det förekommer i Labb 2 i kursen

Kom ihåg: Labb 2

- Uppgiften i labb 2 är att skapa ett spel där två spelare ställer in riktning och hastighet på var sin kanon för att försöka träffa den andres kanon
- Kanonkulor påverkas av gravitation och vind
- Spelare får poäng för varje träff



- Designen ni ska använda för er kod är till största delen redan bestämd, men jag tänkte tala lite om hur man kan komma fram till en design på egen hand

Steg 1: Identifiera objekt och klasser

- Ett bra första steg är att utifrån en beskrivning försöka komma på några klasser av objekt som naturligt beskriver programmets delar
- Designtips: Ringa in ord (oftast substantiv) i beskrivningen av programmet som skulle kunna vara klasser
- I labb 2 identifierade vi Spelare och Projektiler som klasser
- Vi skapade också en klass Game som representerar en hel spelomgång, och håller reda på allt som har med spelomgången att göra
  - I en körning av programmet kommer det bara finnas ett Game-objekt, men man skulle lätt kunna skapa flera - till exempel om man implementerar en multiplayer-server för spelet!

Steg 2: Identifiera attribut

- Vilka attribut skulle varje klass vi identifierat ha?
- Utgå från beskrivningen och placera ut de olika värdena som förekommer
  - Exempel: Vindhastighet är en global egenskap och sparas lämpligen i Game
  - Exempel: Varje spelare ska ha en egen nuvarande poäng, så det är ett attribut hos Player
- Jobba från andra änden och kolla på dina klasser, vad behöver de veta?
  - Projektiler behöver hastighet och position, och även vindhastighet
  - Player behöver en position och en färg som identifierar den

Game är klassen som håller reda på vindhastighet, på något sätt måste den informationen skickas till varje kanonkula

Steg 3: Identifiera gränssnitt/metoder

- Vilka operationer ska man kunna säga till olika objekt att utföra?
- Exempel: Game ska kunna starta en ny omgång, player ska kunna avfyra en kanonkula med en viss vinkel och hastighet …
- Leta efter verb i beskrivningen, meningar som beskriver saker som händer.
- Hur ska klassen se ut utifrån? Vad ska spelaren redan hålla reda på internt och vad ska vara parametrar till metoder? Skissa en bit kod som använder klassen, till exempel skapar en spelare och avfyrar dess kanon.

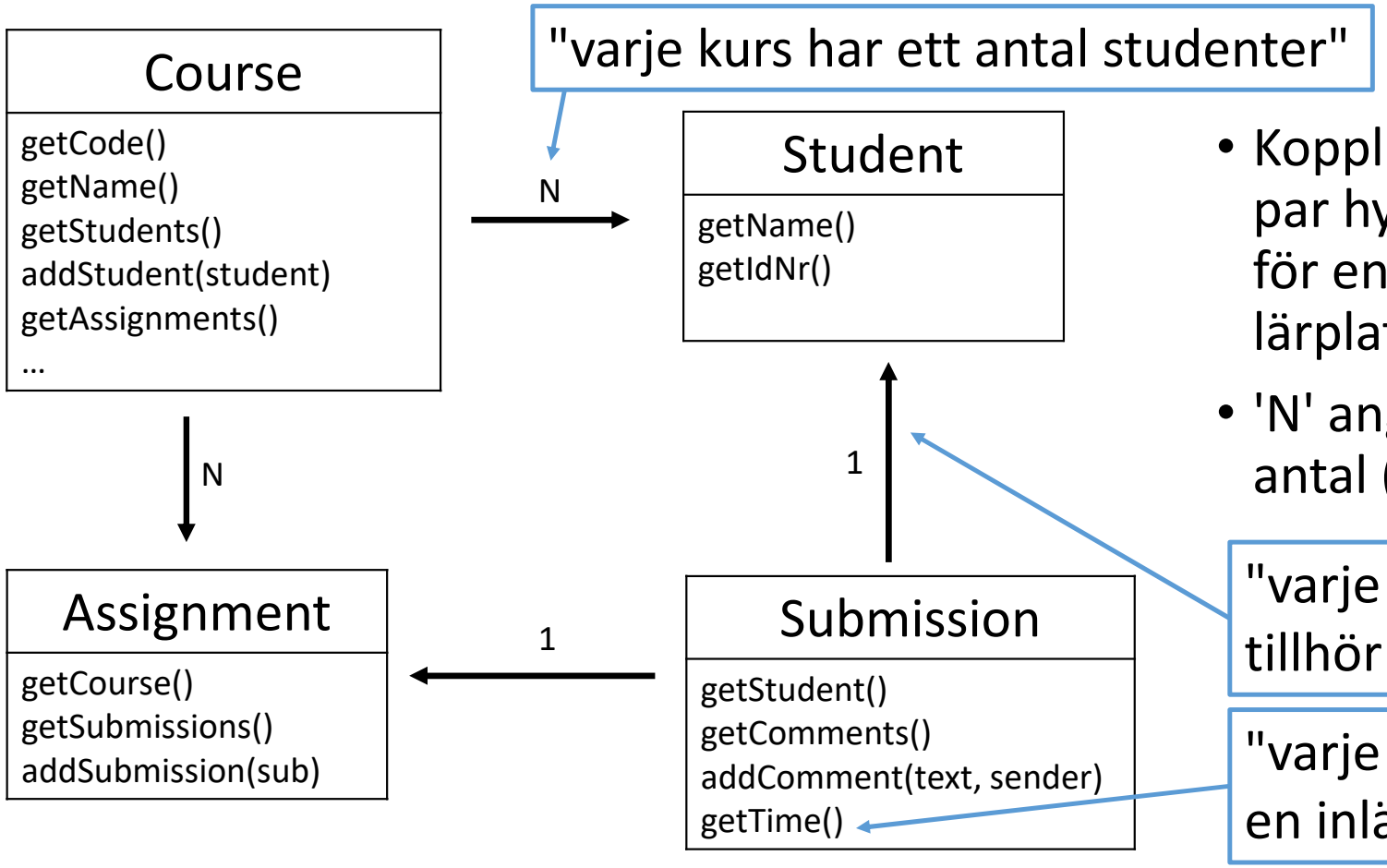
Skrota och gör om

- När som helst under utvecklingsprocessen kan man upptäcka att designen behöver ändras och gå tillbaks och göra om delar av tidigare steg
- Exempel: Vi kanske ringade in ”poäng” i steg 1 och tänkte att vi skulle ha en poäng-klass, men upptäckte när vi funderade på attribut och metoder att poäng bara är ett heltal utan några andra intressanta attribut, så vi använder heltalstypen i Python istället för en klass
- Exempel: När vi programmera upptäcker vi ofta saker som att spelare måste ha tillgång till Game-objektet eller liknande och får ändra vår design för att skicka runt olika objekt till fler ställen, lägga till ytterligare attribut och metoder osv.

Klassdiagram

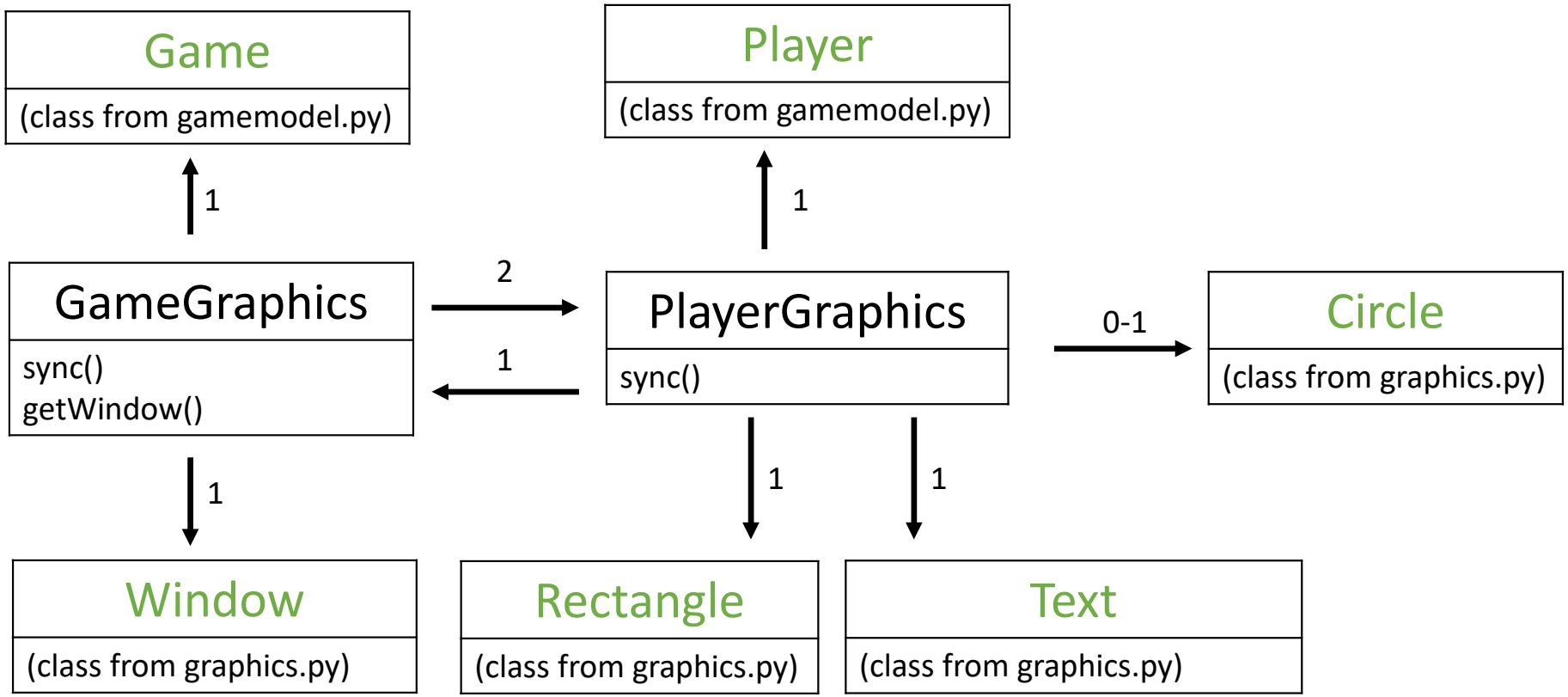
- Att rita tjusiga diagram över klasser och hur de är sammankopplade är vanligt i objektorienterad design
- Kan visa vad klasser innehåller, vilka metoder de har och hur klasserna är kopplade till varandra
- Två syften med klassdiagram
  - Ett hjälpmedel under designprocessen att ha tillhands när man kodar
  - Dokumentation för en färdig design så att den kan användas av andra

Klassdiagram, exempel



- Kopplingar mellan ett par hypotetiska klasser för en Canvas-liknande lärplattform
- 'N' anger godtyckligt antal (typiskt i en lista)

Klassdiagram för labb 2 (från Labb-pm)



UML-diagram

- De enkla klassdiagrammen jag visat här är en form av UML-diagram (Unified Modelling Language)
- Varianter av dem förekommer ofta i objektorienterad design

Polymorfism

Polymorfism

- Ordet polymorfism härstammar från grekiska ”flera former”
- Ett brett begrepp för kod som fungerar för flera olika typer, eller där beteendet kan ändras genom att ändra vilken typ den arbetar med
- Ett vanligt exempel är listor som innehåller objekt från flera olika klasser
  - Klasserna behöver ha några gemensamma metoder (samma namn och parametrar) för att det ska vara meningsfullt

Polymorfism, enkelt exempel

- Skapa en lista med blandade grafikkomponenter och rita ut alla i samma fönster:

```
circle = Circle(Point(0,0), 10)
circle.setFill('red')

letter = Text(Point(0,0), "Hello")

graphics = [circle, letter]

for component in graphics:
    component.draw(win)
```

En lista som innehåller både ett Circle-objekt och ett Text-objekt

Anropar först draw-metoden från klassen Circle, sedan från Text

Ett och samma anrop köra olika kod beroende på vad som finns i listan!

Varför använda polymorfism?

- Polymorfism låter oss skriva funktioner/metoder/klasser som fungerar för flera olika sorters data, och därmed är mer användbara
- Till exempel en metod som ger längden för en lista men som också kan ge längden för en sträng
  - Betyder framför allt att vi slipper komma ihåg två olika funktionsnamn!
- Polymorfism låter oss också ändra beteendet på en klass eller funktion genom att byta ut ett objekt
  - Till exempel byta mellan grafiskt gränssnitt eller textbaserat gränssnitt för ett program genom att bara byta vilken klass som används på ett ställe

Exempel: Kombinera grafikkomponenter

- I racer3.py ritade vi ut en bil som bestod av en cirkel och en bokstav
- Varje gång vi flyttar cirkeln måste vi också flytta bokstaven (garanteras genom inkapsling!)
- Vi skulle kunna skriva en mer allmän klass för att kombinera två grafikkomponenter från graphics.py (ska fungera med vilka komponenter som helst, inte bara en cirkel och en text)

Vi vill kunna slå ihop self.circle och self.letter till ett objekt

```
self.circle = Circle(Point(0,0), 10)
self.circle.setFill('blue')
self.circle.draw(win)

self.letter = Text(Point(0,0), "R")
self.letter.setTextColor('red')
self.letter.draw(win)
```

Klassen Combined

```
class Combined:
    def __init__(self, comp1, comp2):
        self.comp1 = comp1
        self.comp2 = comp2

    def move(self, deltaX, deltaY):
        self.comp1.move(deltaX, deltaY)
        self.comp2.move(deltaX, deltaY)

    def draw(self, win):
        self.comp1.draw(win)
        self.comp2.draw(win)
```

Tar två grafikkomponenter av vilken sort som helst (polymorfisk!)

"Härmar" grafikklasserna (Circle etc.) genom att ha move- och draw-metoder

Anropar möjligen move för olika klasser

Vi vet inte vilka klasser comp1 och comp2 tillhör, men de måste ha move- och draw-metoder!

Använda Combined

- Den här koden skapar en cirkel med en text, ritar ut den och flyttar runt den lite

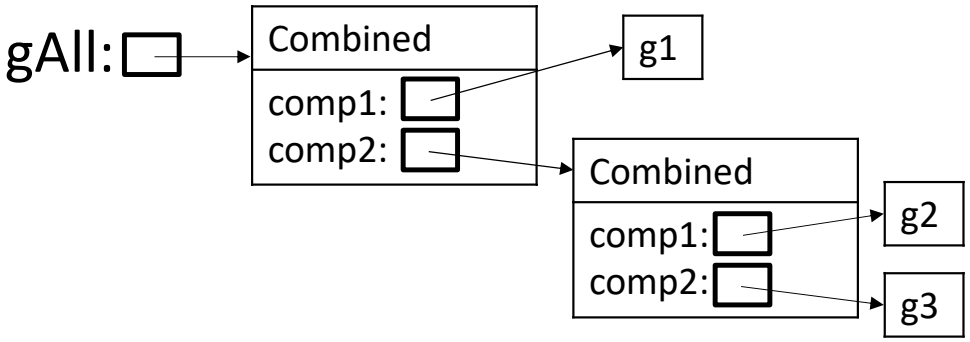
```
class Combined:
    def __init__(self, comp1, comp2):
        self.comp1 = comp1
        self.comp2 = comp2

    def move(self, deltaX, deltaY):
        self.comp1.move(deltaX, deltaY)
        self.comp2.move(deltaX, deltaY)

    def draw(self, win):
        self.comp1.draw(win)
        self.comp2.draw(win)
```

```
circle = Circle(Point(0,0), 10)
circle.setFill('blue')

letter = Text(Point(0,0), "R")
letter.setTextColor('red')

car = Combined(circle, letter)
car.draw(win)

for i in range(100):
    car.move(1,0)
    update(25)
```

Flyttar både cirkeln och texten

Combined i minnet

- Efter att vi kört **`gAll = Combined(g1, Combined(g2, g3))`** för några grafikkomponenter g1, g2 och g3



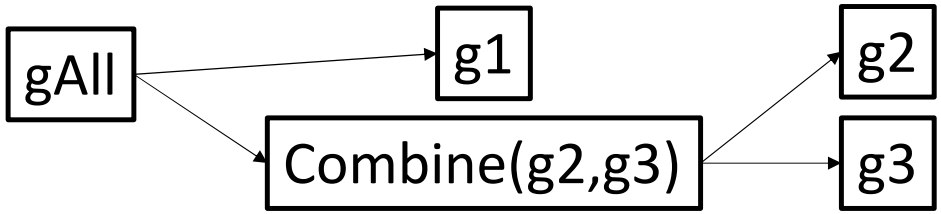
Vad ger uttrycket gAll.comp2.comp1?

Svar: g2

```
class Combined:
    def __init__(self, comp1, comp2):
        self.comp1 = comp1
        self.comp2 = comp2

    def draw(self, win):
        self.comp1.draw(win)
        self.comp2.draw(win)
    …
```

Kombinera fler komponenter?

- Om vi vill kombinera tre eller fler komponenter kan vi lätt ändra Combine till att ta en lista på komponenter istället för bara två
- … eller så kan vi använda oss av följande faktum:
  - Combine fungerar för allla klasser som har move- och draw-metoder, inklusive Combine-klassen själv!
  - Exempel: **`gAll = Combined(g1, Combine(g2, g3))`** slår ihop tre grafikkomponenter g1, g2 och g3. Kör man move eller draw på `gAll` körs metoden på alla tre.
  - Lurig fråga: Om jag kör `gAll.move(1,0)`, hur många objekts draw-metod anropas total?
    - Svaret är fem! Förutom g1, g2 och g3 körs den två stycken Combine-objekt



Polymorfism i Combined-exemplet

- Combined är polymorf på så vis att den kan innehålla vilka objekt som helst vars klasser har en move- och en draw-metod
  - Cirklar, Text, andra Combined-objekt …
- Egentligen är nästan all kod i Python polymorf, vi kan alltid byta ut ett objekt mot objekt från andra klasser så länge de har likadana metoder (men vad programmet gör kommer ändras)

Begränsningar med Combined

- Vi kan lägga till fler metoder i Combined, men bara sådana som alla grafikkomponenter har
- Vi kan till exempel inte lägga till en getX- eller getCenter-metod, för olika komponenter har olika metoder för att ge deras position
  - Även om det fanns en gemensam metod för att hämta position så skulle vi få två positioner (en från varje komponent) och skulle behöva bestämma hur de kombineras till ett resultat

Arv

Arv

- När en klass skapas kan den ärva från andra klasser
  - Klassen som ärver kallas subklass
  - Klassen subklassen ärver från kallas superklass
- Subklassen får automatiskt alla metoder och attribut från superklassen
- Tanken är att varje objekt av subklassen också *är* ett objekt av superklassen och beter sig likvärdigt, men kan ha ytterligare funktionalitet
- Arv är ett stort och svårt begrepp i OO-programmering och vi kommer bara gå igenom det ytligt!
  - Målet är att ni ska förstå vad som menas när det står att en klass ärver från en annan

Varför använda arv?

- En viktig anledning är att inte upprepa kod
  - Om du har två klasser som är väldigt lika är det ofta en bra idé att flytta ut de gemensamma delarna till en basklass som båda klasserna ärver från
  - Att bara klippa-och-klistra leder snabbt till problem, dels får du en större volym kod som gör allt lite trögare att navigera, dels får du buggar när du ändrar något på ena ställen men inte på andra

Hur arv ser ut i koden

- Om vi vill skapa en klass Teacher som ärver från Person (se förra föreläsningen):

```
class Teacher(Person):
    ...
```

- Det här betyder att Teacher automatiskt har alla metoder och attribut som Person har
- Vi kan lägga till ytterligare metoder i Teacher som kommer finnas för Teacher-objekt men inte för andra Person-objekt
- Lite ytterligare tekniska detaljer behövs för konstuktor osv. som inte visas här
  - Ni kommer inte behöva skriva egna klasser som använder arv i kursen

Arv och Är-relationen (IS-A)

- I objektorienterad design talar man om olika relationer mellan klasser:
  - Innehåller-relation: KlassA innehåller KlassB betyder att varje objekt av KlassA har ett attribut med ett värde av typen KlassB
  - Använder-relation: KlassA använder KlassB till exempel om den tar ett KlassB-objekt som parameter eller skapar och returnerar ett KlassB-objekt
  - Det är vanligt att man ritar upp diagram som visar dessa relationer som en del av designprocessen
- Arv är en ytterligare sorts relation, en ”är-relation”. Om till exempel klassen Teacher ärver från klassen Person säger vi att Teacher är Person
  - Betyder att varje Teacher också är en Person (men inte tvärt om!)
- Det är ett bra sätt att testa om arv är lämpligt: Man kan till exempel säga Teacher är en Person men kanske inte t.ex. PersonList är en Person

Exempel: Arv i graphics.py

- Här är en bit av koden från graphics.py

Alla klasser som kan ritas ut ärver direkt eller indirekt från GraphicsObject, som har draw, move etc.

En ”intern klass” för grafikobjekt som har en bounding box
(oviktigt vad det är, men tydligen har cirklar och rektanglar det!)

```
class GraphicsObject:
    def draw(self, graphwin): …
    def move(self, dx, dy): …
    …
class Point(GraphicsObject):
    …
class Text(GraphicsObject):
    …
class _BBox(GraphicsObject):
    …
class Rectangle(_BBox):
  …
class Oval(_BBox):
  …
class Circle(Oval):
  …
```

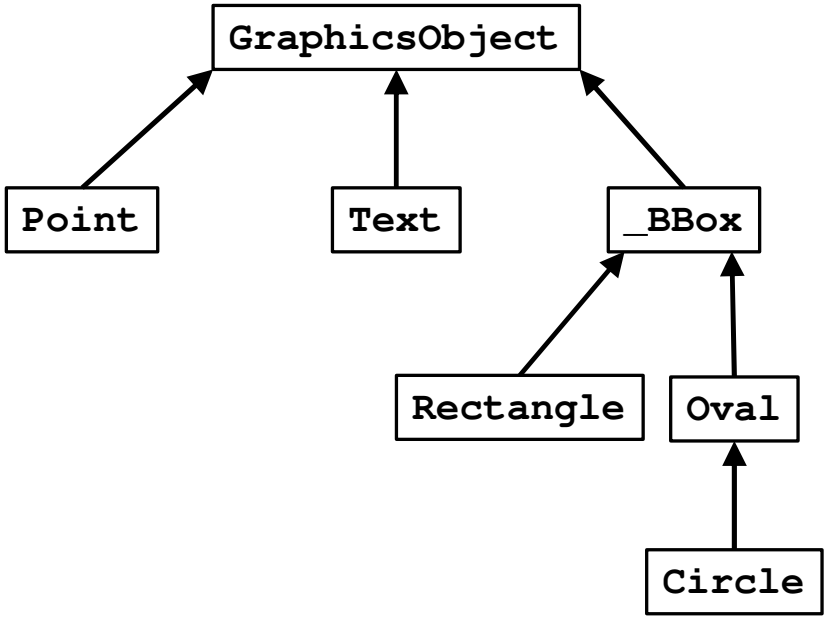
Klasshierarki

- Om en A är ”barnbarn” till B, alltså om A ärver från klass som i sin tur ärver från B, så har A alla metoder och attribut från B precis som om den ärvt direkt
  - A har också allt från den mellanliggande klassen
- När ett stort antal klasser ärver från varandra i flera nivåer skapas en klasshierarki
  - Överst i hierarkin finns väldigt allmänna klasser, klasser vars metoder finns i alla klasser i hierarkin
  - Längre ner blir klasser mer specialiserade
  - Exempel: I graphics.py har GraphicObject bara de mest allmänna metoderna för att till exempel rita ut och flytta objekt, medan Circle har t.ex. en metod som ger radien på cirkeln

Klasshierarki i graphics.py

(pilar går från subklass till superklass)

```
class GraphicsObject:
    def draw(self, graphwin): …
    def move(self, dx, dy): …
    …
class Point(GraphicsObject):
    …
class Text(GraphicsObject):
   …
class _BBox(GraphicsObject):
   …
class Rectangle(_BBox):
  …
class Oval(_BBox):
  …
class Circle(Oval):
  …
```



Klasser blir mer specialiserade längre ner i hierarkin

Specifikationsarv och implementationsarv

- Den sortens arv vi talar om här kallas implementationsarv
  - Implementationen är all kod i klassen
  - Specifikationen är det yttre gränssnittet, vilka metoder som finns, vilka parametrar de tar och vilken typ de returnerar
- Specifikationsarv i Python innebär bara innebär att två klasser har ”likadana” metoder (namn, parametrar, typ av returvärde osv.)

Överkurs: Använda arv för Racer-programmet

- I förra föreläsningen visade jag hur vi kan dela upp en klass som blandade modell och grafik i en ren modellklass och en ren grafikklass
- En nackdel med upplägget är att våran huvudloop måste anropa metoder från båda klasserna
- Ett annat sätt att göra samma sak är att låta grafikklassen utöka modellklassen
  - Modellklassen är helt oförändrad, en logisk representation av en bil
  - Grafikklassen utökar modellklassen, och är alltså en särskild sorts bil som har alla egenskaper från modellklassen men också ritar grafik

Grundklassen Racer

- Modellklassen är kvar helt oförändrad

```
class Racer:
    def __init__(self):
        self.xvelocity = 0
        self.yvelocity = 0
        self.xpos = 0
        self.ypos = 0

    def _moveRelative(self, xDiff,yDiff):
        …

    def _moveAbsolute(self, x,y):
        …

    def drive(self):
        …
```

Överkurs: Början på en grafisk-racerbil

```
"""
  Extends the Racer class, adding graphics.
  This class will automatically have all methods that Racer has.
"""
class GraphicRacer(Racer):
    """
      Constructor for GraphicRacer. Note that unlike
        the constructor for Racer it takes one argument.
    """
    def __init__(self, win):
        # This line calls the constructor of the superclass (Racer)
        super().__init__()

        # Create graphic components for the race car
        self.circle = Circle(Point(0,0), 10)
        self.circle.setFill('blue')
        self.circle.draw(win)

        self.letter = Text(Point(0,0), "R")
        self.letter.setTextColor('red')
        self.letter.draw(win)
```

En grafisk Racer är en slags Racer

Konstruktorer i subklasser börjar i regel med att anropa konstruktorn försuperklassen

Resten av subklassen konstruktor skapar grafiken på samma sätt som tidigare

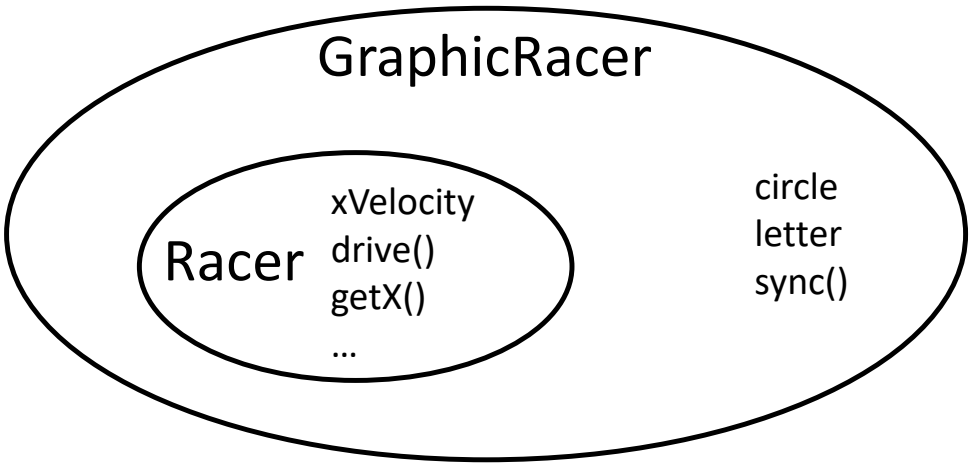
Lägg till en sync-metod

```
class GraphicRacer(Racer):
    def __init__(self, win):
        …

    """
      Synchronize the graphics with the current state of the model
    """
    def sync(self):
        # Move the circle and text to wherever the car is
        c = self.circle
        t = self.letter

        # Move the circle and text elements
        c.move(self.getX()-c.getCenter().getX(), self.getY()-c.getCenter().getY())
        t.move(self.getX()-t.getAnchor().getX(), self.getY()-t.getAnchor().getY())
```

Notera att vi använder ex.vis. self.getX(), där getX() är en metod från Racer!

Använda GraphicRacer

- I huvudprogrammet skriver vi `racer = GraphicRacer(win)` för att skapa en racerbil med grafik (win är fönstret den målas ut i)
- Vi behöver aldrig skapa något objekt av klassen Racer, en GraphicRacer är en racer och har alla metod från båda klasserna
  - racer har både drive() och sync()!
- Ett sätt att tänka på arvet är att varje GraphicRacer innesluter en Racer



Överkurs i överkurs: Overriding av metoder

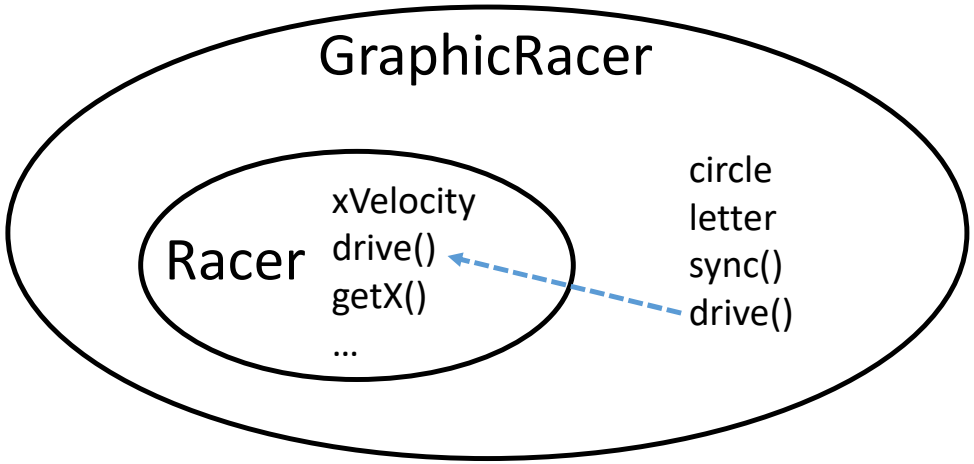
- Vi kan göra så att drive() automatiskt anropar sync()
  - Ändrar (eller utökar) beteendet av drive() i subklassen
  - Huvudloopen blir helt opåverkad av grafik

```
class GraphicRacer(Racer):
    def __init__(self, win):
        …
    def sync(self):
        …

    def drive(self):
        # This line calls the drive method from the superclass (Racer)
        super().drive()

        # After running drive() from Racer, we synchronize the graphics
        self.sync()
```

Tanke: ”gör först drive() som vanligt, uppdatera sedan grafiken”

Överkurs: Overriding

```
def drive(self):
        # Calls drive() from the superclass
        super().drive()

        self.sync()
```



Utifrån syns bara den nya drive()-metoden

Inifrån sett har en graphicRacer nu två drive()-metoder, den från superklassen och sin egen

Ett exempel på polymorfism!

Resultat

- Huvudloopen ser ut precis som i vår första objektorienterade lösning
- racer.drive flyttar både bilens logiska och grafiska representation
- Skillnaden är att vår kod genom arv ändå separerar alla grafikkod från den rena modellkoden!
  - Framtida ändringar är enklare
  - Att byta grafikbibliotek är enkelt och påverkar inte modellen

```
while(True):
    key = win.checkKey()

    if key == "Up":
        racer.accelerate(0,1)
    elif key == "Down":
        racer.accelerate(0,-1)
    elif key == "Right":
        racer.accelerate(1,0)
    elif key == "Left":
        racer.accelerate(-1,0)
    elif key == "space":
        racer.slowDown()
    elif key == "BackSpace":
        racer.reset()

    # Move the racer
    racer.drive()

    update(animationSpeed)
```