Python OO Programming

Les 4: OO design patronen: overerving / compositie

Kristof Michiels

Topics

Topics

- Opdracht: object georiënteerde-toepassing met zelfgekozen invalshoek
- OO patronen: gebruik van overerving / compositie
- UML
- Voorbeeldtoepassing

Opdracht: OO-toepassing met zelfgekozen invalshoek

Opdracht: OO-toepassing met zelfgekozen invalshoek

- Labo: maak in 3 weken een eigen toepassing
- Toepassing mag command-line geörienteerd zijn en in de lijn liggen met de oefeningen die we de voorbije weken hebben gemaakt
- Af te werken en in te dienen tegen vrijdag 29/10 (deadline spreken we komende maandag verder af)
- Jullie hebben allemaal een onderwerp gekozen
- Belangrijk: implementatie dient object-geörienteerd te gebeuren
- Komende lessen: tijd voor vragen / jij werkt autonoom aan je project
- Parallel: voorbeeldtoepassingen worden ter inspiratie getoond

Overerving vs Compositie

- Een OO-aanpak in een toepassing is vaak niet eenduidig
- Je kan het met andere woorden op verschillende manieren benaderen
- We leerden in de eerdere oefeningen werken met overerving en compositie
- Beiden hebben hun plaats binnen OO-development en in de meeste projecten worden ze samen gebruikt
  - Waar we bij overerving van een "is een"-relatie bestaat en je afgeleide klassen definieert die specifieker zijn dan de basisklassen waar ze van erven
  - Ga je bij compositie vaker een "heeft een of meerdere" relatie vorm geven
  - Je werkt vanuit een los verband (de klasses hoeven elkaar niet te kennen) en gebruikt een component klasse als een niet-standaard data type

Overerving vs Compositie

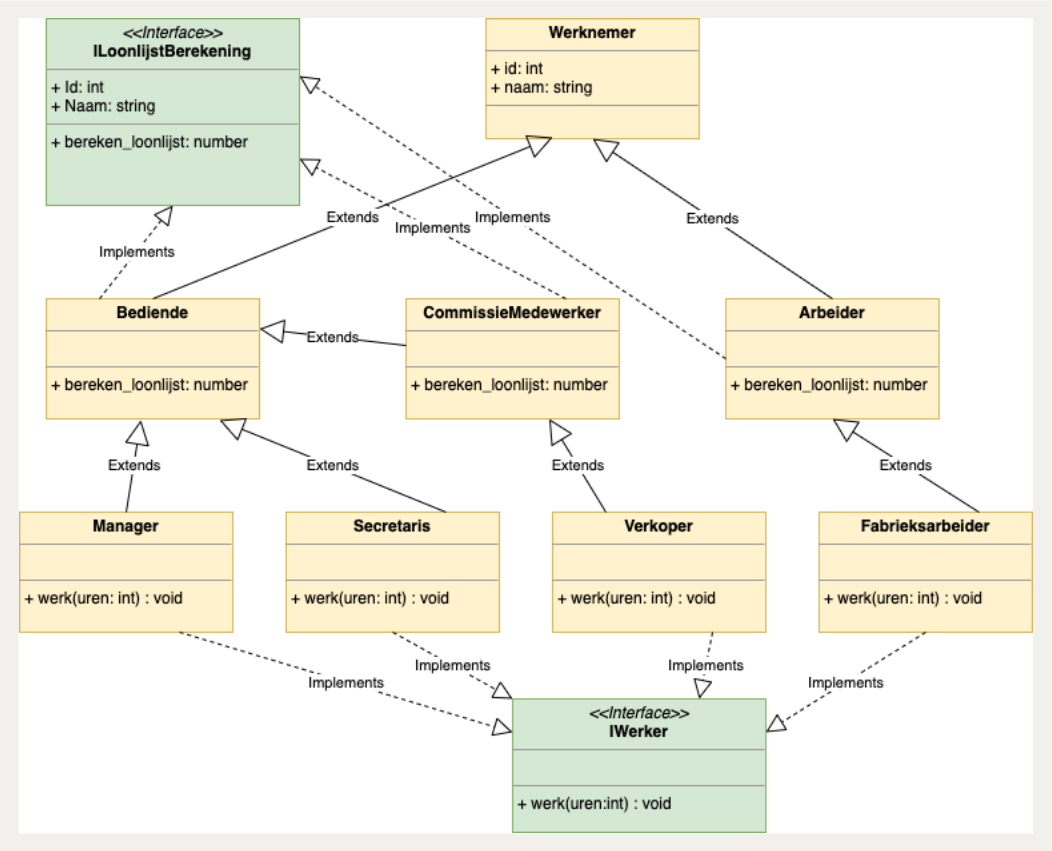
- Er bestaan een reeks patronen die kunnen gebruikt worden in het schrijven van OO-applicaties
- Aan de hand van voorbeelden zijn die het best te illustreren en dat is wat we de komende weken gaan bekijken

Unified Modeling Language (UML)

Een applicatie op voorhand plannen

- Applicaties worden in de praktijk niet zonder voorafgaande planning ontwikkeld
- Een project plannen laat ons toe belangrijke design-beslissingen te nemen alvorens de code effectief te schrijven
- UML (= Unified Modeling Language) kan ons hierbij helpen
- Het biedt een gestandaardiseerde manier om software architecturen mee te ontwerpen en beschrijven
- UML ondersteunt overerving en compositie en wordt vaak gebruikt om klassen en hun onderlinge samenhang te beschrijven

Een voorbeeld van een UML-diagram



UML-diagrammen

- Je kan heel eenvoudig online je eigen UML-diagrammen maken via https://app.diagrams.net/
- Doorheen de voorbeeldtoepassing zal je voorbeelden zien van UML-diagrammen
- Ik nodig jullie uit om ook voor je eigen applicatie een UML-diagram op te stellen

Een UML-klassebeschrijving

- Een klassebeschrijving bestaat uit de naam, de attributen, en de methods
- De diagrammen laten een + of - voorafgaan aan elk onderdeel van een klasse. Deze staan voor publieke of private toegangkelijkheid.
    - Bij Python is er niet zoiets als public of private. Je plaatst dus eigenlijk steeds een plus (we weten ondertussen dat je als developer met een underscore kunt aangeven of een attribuut buiten de klasse mag opgevraagd of aangepast worden)
- Je vermeldt steeds het data type bij de attributen
- Bij de methods vermeld je ook de argumenten met hun data types en tot slot ook het return data type

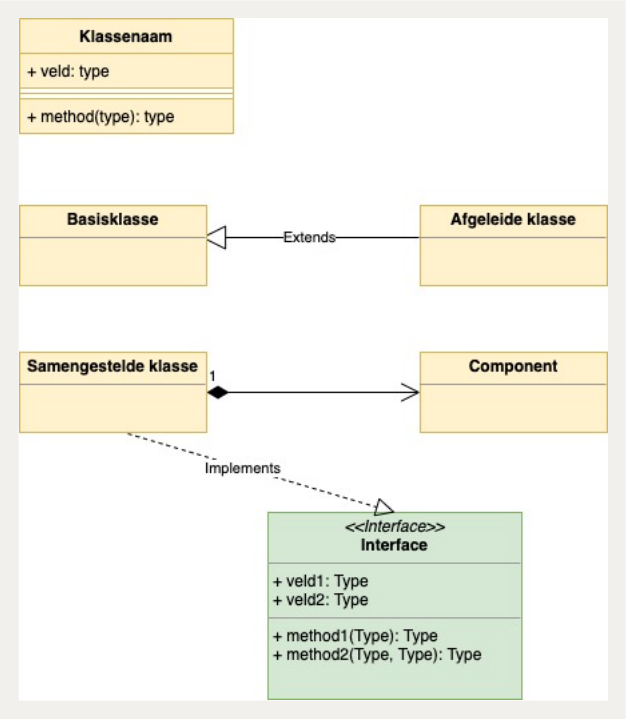
Uitdrukken van overerving en compositie in UML

- Overerving
  - Een witte pijl van de afgeleide klasse naar de basisklasse
  - We vermelden daarbij het woord "Extends", waarmee we duidelijk maken dat het hier om overerving gaat
- Compositie
  - Een holle pijl die begint met een zwarte ruit
  - Een getal geeft weer hoeveel geïnstantieerde componenten de samengestelde klasse kan bevatten
  - Een * symbool ipv een getal verwijst naar een variabel aantal geïnstantieerde componenten
  - Je kan via een range ook een bereik aan mogelijkheden weergeven: bvb (1..4) betekent tussen 1 en 4, (1..*) betekent minstens één

Beschrijven van interfaces in UML

- Een interface (niet te verwarren met een gebruikersinterface) beschrijft de kenmerken en het gedrag van een object
- In het geval van de methods: niet de implementatie maar de declaratie
- Sommige programmeertalen bieden effectief ondersteuning voor het beschrijven van interfaces. Python biedt die ondersteuning niet
- Voegen we interfaces toe aan onze UML-diagrammen dan denken we dus eerder conceptueel
- Om zich te conformeren aan een interface moet een klasse voorzien zijn van de in de interface beschreven attributen en methods

Onderdelen van een UML-diagram



Voorbeeldtoepassing

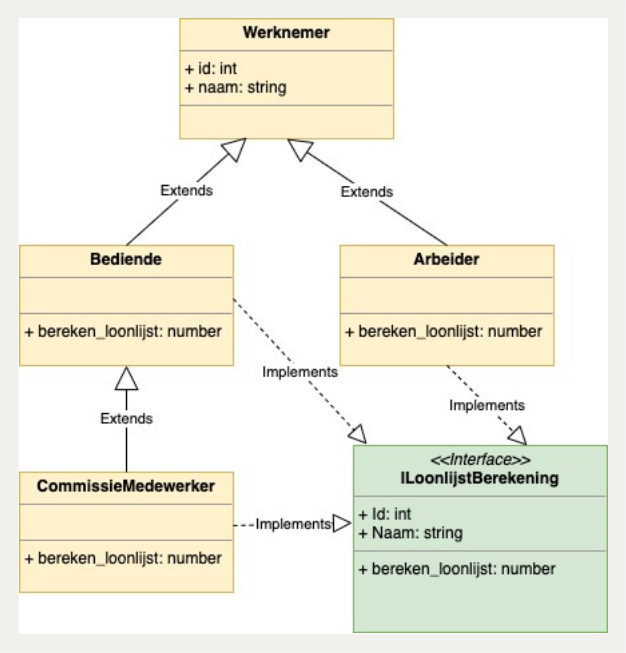
Voorbeeldtoepassing

- Deze Human Resources-toepassing toont een stapsgewijze implementatie van een klasse hiërarchie
- Het betreft een eenvoudige toepassing die door een bedrijf kan gebruikt worden om de lonen te berekenen en de productiviteit van de werknemers in kaart te brengen
- Er wordt zowel gebruik gemaakt van overerving als van object compositie
- Je zal de toepassing een aantal keren aangepast zien worden
- Aan de hand hiervan zal je een aantal OO-patronen gebruikt zien worden
- Het rustig bestuderen van de code kan je helpen om de juiste keuzes te maken in je eigen toepassingen

Stap 1: het loonsysteem

- Een basisimplementatie met aantal klassen in afzonderlijke module die met overerving in elkaar verweven zitten
- Het loonsysteem berekent het loon van alle werknemers
- Klasse loonsysteem accepteert een list met werknemers als argument en drukt hun loon af
- Alle werknemers zijn geïnstantieerde objecten van de klassen Bediende, Arbeider of CommissieMedewerker.
- De twee eerste klassen erven van een basisklasse Werknemer, de CommissieMedewerker erft onrechtstreeks, via de klasse Bediende
- We laten CommissieMedewerker overerven van Bediende, omdat ze het krijgen van een maandloon gemeen hebben. Daarmee maken we van CommissieMedewerker een afgeleide klasse van de klasse Bediende

UML-diagram (stap 1)



Stap 1: het loonsysteem

- Werknemer is hier een abstracte basisklasse waardoor deze niet kan geïnstantieerd worden
- De aanwezigheid van de bereken_loonlijst-method wordt afgedwongen van alle afgeleide klassen door de aanwezigheid van de @abstractmethod decorator
- Elk van de afgeleide klassen beschikt dus over deze method, die wordt aangeroepen binnen de bereken_loonlijst-method van de LoonSysteem-klasse

hrm.py

```
from abc import ABC, abstractmethod

class LoonSysteem:
    def bereken_loonlijst(self, werknemers):
        for werknemer in werknemers:
            print(f'Loonlijst voor: {werknemer.id} - {werknemer.naam}')
            print(f'- Netto bedrag: {werknemer.bereken_loonlijst()}')

class Werknemer(ABC):
    def __init__(self, id, naam):
        self.id = id
        self.naam = naam

    @abstractmethod
    def bereken_loonlijst(self):
        pass
```

hrm.py (vervolg)

```
class Bediende(Werknemer):
    def __init__(self, id, naam, maandloon):
        super().__init__(id, naam)
        self.maandloon = maandloon

    def bereken_loonlijst(self):
        return self.maandloon

class Arbeider(Werknemer):
    def __init__(self, id, naam, uren_gewerkt, uurtarief):
        super().__init__(id, naam)
        self.uren_gewerkt = uren_gewerkt
        self.uurtarief = uurtarief

    def bereken_loonlijst(self):
        return self.uren_gewerkt * self.uurtarief
```

hrm.py (vervolg)

```
class CommissieMedewerker(Bediende):
    def __init__(self, id, naam, maandloon, commissie):
        super().__init__(id, naam, maandloon)
        self.commissie = commissie

    def bereken_loonlijst(self):
        vast = super().bereken_loonlijst()
        return vast + self.commissie
```

programma.py (het ingangspunt van onze toepassing)

```
import hrm

oBediende1 = hrm.Bediende(1, 'Ben Segers',2500)
oArbeider1 = hrm.Arbeider(2, 'Staf Peeters', 40, 20)
oCommissieMedewerker1 = hrm.CommissieMedewerker(3, 'Eva De Lange', 2000, 1500)

oLoonSysteem1 = hrm.LoonSysteem()
oLoonSysteem1.bereken_loonlijst([
    oBediende1,
    oArbeider1,
    oCommissieMedewerker1
])
```

Stap 1: het loonsysteem - merk op dat:

- We gebruiken de super() functie om de __init__ method van Werknemer aan te spreken
- Deze wordt niet meer automatisch uitgevoerd omdat we deze method hebben overschreven in de afgeleide klasse
- In de CommissieMedewerker-klasse roepen we de method bereken_loonlijst bij de base-klasse bediende op via super().bereken_loonlijst()
- Waarom? Indien de berekening van het loon in de toekomst verandert, dan blijft onze afgeleide klasse correct functioneren

Stap 2: het productiviteitssysteem

- We willen een productiviteitssysteem toevoegen aan onze applicatie
- We gaan deze productiviteit berekenen gebaseerd op een aantal (4) als klassen gedefinieerde rollen: manager, secretaris, verkoper, fabrieksarbeider
- We verhuizen de werknemer-gerelateerde klassen naar een nieuw bestand: werknemers.py
- We voegen de 4 nieuwe klassen toe aan dit bestand
- Voor het productiviteitssysteem maken we eveneens een nieuw bestand aan: productiviteit.py

werknemers.py (gedeeltelijke code)

```
class Manager(Bediende):
    # Geen __init__ => erft dus deze method van Bediende
    def werk(self, uren):
        print(f'{self.naam} leidt het team gedurende {uren} uur.')

class Secretaris(Bediende):
    def werk(self, uren):
        print(f'{self.naam} wijdt zich {uren} uren aan zijn administratieve taken.')

class Verkoper(CommissieMedewerker):
    def werk(self, uren):
        print(f'{self.naam} spendeert {uren} uren aan verkoopstaken.')

class Fabrieksarbeider(Arbeider):
    def werk(self, uren):
        print(f'{self.naam} assembleert producten voor {uren} uren.')
```

productiveit.py

Gelijkaardig opgevat als het loonsysteem

```
class ProductiviteitsSysteem:
    def volg(self, werknemers, uren):
        print('Opvolgen Productiviteit van de werknemers')
        print('========================================')
        for werknemer in werknemers:
            werknemer.werk(uren)
        print('')
```

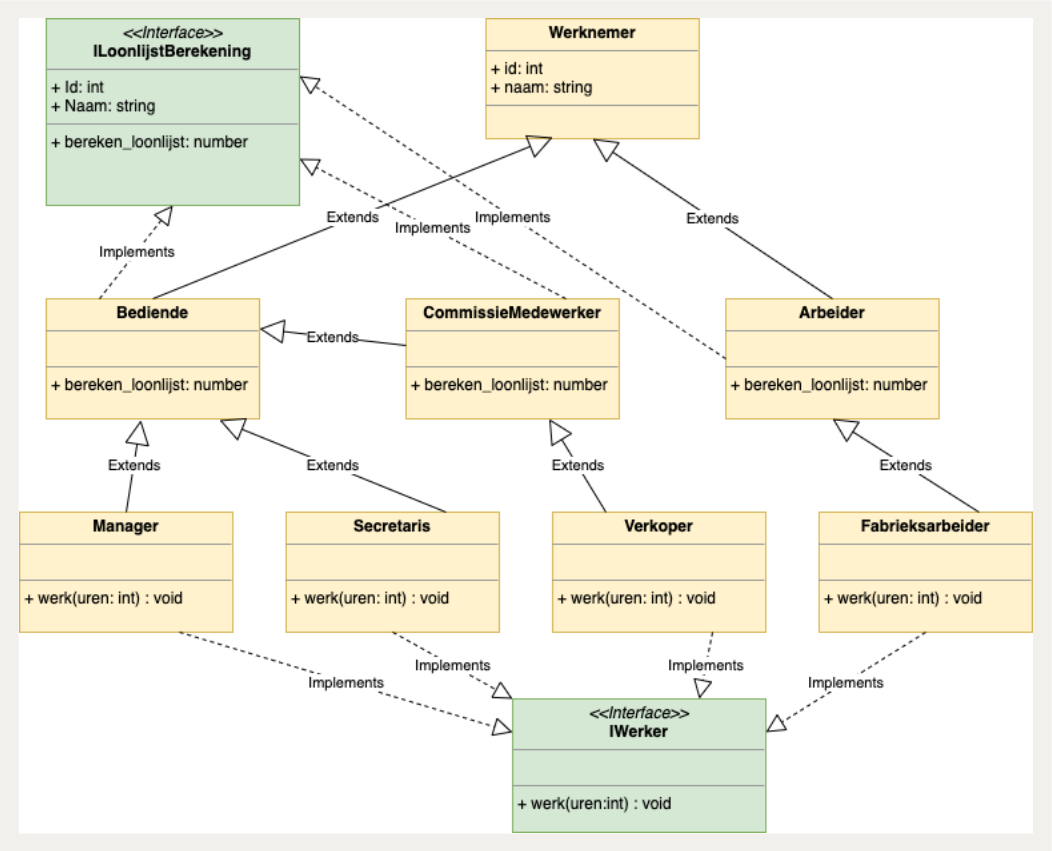
Stap 2: het productiviteitssysteem

- programma.py wordt aangepast:
  - Importeren van de verschillende modules
  - Aanpassen van de klasseverwijzingen
  - Aanmaak productiviteitssysteem object en aanroepen van de werk-method
  - De geïnstantieerde werknemer-gerelateerde klassen worden nu vervangen door de 4 nieuw gecreëerde rollen/klassen
  - Deze erven van de eerder gecreëerde klassen dus alle functionaliteit blijft intact

programma.py

```
import hrm
import werknemers
import productiviteit

oManager1 = werknemers.Manager(1, 'Ben De Vuyst', 2400)
oSecretaris1 = werknemers.Secretaris(2, 'Kenneth De Prins', 2100)
oVerkoper1 = werknemers.Verkoper(3, 'Kathy Kopers', 2300, 450)
oFabrieksArbeider1 = werknemers.Fabrieksarbeider(4, 'Piet Meesters', 40, 25)

werknemers = [oManager1,oSecretaris1,oVerkoper1, oFabrieksArbeider1]

oProductiviteitsSysteem1 = productiviteit.ProductiviteitsSysteem()
oProductiviteitsSysteem1.volg(werknemers, 40)

oLoonSysteem1 = hrm.LoonSysteem()
oLoonSysteem1.bereken_loonlijst(werknemers)
```

Het UML-diagram voor stap 2



Stap 2: het productiviteitssysteem

- Implementeren een nieuwe interface IWerker
- Een klasse die hier aan voldoet kan gevolgd worden in het productiviteitssysteem
- We bemerken hier al een probleem: als de vereisten veranderen, nieuwe features moeten worden toegevoegd, nieuwe rollen enz => complexiteit gaat snel toenemen

Gebruik meervoudige overerving

- Tot nu toe erfde elke klasse van hooguit één andere klasse
- Python ondersteunt meervoudige overerving
- Beeldt je in: klasse TijdelijkeSecretaris die erft van Secretaris en Arbeider
- Welke init method wordt uitgevoerd? Method resolution order (MRO)!
- Eigen init en bereken_loonlijst methods
  - Bypass MRO: we roepen hier zelfgekozen base klasse aan
- Complexiteit! Tijd om de klasses en hun onderlinge samenhang te herdenken
- Pleidooi om alles goed op voorhand uit te denken

Gebruik meervoudige overerving

Gedeeltelijke code voor werknemers.py (zie code stap 2)

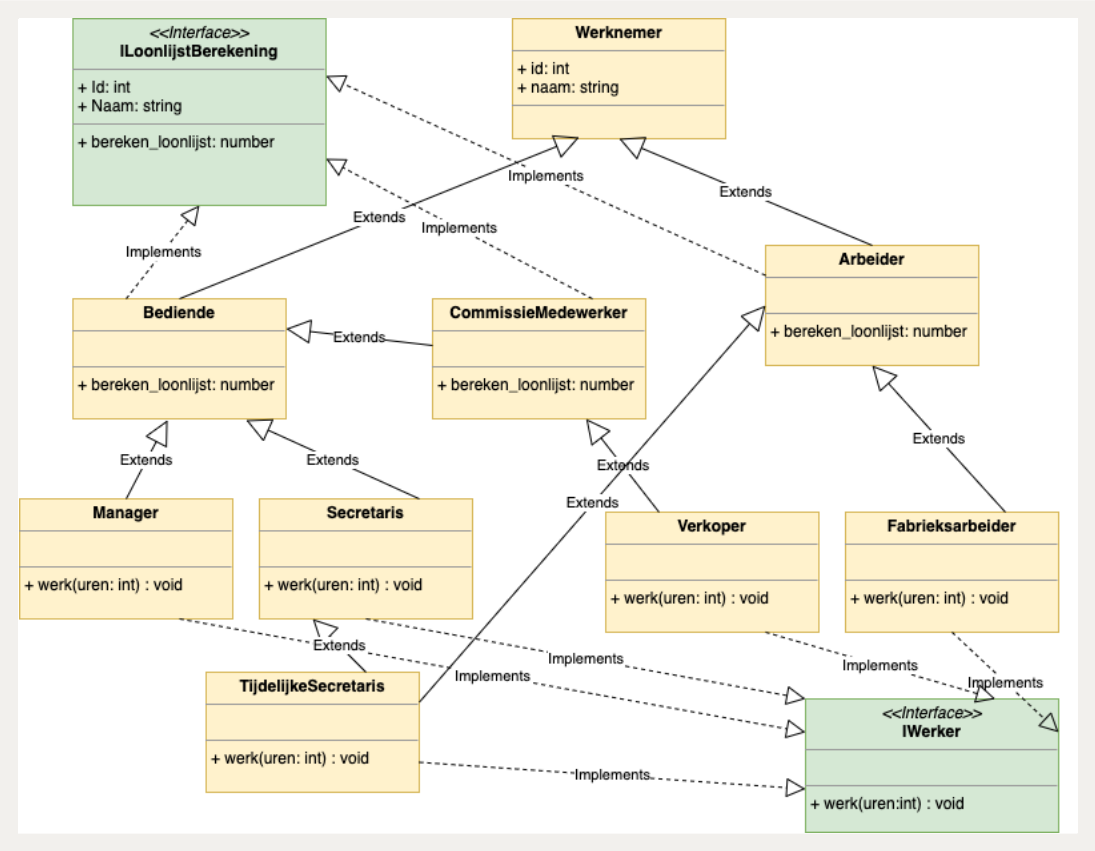
```
class TijdelijkeSecretaris(Secretaris, Arbeider):
    def __init__(self, id, naam, uren_gewerkt, uurtarief):
        Arbeider.__init__(self, id, naam, uren_gewerkt, uurtarief)

    def bereken_loonlijst(self):
        return Arbeider.bereken_loonlijst(self)
```

Gebruik meervoudige overerving

Gedeeltelijke code voor programma.py (zie code stap 2)

```
print(werknemers.TijdelijkeSecretaris.__mro__)
oTijdelijkeSecretaris1 = werknemers.TijdelijkeSecretaris(5, 'Brent Willems', 40, 30)

werknemers = [
    oManager1,
    oSecretaris1,
    oVerkoper1,
    oFabrieksArbeider1,
    oTijdelijkeSecretaris1
]
```

UML-diagram


<<Interface>>
ILoonlijstBerekening
+ Id: int
+ Naam: string
+ bereken_loonlijst: number
Werknemer
+ id: int
+ naam: string
Arbeider
+ bereken_loonlijst: number
Bediende
+ bereken_loonlijst: number
CommissieMedewerker
+ bereken_loonlijst: number
Manager
+ werk(uren: int) : void
Secretaris
+ werk(uren: int) : void
Verkoper
+ werk(uren: int) : void
Fabrieksarbeider
+ werk(uren: int) : void
TijdelijkeSecretaris
+ werk(uren: int) : void
<<Interface>>
IWerker
+ werk(uren:int) : void
Implements
Extends

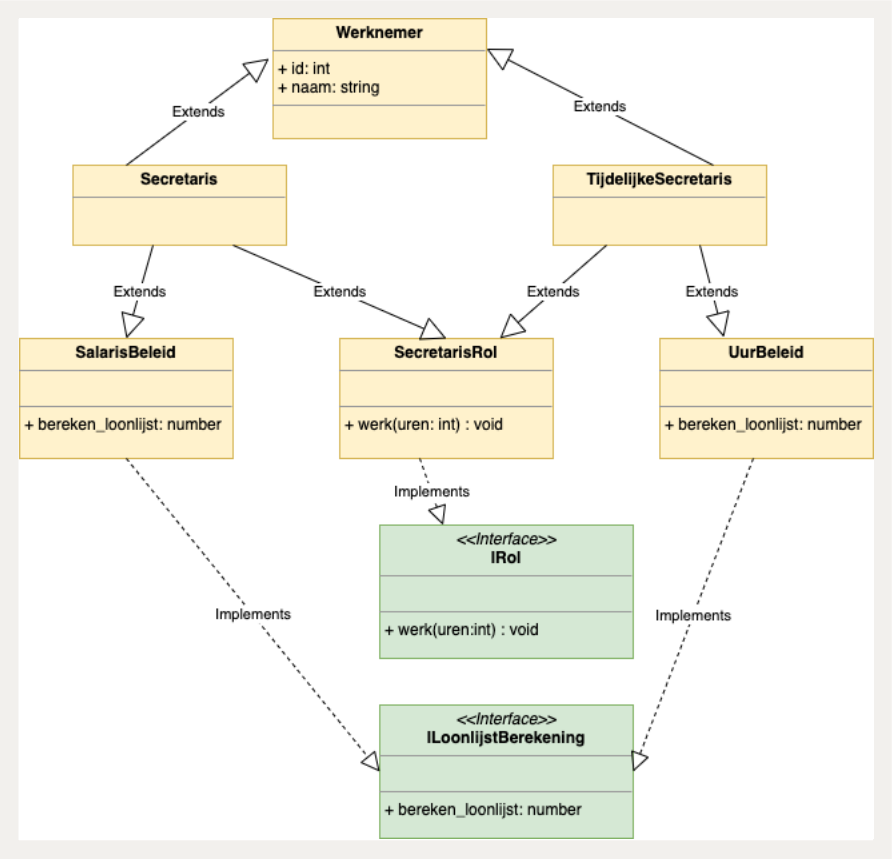
Stap 3: herdenken van onze projectstructuur

- hrm.py: klassen voor loonlijstbeleid
- productiviteit.py: klassen voor rollen mbt werknemers
- werknemers.py: klassen voor elk type werknemer
- programma.py: start script, instantieert werknemers en systemen => blijft hetzelfde

Stap 3: herdenken van onze projectstructuur

- Meervoudige overerving gebruikt om klassen los te koppelen van de verschillende systemen die ze gebruiken
- Alle code voor één systeem leeft nu in de betreffende module
- Allemaal vertrouwde code, maar op een nieuwe plaats
- Grootste wijziging: werknemers.py

UML-diagram


Werknemer
+ id: int
+ naam: string
Extends
Extends
Secretaris
TijdelijkeSecretaris
Extends
Extends
Extends
Extends
SalarisBeleid
+ bereken_loonlijst: number
SecretarisRol
+ werk(uren: int) : void
UurBeleid
+ bereken_loonlijst: number
Implements
<<Interface>>
IRol
+ werk(uren:int) : void
Implements
Implements
<<Interface>>
ILoonlijstBerekening
+ bereken_loonlijst: number

hrm.py (gedeeltelijke code)

```
class SalarisBeleid:
    def __init__(self, maandloon):
        self.maandloon = maandloon

    def bereken_loonlijst(self):
        return self.maandloon


class UurBeleid:
    def __init__(self, uren_gewerkt, uurtarief):
        self.uren_gewerkt = uren_gewerkt
        self.uurtarief = uurtarief

    def bereken_loonlijst(self):
        return self.uren_gewerkt * self.uurtarief
```

hrm.py (gedeeltelijke code)

```
class CommissieBeleid(SalarisBeleid):
    def __init__(self, maandloon, commissie):
        super().__init__(maandloon)
        self.commissie = commissie

    def bereken_loonlijst(self):
        vast = super().bereken_loonlijst()
        return vast + self.commissie
```

productiviteit.py (gedeeltelijke code)

```
class ManagersRol:
    def werk(self, uren):
        return f'leidt het team gedurende {uren} uur.'

class SecretarisRol:
    def werk(self, uren):
        return f'wijdt zich {uren} uren aan zijn administratieve taken.'

class VerkopersRol:
    def werk(self, uren):
        return f'spendeert {uren} uren aan verkoopstaken.'

class FabrieksarbeidersRol:
    def werk(self, uren):
        return f'assembleert producten voor {uren} uren.'
```

werknemers.py (gedeeltelijke code)

Maakt gebruik van meervoudige overerving!

```
class Secretaris(Werknemer, SecretarisRol, SalarisBeleid):
    def __init__(self, id, naam, maandloon):
        SalarisBeleid.__init__(self, maandloon)
        super().__init__(id, naam)

class TijdelijkeSecretaris(Werknemer, SecretarisRol, UurBeleid):
    def __init__(self, id, naam, hours_worked, hour_rate):
        UurBeleid.__init__(self, hours_worked, hour_rate)
        super().__init__(id, naam)
```

Stap 4: object compositie

- Object compositie is een OO design concept waarmee je een *heeft-een* of *deel-van* relatie uitdrukt
- Technisch gezien hebben we in ons voorbeeldproject al gebruik gemaakt van compositie
  - Onze Werknemer-klasse bvb heeft 2 attributen: id en naam
  - Deze worden geïnitialiseerd met integers en strings als data type
- Interessanter is het toevoegen van attributen met data types die we zelf creëren
- Object compositie is niets anders dan het creëren van nieuwe klasse die je als data type gebruikt binnen een andere klasse

Stap 4: object compositie

- We maken een nieuwe klasse Adres aan in contactgegevens.py
- Adres wordt dan gebruikt als een custom datatype in de Werknemer-klasse
- Adres wordt toegevoegd nadat object geïnstantieerd is
- Elke werknemer kan geen of 1 adres hebben
- Dus niet elke medewerker hoeft een adres te hebben
- We spreken hier over een "loosely coupled" relatie

contactgegevens.py

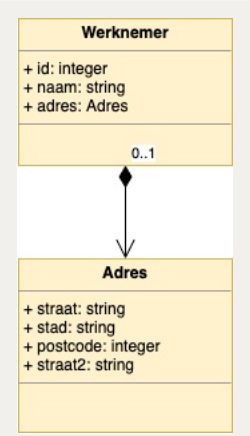
```
class Adres:
    def __init__(self, straat, stad, postcode, straat2=''):
        self.straat = straat
        self.straat2 = straat2
        self.stad = stad
        self.postcode = postcode

    def __str__(self):
        regels = [self.straat]
        if self.straat2:
            regels.append(self.straat2)
        regels.append(f'{self.postcode} {self.stad}')
        return '\n'.join(regels)
```

werknemers.py (gedeeltelijke code)

- We voegen een nieuw instance attribuut toe aan de klasse Werknemer
- Een adres wordt niet toegevoegd bij de instantiëring, maar achteraf via een method
- We hoeven op deze manier niet aan elke medewerker een adres toekennen

```
class Werknemer:
    def __init__(self, id, naam):
        self.id = id
        self.naam = naam
        # Een losse relatie: Werknemer hoeft niets te weten over Adres
        self.adres = None
```

UML-diagram


Werknemer
+ id: integer
+ naam: string
+ adres: Adres
0..1
Adres
+ straat: string
+ stad: string
+ postcode: integer
+ straat2: string

programma.py + hrm.py (gedeeltelijke code)

```
import werknemers
import contactgegevens

oManager1 = werknemers.Manager(1, 'Ben De Vuyst', 2400)
oManager1.adres = contactgegevens.Adres('Kloosterstraat 15', 'Antwerpen', 2000)
```

```
class LoonSysteem:
    ...
    for werknemer in werknemers:
        ...
        if werknemer.adres:
            print('Opsturen naar :')
            print(werknemer.adres)
        print('')
```

Stap 5: flexibel ontwerpen met object compositie

- We kunnen meer doen met object compositie en gaan hiervoor verder aanpassen in ons voorbeeldproject
- We buigen ons eerst over de klasse ProductiviteitsSysteem en voegen daar twee extra methods toe
- We gaan deze gebruiken om nieuwe rollen te instantiëren
- Merk op dat de method geef_rol() een instance teruggeeft van de gevraagde rol en geen object (zie de haakjes!)
- We maken ook een aanpassing in de volg()-method

productiviteit.py (gedeeltelijke code)

```
class ProductiviteitsSysteem:
    def __init__(self):
        # _rollen: een dictionary type
        self._rollen = {
            'manager': ManagersRol,
            'secretaris': SecretarisRol,
            'verkoop': VerkopersRol,
            'fabriek': FabrieksarbeidersRol
        }

    def geef_rol(self, rol_id):
        # dictionary: get-method
        rol_type = self._rollen.get(rol_id)
        if not rol_type:\
            raise ValueError('ongeldige rol_id')
        # rol_type() verwijst naar één van onze klassen
        return rol_type()
```

productiviteit.py (gedeeltelijke code)

```
    ...
    def volg(self, werknemers, uren):
        print('Opvolgen Productiviteit van de werknemers')
        print('========================================')
        for werknemer in werknemers:
            werknemer.werk(uren)
            #resultaat = werknemer.werk(uren)
            #print(f'{werknemer.naam}: {resultaat}')
        print('')
```

Stap 5: flexibel ontwerpen met object compositie

- We maken vervolgens aanpassingen aan het loonsysteem
- We gaan een interne databank bijhouden met loonbeleid voor elke werknemer
- We mappen hiervoor voor elke medewerker een specifiek loonbeleid aan zijn/haar id
- We slaan hiervoor specifieke loonbeleid-objecten op, en niet hun klassen
- Returnwaarden hoeven dus niet geïnstantieerd worden

hrm.py (gedeeltelijke code)

- We voegen een klasse LoonlijstBeleid toe die een base-klasse zal worden voor elke andere beleidsklasse
- Zal bijhouden hoeveel uren effectief werd gewerkt
- SalarisBeleid en UurBeleid laten we dan erven van deze base-klasse

```
class LoonlijstBeleid:
    def __init__(self):
        self.uren_gewerkt = 0

    def volg_werk(self, uren):
        self.uren_gewerkt += uren
```

hrm.py (gedeeltelijke code)

- De CommissieBeleid klasse erft nu via SalarisBeleid ook van LoonlijstBeleid
- We zorgen nu voor een commissie_per_verkoop attribuut en een commissie-method

```
class CommissieBeleid(SalarisBeleid):
    def __init__(self, maandloon, commissie_per_verkoop):
        super().__init__(maandloon)
        self.commissie_per_verkoop = commissie_per_verkoop

    def commissie(self):
        verkopen = self.uren_gewerkt / 5
        return verkopen * self.commissie_per_verkoop

    def bereken_loonlijst(self):
        vast = super().bereken_loonlijst()
        return vast + self.commissie()
```

contactgegevens.py (gedeeltelijke code)

- En, gelijkaardig aan wat we reeds gedaan hebben...

```
class AdresLijst:
    def __init__(self):
        self._werknemer_adressen = {
            1: Adres('Lange Leemstraat 34', 'Antwerpen', 2018, 'Bus 202'),
            2: Adres('Kloosterstraat 15', 'Antwerpen', 2000),
            3: Adres('Grote Markt 4', 'Antwerpen', 2000),
            4: Adres('Ellermanstraat 33', 'Antwerpen', 2060),
            5: Adres('Esmoreitlaan 7', 'Antwerpen', 2050)
        }

    def geef_werknemer_adres(self, werknemer_id):
        adres = self._werknemer_adressen.get(werknemer_id)
        if not adres:
            raise ValueError(werknemer_id)
        return adres
```

In werknemers.py

- Hier de grootste wijzigingen
- Aanmaak van een WerknemersDatabase klasse
- Een voorbeeld van gebruik van compositie
- _werknemers bevat een lijst van dictionaries, elke dictionary = een werknemer
- Deze klasse heeft ook instances nodig van het ProductiviteitsSysteem, LoonSysteem en AdresLijst.

In werknemers.py

```
from productiviteit import ProductiviteitsSysteem
from hrm import LoonSysteem
from contactgegevens import AdresLijst

class WerknemersDatabase:
    def __init__(self):
        self._werknemers = [
            {'id': 1,'naam': 'Ben De Vuyst','rol': 'manager'},
            ...
        ]
        self.productiviteit = ProductiviteitsSysteem()
        self.loonlijst = LoonSysteem()
        self.werknemer_adressen = AdresLijst()
```

In werknemers.py

- Vervolgens twee methods die voor ons een list van werknemer-objecten construeren
- Een voorbeeld van gebruik van compositie

```
def werknemers(self):
    return [self._werknemer_aanmaken(**data) for data in self._werknemers]

def _werknemer_aanmaken(self, id, naam, rol):
    adres = self.werknemer_adressen.geef_werknemer_adres(id)
    werknemer_rol = self.productiviteit.geef_rol(rol)
    loonlijst_beleid = self.loonlijst.geef_beleid(id)
    return Werknemer(id, naam, adres, werknemer_rol, loonlijst_beleid)
```

In werknemers.py nog Werknemer-klasse aanpassen

```
class Werknemer:
    def __init__(self, id, naam, adres, rol, loonlijst):
        self.id = id
        self.naam = naam
        self.adres = adres
        self.rol = rol
        self.loonlijst = loonlijst

    def werk(self, uren):
        verplichtingen = self.rol.werk(uren)
        print(f'Werknemer {self.id} - {self.naam}:')
        print(f'- {verplichtingen}')
        print('')
        self.loonlijst.volg_werk(uren)

    def bereken_loonlijst(self):
        return self.loonlijst.bereken_loonlijst()
```

Tot slot: programma.py aanpassen

```
from hrm import LoonSysteem
from productiviteit import ProductiviteitsSysteem
from werknemers import WerknemersDatabase

oProductiviteitssysteem1 = ProductiviteitsSysteem()
oLoonlijstsysteem1 = LoonSysteem()
oWerknemersdatabase1 = WerknemersDatabase()

werknemers = oWerknemersdatabase1.werknemers()
oProductiviteitssysteem1.volg(werknemers, 40)
oLoonlijstsysteem1.bereken_loonlijst(werknemers)
```

Dit design noemen we een policy-based design

- Modules zijn samengesteld uit meerdere policies die belast zijn met het uitvoeren van het werk
- Andere klasses, zoals LoonSysteem nemen informatie in en passen de juiste policy toe
- Geeft flexibiliteit in geval dat vereisten veranderen in de toekomst => zie onder
- ! Moesten we beroep gedaan hebben op inheritance, dan hadden we nu een afzonderlijke klasse moeten creëren hebben

```
manager = werknemers[0]
manager.loonlijst = UurBeleid(55)
```

Python OO - les 4 - kristof.michiels01@ap.be