OO og önnur paradigms

Hönnun og smíði hugbúnaðar

Haust 2022

Hvað er Programming Paradigm

- Forritunar „stíll“ eða „way of programming“
- Gefur þér ákveðin tól
- Setur Skorður á það sem er hægt að gera
- Programming paradigm er dæmi um Architectural þátt
- Forrit getur nýst við mörg paradigm þótt eitt sé yfirleitt mest áberandi

Dæmi

- Imperative
  - Lista niður þau skref sem forritið tekur til að fá loka niðurstöðuna
  - Object oriented programming
    - Forrit eru byggð upp af klösum/hlutum og hvernig þeir spila saman
    - Forritunarmál: Java, C#, python etc.
  - Procedural
    - Einfaldlega bara listi af leiðbeiningum þar sem föll eru notuð
    - Forritunarmál dæmi: C, BASIC, Java, C# etc.
  - Constraint
    - Constraint eru specify-uð og og ákveðinn “engine” finnur þau gildi sem uppfylla skilyrðin(constraintin)
    - Tæpt á að vera flokkað sem imperative paradigm en er oft gert í java/C#/python
    - Forritunarmál dæmi: MiniZinc

- Declerative
  - High-level rökfærsla á hvað forrit ætti að gera
  - Lýsa því sem er óskast af forritinu án þess að taka sérstaklega fram þau skref sem þarf að framkvæma (t.d. SQL: select * from users where ...)
  - Functional programming
    - Falla köll sem eru keðjuð saman, breytur eru yfirleitt „óbreytilegar (immutable)“
    - Forritunarmál dæmi: Haskell, Clojure, F#, Scala etc.
  - Logic
    - Byggt á formal logic (t.d. $(p \rightarrow q) \wedge (\neg r \rightarrow q)$),
    - Forritunarmál dæmi: Prolog

Functional Programming

- Forrit smíðuð úr „pure functions“
  - Engin side effect
  - Skila alltaf sama gildi við sama inntak
- Breytur eru yfirleitt „óbreytilegar(immutable)“
- Ekkert shared state(hvorki global né local)
- Styður ekki lykkjur eða if-else statements eins og það þekkist í imperative paradigms
- Oft notað fyrir concurrency eða stærðfræðilega reikninga

Kostir

- Engin óþekkt side effect
- Modular og testable
- Maintainable
- Gerir concurrency „auðvelt“

Gallar

- Erfiðara en imperative
- Ólæsilegra
- Getur verið meira computation heavy
- Getur tekið upp meira minni

Functional vs Imperative dæmi

```
const numList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
let result = 0;
for (let i = 0; i < numList.length; i++) {
  if (numList[i] % 2 === 0) {
    result += numList[i] * 10;
  }
}
```

Imperative

```
const result = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
               .filter(n => n % 2 === 0)
               .map(a => a * 10)
               .reduce((a, b) => a + b);
```

Functional

Object oriented programming

- Meginn fókus áfangans
- Imperative
- Forrit byggð upp með klösum/hlutum og hvernig þeir tala saman
- Klasar eru ákveðin blueprint fyrir því hvernig á að smíða hluti, þeir lýsa hlutunum
- Hlutir eru síðan „instances“ af klösunum og hafa sitt eigið state, hegðun og identity
- Forritarar eiga til með að gleyma mátt OOP

Kostir

- Brýtur forrit upp í einfaldar, endurnýtanlegar einingar
- Auðvelt að lesa og skrifa
- Verndar upplýsingar með encapsulation
- Hefur inheritance, polymorphism og abstractions

Gallar

- Shared state
- Side effects
- Concurrency

OOP frh. Klasi vs Hlutur

```python
class Dog:
  def __init__(self, name: str) -> None:
      self.name = name

  def bark(self):
    print(f'My name is {self.name}')
```

Klasi

```python
dog = Dog('Anton')
dog.bark() # prints: My name is Anton
```

Hlutur

The Four Pillars Of OOP

- Inheritance
- Polymorphism
- Encapsulation
- Abstractions

Inheritance

- Leyfir klösum að erfa virkni frá öðrum klösum
- Ein leið til að extend-a virkni frá öðrum klösum
- Ein leið til að styðja við endurnýtanleika
- Subclass meira specific
  - Golden Retriever er meira specific en Hundur
  - Subclass-ar eiga að geta hagað sér sem parent klasar (*Liskov substitution principle*)

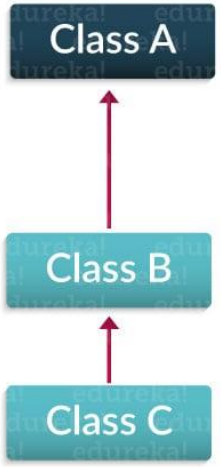
Inheritance dæmi

```python
class Dog:
  def __init__(self, name: str) -> None:
      self._name = name

  def bark(self):
    print(f'My name is {self._name}')


class GoldenRetriever(Dog):
  def retrieve_gold(self):
    return "gold"

  def bark(self):
    super().bark()
    print("and I am a Golden Retriver")
```

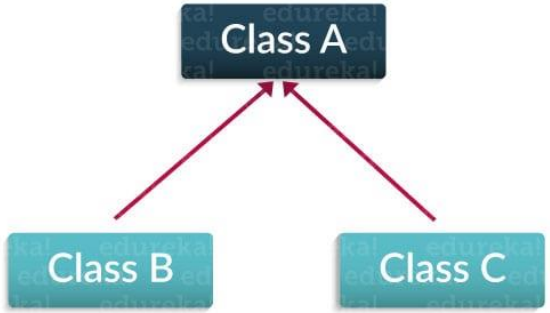
```python
goldenRetriver = GoldenRetriever('Anton')
goldenRetriver.bark() # prints: My name is Anton
                      #         and I am a Golden Retriever

print(goldenRetriver.retrieve_gold()) # prints: gold
```

Types Of Inheritance

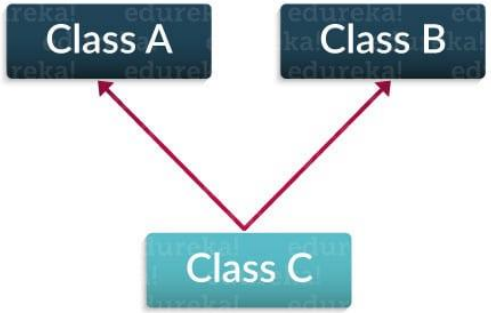
Single Inheritance



Multilevel Inheritance



Hierarchical Inheritance



Multiple Inheritance

Polymorphism

- Bein þýðing:
  - „ the condition of occurring in several different forms.“
- Nokkrar mismunandi tegundir af Polymorphism:
  - Polymorphism sem method overloading
  - Polymorphism með shared interface
  - Polymorphism sem method overriding

Polymorphism sem method overloading

- Fall með sama nafn en tekur inn aðra parametra og/eða skilar út öðrum gildum
- Í raun ekki hægt í Python, að minnsta kosti ekki innbyggð virkni eins og í tungumálum eins og Java og C#

```java
public class CustomPrinter{
    public void Print(String string) {
        System.out.println(string);
    }

    public void Print(String[] strings) {
        for (int i = 0; i < strings.length; i++) {
            Print(strings[i]);
        }
    }

   public static void main(String []args){
      CustomPrinter printer = new CustomPrinter();
      printer.Print("hello"); // output: hello

      String[] strings = {"hello", "there"};
      printer.Print(strings); // output: hello \n there
    }
}
```

Java

```python
class CustomPrinter():
    def __is_list_of_strings(self, object: any) -> bool:
        bla = isinstance(object, list)
        blabla = all(isinstance(s, str) for s in object)
        return bla and blabla

    def print(self, object_to_print: any):
        if isinstance(object_to_print, str):
            print(object_to_print)
        elif self.__is_list_of_strings(object_to_print):
            for elem in object_to_print:
                print(elem)


if __name__ == '__main__':
    test = CustomPrinter()
    test.print("hello")  # output: hello
    test.print(["hello", "there"])  # output: hello \n there
```

Python

Polymorphism með shared interface

```python
class Cat:
    def make_noise(self):
        print("rawr")


class Car:
    def make_noise(self):
        print("vroom vroom")


noise_makers = [Cat(), Car()]
for object in noise_makers:
    object.make_noise()

# ouput:
  # rawr
  # vroom vroom
```

Án explicit interface

```python
from abc import ABC, abstractmethod
from typing import List


class NoiseMakerInterface(ABC):
    @abstractmethod
    def make_noise(self):
        pass


class Cat(NoiseMakerInterface):
    def make_noise(self):
        print("rawr")


class Car(NoiceMakerInterface):
    def make_noise(self):
        print("vroom vroom")


noise_makers = [Cat(), Car()]  # type: List[NoiceMakerInterface]
for object in noise_makers:
    object.make_noise()

# ouput:
  # rawr
  # vroom vroom
```

Með explicit interface

Polymorphism með method overriding

- Method overriding er þegar klasi erfir annan klasa og yfirskrifar síðan fall sem var nú þegar skilgreint í parent klasanum

```python
class Dog:
    def __init__(self, name: str) -> None:
        self._name = name

    def bark(self):
        print(f'My name is {self._name}')


class GoldenRetriever(Dog):
    def retrieve_gold(self):
        return "gold"

    def bark(self):
        print(
            f"bark bark I am a Golden Retriver and my name is {self._name}")


goldenRetriver = GoldenRetriever('Anton')
goldenRetriver.bark()  # outputs: bark bark I am a Golden Retriver and my name is An
ton
```

Encapsulation

- Einfaldlega það að fela gögn og virkni
- Hægt að fela breytur og virkni í hlutum með því að merkja það sem private/protected
- Hver sem er á ekki að geta lesið state-ið á hlut eða verra, breytt því
- Notandi klasans á ekki að vera coupled við innri virkni hans

```python
class Dog:
    def __init__(self, name: str, weight: float) -> None:
        self._name = name
        self.__weight = weight

    def set_weight(self, weight: float):
        if weight <= 0:
            raise Exception("Weight cannot be less or equal to 0")

        self.__weight = weight

    def walk(self):
        if self.__weight < 60:
            print("tip tap")
        elif 60 <= self.__weight < 100:
            print("clonk")
        else:
            print("thunk")

    def bark(self):
        print(f'My name is {self._name}')


class GoldenRetriever(Dog):
    def retrieve_gold(self):
        return "gold"

    def bark(self):
        super().bark()
        print(f"and I am a Golden Retriver who weighs {self.__weight}")


class SiberianHusky(Dog):
    def bark(self):
        print(
            f"bark bark I am a husky and my name is {self._name}")
```

```python
# kastar ekki villu því _name er protected en ekki private sem þýðir að börnin hafa aðgang að honum
dog = SiberianHusky("Moon Moon", 50)
dog.bark()  # outputs: bark bark I am a husky and my name is Moon Moon
```

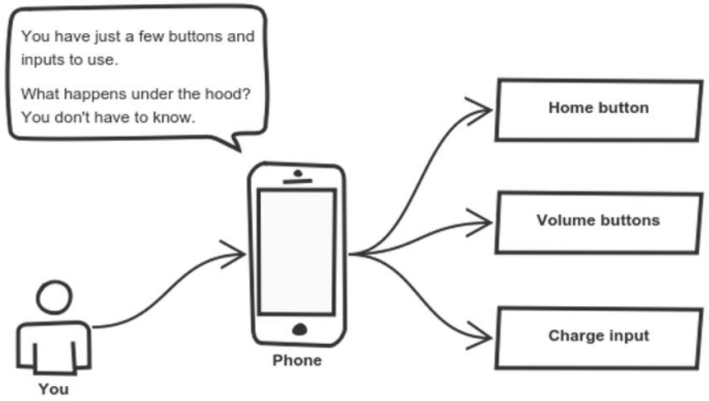
```python
dog.walk()  # outputs: tip tap
dog.set_weight(100)
dog.walk()  # outputs: thunk
```

```python
# kastar villu því __weight er private breyta í Dog
dog = GoldenRetriever("Anton", 60)
dog.bark()
```

```python
dog = Dog("Snati", 50)
print(dog._name)  # kastar villu því breytan er protected
print(dog.__weight)  # kastar villu því breytan er private
dog.set_weight(-10)  # kastar villu því við leyfum ekki neikvæðar tölur
```

Abstractions

- Á íslensku: útdráttur, samantekt
- Klasar expose-a bara nokkrum high-level föllum og breytum sem notandi getur notað til að vinna með instance af klasanum þ.e.a.s public föll og public breytur. Þessi public föll/breytur mynda til samans *abstraction* af innri virkni/útfærslu klasans.
- Extension af Encapsulation
- High-level user interface
- Felur í burtu implementation detail




```python
class Cat:
    def __init__(self, weight: float) -> None:
        self.__weight = weight

    def make_noise(self):
        print(self.__get_noise())

    def __get_noise(self):
        if self.__weight > 60:
            return "rawr"
        else:
            return "meow"
```

```python
cat = Cat(100)
cat.make_noise()

# throws AttributeError:
#'Cat' object has no attribute '__get_noise'
print(cat.__get_noise())
```

Implicit interface

```python
class NoiseMakerInterface(ABC):
    @abstractmethod
    def make_noise(self):
        pass


class Cat(NoiseMakerInterface):
    def __init__(self, weight: float) -> None:
        self.__weight = weight

    def make_noise(self):
        print(self.__get_noise())

    def __get_noise(self):
        if self.__weight > 60:
            return "rawr"
        else:
            return "meow"
```

```python
cat = Cat(100)
cat.make_noise()
# throws AttributeError:
'Cat' object has no attribute '__get_noise'
print(cat.__get_noise())
```

Explicit interface

Abstractions frh. Dæmi í C#

```
public interface NoiseMakerInterface {
  public void MakeNoise();
}

public class Cat : NoiseMakerInterface {
  private double weight;

  public Cat(double weight) {
    this.weight = weight;
  }

  public void MakeNoise() {
    Console.WriteLine(getNoise());
  }

  private void getNoise() {
    if (weight > 60) {
      return "rawr"
    }
    return "meow"
  }
}
```

Interface í C#

```
public abstract class NoiseMaker {
  public abstract MakeNoise();
}

public class Cat : NoiseMaker {
  private double weight;

  public Cat(double weight) {
    this.weight = weight;
  }

  public override void MakeNoise() {
    Console.WriteLine(getNoise());
  }

  private void getNoise() {
    if (weight > 60) {
      return "rawr"
    }
    return "meow"
  }
}
```

Abstract klasi í C#