Almene programmerings principper

Det er muligt at skrive kode der virker og kører uden at bekymre sig om programmerings principper og paradigmer, men det bliver hurtigt et problem når systemer skal skaleres op.
At følge diverse gode programmerings principper kan gøre det nemmere at skalere systemer op og udvide, kan gøre udviklingen hurtigere, koden nemmere at forstå og nemmere at vedligeholde.
I dette afsnit følger en række udvalgte programmerings principper som er gode at følge og som vil øge kvaliteten af koden.

SOLID

SOLID er en samling af fem designprincipper, der hjælper med at opbygge fleksible og vedligeholdelsesvenlige systemer:

1. **Single Responsibility Principle (SRP)**:
   En klasse bør have én, og kun én, grund til at ændre sig. Dette betyder, at hver klasse skal have et klart og specifikt ansvar, hvilket gør systemet lettere at forstå og vedligeholde. En, af forfatteren, opdateret version af princippet siger at et modul bør være ansvarligt overfor én, og kun én aktør. Begge synspunkter understøtter princippet om at holde ansvar klart afgrænset for at undgå kompleksitet og utilsigtede afhængigheder.
2. **Open/Closed Principle (OCP)**:
   Klasser, moduler, funktioner osv. skal være åbne for udvidelse, men lukkede for ændringer. Dette betyder, at du kan tilføje ny funktionalitet uden at ændre eksisterende kode, hvilket beskytter mod utilsigtede fejl.
3. **Liskov Substitution Principle (LSP)**:
   Objekter af en basisklasse skal kunne erstattes af objekter af deres afledte klasser uden at ændre den korrekte funktionalitet af programmet. Dette sikrer, at polymorfi fungerer korrekt og forudsigeligt.
4. **Interface Segregation Principle (ISP)**:
   Klienter skal ikke tvinges til at afhænge af interfaces, de ikke bruger. Dette betyder, at interfaces skal være specifikke og tilpasset de nøjagtige behov, hvilket gør systemet mere fleksibelt. Hvis princippet virker lidt mærkeligt, eller hvis der er noget forviring omkring hvad der præcis menes kan man kigge her: Design Principles and Design Patterns og The Interface Segregation Principle
5. **Dependency Inversion Principle (DIP)**:
   Højniveau-moduler bør ikke afhænge af lavniveau-moduler; begge skal afhænge af abstraktioner. Dette gør systemet mere robust og lettere at ændre eller udvide uden at påvirke andre dele.

OOP (Object-Oriented Programming)

OOP er et programmeringsparadigme der organiserer kode i objekter, der kombinerer data og funktionalitet. Dette gør koden mere struktureret og genanvendelig:

1. **Klasser og objekter**:
   Klasser definerer strukturen for objekter, som kombinerer data (attributter) og funktioner (metoder).
   Objekter er instanser af klasser og repræsenterer individuelle enheder med deres egen tilstand og adfærd.
2. **Indkapsling**:
   Beskytter klassens interne tilstand ved at skjule data og kun tillade adgang via definerede metoder, hvilket reducerer utilsigtet adgang og manipulation.
3. **Arv**:
   Tillader en klasse at arve egenskaber og metoder fra en anden klasse, hvilket fremmer kodegenbrug og skabelse af klassehierarkier.
4. **Polymorfi**:
   Tillader at forskellige klasser behandles ens via en fælles superklasse eller interface, hvilket giver fleksibilitet og dynamisk adfærd baseret på objekttype.
5. **Abstraktion**:
   Fokuserer på essentielle egenskaber og skjuler detaljer, hvilket gør koden enklere at arbejde med og lettere at ændre eller udvide.

CRUD (Create, Read, Update, Delete)

CRUD repræsenterer de fire basale operationer for at håndtere data i en applikation:

- **Create**: Tilføjelse af nye poster til databasen.
- **Read**: Hentning af eksisterende data fra databasen.
- **Update**: Ændring af eksisterende data i databasen.
- **Delete**: Fjernelse af data fra databasen.

DRY (Don't Repeat Yourself)

DRY-princippet sikrer, at hver del af applikationen kun har én kilde til sandhed. Dette princip hjælper med at reducere redundans i koden, hvilket gør det lettere at vedligeholde og reducere fejl. I vores projekter vil vi anvende DRY til at centralisere fælles logik, såsom datahåndtering fra API'er, så vi undgår at duplikere kode flere steder i systemet.

KISS (Keep It Simple, Stupid)

KISS-princippet minder os om at holde koden så enkel som muligt. Unødvendig kompleksitet bør undgås, hvilket gør systemet lettere at forstå, teste og vedligeholde. I vores projekter betyder det, at vi fokuserer på klare og direkte løsninger, især når vi bygger brugergrænsefladen og håndterer brugerinteraktioner.

YAGNI (You Aren't Gonna Need It):

YAGNI-princippet handler om at undgå at implementere funktionalitet, før den faktisk er nødvendig. Det hjælper med at reducere unødig kompleksitet og holder projektets fokus på det, der er aktuelt og relevant. I vores system betyder det, at vi kun implementerer de funktioner, der er nødvendige for den nuværende version af produktet, og undgår at forudse fremtidige behov, som måske aldrig opstår.

Functional Programming:

Funktionel programmering fokuserer på brugen af rene funktioner og undgåelse af tilstand og bivirkninger. Dette er stærk i modsætning til OOP, som gør ekstremt brug af tilstande og bivirkninger. Dette princip hjælper med at skrive kode, der er forudsigelig, nem at teste og lettere at vedligeholde. I vores projekter kan funktionel programmering f.eks. anvendes til at behandle og transformere data hentet fra API'er på en måde, der er sikker og effektiv. Dog bliver dette ikke anvendt så meget ude i industrien, som trækker sig imod større brug af OOP. Derfor vil vi lægge større vægt på brugen af OOP her på kurset.

Ressourcer

Her er en samling af forskellige ressourcer relateret til dette dokument.

Øvelser i polymorfisme.

- Python-Course.eu - Polymorphism: https://www.python-course.eu/python3_polymorphism.php

Abstrakte klasser og metoder, flere arv, og mixins.

- JournalDev - Python Abstract Class: https://www.journaldev.com/15911/python-abstract-class
- Real Python - Inheritance and Composition: A Python OOP Guide: https://realpython.com/inheritance-composition-python

Solid.

- SOLID: Design Principles and Design Patterns
- Interface Segregation Principle: The Interface Segregation Principle