Lecture 03: Object-Oriented Analysis & Design
IN710: Object-Oriented Systems Development
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Tuesday, 25 February

Lecture 02: Python 2 Recap

- Functional programming
- Context managers
- Other in-built functions

New GitHub Setup

- Course materials repository has been archived
- Create new repository for OOSD
- Submoduling practicals & course materials repositories
- Git commands:
  - git submodule add <url to repo>
  - git submodule foreach git pull

Lecture 03: Object-Oriented Analysis & Design Topics

- Object-oriented paradigm
- Object-oriented analysis, design & modeling
- KISS
- DRY
- YAGNI
- SOLID

Object-Oriented Paradigm

- First OO programming language - Simula
  - Developed in the 1960s
  - Norwegian Computing Center
- First pure OO programming language - Smalltalk
  - Developed in the 1970s
  - Learning Research Group at Xerox PARC
- Objective-C & C++
  - Developed in the mid-1980s
  - Brad Cox & Bjarne Stroustrup
- Dominant programming paradigm
  - Early & mid-1990s
  - Rising popularity of GUIs. For example, Cocoa on macOS

Object-Oriented Analysis

- Model of the system's functional requirements
- Independent of implementation constraints
- Requirements organised around objects
- Behaviours (processes) & states (data) modeled after real world objects
  - Considered separately. For example, flows charts (behaviours) & entity-relationship diagrams (states)
- Common models - use cases & object models

Object-Oriented Design

- Planning a system of interacting objects
- Purpose of solving a software problem
- Applying implementation constraints to the conceptual model
- Designing software architectures by applying architectural patterns & design patterns

Object-Oriented Modeling

- Common approach to modeling applications & systems
- A technique heavily used by both OOA & OOD
- Modeling of dynamic behaviour
- Modeling of static structures
- Benefits of OOM:
  - Efficient & effective communication
  - Useful & stable abstraction

Learning Activity: System Design (30 minutes)

- In groups of three or four, design a URL shortener
- Why do we need a URL shortener?
- System requirements
  - Functional
  - Non-functional
- Constraints
- Database design
- System design & algorithm

KISS

- Keep it simple stupid
- Noted by the U.S. Navy in 1960
- Simplicity in design
- Unnecessary complexity should be avoided
- Cyclomatic complexity (code smell)
- ValueError exception

```
def cyclomatic_complexity(day):
    if day == 1:
        return 'Monday'
    elif day == 2:
        return 'Tuesday'
    elif day == 3:
        return 'Wednesday'
    elif day == 4:
        return 'Thursday'
    elif day == 5:
        return 'Friday'
    elif day == 6:
        return 'Saturday'
    elif day == 7:
        return 'Sunday'
    else:
        raise ValueError('day must be between 1 & 7.')

print(cyclomatic_complexity(6))
```

KISS

- Code refactoring

```
def cyclomatic_complexity(day):
    days = ('Monday', 'Tuesday', 'Wednesday',
            'Thursday', 'Friday', 'Saturday', 'Sunday')
    if day >= 1 and day <= 7:
        return days(day - 1)
    else:
        raise ValueError('day must be between 1 & 7.')

print(cyclomatic_complexity(6))
```

DRY

- Don’t repeat yourself
- Reduce repetition of information
- Single representation
- WET

YAGNI

- You aren't gonna need it
- Principle of extreme programming (XP)
- Functionality shouldn't be added until deemed necessary

SOLID

- Single responsibility
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Single Responsibility

- Every module, class & function should have a single responsibility
- Responsibility should be encapsulated by the module, class or function
- God object (anti-pattern)

Single Responsibility

- Computer class
- Two responsibilities - CPU & RAM

```
class Computer:
    def fetch(self):
        print('Fetching...')

    def decode(self):
        print('Decoding...')

    def execute(self):
        print('Executing...')

    def read(self):
        print('Reading...')

    def write(self):
        print('Writing...')
```

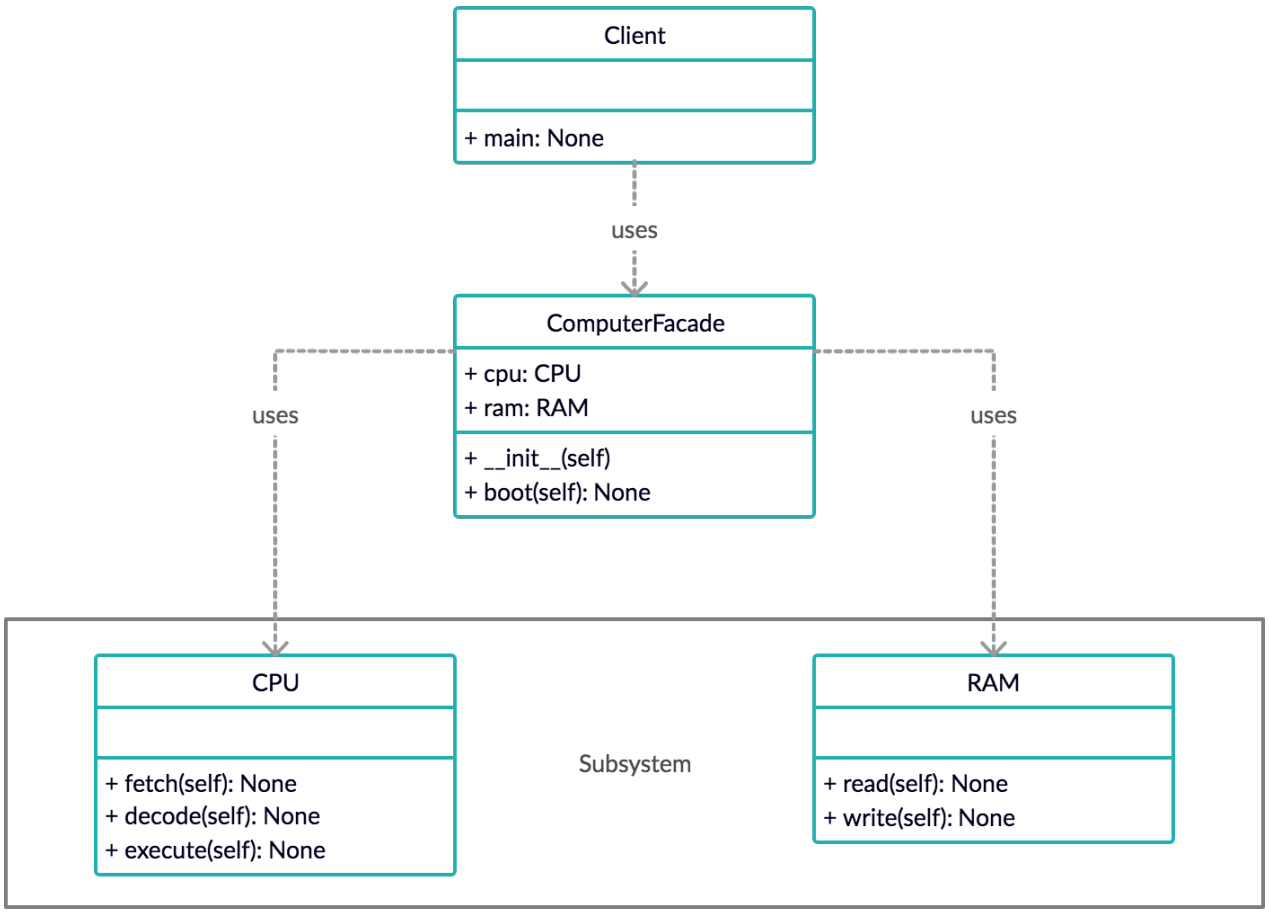
Single Responsibility

- CPU & RAM class

```
class CPU:
    def fetch(self):
        print('Fetching...')

    def decode(self):
        print('Decoding...')

    def execute(self):
        print('Executing...')

class RAM:
    def read(self):
        print('Reading...')

    def write(self):
        print('Writing...')
```

Façade Pattern: UML

- Consider the following UML diagram:



Façade Pattern

- Structural pattern
- A simple interface to the complex logic of one or several subsystems
- Delegates the client's request(s) to the appropriate objects within the subsystem(s)

```
class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.ram = RAM()

    def boot(self):
        self.cpu.fetch()
        self.ram.read()
        self.cpu.decode()
        self.ram.write()
        self.cpu.execute()

def main():
    computer_facade = ComputerFacade()
    computer_facade.boot()

if __name__ == '__main__':
    main()
```

Open-Closed

- Open for extension
- Closed for modification
- Modules, classes & methods

Open-Closed

- Animal class

```
class Animal:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

def main():
    animals = (
        Animal('Cow'),
        Animal('Pig')
    )

    for a in animals:
        if a.name == 'Cow':
            print('Moo!')
        elif a.name == 'Pig':
            print('Oink!')

if __name__ == '__main__':
    main()   # Moo!
             # Oink!
```

Open-Closed

- If we want to add a new animal, we have to modify the main function

```
class Animal:
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

def main():
    animals = (
        Animal('Cow'),
        Animal('Pig'),
        Animal('Sheep')
    )

    for a in animals:
        if a.name == 'Cow':
            print('Moo!')
        elif a.name == 'Pig':
            print('Oink!')
        elif a.name == 'Sheep':
            print('Baa!')

if __name__ == '__main__':
    main()  # Moo!
            # Oink!
            # Baa!
```

Open-Closed

- Cow, Pig & Sheep class extend from the Animal class
- Each child class has their own implementation of the sound() abstract method

```
from abc import ABC, abstractmethod

class Animal(ABC):
    def __init__(self, name):
        self.__name = name

    @property
    def name(self):
        return self.__name

    @abstractmethod
    def sound(self):
        pass

class Cow(Animal):
    def sound(self):
        print('Moo!')

class Pig(Animal):
    def sound(self):
        print('Oink!')

class Sheep(Animal):
    def sound(self):
        print('Baa!')
```

Liskov Substitution

- Barbara Liskov - Turing Award winner
- If S is a subtype of T then objects of type T may be replaced with objects of type S without altering any of the desirable properties of the program
- Strong behavioural subtyping
- **Task 2 in today's practical - square/rectangle problem**

Interface Segregation

- First used by Robert C. Martin while consulting at Xerox
- Xerox had created a new printer system that could perform a variety of tasks
- The software for this system was created from the ground up
- As the software grew, making modifications became more & more difficult
- The design problem was a single class was used by almost all of the tasks
- Clients shouldn't be forced to depend upon interfaces that they don't use

Interface Segregation

- Circle & Square class extend from the Shape class
- Each child class has their own implementation of the draw_circle() & draw_square() abstract method

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw_circle(self):
        pass

    @abstractmethod
    def draw_square(self):
        pass

class Circle(Shape):
    def draw_circle(self):
        pass

    def draw_square(self):
        pass

class Square(Shape):
    def draw_circle(self):
        pass

    def draw_square(self):
        pass
```

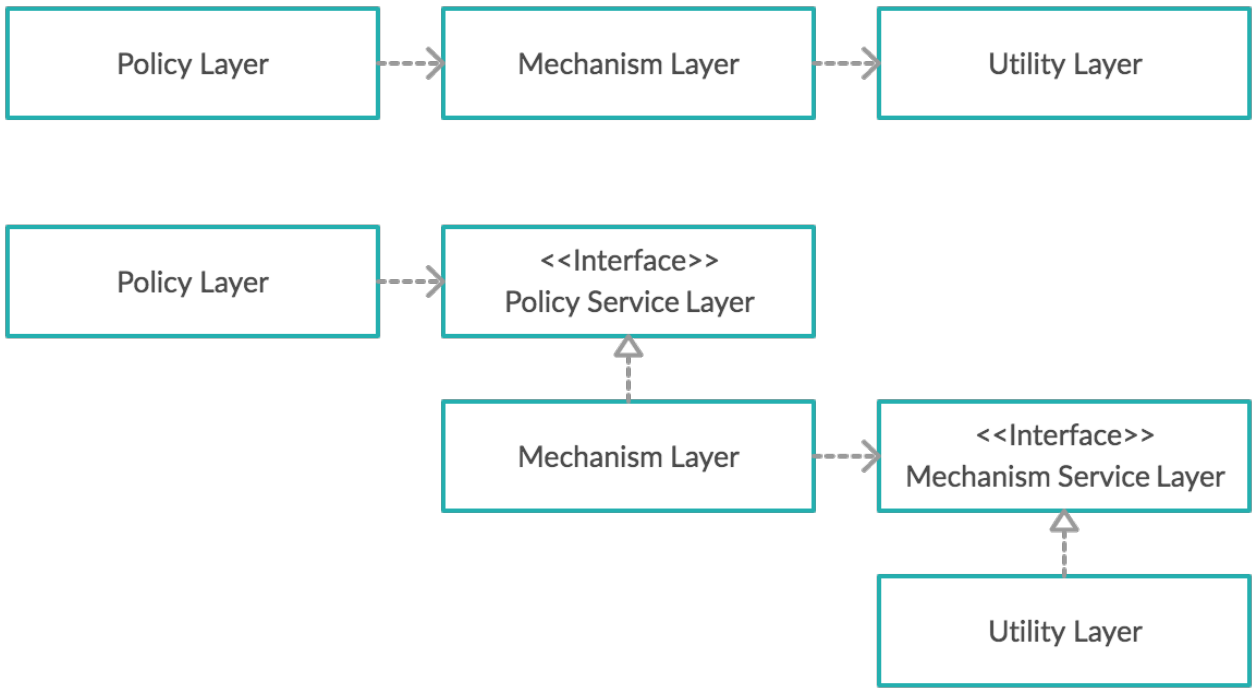
Interface Segregation

- Each child class has their own implementation of the draw() abstract method

```
from abc import ABC, abstractmethod

class Shape(ABC):
    @abstractmethod
    def draw(self):
        pass

class Circle(Shape):
    def draw(self):
        pass

class Square(Shape):
    def draw(self):
        pass
```

Dependency Inversion

- High-level modules shouldn't depend on low-level modules
- Both should depend on abstractions
- Abstractions shouldn't depend on details
- Details should depend on abstractions



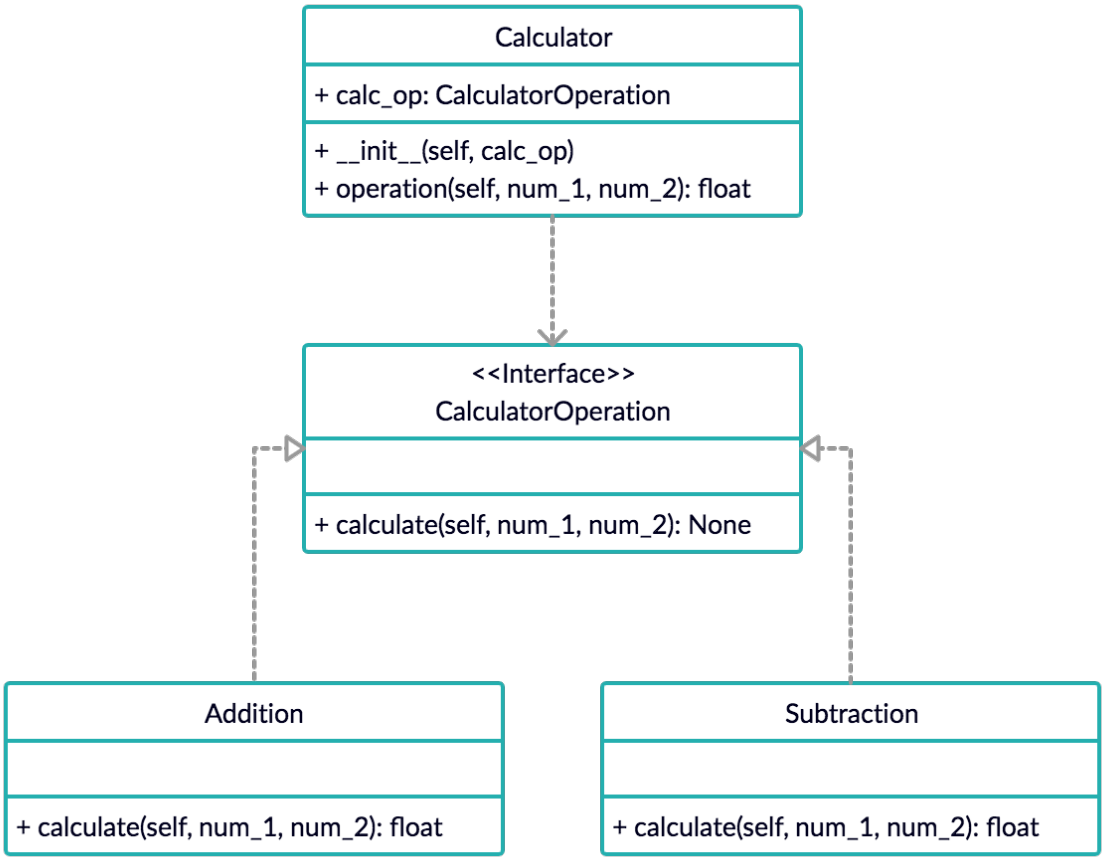
Dependency Inversion

- Calculator class

```
class Calculator:
    def addition(self, num_1, num_2):
        return num_1 + num_2

    def subtraction(self, num_1, num_2):
        return num_1 - num_2

def main():
    calc = Calculator()
    print(calc.addition(10, 5))
    print(calc.subtraction(10, 5))

if __name__ == '__main__':
    main()  # 15
            # 5
```

Dependency Inversion: UML

- Consider the following UML diagram:



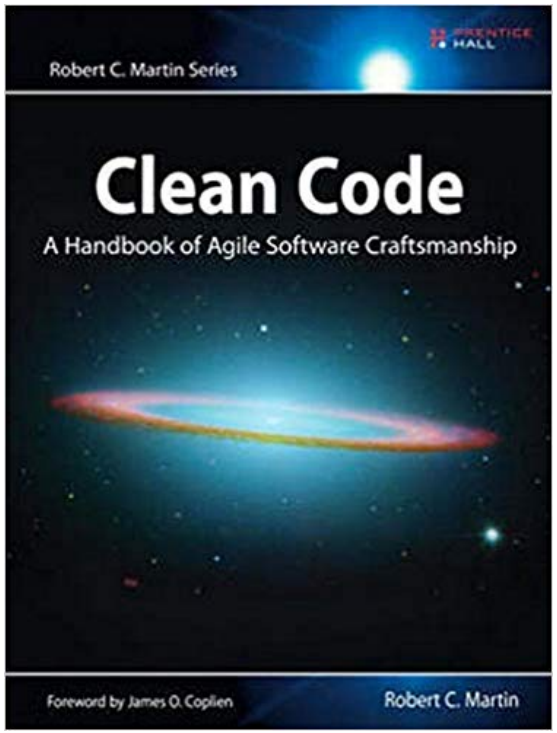
Dependency Inversion

- Dependency injection via constructor

```
from abc import ABC, abstractmethod

class CalculatorOperation(ABC):
    @abstractmethod
    def calculate(self, num_1, num_2):
        pass

class Calculator:
    def __init__(self, calc_op):
        self.calc_op = calc_op

    def operation(self, num_1, num_2):
        return self.calc_op.calculate(num_1, num_2)

class Addition(CalculatorOperation):
    def calculate(self, num_1, num_2):
        return num_1 + num_2

class Subtraction(CalculatorOperation):
    def calculate(self, num_1, num_2):
        return num_1 - num_2

def main():
    addition = Calculator(Addition())
    subtraction = Calculator(Subtraction())
    print(addition.operation(10, 5))
    print(subtraction.operation(10, 5))

if __name__ == '__main__':
    main()  # 15
            # 5
```

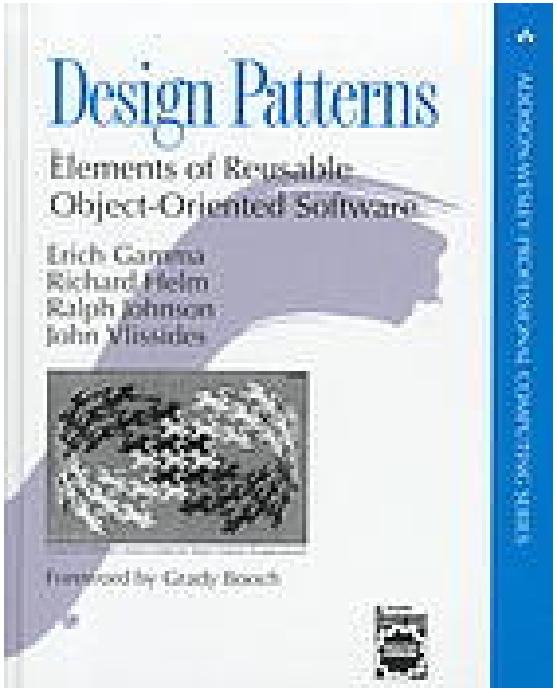
Recommended Reading

- Clean Code: A Handbook of Agile Software Craftsmanship
- Robert C. Martin



Recommended Reading

- Design Patterns: Elements of Reusable Object-Oriented Software
- Gang of Four (GoF)



Practical

- Series of tasks covering today's lecture
- Worth 1% of your final mark for the Object-Oriented Systems Development course
- Deadline: Tuesday, 10 March at 5pm

Lecture 04: Exceptions & Automation Testing Topics

- Syntax errors
- Exceptions
- Automation testing
  - Unit testing
  - Integration testing
  - End-to-end testing
  - User acceptance testing
- Software development testing practices
  - Test-driven development
  - Continuous integration