



# Lecture 14: Proxy Pattern

## IN710: Object-Oriented Systems Development

### Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Thursday, 23 April

# LECTURE 13: TEMPLATE PATTERN RECAP

- ▶ Design pattern 09: template pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ Real world analogy
  - ▶ UML & implementation
  - ▶ Pros & cons

# LECTURE 14: PROXY PATTERN TOPICS

- ▶ Design pattern 10: proxy pattern
  - ▶ Definition
  - ▶ Problem/solution
  - ▶ UML & implementation
  - ▶ Pros & cons

# PROXY PATTERN: GoF

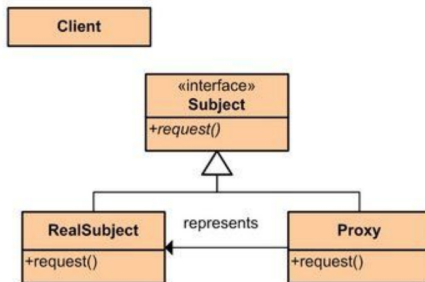
## ► GoF definition & UML

### Proxy

**Type:** Structural

**What it is:**

Provide a surrogate or placeholder for another object to control access to it.



# PROXY PATTERN: DEFINITION

- ▶ Structural pattern
- ▶ A class functioning as an interface
- ▶ The proxy could interface with anything, for example, a network connection
- ▶ A wrapper called by the client to access serving objects behind the scenes

# PROXY PATTERN: PROBLEM

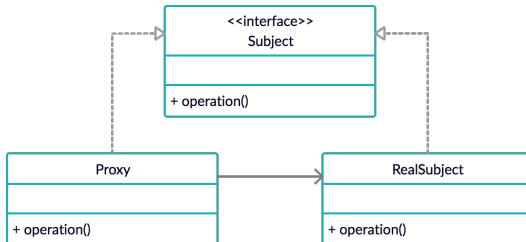
- ▶ An object that consumes a large amount of system resources
- ▶ Lazy initialisation

# PROXY PATTERN: SOLUTION

- ▶ Create a new proxy class with the same interface as the service object
- ▶ Pass the proxy object to all of the client's objects
- ▶ Proxy creates a service object and delegates the work to it

# PROXY PATTERN: UML

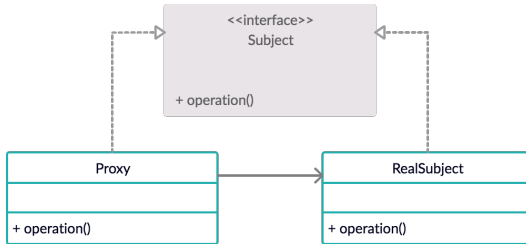
- Consider the following UML diagram:





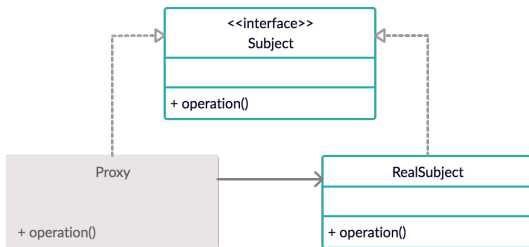
# PROXY PATTERN: UML

## ► Subject interface class



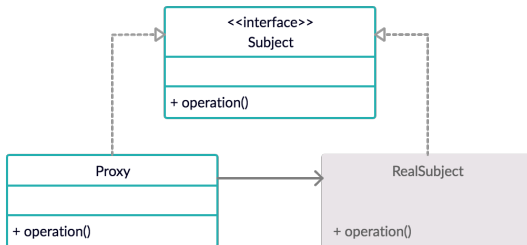
# PROXY PATTERN: UML

- ▶ Proxy class
- ▶ Implements subject interface class
- ▶ Can act as substitute for subject objects
- ▶ Maintains a reference to the substituted object



# PROXY PATTERN: UML

- RealSubject class
- Substituted object



# PROXY PATTERN: IMPLEMENTATION

```
from abc import ABC, abstractmethod

class AbstractCar(ABC):
    @abstractmethod
    def drive(self):
        pass

class Car(AbstractCar):
    def drive(self):
        print('The driver is old enough to drive')

class Driver:
    def __init__(self, age):
        self.age = age
```

# PROXY PATTERN: IMPLEMENTATION

```
class ProxyCar(AbstractCar):
    def __init__(self, driver):
        self.car = Car()
        self.driver = driver

    def drive(self):
        if self.driver.age <= 16:
            print('The driver is too young to drive')
        else:
            self.car.drive()

def main():
    driver = Driver(16)
    car = ProxyCar(driver)
    car.drive()

    driver = Driver(25)
    car = ProxyCar(driver)
    car.drive()

if __name__ == '__main__':
    main() # The driver is too young to drive
          # The driver is old enough to drive
```

# PROXY PATTERN: PROS

- ▶ New proxies can be introduced without having to change the existing services or clients
- ▶ Service object can be controlled without the clients knowing about it
- ▶ The proxy works even if the service object isn't available

## PROXY PATTERN: CONS

- Overall, the complexity of the code increases - new classes need to be introduced

# PRACTICAL

- ▶ Series of tasks covering today's lecture
- ▶ Worth 1% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Friday, 12 June at 5pm