Lecture 05: Strategy Pattern
IN710: Object-Oriented Systems Development
Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Tuesday, 3 March

Lecture 04: Exceptions & Automation Testing Recap

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

Lecture 05: Strategy Pattern Topics

- Design pattern 01: strategy pattern
  - Definition
  - Problem & solution
  - Real world analogy
  - UML & implementation
  - Open-closed principle
  - Pros & cons

Strategy Pattern: Definition

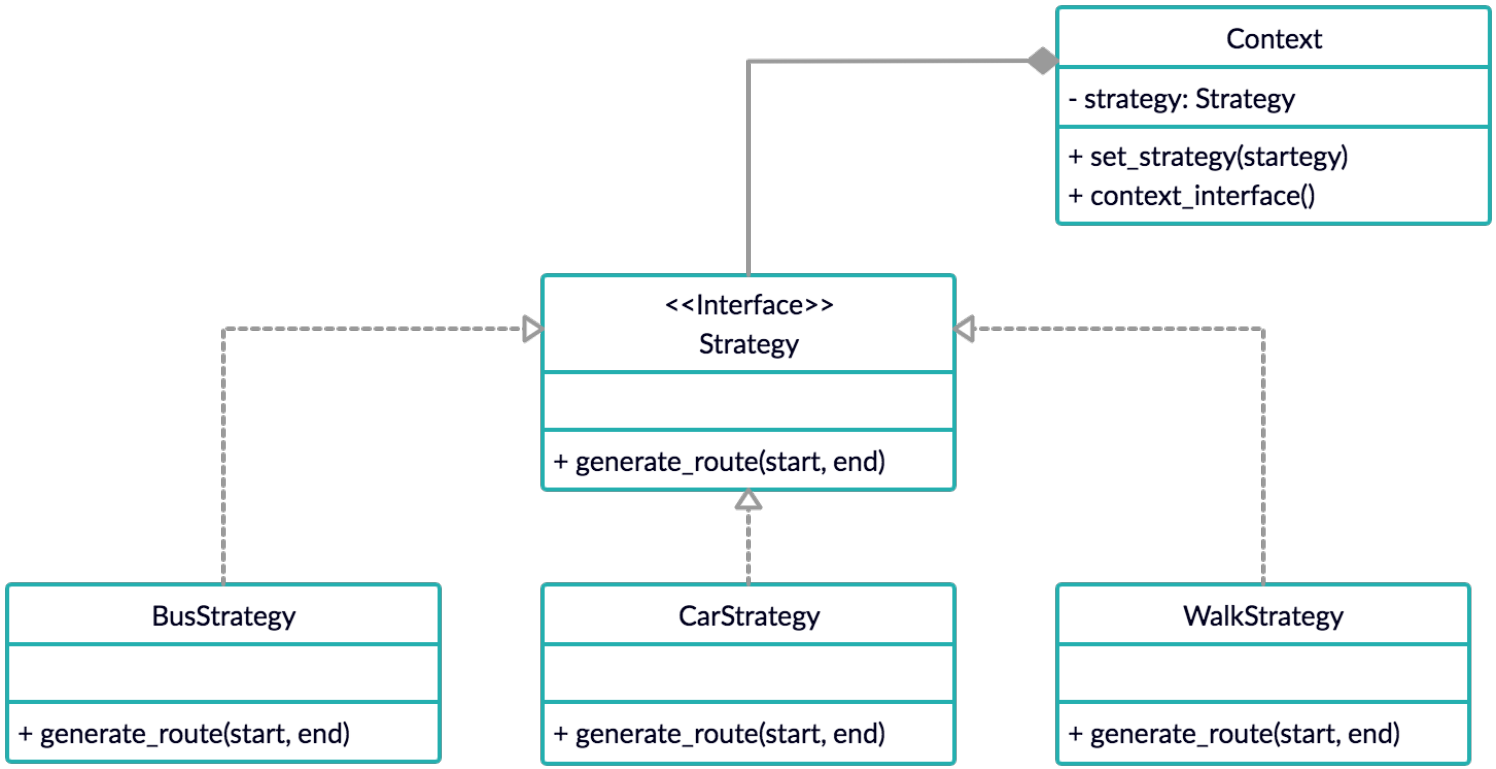
- Policy pattern
- Behavioural pattern
- Defining a family of algorithms
- Encapsulating each algorithm
- Enabling an algorithm to be selected at runtime
- Each algorithm is interchangeable

Strategy Pattern: Problem

- Navigation application

Strategy Pattern: Solution

- Three separate strategy classes - bus, car & walk

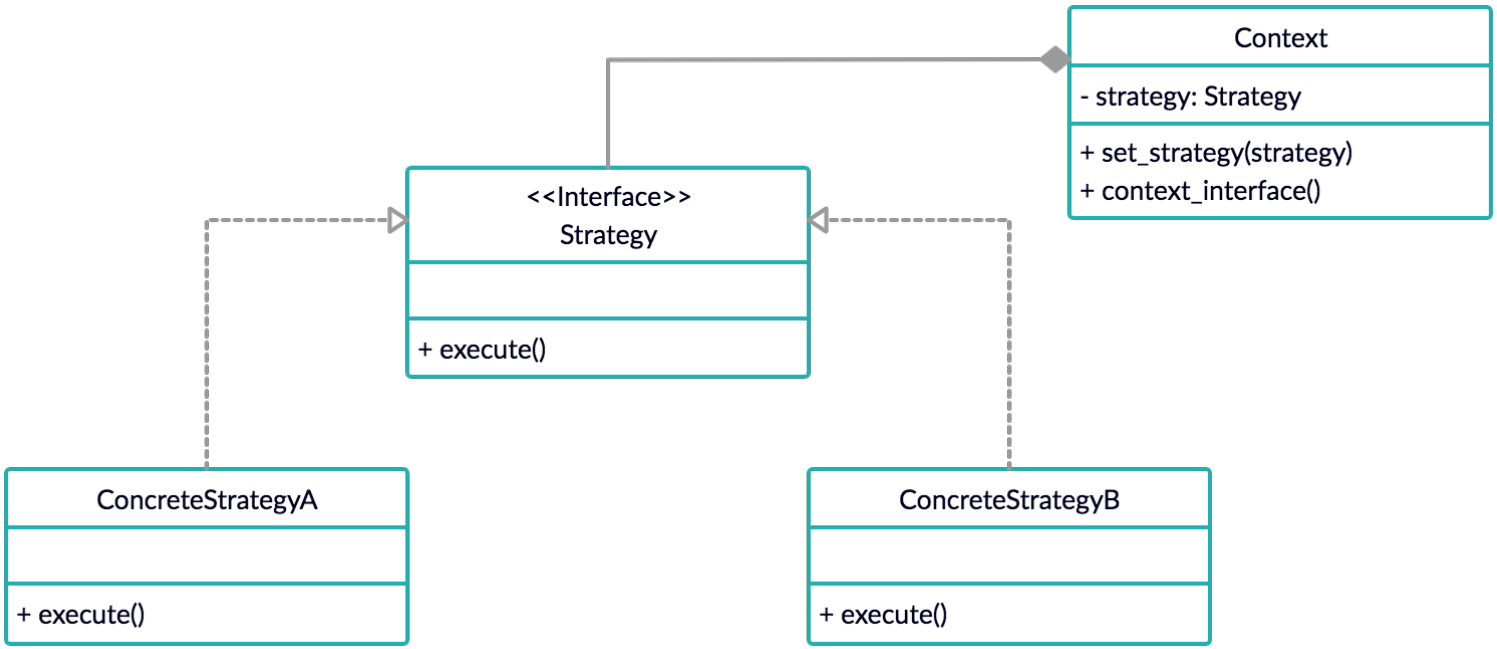


Strategy Pattern: Real World Analogy

- Transport to Dunedin airport
- Transportation strategies - car, shuttle, taxi, etc
- Constraints - cost & time

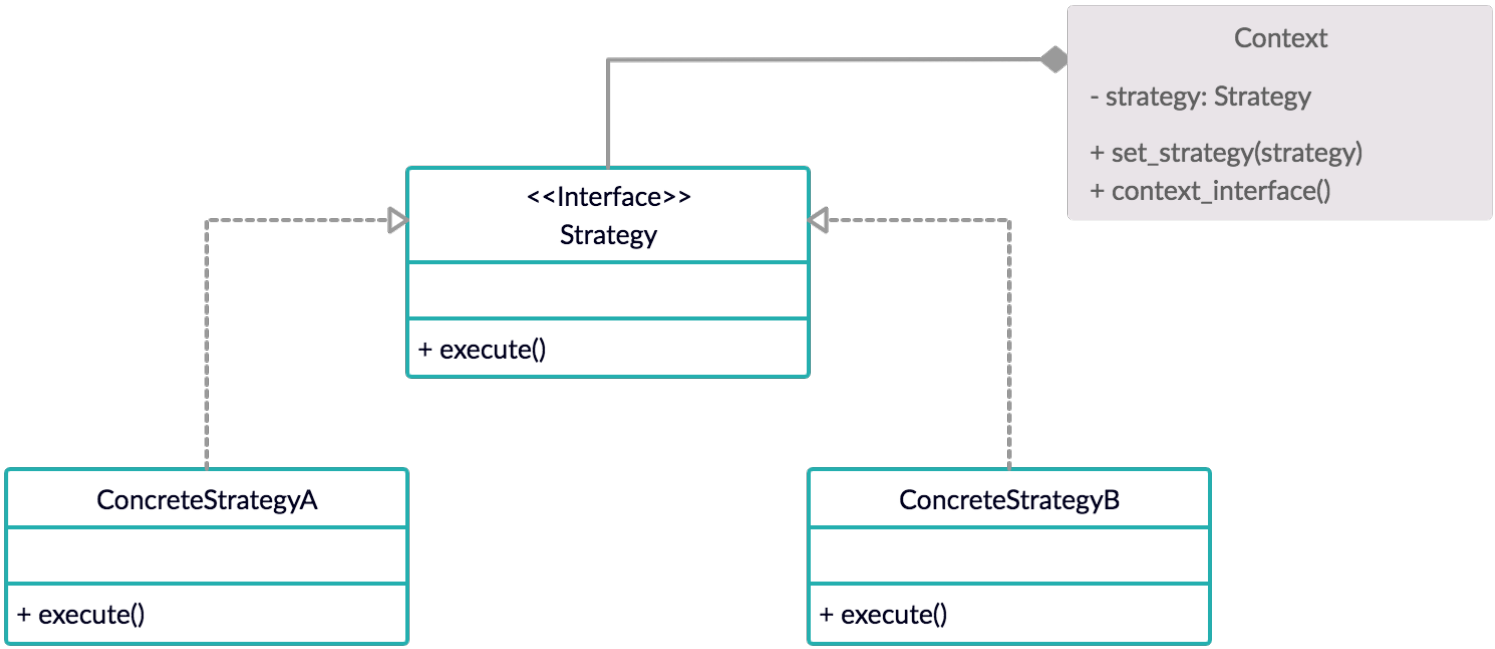
Strategy Pattern: UML

- Consider the following UML diagram:



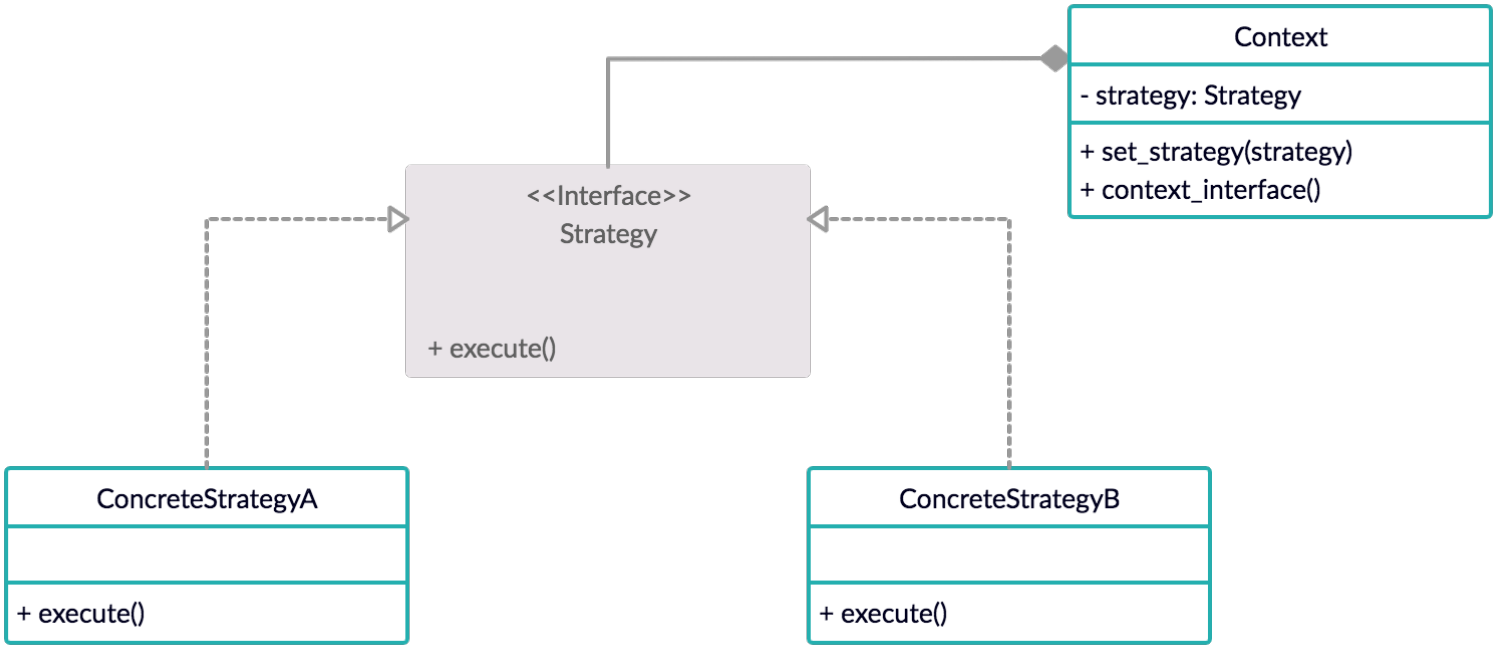
Strategy Pattern: UML

- Context class
- An algorithm isn't implemented directly
- Refers to the strategy interface for executing an algorithm
- Independent of how an algorithm is implemented



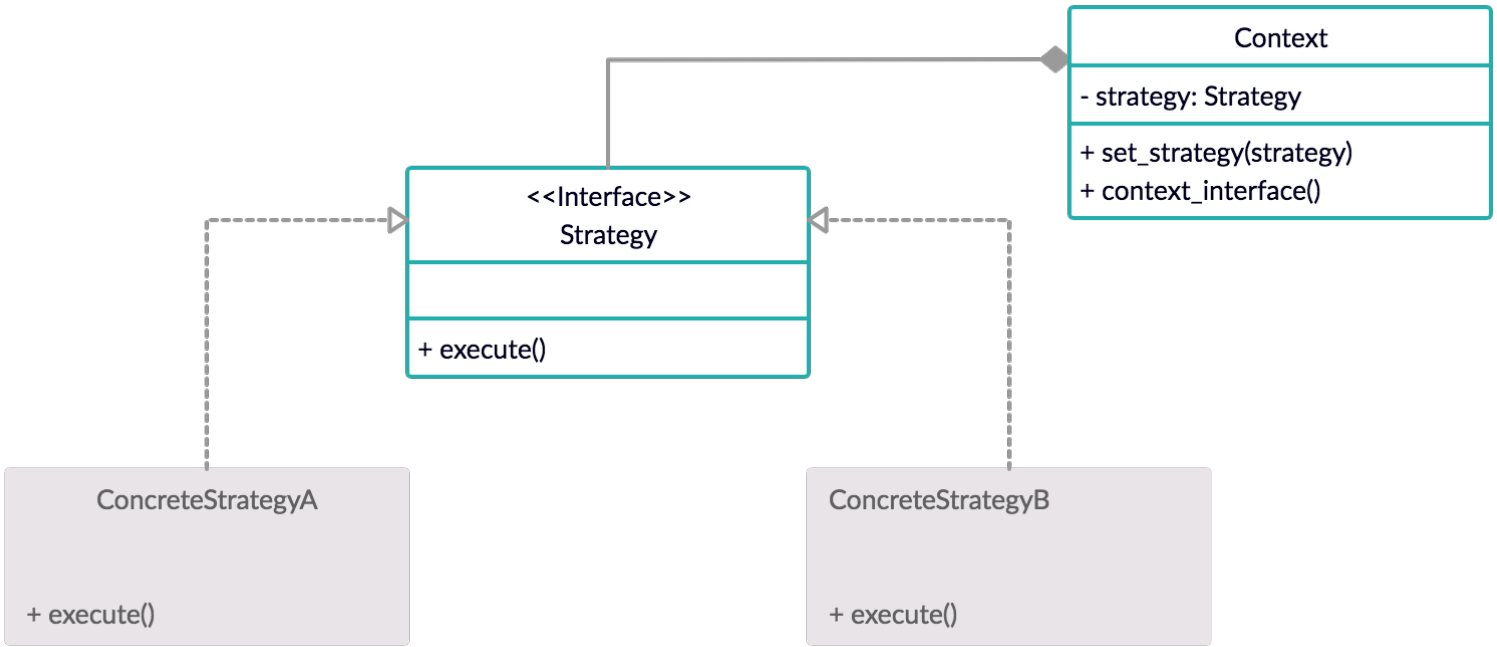
Strategy Pattern: UML

- Strategy interface class
- Declares a method which the context uses to execute an algorithm



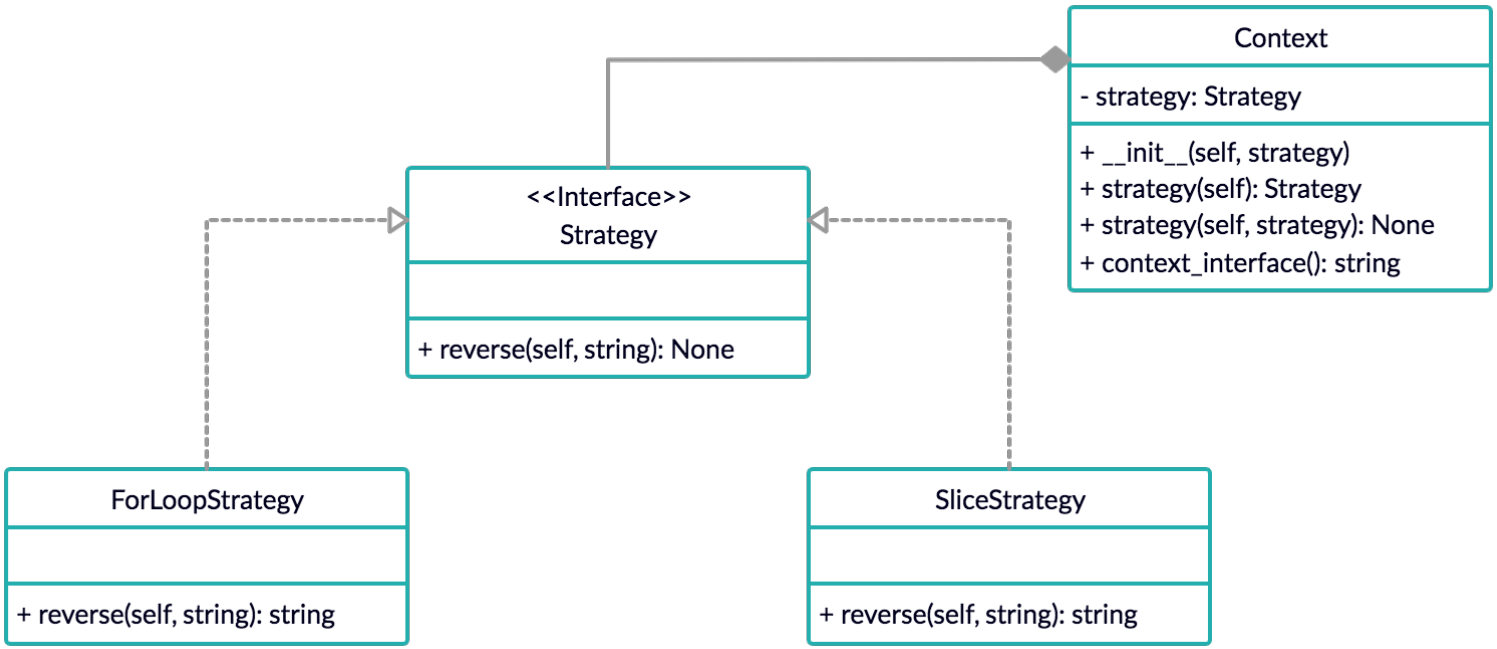
Strategy Pattern: UML

- Concrete strategy classes
- Implement the strategy interface
- Encapsulate the algorithm



Strategy Pattern: UML

- Consider the following UML diagram:



Strategy Pattern: Implementation

```
from abc import ABC, abstractmethod

class Context:
    def __init__(self, strategy):
        self.__strategy = strategy

    @property
    def strategy(self):
        return self.__strategy

    @strategy.setter
    def strategy(self, strategy):
        self.__strategy = strategy

    def context_interface(self):
        return self.__strategy.reverse('abcde')
```

Strategy Pattern: Implementation

```
class Strategy(ABC):
    @abstractmethod
    def reverse(self, string):
        pass

class ForLoopStrategy(Strategy):
    def reverse(self, string):
        reverse_string = ''
        for s in string:
            reverse_string = s + reverse_string
        return reverse_string

class SliceStrategy(Strategy):
    def reverse(self, string):
        return string[::-1]

def main():
    context = Context(ForLoopStrategy())
    print(context.context_interface())
    context.strategy = SliceStrategy()
    print(context.context_interface())

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

Strategy Pattern: Implementation

```
from abc import ABC, abstractmethod

class Context:
    def __init__(self, strategy, string):
        self.__strategy = strategy
        self.__string = string

    @property
    def strategy(self):
        return self.__strategy

    @strategy.setter
    def strategy(self, strategy):
        self.__strategy = strategy

    @property
    def string(self):
        return self.__string

    @string.setter
    def string(self, string):
        self.__string = string

    def context_interface(self):
        return self.__strategy.reverse(self.__string)
```

Strategy Pattern: Implementation

```
class Strategy(ABC):
    @abstractmethod
    def reverse(self, string):
        pass

class ForLoopStrategy(Strategy):
    def reverse(self, string):
        reverse_string = ''
        for s in string:
            reverse_string = s + reverse_string
        return reverse_string

class SliceStrategy(Strategy):
    def reverse(self, string):
        return string[::-1]

def main():
    context = Context(ForLoopStrategy(), 'abcde')
    print(context.context_interface())
    context.strategy = SliceStrategy()
    context.string = 'fghij'
    print(context.context_interface())

if __name__ == '__main__':
            main()  # edcba
                    # jihgf
```

Strategy Pattern: Open-Closed Principle

- Behaviours of a class shouldn't be inherited
- instead a class should be encapsulated using interfaces
- Strategy pattern uses composition instead of inheritance
- Behaviours are defined as separate interfaces & specific classes that implement these interfaces
- Allows better decoupling between the behavior & the class that uses the behaviour
- The behaviour can be changed without breaking the classes that use it

Strategy Pattern: Pros

- At runtime, algorithms are interchangeable
- An algorithm's implementation details are isolated
- New strategies can be introduced without having to change the context's code

Strategy Pattern: Cons

- The client must know the difference between strategies
- The number of objects in an application increases

Practical

- Series of tasks covering today's lecture
- Worth 1% of your final mark for the Object-Oriented Systems Development course
- Deadline: Tuesday, 17 March at 5pm

Reminder: Exam 01

- Series of tasks covering lectures 01-04
- Worth 6% of your final mark for the Object-Oriented Systems Development course
- Deadline: Thursday, 5 March at 5pm

Lecture 06: Observer Pattern Topics

- Design pattern 02: observer pattern
  - Definition
  - Problem/solution
  - Real world analogy
  - UML & implementation
  - Strong vs. weak reference
  - Pros & cons