



Lecture 17: Strategy Pattern

IN628: Programming 4

Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

LECTURE 04: EXCEPTIONS & AUTOMATION TESTING RECAP

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

WHAT ARE DESIGN PATTERNS?

- ▶ Design patterns are typical solutions to common problems in software design
- ▶ Each pattern is like a blueprint that you can customize to solve a particular design problem in your code
- ▶ Classified into three categories:
 - ▶ Structural
 - ▶ Behavioural
 - ▶ Creational

STRUCTURAL

- ▶ Identifying a simple way to realise relationships among entities
- ▶ Patterns to be familiarise yourself with:
 - ▶ Adapter
 - ▶ Flyweight
 - ▶ Proxy
 - ▶ Façade - talked about this in lecture 03

BEHAVIOURAL

- ▶ Identify common communication patterns among objects & realise these patterns
- ▶ Patterns to be familiarise yourself with:
 - ▶ Strategy
 - ▶ Observer
 - ▶ State
 - ▶ Template

CREATIONAL

- ▶ Deals with object creation mechanisms
- ▶ Create objects in a manner suitable to the situation
- ▶ Two key ideas:
 - ▶ Encapsulating knowledge about which concrete classes the system uses
 - ▶ Hiding how instances of these concrete classes are created & combined
- ▶ Patterns to be familiarise yourself with:
 - ▶ Factory
 - ▶ Singleton
 - ▶ Builder

DESIGN PATTERNS TABLE

C	Abstract Factory	S	Facade	S	Proxy
S	Adapter	C	Factory Method	B	Observer
S	Bridge	S	Flyweight	C	Singleton
C	Builder	B	Interpreter	B	State
B	Chain of Responsibility	B	Iterator	B	Strategy
B	Command	B	Mediator	B	Template Method
S	Composite	B	Memento	B	Visitor
S	Decorator	C	Prototype		

► Reference: Jason S. McDonald

STRATEGY PATTERN: GoF

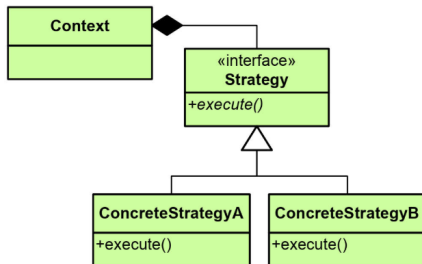
► GoF definition & UML

Strategy

Type: Behavioral

What it is:

Define a family of algorithms, encapsulate each one, and make them interchangeable. Lets the algorithm vary independently from clients that use it.



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 1

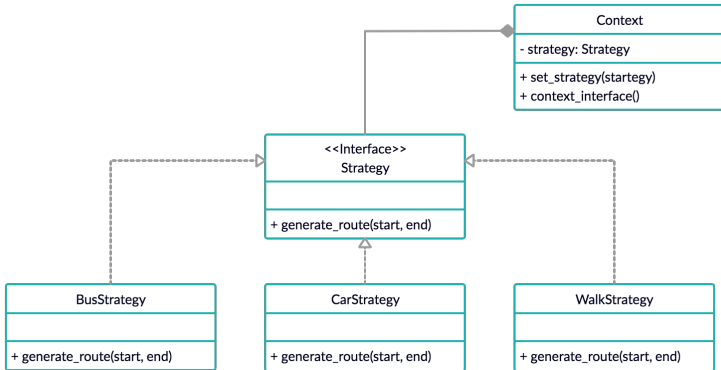
- ▶ Performing validation on incoming data
 - ▶ Many ways to validate data:
 - ▶ Data-type checking
 - ▶ Simple range & constraint checking
 - ▶ Code & cross reference checking
- ▶ Select a validation algorithm depending on different factors
 - ▶ Factors are not known until runtime & may require a different validation algorithm to be performed
- ▶ The validation algorithms may be used by other validation objects in different areas of the system without code duplication

STRATEGY PATTERN: PROBLEM 2

- ▶ Navigation application

STRATEGY PATTERN: SOLUTION 2

- Three separate strategy classes - bus, car & walk

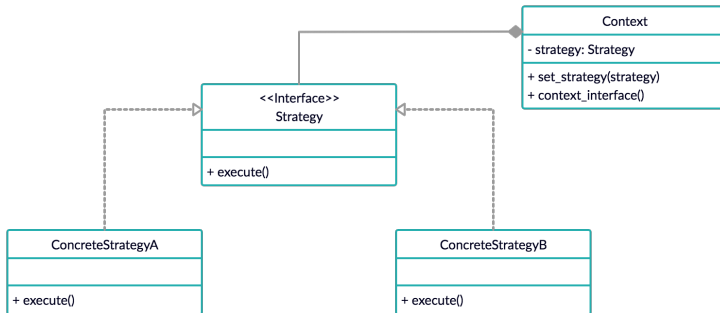


STRATEGY PATTERN: REAL WORLD ANALOGY

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

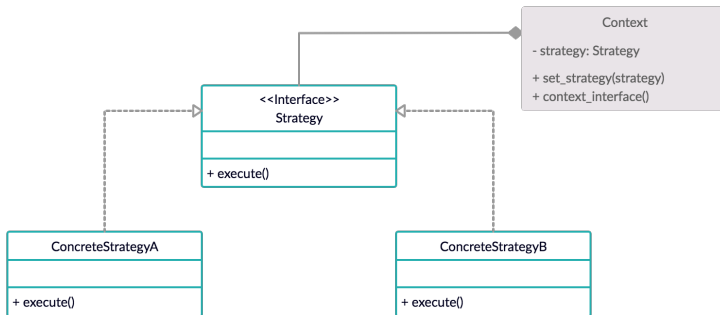
STRATEGY PATTERN: UML

- Consider the following UML diagram:



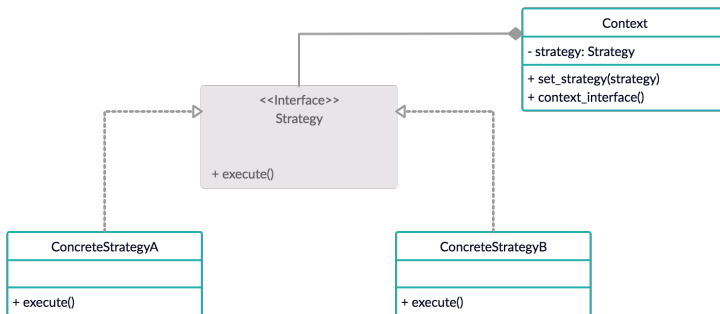
STRATEGY PATTERN: UML

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



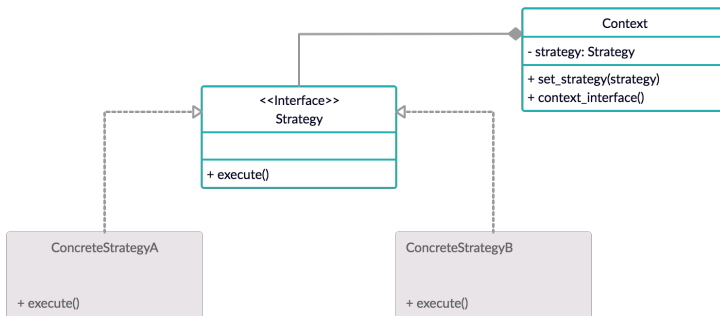
STRATEGY PATTERN: UML

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



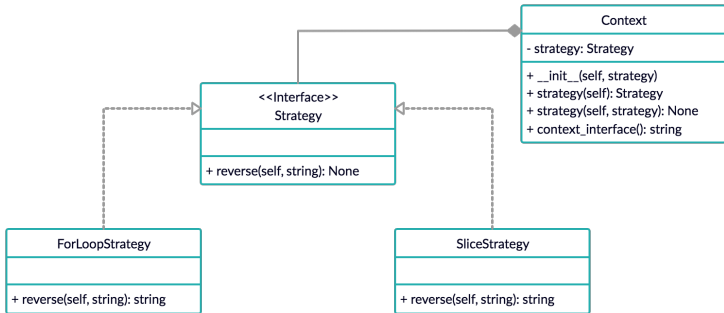
STRATEGY PATTERN: UML

- 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

STRATEGY PATTERN: VIDEOS

- ▶ <https://www.youtube.com/watch?v=-NCgRD9-C6o>
- ▶ Note: Code examples are Java. Concepts still apply in Python

PRACTICAL

- ▶ Series of tasks covering today's lecture
- ▶ Worth 1% of your final mark for the Programming 4 course
- ▶ Deadline: Friday, 12 June at 5pm