



Lecture 09: Adapter Pattern

IN710: Object-Oriented Systems Development

Semester One, 2020

Kaiako: Grayson Orr

Te Kura Matatini ki Otago, Ōtepoti, Aotearoa

Tuesday, 17 March

LECTURE 08: SINGLETON PATTERN RECAP

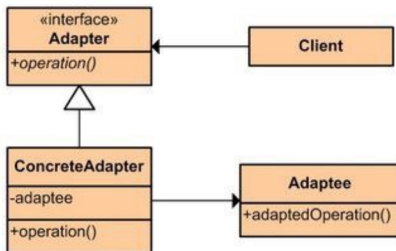
- ▶ Design pattern 04: singleton pattern
 - ▶ Definition
 - ▶ Problem/solution
 - ▶ Real world analogy
 - ▶ UML & implementation
 - ▶ Pros & cons

LECTURE 09: ADAPTER PATTERN TOPICS

- ▶ Design pattern 05: adapter pattern
 - ▶ Definition
 - ▶ Problem/solution
 - ▶ Real world analogy
 - ▶ UML & implementation
 - ▶ Pros & cons

ADAPTER PATTERN: GoF

► GoF definition & UML



Adapter

Type: Structural

What it is:

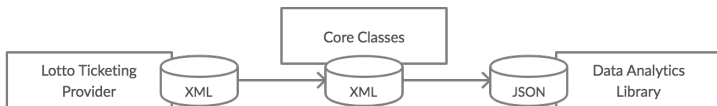
Convert the interface of a class into another interface clients expect. Lets classes work together that couldn't otherwise because of incompatible interfaces.

ADAPTER PATTERN: DEFINITION

- ▶ Structural pattern
- ▶ Also known as wrapper
 - ▶ An alternative name shared with the decorator pattern
- ▶ Allows the interface of an existing class to be used as another interface
- ▶ Often used to make existing classes work with others without modifying their code

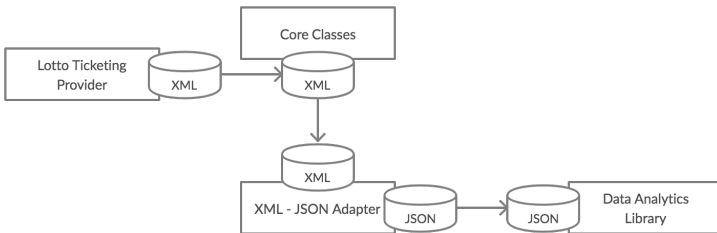
ADAPTER PATTERN: PROBLEM

- Lotto ticketing application



ADAPTER PATTERN: SOLUTION

- Create an XML to JSON adapter

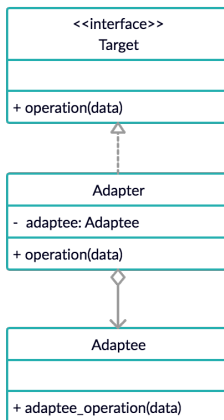


ADAPTER PATTERN: REAL WORLD ANALOGY

- ▶ Power plugs & sockets
- ▶ Standards are different in different countries
- ▶ Example: a US power plug may not fit in a European socket
- ▶ The problem can be solved by using a travel adapter

OBJECT ADAPTER PATTERN: UML

- Consider the following UML diagram:

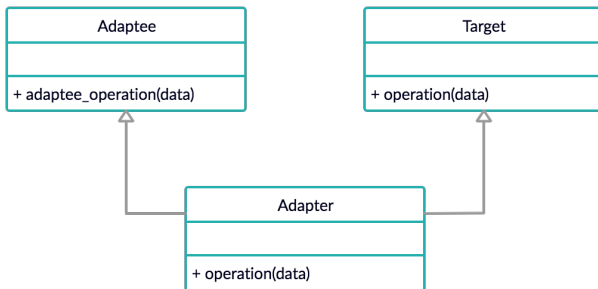


OBJECT ADAPTER PATTERN

- ▶ Contains an instance of the class it wraps
- ▶ Makes calls to the instance of the wrapped object

CLASS ADAPTER PATTERN: UML

- Consider the following UML diagram:



CLASS ADAPTER PATTERN

- ▶ Multiple polymorphic interfaces
- ▶ Implement interfaces from both objects at the same time
- ▶ Can only be implemented in programming languages that support multiple inheritance

ADAPTER PATTERN: IMPLEMENTATION

```
from abc import ABC, abstractmethod
```

```
class EuropeanSocket(ABC):
```

```
    @abstractmethod
```

```
    def voltage(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def live(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def neutral(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def earth(self):
```

```
        pass
```

```
class USSocket(ABC):
```

```
    @abstractmethod
```

```
    def voltage(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def live(self):
```

```
        pass
```

```
    @abstractmethod
```

```
    def neutral(self):
```

```
        pass
```

ADAPTER PATTERN: IMPLEMENTATION

```
class AdapteeSocket(EuropeanSocket):  
    def voltage(self):  
        return 220  
  
    def live(self):  
        return 1  
  
    def neutral(self):  
        return -1  
  
    def earth(self):  
        return 0  
  
class Adapter(USSocket):  
    def __init__(self, socket):  
        self.socket = socket  
  
    def voltage(self):  
        return 110  
  
    def live(self):  
        return self.socket.live()  
  
    def neutral(self):  
        return self.socket.neutral()
```

ADAPTER PATTERN: IMPLEMENTATION

```
class FlatIron:
    def __init__(self, adapter):
        self.adapter = adapter

    def heating(self):
        if self.adapter.voltage() > 110:
            print('Your_flat_iron_is_overheating')
        elif self.adapter.live() == 1 and self.adapter.neutral() == -1:
            print('Your_hair_is_ready_to_be_straighten')
        else:
            print('The_power_is_off')

def main():
    adaptee_socket = AdapteeSocket()
    adapter = Adapter(adaptee_socket)
    flat_iron = FlatIron(adapter)
    flat_iron.heating()

if __name__ == '__main__':
    main() # Your hair is ready to be straighten
```

ADAPTER PATTERN: PROS

- ▶ Separate the interface from the primary business logic of the program
- ▶ New types of adapters can be introduced without having to change the client's code

ADAPTER PATTERN: CONS

- Overall, the complexity of the code increases - new interfaces & 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

REMINDER: EXAM 02

- ▶ Series of tasks covering lectures 05-08
- ▶ Worth 6% of your final mark for the Object-Oriented Systems Development course
- ▶ Deadline: Thursday, 19 March at 5pm

LECTURE 10: BUILDER PATTERN TOPICS

- ▶ Design pattern 06: builder pattern
 - ▶ Definition
 - ▶ Problem/solution
 - ▶ UML & implementation
 - ▶ Applicability
 - ▶ Pros & cons