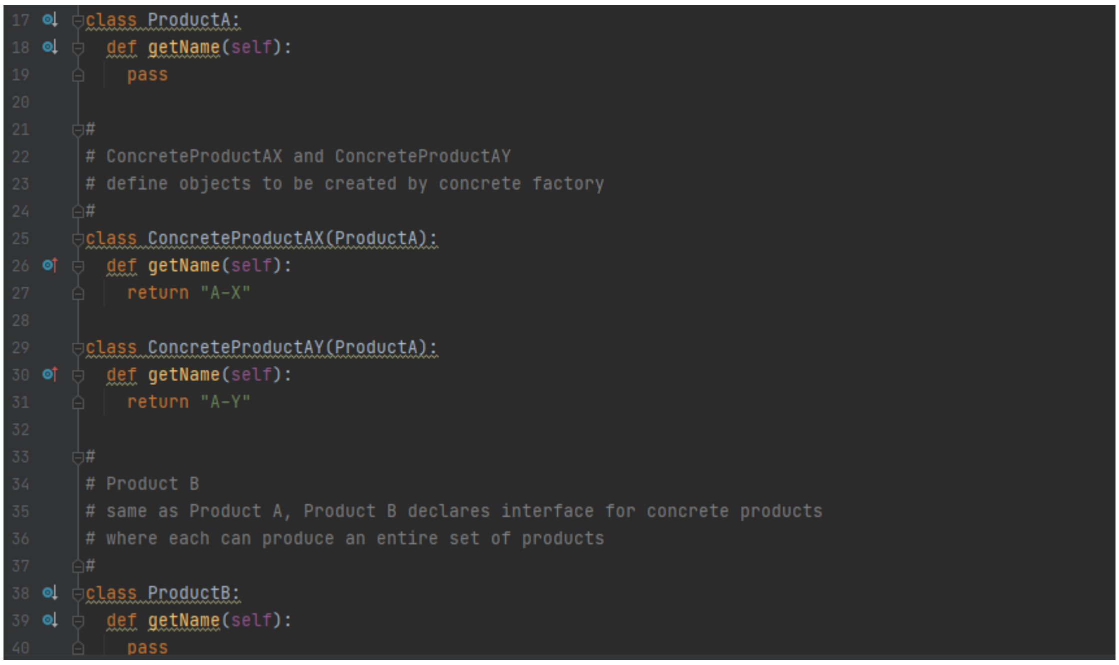
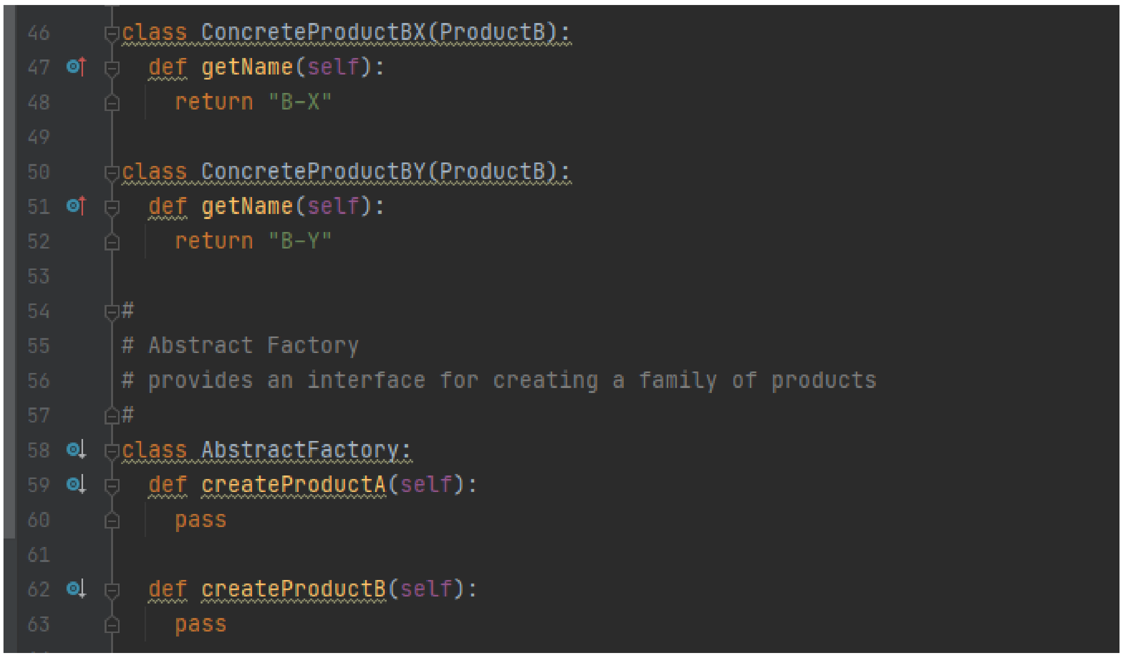
LAB-13

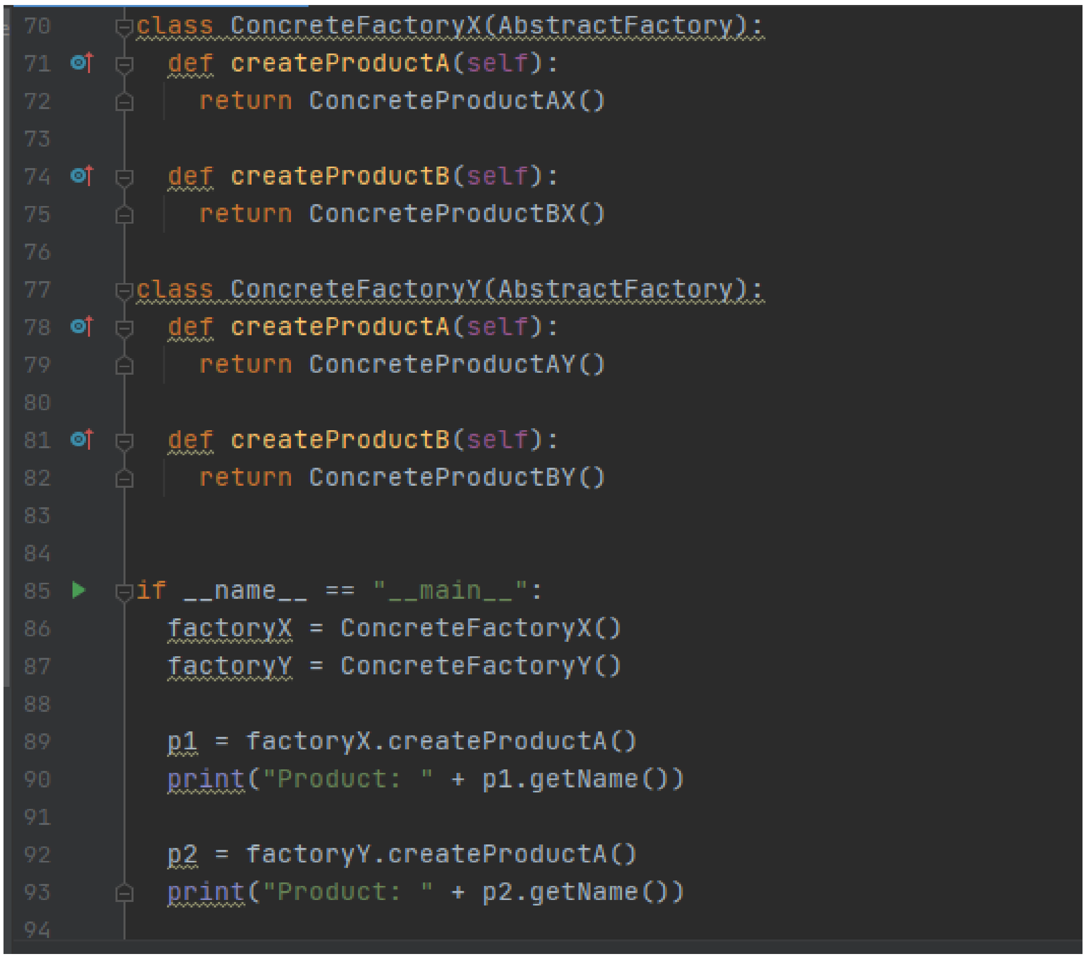
Abstract factory.py



```
class ProductA:
    def getName(self):
        pass

#
# ConcreteProductAX and ConcreteProductAY
# define objects to be created by concrete factory
#
class ConcreteProductAX(ProductA):
    def getName(self):
        return "A-X"

class ConcreteProductAY(ProductA):
    def getName(self):
        return "A-Y"

#
# Product B
# same as Product A, Product B declares interface for concrete products
# where each can produce an entire set of products
#
class ProductB:
    def getName(self):
        pass
```

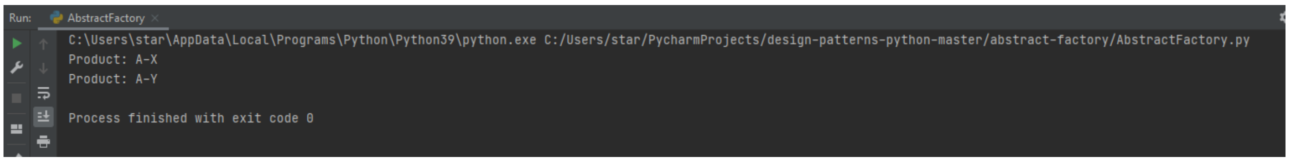


```
class ConcreteProductBX(ProductB):
    def getName(self):
        return "B-X"

class ConcreteProductBY(ProductB):
    def getName(self):
        return "B-Y"

#
# Abstract Factory
# provides an interface for creating a family of products
#
class AbstractFactory:
    def createProductA(self):
        pass

    def createProductB(self):
        pass
```



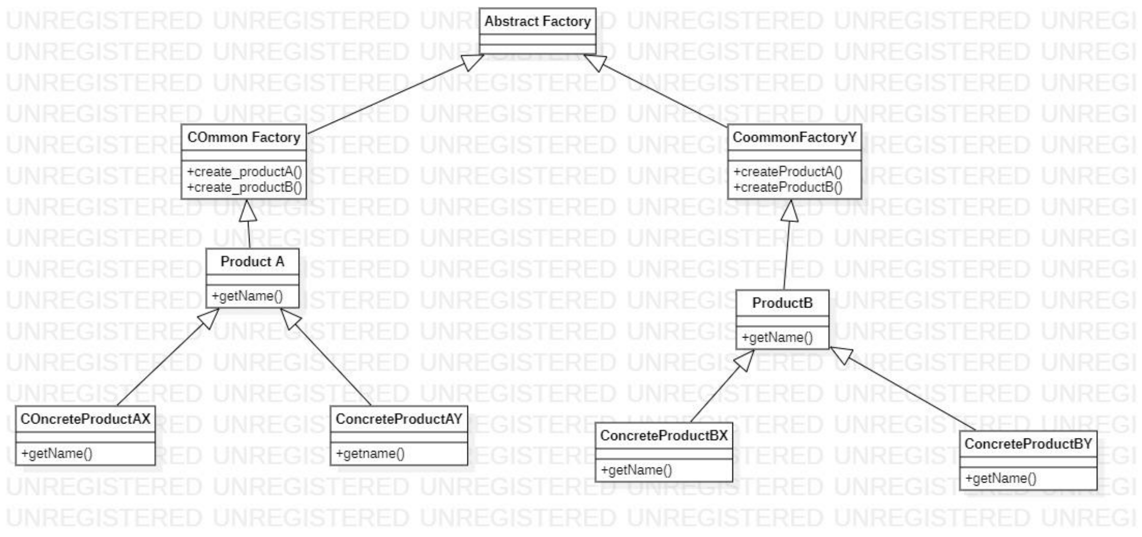
```python
class ConcreteFactoryX(AbstractFactory):
    def createProductA(self):
        return ConcreteProductAX()

    def createProductB(self):
        return ConcreteProductBX()

class ConcreteFactoryY(AbstractFactory):
    def createProductA(self):
        return ConcreteProductAY()

    def createProductB(self):
        return ConcreteProductBY()


if __name__ == "__main__":
    factoryX = ConcreteFactoryX()
    factoryY = ConcreteFactoryY()

    p1 = factoryX.createProductA()
    print("Product: " + p1.getName())

    p2 = factoryY.createProductA()
    print("Product: " + p2.getName())
```

<u>Output:</u>

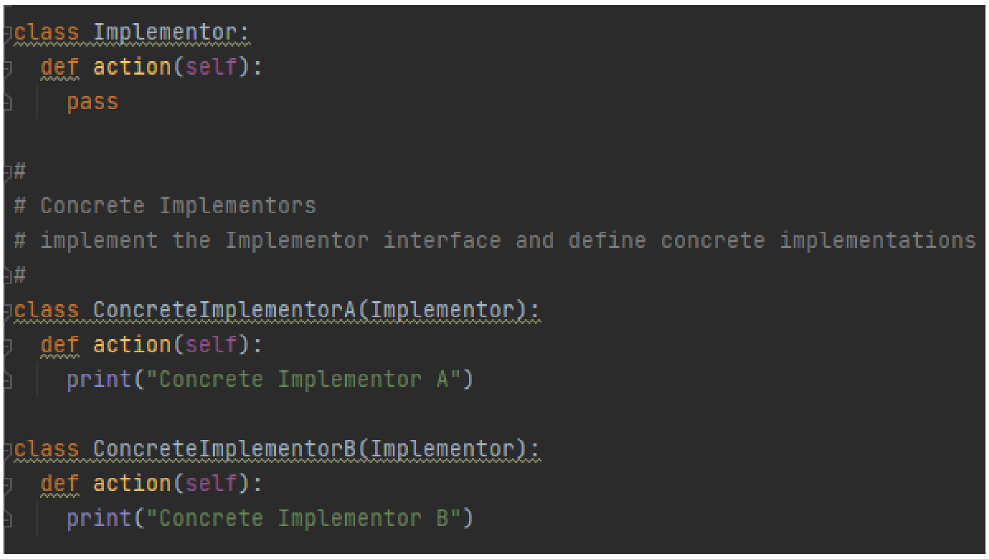


```
C:\Users\star\AppData\Local\Programs\Python\Python39\python.exe C:/Users/star/PycharmProjects/design-patterns-python-master/abstract-factory/AbstractFactory.py
Product: A-X
Product: A-Y

Process finished with exit code 0
```

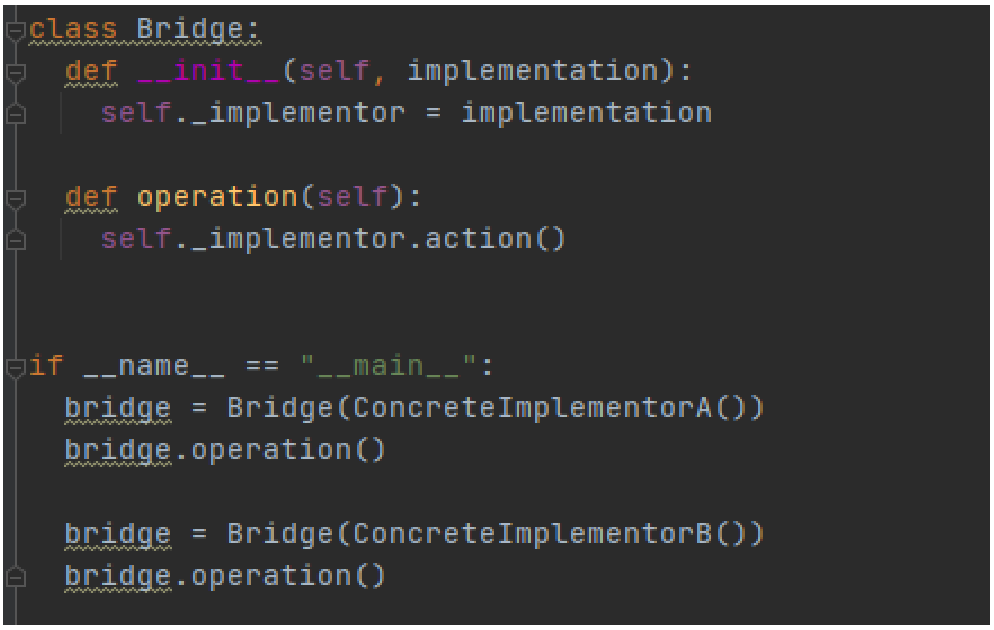
<u>UML Diagram:</u>



Bridge.py:

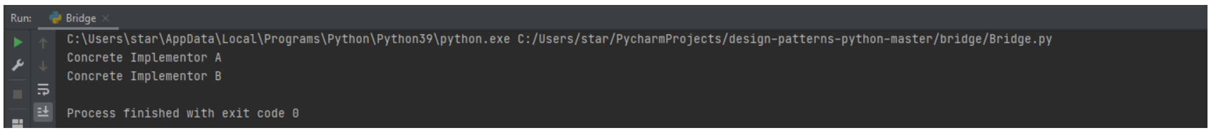


```
class Implementor:
    def action(self):
        pass

#
# Concrete Implementors
# implement the Implementor interface and define concrete implementations
#
class ConcreteImplementorA(Implementor):
    def action(self):
        print("Concrete Implementor A")

class ConcreteImplementorB(Implementor):
    def action(self):
        print("Concrete Implementor B")
```



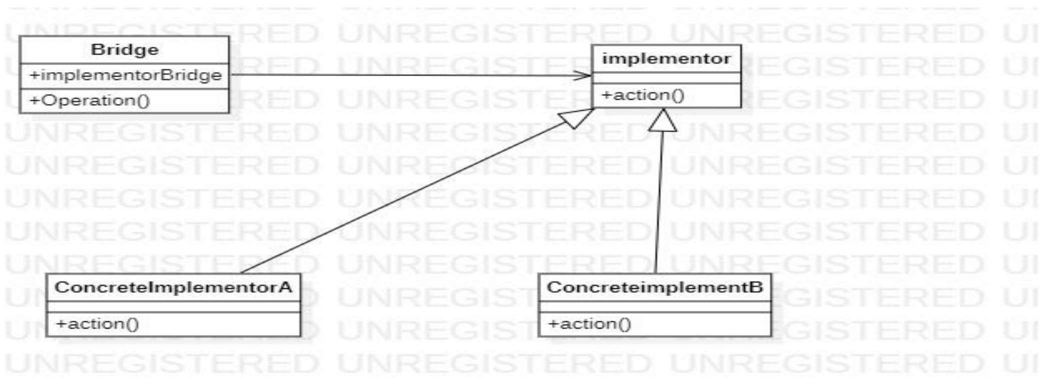
```
class Bridge:
    def __init__(self, implementation):
        self._implementor = implementation

    def operation(self):
        self._implementor.action()


if __name__ == "__main__":
    bridge = Bridge(ConcreteImplementorA())
    bridge.operation()

    bridge = Bridge(ConcreteImplementorB())
    bridge.operation()
```

Output:

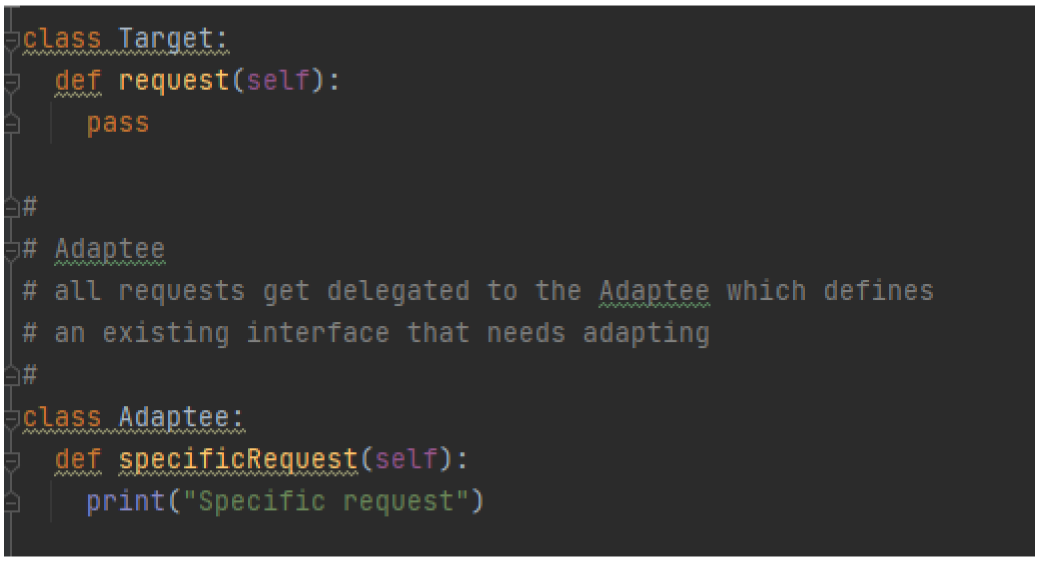


```
C:\Users\star\AppData\Local\Programs\Python\Python39\python.exe C:/Users/star/PycharmProjects/design-patterns-python-master/bridge/Bridge.py
Concrete Implementor A
Concrete Implementor B

Process finished with exit code 0
```

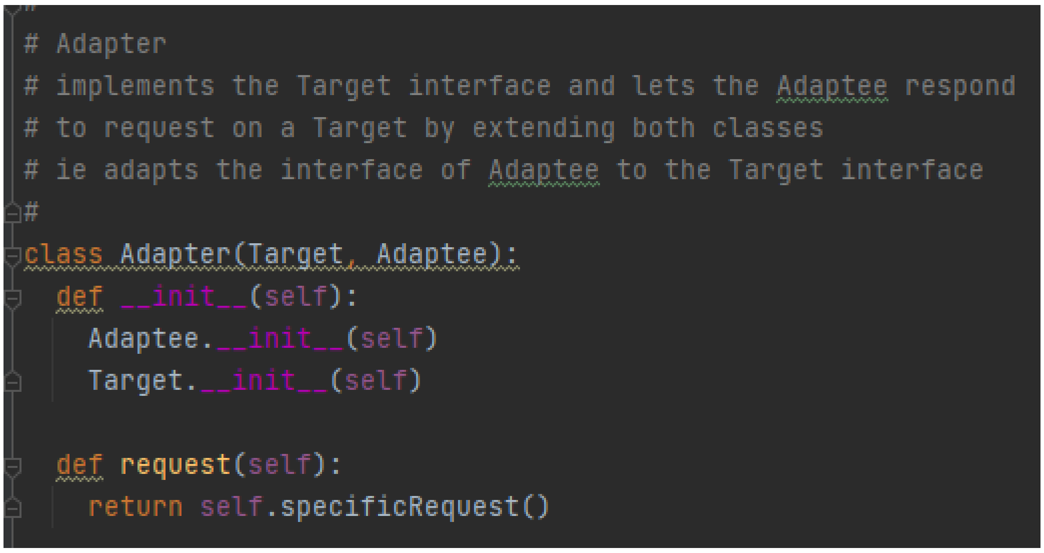
UML Diagram:



Adapter.py:

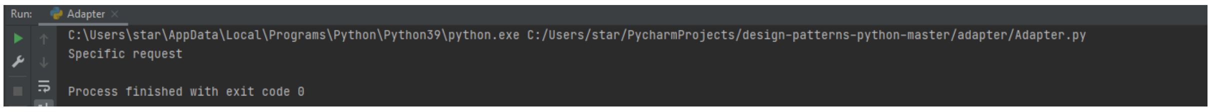


```
class Target:
    def request(self):
        pass

#
# Adaptee
# all requests get delegated to the Adaptee which defines
# an existing interface that needs adapting
#
class Adaptee:
    def specificRequest(self):
        print("Specific request")
```



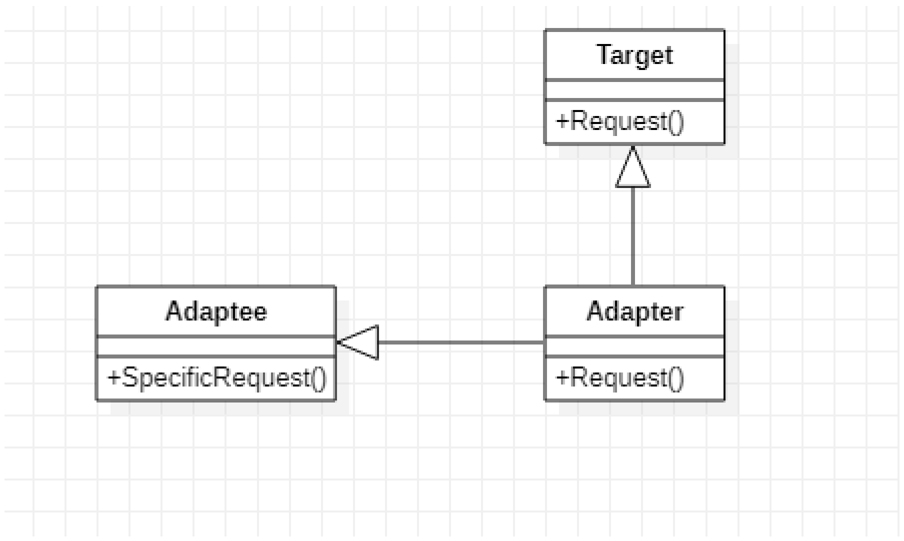
```
# Adapter
# implements the Target interface and lets the Adaptee respond
# to request on a Target by extending both classes
# ie adapts the interface of Adaptee to the Target interface
#
class Adapter(Target, Adaptee):
    def __init__(self):
        Adaptee.__init__(self)
        Target.__init__(self)

    def request(self):
        return self.specificRequest()
```

Output:



```
C:\Users\star\AppData\Local\Programs\Python\Python39\python.exe C:/Users/star/PycharmProjects/design-patterns-python-master/adapter/Adapter.py
Specific request

Process finished with exit code 0
```

UML Diagram:


Target
+Request()
Adaptee
+SpecificRequest()
Adapter
+Request()