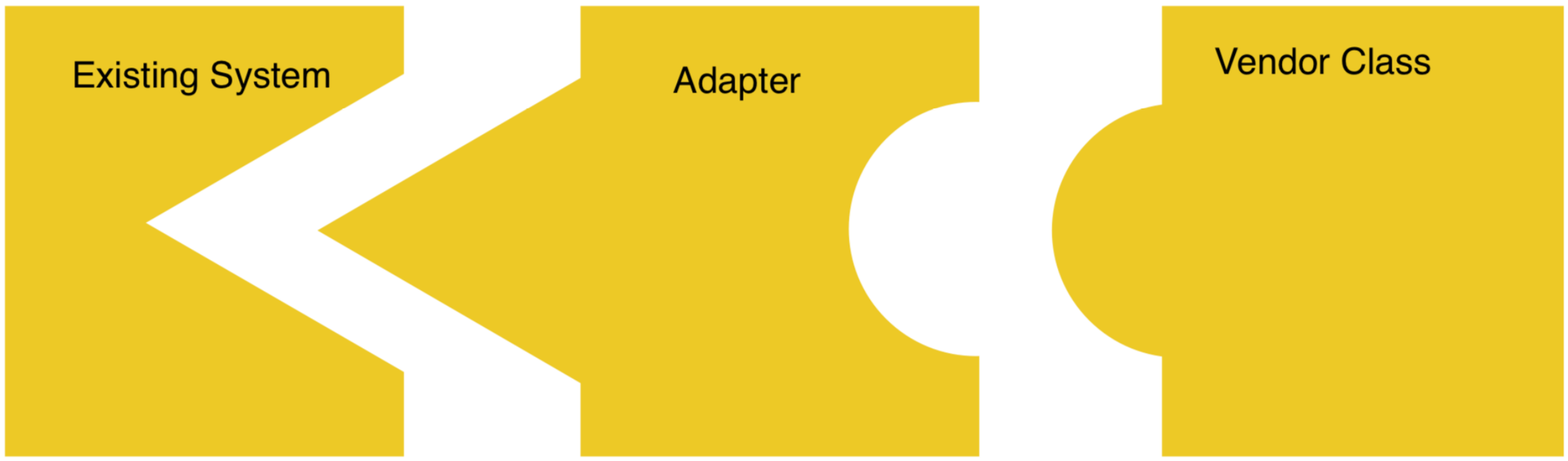
Adapter Pattern

Introduction

- You will learn how to wrap some objects with a different purpose to make their interfaces look like something they're not. So, we can adapt a design expecting one interface to a class that implements a different interface. In addition, we will look at another pattern that wraps objects to simplify their interface

Object oriented adapters

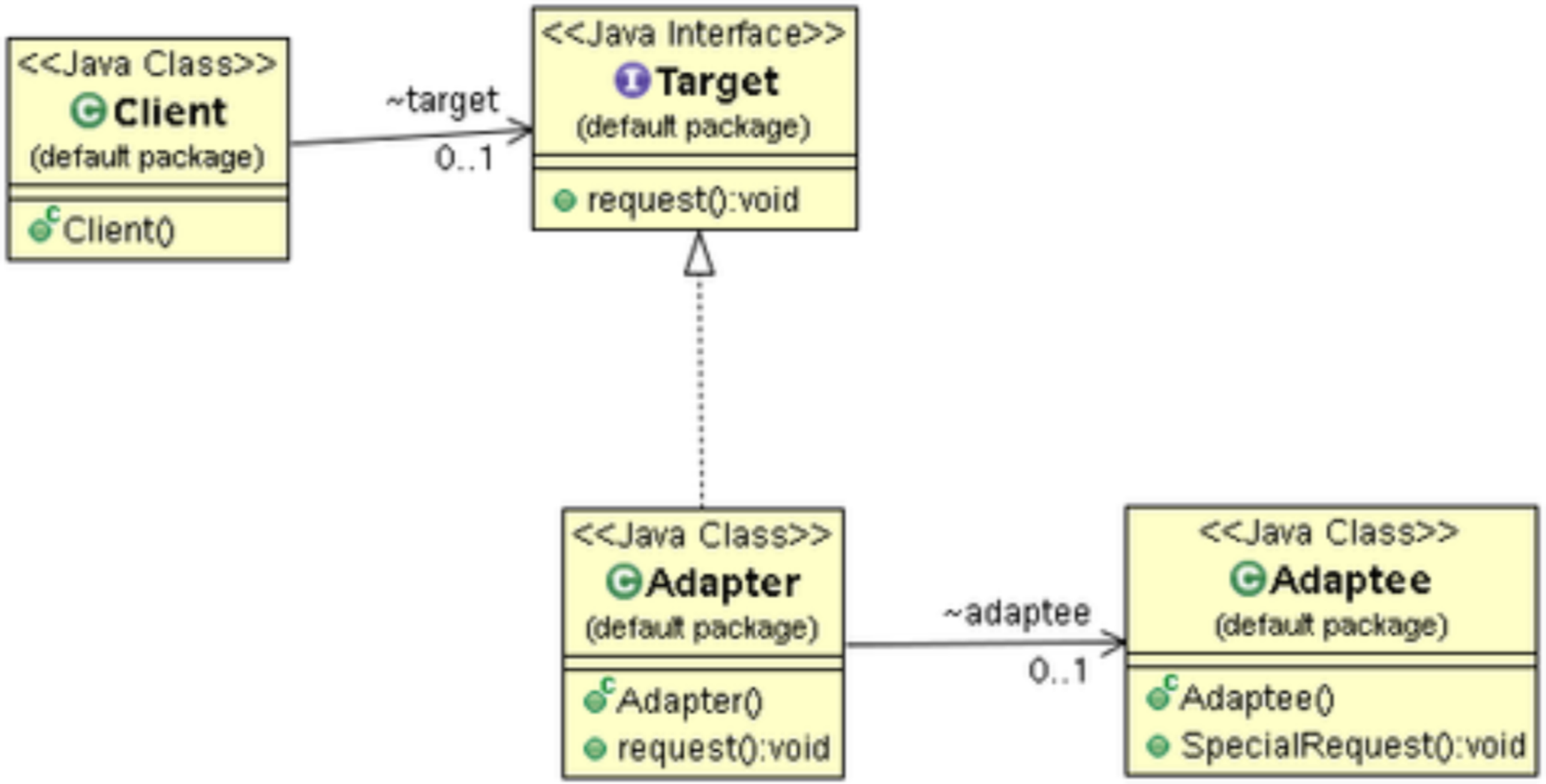
- Let say your existing software needs to use a vector class library, but the new interfaces are different from the last vector's library.
- You don't want to change your existing code and you can't change the vendor's code. Then you can write a class that adapts new vendor interface into one you are expecting.
- Here's how the client uses the adapter
  - The client makes a request to the adapter by calling a method on it using the target interface
  - The adapter translates that request in to one or more calls on the adaptee using the adaptee interface.
  - The client receives the results of the call and never knows there is an adapter doing the translation.



The Adapter Pattern

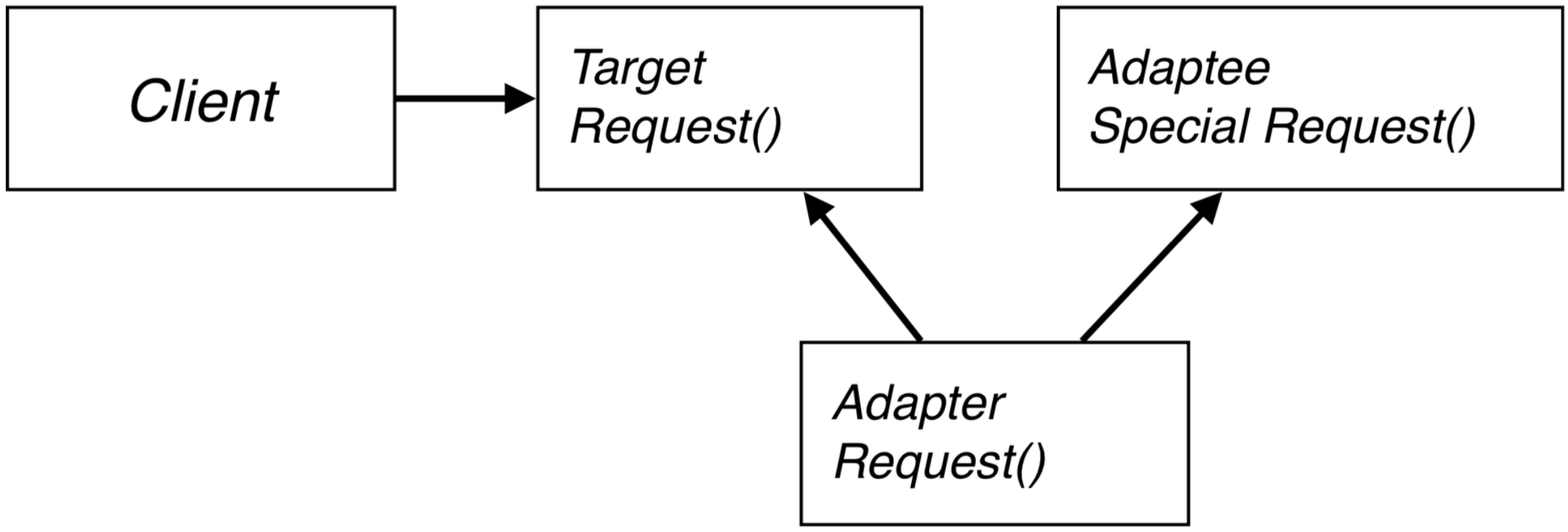
- It converts the interface of a class into another interface the clients use. An adapter allows classes to work together that normally could not because of incompatible interfaces by providing its interface to clients while using the original interface. The adapter translates calls to its interface into calls to the original interface. The adapter is also responsible to transforming data into appropriate forms.
- Again this pattern allows us to use an incompatible interface by creating an adapter that converts it. So it helps to decouple your client code from the actual implemented interface that could be changed over time.
- The Adapter pattern uses the object composition to wrap the adaptee. This way we can use the Adapter to wrap any child class of the Adaptee.

The Adapter Pattern



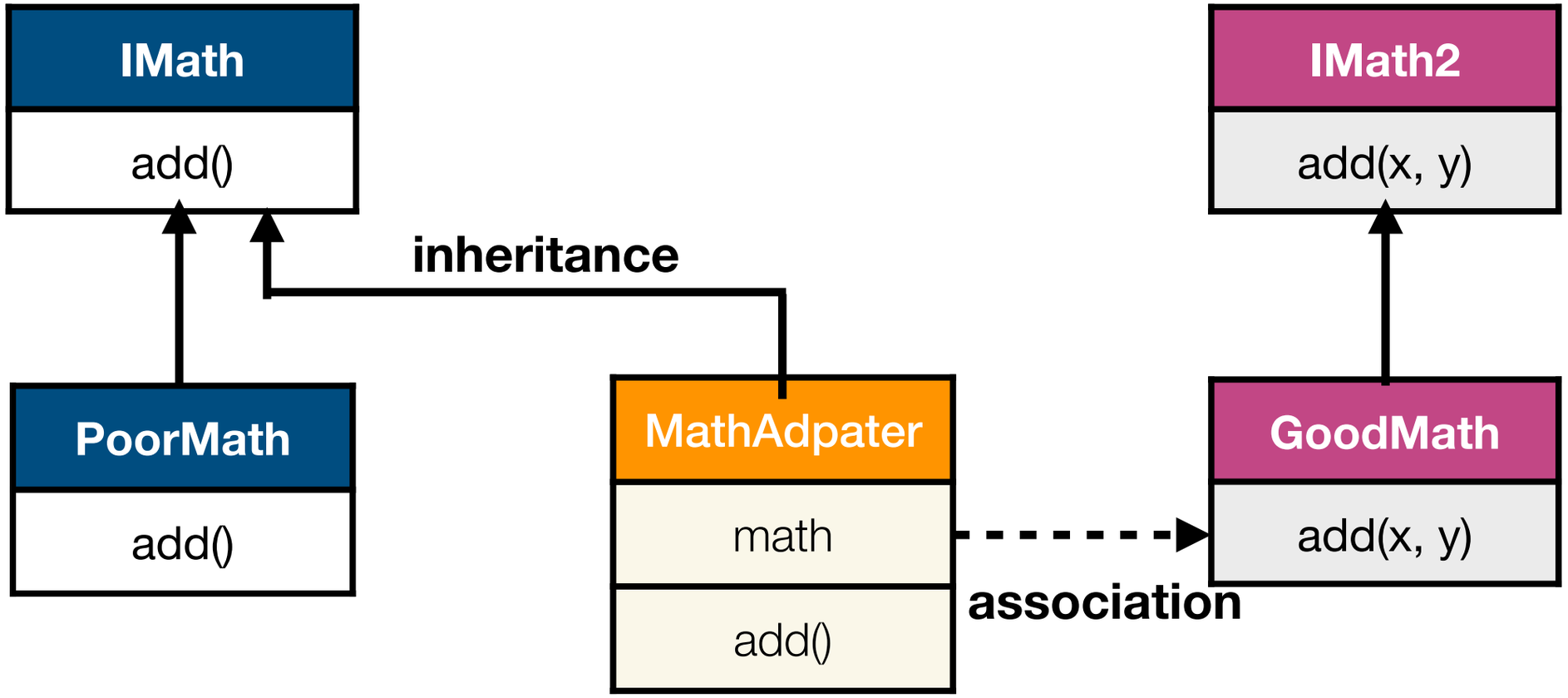
Object Adapter vs Class Adapter

- So the way we created the adapter is the object adapter, but you can also use the multiple inheritance to create the adapter that is called Class Adapter.
- The only difference is that the class adapter uses the inheritance to access the Target and the Adaptee. And the object adapter uses composition to pass requests to an Adaptee.



Example of Math Library

- In this example, there is an old library called PoorMath. This library provides one method add(). And many client applications have been using this method. But, now we have an improved Math library called GoodMath, the add method has been changed its interface to add(x, y). Therefore, in order for the client applications to use this new improved library without changing their codes, we will need ton provide an adapter.



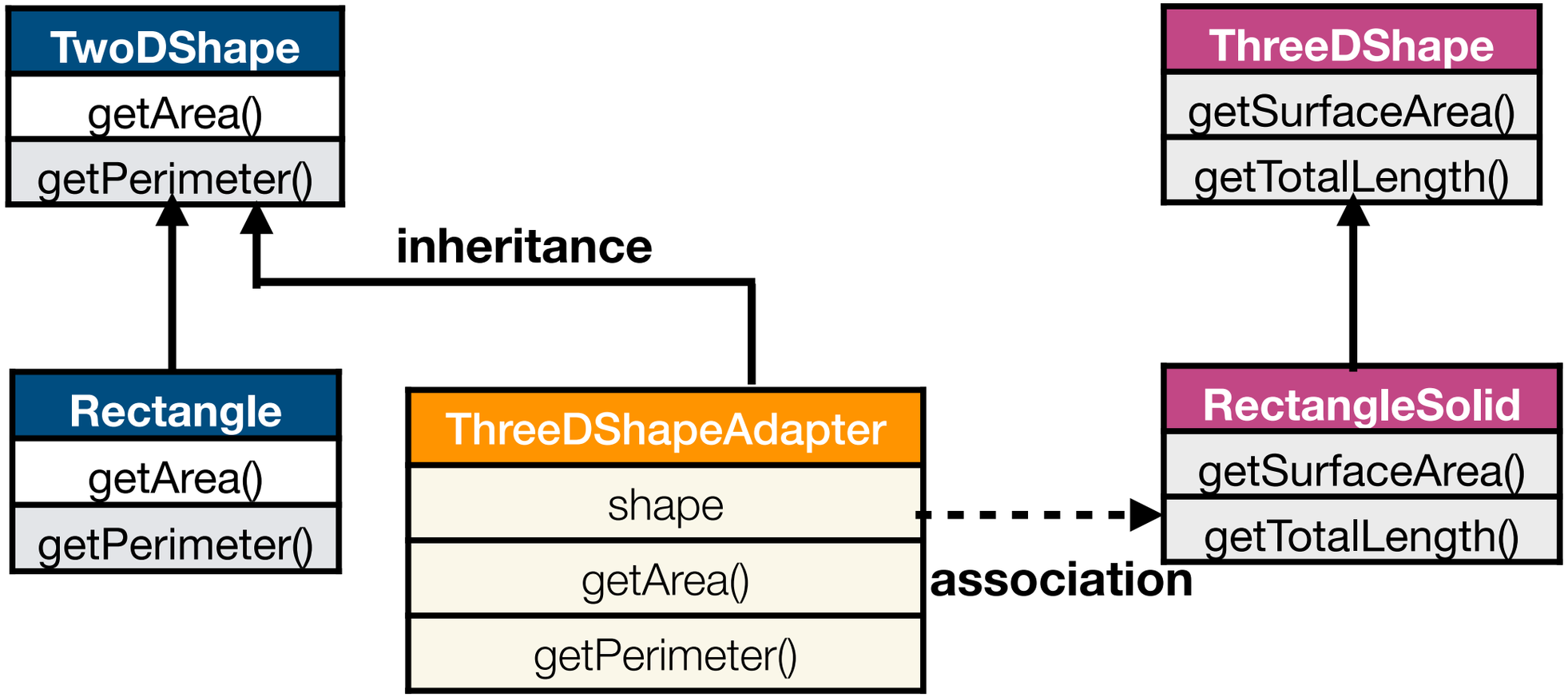
Example of Math Library

```
class IMath(ABC):
    @abstractmethod
    def add(self):
        pass

class PoorMath(ABC):
    def add(self):
        try:
            x = int(input("Enter the first integer: "))
            y = int(input("Enter the second integer: "))
        except ValueError:
            x = y = 0

        return x + y
```

```
class IMath2(ABC):
    @abstractmethod
    def add(self, x, y):
        pass

class GoodMath(ABC):
    def add(self, x, y):
        return x + y

class MathAdapter(IMath):
    def __init__(self, math):
        self.math = math

    def add(self):
        try:
            x = int(input("Enter the first integer: "))
            y = int(input("Enter the second integer: "))
        except ValueError:
            x = y = 0

        return self.math.add(x, y)

class Factory:
    def getMath(self):
        math = GoodMath()
        return MathAdapter(math)

def main():
    math = Factory().getMath()
    print("Sum =", math.add())
```

Example of Shape Library

- In this example, there is an old library called TwoDShape. This library provides functions for two dimensional shapes. The new library named ThreeDShape provides advanced functions for three-dimensional shapes. Therefore, in order for the client applications to use this new improved library without changing their codes, we will need ton provide an adapter.



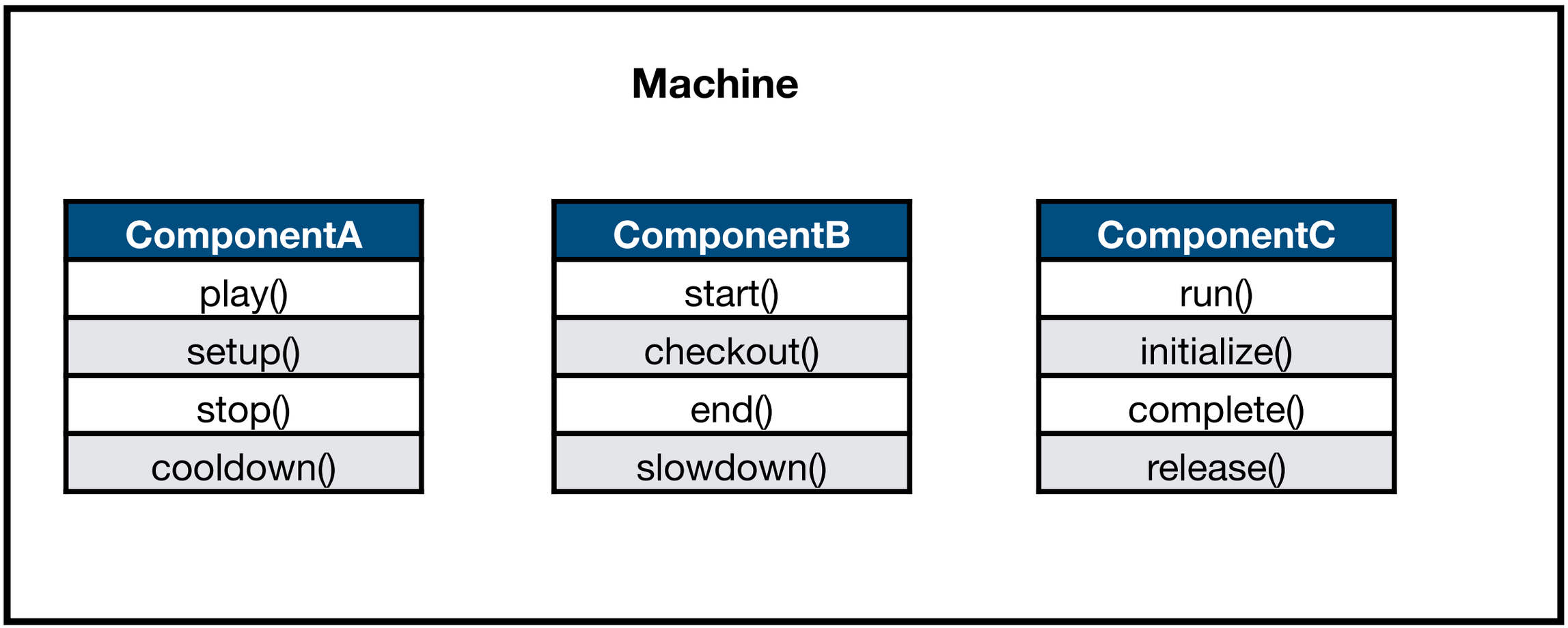
Example of Shape Library

```
class TwoDShape(ABC):
    @abstractmethod
    def getArea(self):
        pass

    @abstractmethod
    def getPerimeter(self):
        pass

class Rectangle(TwoDShape):
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def getArea(self):
        return self.length * self.width

    def getPerimeter(self):
        return 2 * (self.length + self.width)
```

```
class ThreeDShape(ABC):
    @abstractmethod
    def getSurfaceArea(self):
        pass

    @abstractmethod
    def getTotalLength(self):
        pass

class RectangleSolid(ThreeDShape):
    def __init__(self, length, width, height):
        self.length = length
        self.width = width
        self.height = height

    def getSurfaceArea(self):
        return 2 * (self.length * self.width + self.length *
self.height + self.width * self.height)

    def getTotalLength(self):
        return 4 * (self.length + self.width + self.height);

class ThreeDShapeAdapter(TwoDShape):
    def __init__(self, shape):
        self.shape = shape

    def getArea(self):
        return self.shape.getSurfaceArea() / 2

    def getPerimeter(self):
        return self.shape.getTotalLength() / 2
```

The Facade Pattern

- This pattern simplifies and unifies a set of complex classes that belong to some sub-module. Facade defines a user-friendly interface that makes the sub-module easier to use.



Example of Facade Pattern

- To turn on the machine, you need to execute a sequence of method calls on each component as shown in the main method.

```
def main():
    comA = ComponentA()
    comB = ComponentB()
    comC = ComponentC()

    # Turn on the machine
    comA.setup()
    comB.start()
    comC.run()
    comA.play()
    comB.checkout()
    comC.initialize()

    # Turn off the machine
    comA.cooldown()
    comB.end()
    comC.complete()
    comA.stop()
    comB.slowdown()
    comC.release()
```

Example of Facade Pattern

```
class ControlFacade:
    def __init__(self, comA, comB, comC):
        self.comA = comA
        self.comB = comB
        self.comC = comC

    def turnOn(self):
        self.comA.setup()
        self.comB.start()
        self.comC.run()
        self.comA.play()
        self.comB.checkout()
        self.comC.initialize()

    def turnOff(self):
        self.comA.cooldown()
        self.comB.end()
        self.comC.complete()
        self.comA.stop()
        self.comB.slowdown()
        self.comC.release()

def main():
    comA = ComponentA()
    comB = ComponentB()
    comC = ComponentC()

    control = ControlFacade(comA, comB, comC)
    control.turnOn()
    control.turnOff()
```

- We used Facade to encapsulate the sequence of method calls in a new class “ControlFacade”. The new ControlFacade class allows the client code to turn on or off the machine easier.

The Principle of Least Knowledge

- This principle guides us to reduce the interactions between objects to your immediate friends. It helps to prevent us from designing architectures that have a large number of classes coupled together so that changes in one part of the module cascade to the other parts.
- Example,

```
public Book getBook() {
    return library.getSectoon().getBookshelf().getBook();
}
```

- Here are guidelines provided by this pattern: take any object, from any method in that object, we should only invoke methods that belong to:
  - The object itself
  - Objects passed in as a method parameter
  - Any object the method creates
  - Any components of the object.
- So in order to reduce the number of classes we're dependent on, the library class should have a getBook() to return a book directly.

```
public Book getBook() {
    return library.getBook();
}
```

Summary

- When you need to use an existing class and it's interface is not the one you need, use the Adapter pattern.
- An adapter changes an interface into one a client expects
- When you need to simplify and unify a large interface or complex set of interfaces, use the Facade pattern.
- A facade decouples a client from a complex sub-module.

object.__dict__

- According to python documentation object.__dict__ is a dictionary or other mapping object used to store an object's (writable) attributes.
- As from the below code, we can observe that when __dict__ is called for the object of class SomeClass we get the dictionary for the defined attributes.
- Example,

```
def isSmall(value):
    if self.value < 1000:
        return True
    else:
        return False

class SomeClass:
    def __init__(self, name, value, method2):
        self.name = name
        self.value = value
        self.method1 = self.isBig
        self.method2 = method2

    def isBig(self):
        if self.value > 1000:
            return True
        else:
            return False

sc = SomeClass('score', 1001, isSmall)
print(sc.__dict__)
```

```
{
'name': 'score',
'value': 1001,
'method1': <bound method
SomeClass.isBig of <__main__.SomeClass
object at 0xf7be1b98>>,
'method2': <function isSmall at
0xf7bf07c0>
}
```

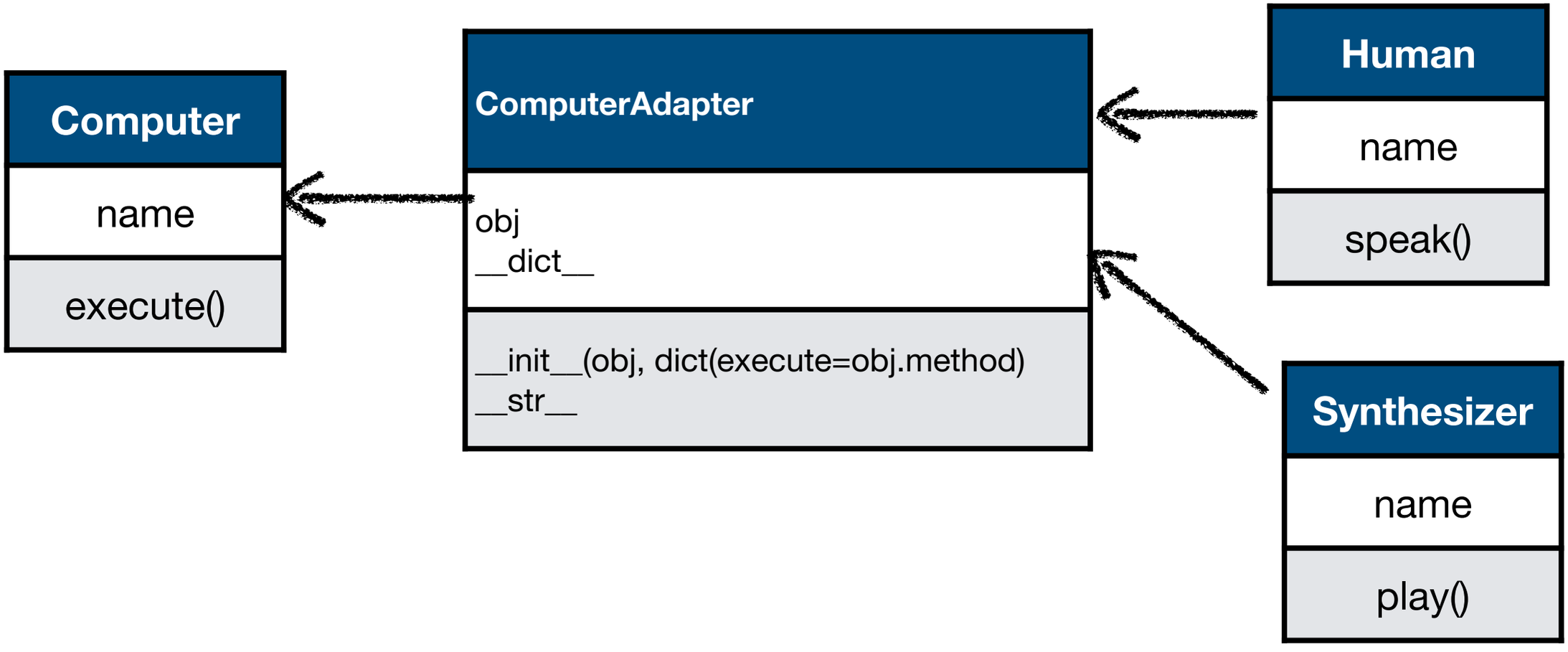
The dictionary update() method

- The update() method inserts the specified items to the dictionary.
- The specified items can be a dictionary, or an iterable object with key value pairs.
- Syntax:
  - dictionary.update(iterable)
    - iterable - a dictionary or an iterable object with key value pairs, that will be inserted to the dictionary
- Example

```
book = {
    "title": "Python Programming",
    "author": "Jack Peterson",
    "publisher": "SAMS"
}

book.update({"price": 100.99})
print(book)
```

Case Study: Computer Adapter in Python way

- The original Computer class has a method called “execute”. This method is called by the client code.
- There are two other classes: Human and Synthesizer. They have different methods: speak() and play().
- If the client code can only call execute method, we will use the Adapter pattern to adapt these two classes into Computer class interface.



Case Study: Computer Adapter in Python way

computer_adapter.py

```
from external import Synthesizer, Human

class Computer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} computer'.format(self.name)

    def execute(self):
        return 'executes a program'

class ComputerAdapter:
    def __init__(self, obj, adapted_methods):
        self.obj = obj
        self.__dict__.update(adapted_methods)

    def __str__(self):
        return str(self.obj)

class Factory:
    def getObject(self, type):
        obj = None
        if type == "Computer":
            obj = Computer("Apple")
        elif type == "Synthesizer":
            synth = Synthesizer("Homepod")
            obj = ComputerAdapter(synth, dict(execute=synth.play))
        elif type == "Human":
            human = Human('Peter')
            obj = ComputerAdapter(human, dict(execute=human.speak))

        return obj

def main():
    type = input("Enter the type of object: ")
    obj = Factory().getObject(type)
    print(obj.execute())

if __name__ == "__main__":
    main()
```

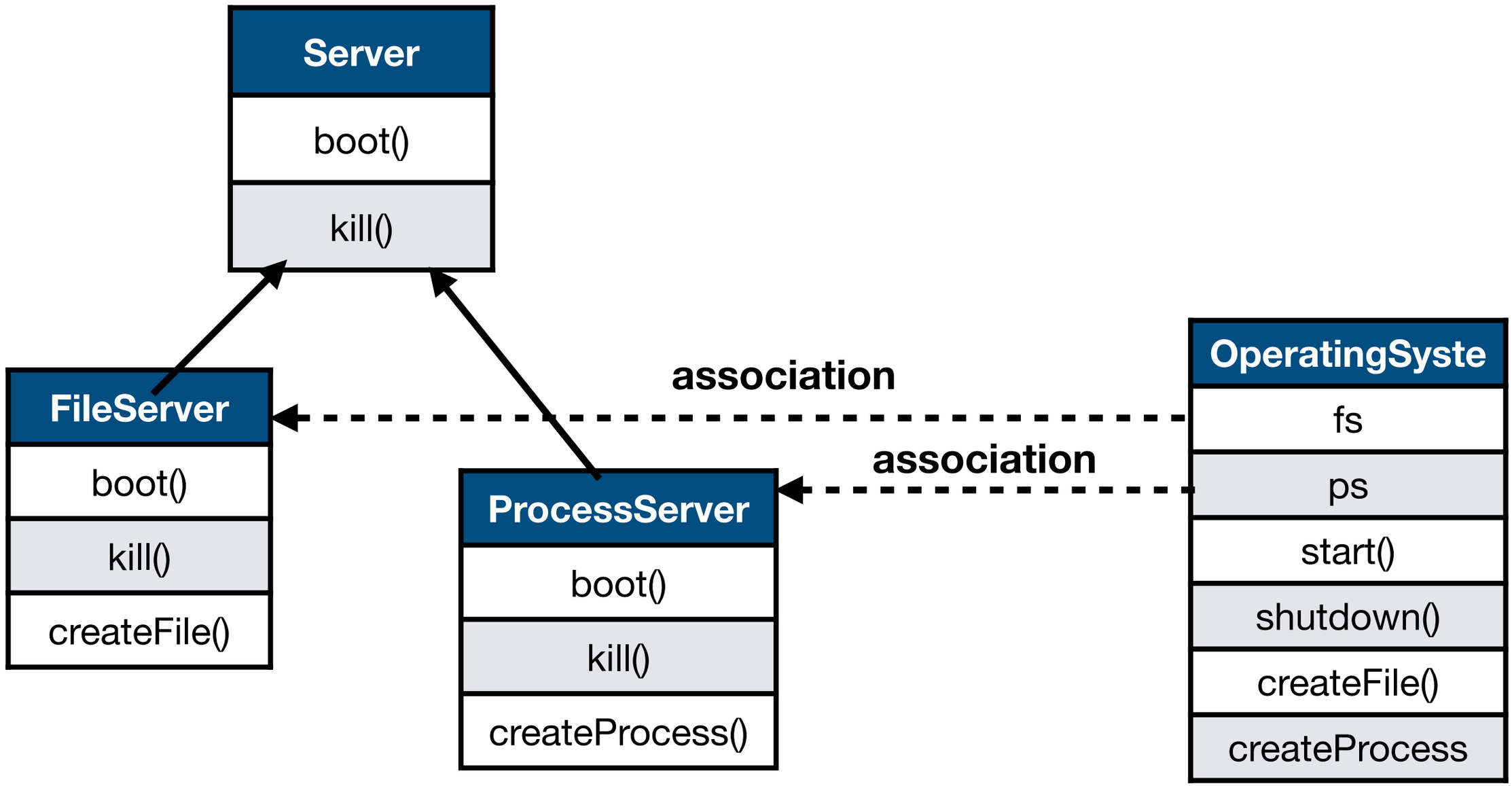
external.py

```
class Synthesizer:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return 'the {} synthesizer'.format(self.name)

    def play(self):
        return 'is playing an electronic song'

class Human:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return '{} the human'.format(self.name)

    def speak(self):
        return 'says hello'
```

```
Enter the type of object: Human
says hello

Enter the type of object: Synthesizer
is playing an electronic song

Enter the type of object: Computer
executes a program
```

Case Study Facade Pattern: Operating System

- A modular operating system can have a number of servers: a file server, a process server, a network server, a graphical/window server, and so forth. The following example includes two servers: the FileServer and the ProcessServer.
- The OperatingSystem class is a façade. In its __init__(), all the necessary server instances are created. The start() method, used by the client code, is the entry point to start all servers within the system.



Case Study: Operating System

```
class OperatingSystem:
    def __init__(self):
        self.fs = FileServer()
        self.ps = ProcessServer()

    def start(self):
        [i.boot() for i in (self.fs, self.ps)]

    def shutdown(self):
        [i.kill() for i in (self.fs, self.ps)]

    def createFile(self, user, name, permissions):
        return self.fs.createFile(user, name, permissions)

    def createProcess(self, user, name):
        return self.ps.createProcess(user, name)
```