CSIT121
Object-Oriented Design and Programming

Dr. Fenghui Ren

School of Computing and Information Technology
University of Wollongong

Lecture 11 outline

- Design pattern in brief
- Iterator pattern
- Strategy pattern
- Decorator pattern
- Observer pattern
- State pattern
- Singleton pattern
- Template pattern
- Adapter pattern
- Facade pattern
- Command pattern
- Abstract factory pattern
- Composite pattern

Design patterns in brief

- Design patterns are an attempt to bring the formal definitions and correct designed structures to software engineering.
- Design patterns can be used to solve general problems in some specific situations. They are ideal solutions for the OOD.
- We will learn different design patterns

Iterator Pattern

- An iterator is an object with a next() function and a done() function.
- The next() function can automatically move from the current item to the next item in a collection.
- The done() function can check if all items in the collection are visited.
- Build-in iterable classes, list, set, dictionary, and etc.

```
while not iterator.done():
    item = iterator.next()
    # do something with the item
```

Iterator Pattern

- You can also built your own iterable class in Python
- Python has the Iterator abstract base class in the collections.abc module
- If you want to create an iterable class, you should implement the __next__ method and the __iter__ method in the Iterator ABC.
- These methods will be called automatically when you call the next() and the iter() functions.
- The StopIteration is used in Python to check if the iteration has been completed.

Iterator Pattern Example

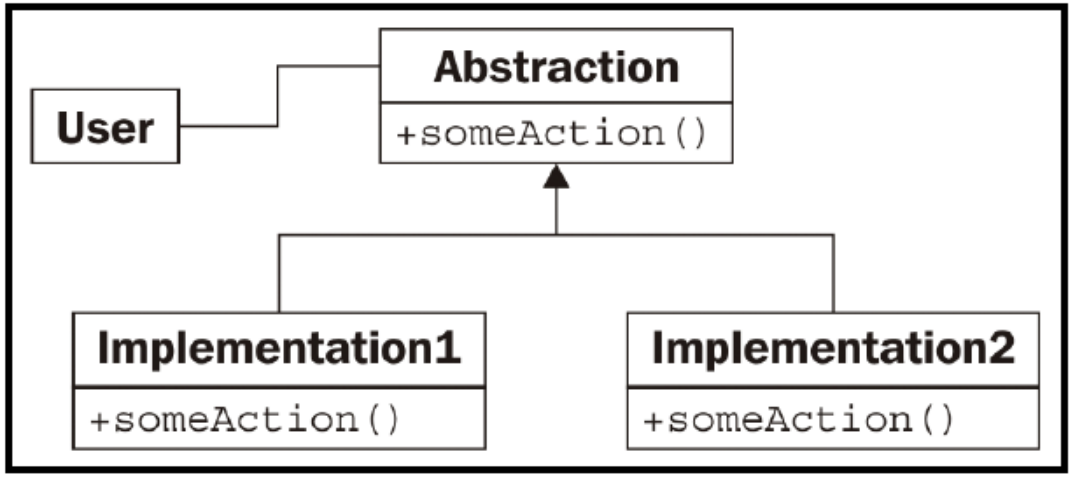
```
class CapitalIterable:
    def __init__(self, string):
        self.string = string

    def __iter__(self):
        return CapitalIterator(self.string)


class CapitalIterator:
    def __init__(self, string):
        self.words = [w.capitalize() for w in string.split()]
        self.index = 0

    def __next__(self):
        if self.index == len(self.words):
            raise StopIteration()

        word = self.words[self.index]
        self.index += 1
        return word

    def __iter__(self):
        return self
```

```
if __name__ == '__main__':
    iterable = CapitalIterable('hello object world')
    iterator = iter(iterable)
    while True:
        try:
            print(next(iterator))
        except StopIteration:
            break
```

```
= RESTART: /Users/fren/Library/Cloud
/CSIT121/Autumn_2023_Python/Python-3
Chapter09/capital_iterator.py
Hello
Object
World
```

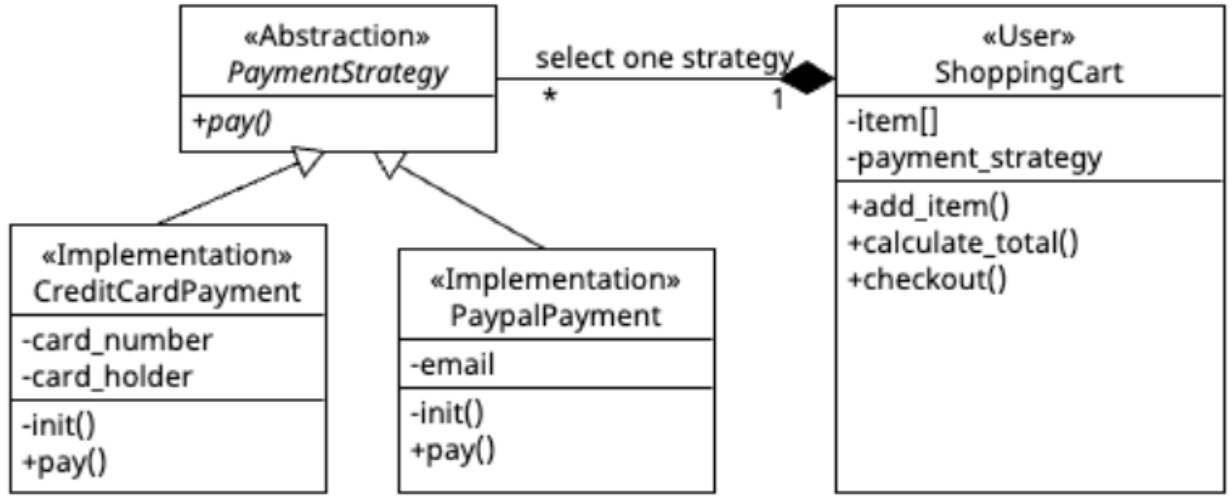
Strategy Pattern

- The strategy pattern is a common demonstration of abstraction in OOP.
- The pattern implements different solutions to a single problem, each in a different object.
- The user can choose the most appropriate implementation dynamically at runtime.



Strategy Pattern Example

```python
class PaymentStrategy:
    def pay(self, amount):
        pass

class CreditCardPayment(PaymentStrategy):
    def __init__(self, card_number, card_holder):
        self._card_number = card_number
        self._card_holder = card_holder

    def pay(self, amount):
        print(f"Paid {amount} via credit card {self._card_number}.")

class PayPalPayment(PaymentStrategy):
    def __init__(self, email):
        self._email = email

    def pay(self, amount):
        print(f"Paid {amount} via PayPal using {self._email}.")

class ShoppingCart:
    def __init__(self, payment_strategy):
        self._items = []
        self._payment_strategy = payment_strategy

    def add_item(self, item):
        self._items.append(item)

    def calculate_total(self):
        total = sum(item['price'] for item in self._items)
        return total

    def checkout(self):
        total = self.calculate_total()
        self._payment_strategy.pay(total)
```

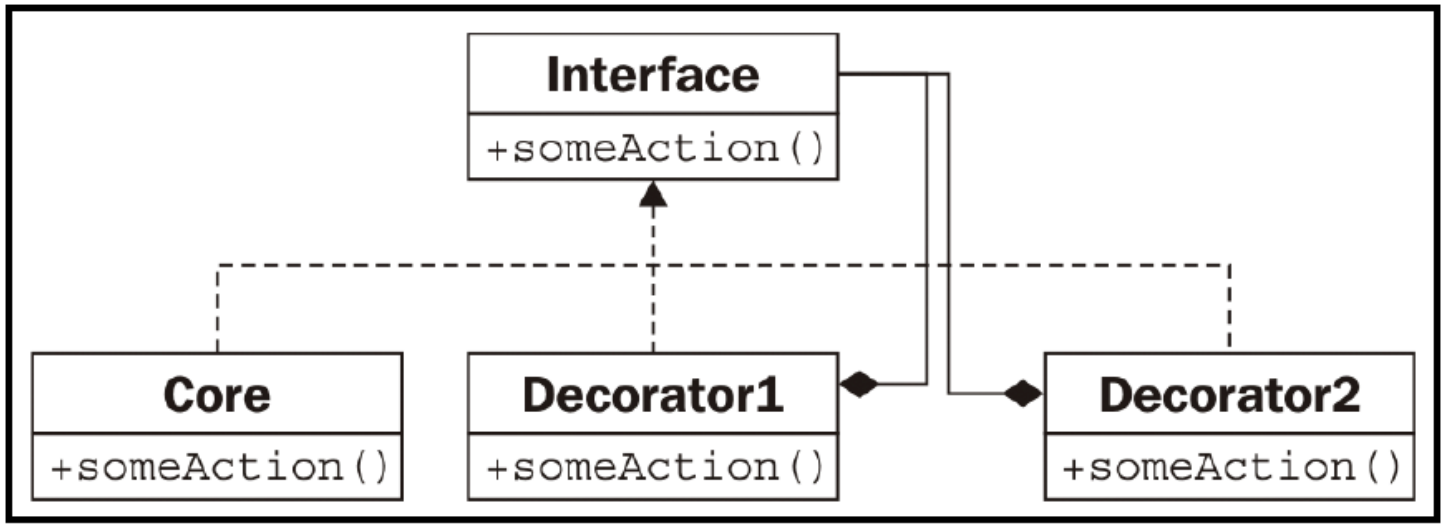



```python
if __name__ == "__main__":
    credit_card = CreditCardPayment("1234-5678-9012-3456", "John Doe")
    paypal = PayPalPayment("john@example.com")

    cart1 = ShoppingCart(credit_card)
    cart1.add_item({'name': 'Product 1', 'price': 50})
    cart1.add_item({'name': 'Product 2', 'price': 30})
    cart1.checkout()

    cart2 = ShoppingCart(paypal)
    cart2.add_item({'name': 'Product 3', 'price': 20})
    cart2.checkout()
```

```
= RESTART: /Users/fren/Library/CloudStorage/C
/CSIT121/Autumn_2023_Python/examples/payment_
Paid 80 via credit card 1234-5678-9012-3456.
Paid 20 via PayPal using john@example.com.
```

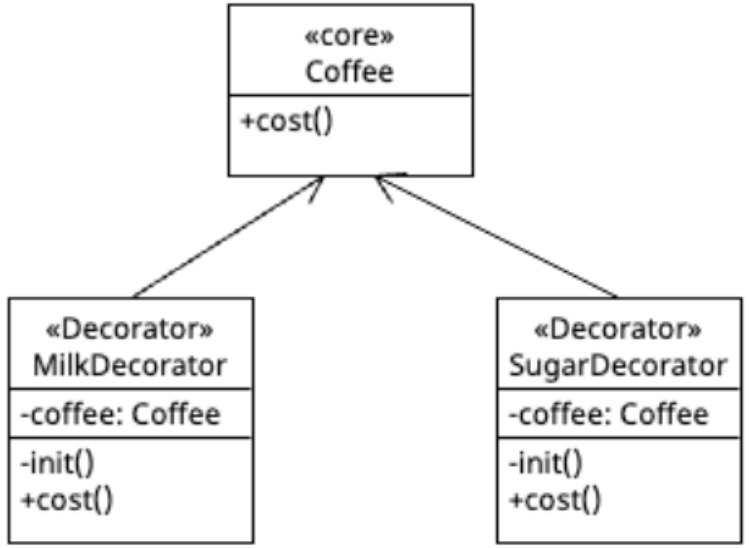
Decorator Pattern

- The decorator pattern is a structural design pattern that allows behavior to be added to individual objects without affecting the behavior of other objects from the same class.



Decorator Pattern Example

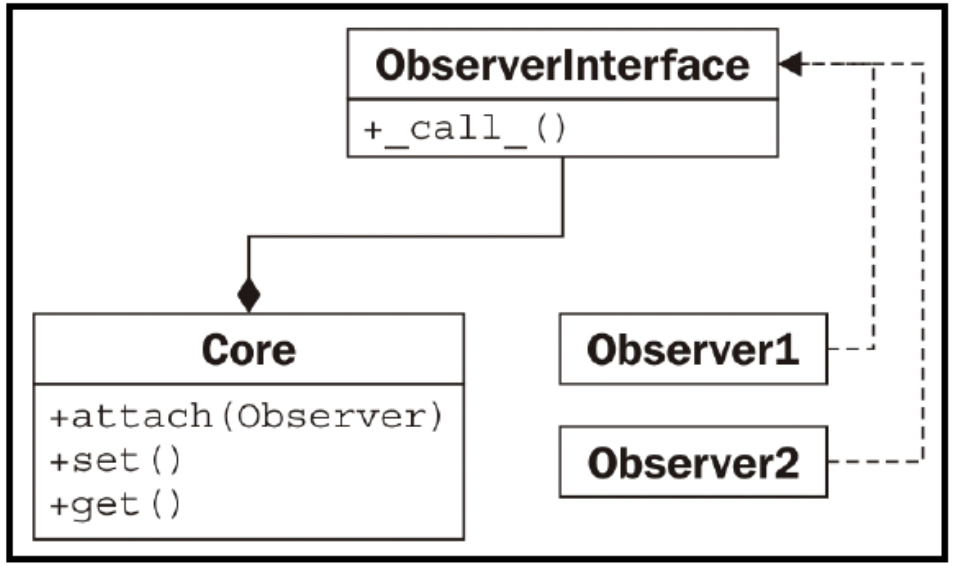
```
class Coffee:
    def cost(self):
        return 5

class MilkDecorator:
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost() + 2

class SugarDecorator:
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost() + 1

if __name__ == "__main__":
    simple_coffee = Coffee()
    coffee_with_milk = MilkDecorator(simple_coffee)
    coffee_with_milk_and_sugar = SugarDecorator(coffee_with_milk)

    print("Cost of simple coffee:", simple_coffee.cost())
    print("Cost of coffee with milk:", coffee_with_milk.cost())
    print("Cost of coffee with milk and sugar:", coffee_with_milk_and_sugar.cost())
```




```
= RESTART: /Users/fren/Library/CloudSto
/CSIT121/Autumn_2023_Python/examples/co
Cost of simple coffee: 5
Cost of coffee with milk: 7
Cost of coffee with milk and sugar: 8
```

Observer Pattern

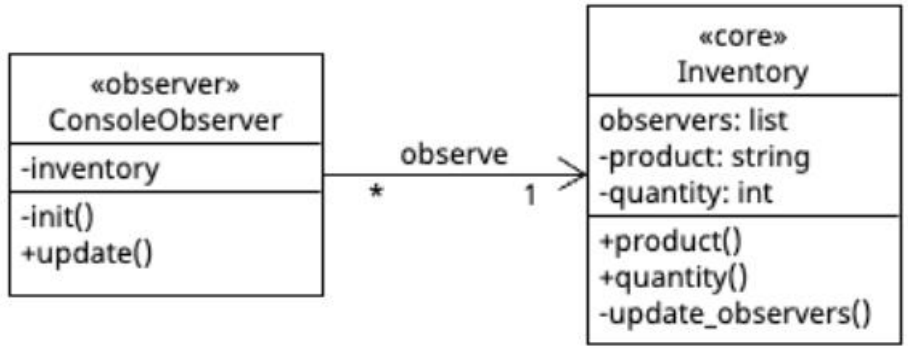
- The observer pattern is useful for state monitoring and event-handling situations.
- This pattern allows a given object to be monitored by an unknown and dynamic group of observer objects.
- Whenever a value on the core object changes, it lets all observer objects know by calling an update() method.
- Each observer may be responsible for different tasks and the core object no need to know those tasks.



Observer Pattern Example

```
class Inventory:
    def __init__(self):
        self.observers = []
        self._product = None
        self._quantity = 0

    def attach(self, observer):
        self.observers.append(observer)

    @property
    def product(self):
        return self._product

    @product.setter
    def product(self, value):
        self._product = value
        self._update_observers()

    @property
    def quantity(self):
        return self._quantity

    @quantity.setter
    def quantity(self, value):
        self._quantity = value
        self._update_observers()

    def _update_observers(self):
        for observer in self.observers:
            observer.update()
```

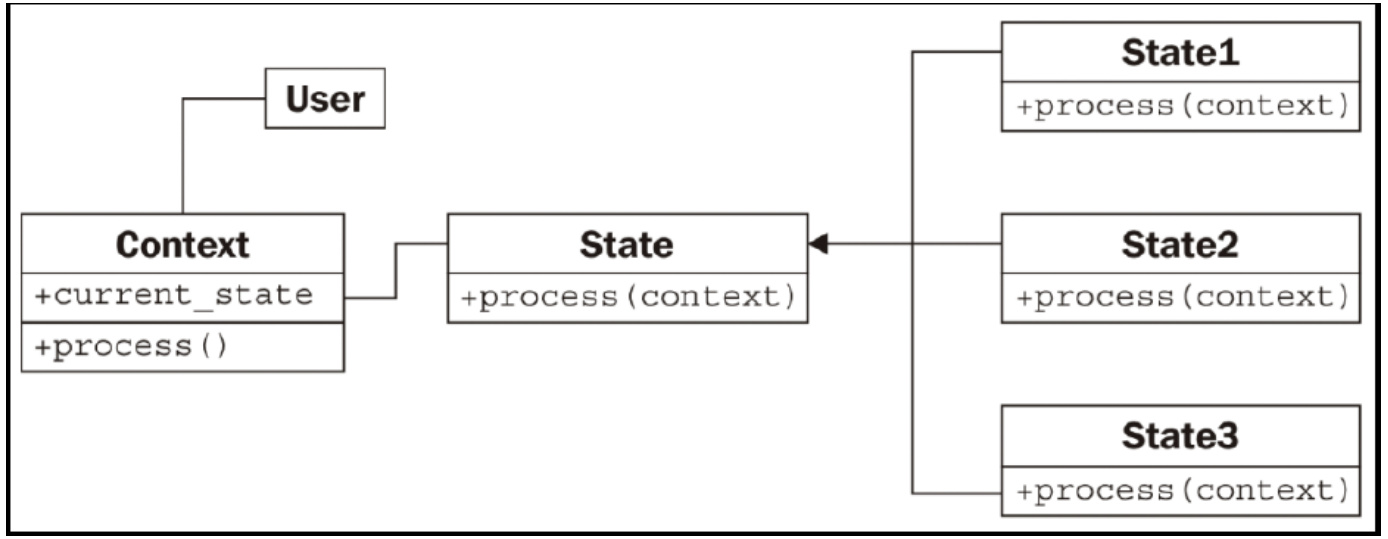
```
class ConsoleObserver:
    def __init__(self, inventory):
        self.inventory = inventory

    def update(self):
        print(self.inventory.product)
        print(self.inventory.quantity)

if __name__ == "__main__":
    i = Inventory()
    c = ConsoleObserver(i)
    i.attach(c)
    i.product = "widget"
    i.quantity = 5
```




```
= RESTART: /Users/fr
/CSIT121/Autumn_2023
widget
0
widget
5
```

State Pattern

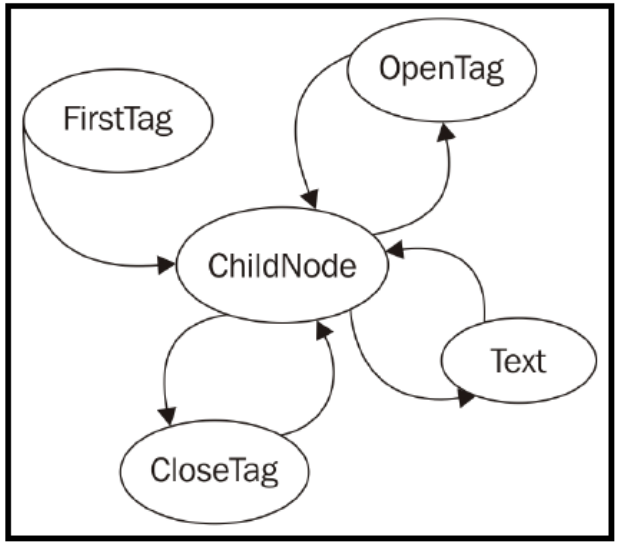
- The goal of the state pattern is to represent state-transition systems: systems where it is obvious that an object can be in a specific state and that certain activities may drive it to a different state, such as Finite-state-machine, UML state diagram
- A manager class that provides an interface for switching states, and maintains the current state.
- Each state knows what other states are allowed to be in and will transition to those states when possible.



State Pattern Example

- We will built an XML parsing tool to create a tree of node objects for each tag and its contents in a XML file.
- We need to consider what states our parser can be in. We need a state for processing opening tags and closing tags. When we are inside a tag with text contents, we will process that as a separate state.
- The FirstTag state will switch to ChildNode state, which is responsible for deciding which of other three states to switch to.
- When those states are finished, they will switch back to ChildNode state.

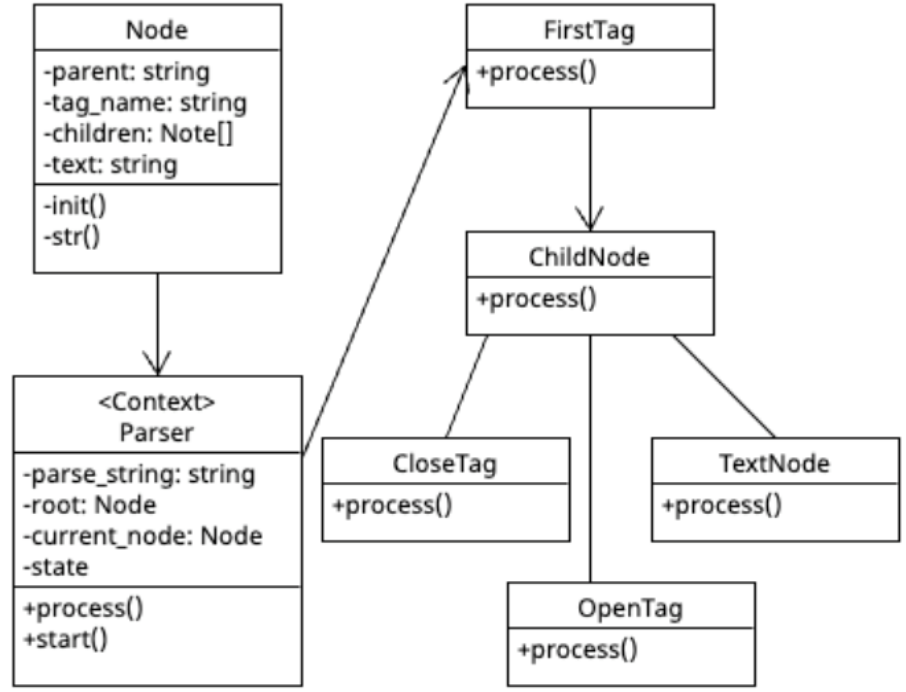
```
<book>
    <author>Dusty Phillips</author>
    <publisher>Packt Publishing</publisher>
    <title>Python 3 Object Oriented Programming</title>
    <content>
        <chapter>
            <number>1</number>
            <title>Object Oriented Design</title>
        </chapter>
        <chapter>
            <number>2</number>
            <title>Objects In Python</title>
        </chapter>
    </content>
</book>
```



State Pattern Example

```
class Node:
    def __init__(self, tag_name, parent=None):
        self.parent = parent
        self.tag_name = tag_name
        self.children = []
        self.text = ""

    def __str__(self):
        if self.text:
            return self.tag_name + ": " + self.text
        else:
            return self.tag_name

class Parser:
    def __init__(self, parse_string):
        self.parse_string = parse_string
        self.root = None
        self.current_node = None

        self.state = FirstTag()

    def process(self, remaining_string):
        remaining = self.state.process(remaining_string, self)
        if remaining:
            self.process(remaining)

    def start(self):
        self.process(self.parse_string)

if __name__ == "__main__":
    import sys

    with open('xml_example.xml') as file:
        contents = file.read()
        p = Parser(contents)
        p.start()

        nodes = [p.root]
        while nodes:
            node = nodes.pop(0)
            print(node)
            nodes = node.children + nodes
```

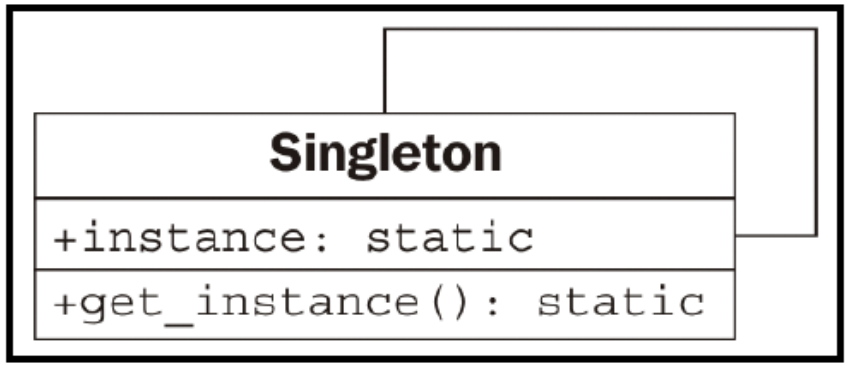
```
class FirstTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find("<")
        i_end_tag = remaining_string.find(">")
        tag_name = remaining_string[i_start_tag + 1 : i_end_tag]
        root = Node(tag_name)
        parser.root = parser.current_node = root
        parser.state = ChildNode()
        return remaining_string[i_end_tag + 1 :]


class ChildNode:
    def process(self, remaining_string, parser):
        stripped = remaining_string.strip()
        if stripped.startswith("</"):
            parser.state = CloseTag()
        elif stripped.startswith("<"):
            parser.state = OpenTag()
        else:
            parser.state = TextNode()
        return stripped


class OpenTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find("<")
        i_end_tag = remaining_string.find(">")
        tag_name = remaining_string[i_start_tag + 1 : i_end_tag]
        node = Node(tag_name, parser.current_node)
        parser.current_node.children.append(node)
        parser.current_node = node
        parser.state = ChildNode()
        return remaining_string[i_end_tag + 1 :]

class CloseTag:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find("<")
        i_end_tag = remaining_string.find(">")
        assert remaining_string[i_start_tag + 1] == "/"
        tag_name = remaining_string[i_start_tag + 2 : i_end_tag]
        assert tag_name == parser.current_node.tag_name
        parser.current_node = parser.current_node.parent
        parser.state = ChildNode()
        return remaining_string[i_end_tag + 1 :].strip()


class TextNode:
    def process(self, remaining_string, parser):
        i_start_tag = remaining_string.find("<")
        text = remaining_string[:i_start_tag]
        parser.current_node.text = text
        parser.state = ChildNode()
        return remaining_string[i_start_tag:]
```




```
= RESTART: /Users/fren/Library/CloudStorage/
/CSIT121/Autumn_2023_Python/examples/xml_sta
book
author: Dusty Phillips
publisher: Packt Publishing
title: Python 3 Object Oriented Programming
content
chapter
number: 1
title: Object Oriented Design
chapter
number: 2
title: Objects In Python
```

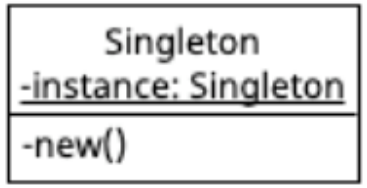
Singleton Pattern

- If your class allows only one instance to exist, you can use the singleton pattern
- In most OOP, singletons are enforced by making the constructor private, so no one can create additional instances of the class.
- A static method will be provided to create (for the first-time call) and retrieve the single instance.



Singleton Pattern Example

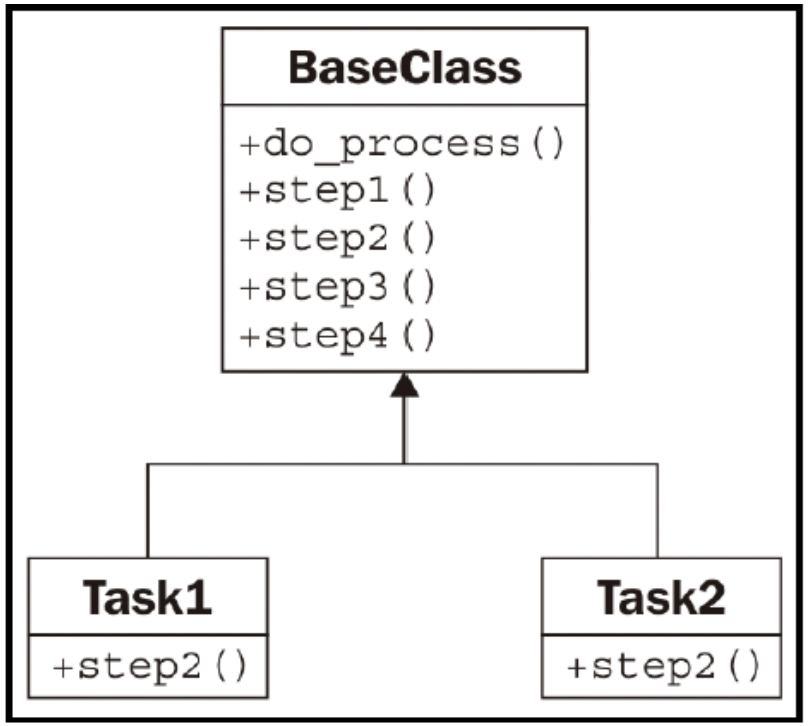
```
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

if __name__ == "__main__":
    instance1 = Singleton()
    instance2 = Singleton()

    print("Is instance1 the same as instance2?", instance1 is instance2)
```




```
= RESTART: /Users/fren/Library/CloudStorage/OneDr
/CSIT121/Autumn_2023_Python/examples/singleton.py
Is instance1 the same as instance2? True
```

Template Pattern

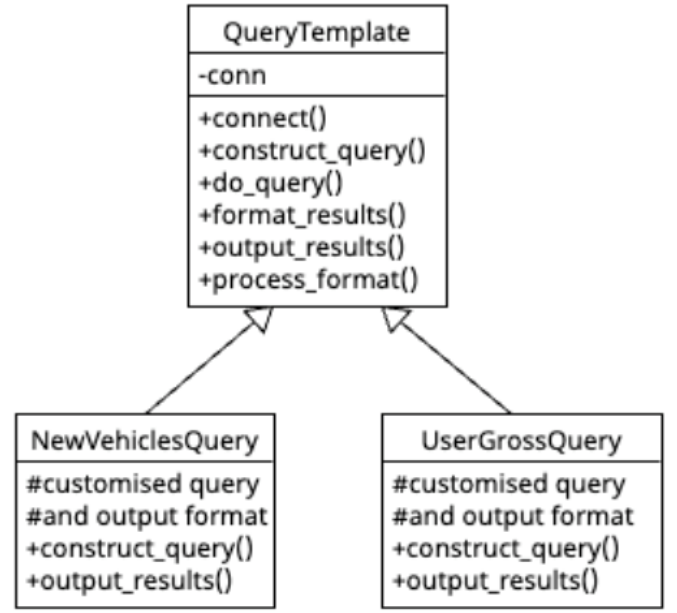
- The template pattern is useful for removing duplicate code.
- It is designed for situations where we have several different tasks to accomplish that have some common steps.
- The common steps are implemented in a base class, and the distinct steps are overridden in subclasses to provide custom behavior.
- In some ways, it is like a generalized strategy pattern.



Template Pattern Example

```
class QueryTemplate:
    def connect(self):
        self.conn = sqlite3.connect("sales.db")

    def construct_query(self):
        raise NotImplementedError()

    def do_query(self):
        results = self.conn.execute(self.query)
        self.results = results.fetchall()

    def format_results(self):
        output = []
        for row in self.results:
            row = [str(i) for i in row]
            output.append(", ".join(row))
        self.formatted_results = "\n".join(outpu

    def output_results(self):
        raise NotImplementedError()

    def process_format(self):
        self.connect()
        self.construct_query()
        self.do_query()
        self.format_results()
        self.output_results()
```

```
class NewVehiclesQuery(QueryTemplate):
    def construct_query(self):
        self.query = "select * from Sales where new='true'"

    def output_results(self):
        print(self.formatted_results)


class UserGrossQuery(QueryTemplate):
    def construct_query(self):
        self.query = (
            "select salesperson, sum(amt) "
            + " from Sales group by salesperson"
        )

    def output_results(self):
        filename = "gross_sales_{0}.txt".format(
            datetime.date.today().strftime("%Y%m%d")
        )
        with open(filename, "w") as outfile:
            outfile.write(self.formatted_results)

if __name__ == "__main__":
    nv = NewVehiclesQuery()
    nv.process_format()

    ug = UserGrossQuery()
    ug.process_format()
```




```
T121/Autumn_2023_Python/examples/car_sales_t
mplate.py
Tim, 16000, 2010, Honda Fit, true
Gayle, 28000, 2009, Ford Mustang, true
Gayle, 50000, 2010, Lincoln Navigator, true
>>>
```

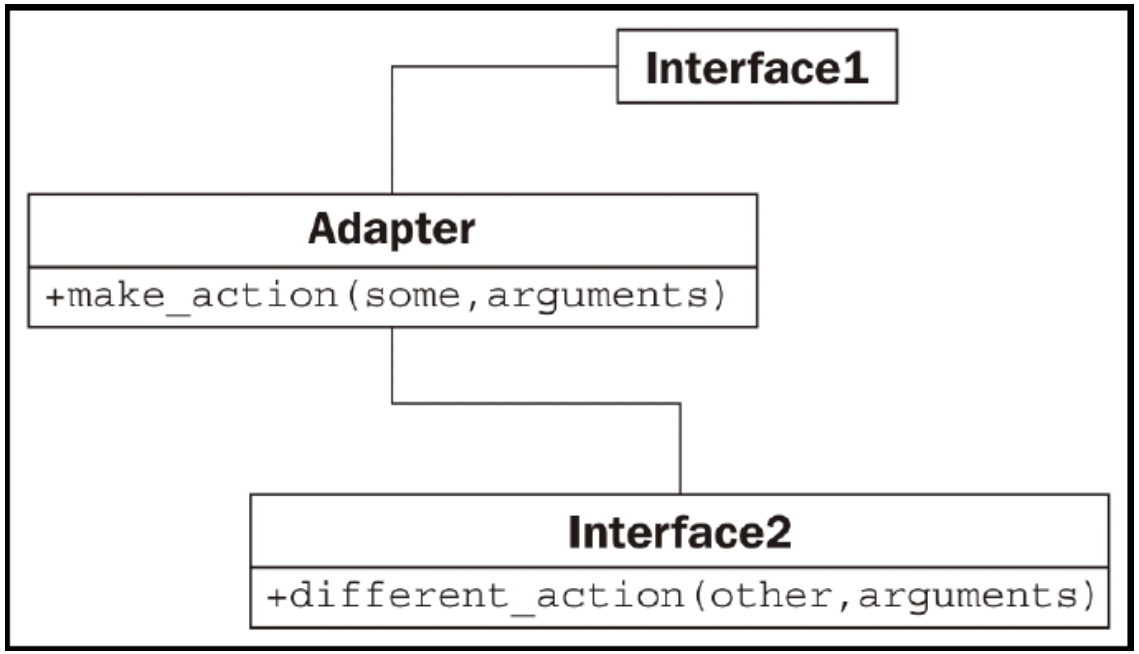
Ln: 275 Col:

gross_sales_20230817.txt — examples

```
1  Don, 20000
2  Gayle, 86000
3  Tim, 25000
```

Adapter Pattern

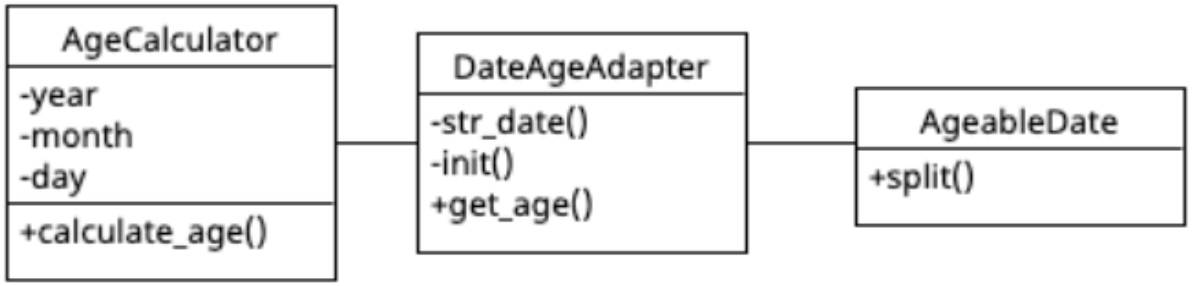
- The purpose of the adapter pattern is to interact between two preexisting objects to work together, in most cases, their interfaces are not compatible.
- The adapting may entail a variety of tasks, rearranging the order of arguments, calling a differently named method, or supplying default arguments.
- The adapter pattern can be considered a simplified decorator pattern.



Adapter Pattern Example

- We will write an adapter to adapt between the DOB and the age.

```
class AgeCalculator:
    def __init__(self, birthday):
        self.year, self.month, self.day = (
            int(x) for x in birthday.split("-")
        )

    def calculate_age(self, date):
        year, month, day = (int(x) for x in date.split("-"))
        age = year - self.year
        if (month, day) < (self.month, self.day):
            age -= 1
        return age


class DateAgeAdapter:
    def _str_date(self, date):
        return date.strftime("%Y-%m-%d")

    def __init__(self, birthday):
        birthday = self._str_date(birthday)
        self.calculator = AgeCalculator(birthday)

    def get_age(self, date):
        date = self._str_date(date)
        return self.calculator.calculate_age(date)


class AgeableDate(datetime.date):
    def split(self, char):
        return self.year, self.month, self.day
```

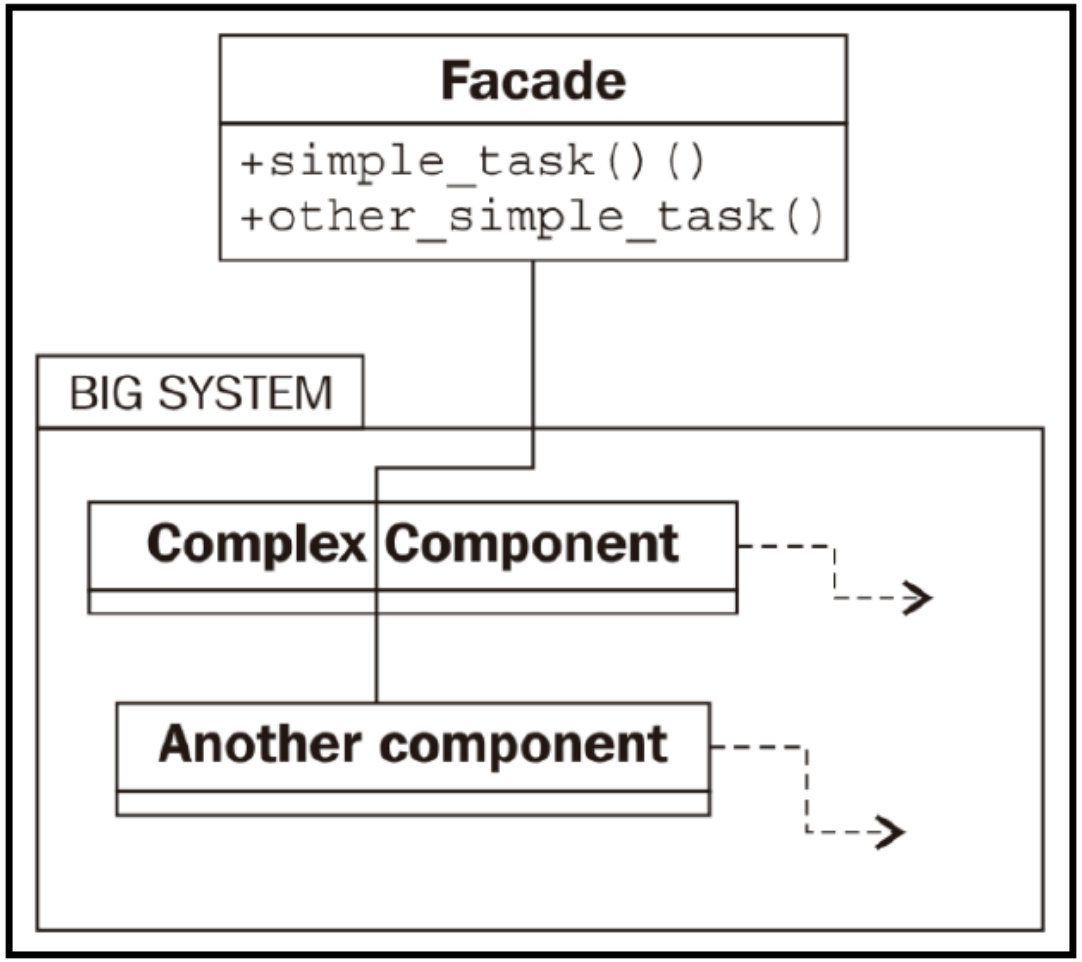



```
if __name__ == "__main__":
    bd = AgeableDate(1981,1,1)
    today = AgeableDate.today()
    print(today)
    age = AgeCalculator(bd)
    print(age.calculate_age(today))
```

```
= RESTART: /Users/fren/Li
/CSIT121/Autumn_2023_Pyth
2023-08-17
42
```

Facade Pattern

- The facade pattern is used to provide a simple interface to complex systems when typical complicated interactions are not necessary.
- The facade pattern allows us to define a new object that encapsulates this particular usage of the system. Any time we want access to common functionality, we can use the simplified interface.



Facade Pattern Example

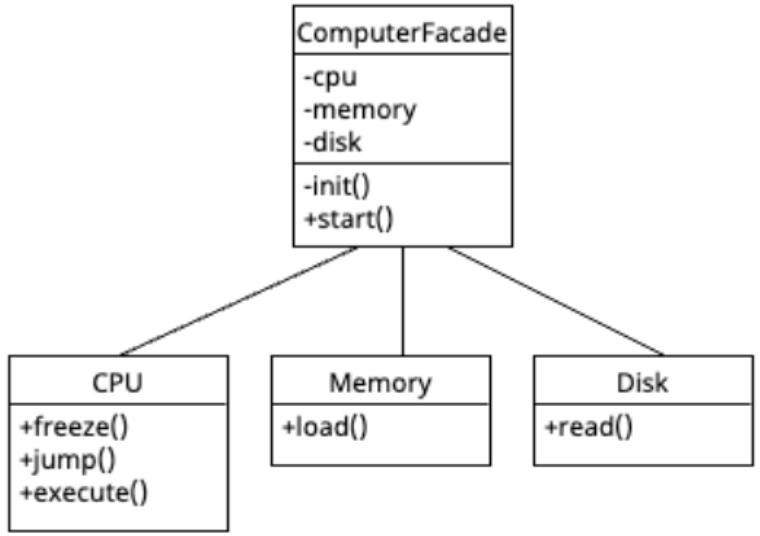
- We have a complex subsystem for handling various parts of a computer, such as CPU, Memory, and Disk. We'll create a Facade class to simplify the interaction with these subsystems.

```
# Complex subsystems
class CPU:
    def freeze(self):
        print("CPU: Freezing")

    def jump(self, position):
        print(f"CPU: Jumping to position {position}")

    def execute(self):
        print("CPU: Executing")

class Memory:
    def load(self, position, data):
        print(f"Memory: Loading data {data} at position {position}")

class Disk:
    def read(self, lba, size):
        print(f"Disk: Reading {size} bytes from LBA {lba}")
```

```
# Facade class
class ComputerFacade:
    def __init__(self):
        self.cpu = CPU()
        self.memory = Memory()
        self.disk = Disk()

    def start(self):
        self.cpu.freeze()
        self.memory.load(0, "BOOT_ADDRESS")
        self.cpu.jump("BOOT_ADDRESS")
        self.cpu.execute()
```

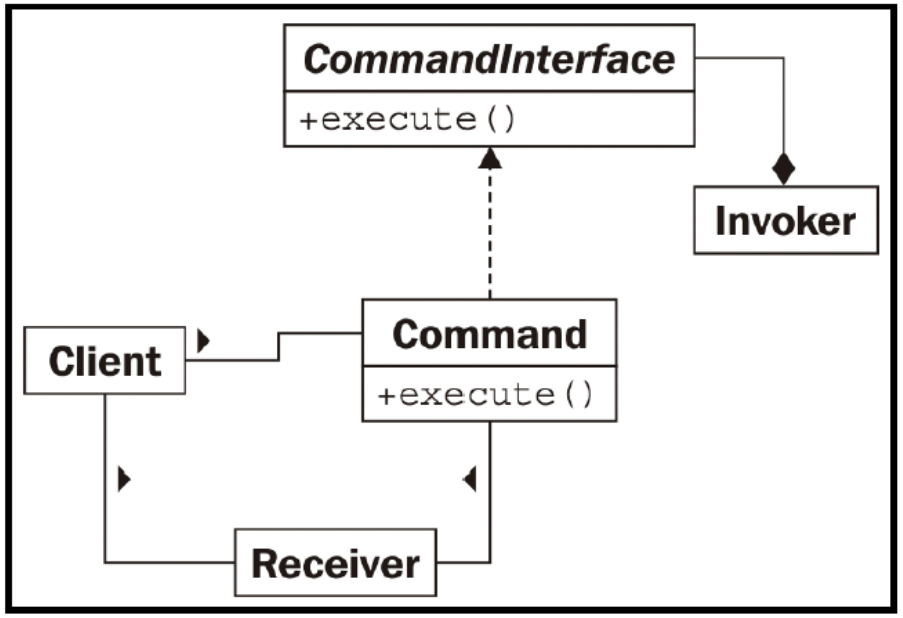
```
# Client code
computer = ComputerFacade()
computer.start()
```

```
= RESTART: /Users/fren/Library/CloudStorage/OneD
/CSIT121/Autumn_2023_Python/examples/cup_facade.
CPU: Freezing
Memory: Loading data BOOT_ADDRESS at position 0
CPU: Jumping to position BOOT_ADDRESS
CPU: Executing
```

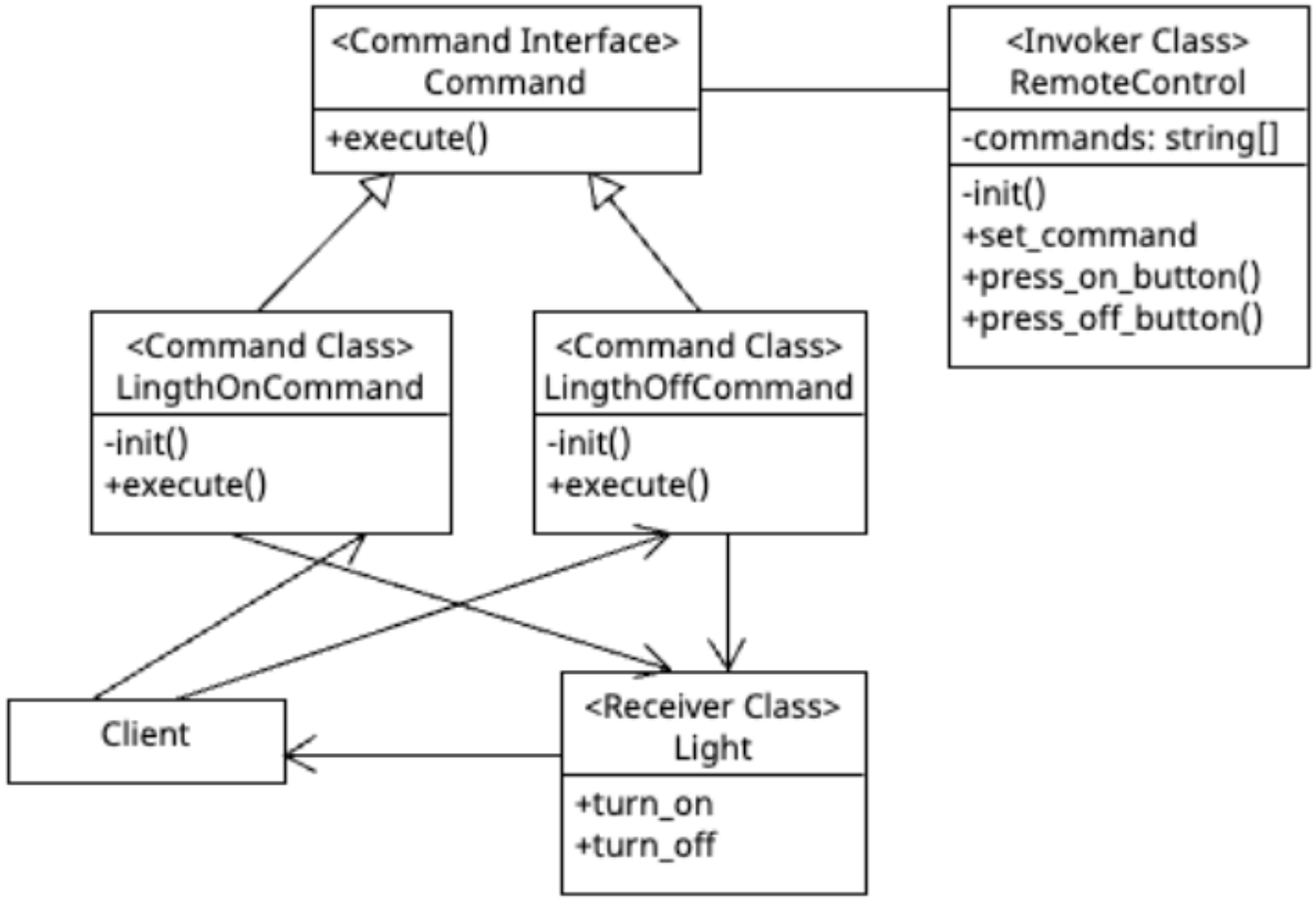


Command Pattern

- The command pattern adds a level of abstraction between actions and action callers.
- In the command pattern, client code creates a Command object which knows about a receiver object that manages its own internal state when the command is executed on it.
- The Command object implements a specific interface and also keeps track of any arguments required to perform the action.
- Finally, one or more Invoker objects execute the command at the correct time.



Command Pattern Example



Command Pattern Example

```
# Command interface
class Command(ABC):
    @abstractmethod
    def execute(self):
        pass

# Concrete command classes
class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()
```

```
# Receiver class
class Light:
    def turn_on(self):
        print("Light is on")

    def turn_off(self):
        print("Light is off")

# Invoker class
class RemoteControl:
    def __init__(self):
        self.commands = [None, None]

    def set_command(self, slot, on_command, off_command):
        self.commands[slot] = (on_command, off_command)

    def press_on_button(self, slot):
        on_command, _ = self.commands[slot]
        on_command.execute()

    def press_off_button(self, slot):
        _, off_command = self.commands[slot]
        off_command.execute()
```

```
# Client code
light = Light()
light_on = LightOnCommand(light)
light_off = LightOffCommand(light)

remote_control = RemoteControl()
remote_control.set_command(0, light_on, light_off)

remote_control.press_on_button(0)
remote_control.press_off_button(0)
```

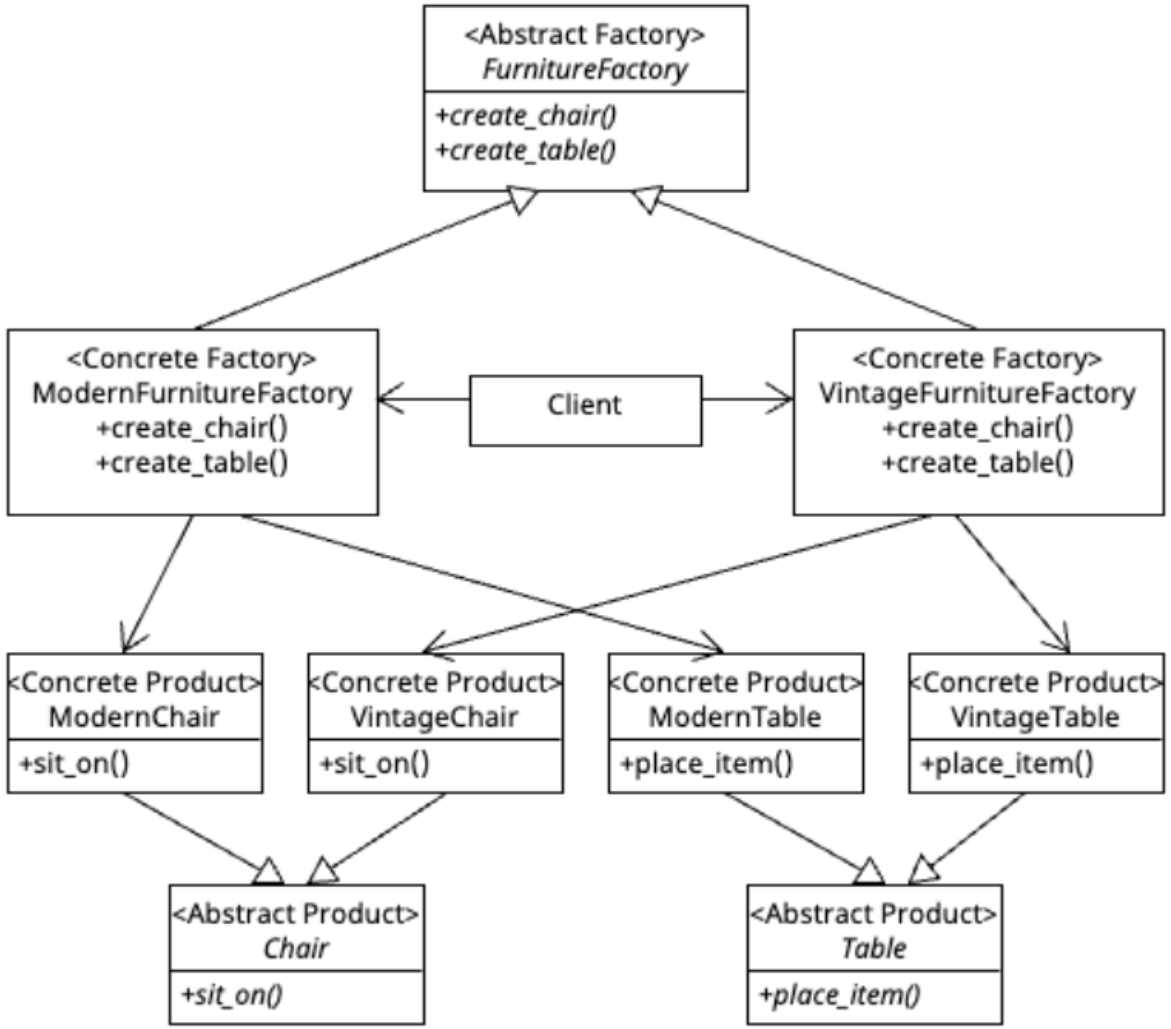
Program outputs

```
= RESTART: /Users/fr
/CSIT121/Autumn_2023
Light is on
Light is off
```

Abstract Factory Pattern

- The abstract factory pattern is normally used when we have multiple possible implementations of a system that depend on some configuration or platform issue.
- The calling code requests an object from the abstract factory, not knowing exactly what class of object will be returned.
- The underlying implementation returned may depend on a variety of factors, such as current locale, operating system, or local configuration.

Abstract Factory Pattern Example



Abstract Factory Pattern Example

```
# Abstract factory
class FurnitureFactory(ABC):
    @abstractmethod
    def create_chair(self):
        pass

    @abstractmethod
    def create_table(self):
        pass

# Concrete factory classes
class ModernFurnitureFactory(FurnitureFactory):
    def create_chair(self):
        return ModernChair()

    def create_table(self):
        return ModernTable()

class VintageFurnitureFactory(FurnitureFactory):
    def create_chair(self):
        return VintageChair()

    def create_table(self):
        return VintageTable()
```

```
# Abstract product classes
class Chair(ABC):
    @abstractmethod
    def sit_on(self):
        pass

class Table(ABC):
    @abstractmethod
    def place_items(self):
        pass

# Concrete product classes
class ModernChair(Chair):
    def sit_on(self):
        print("Sitting on a modern chair")

class ModernTable(Table):
    def place_items(self):
        print("Placing items on a modern table")

class VintageChair(Chair):
    def sit_on(self):
        print("Sitting on a vintage chair")

class VintageTable(Table):
    def place_items(self):
        print("Placing items on a vintage table")
```

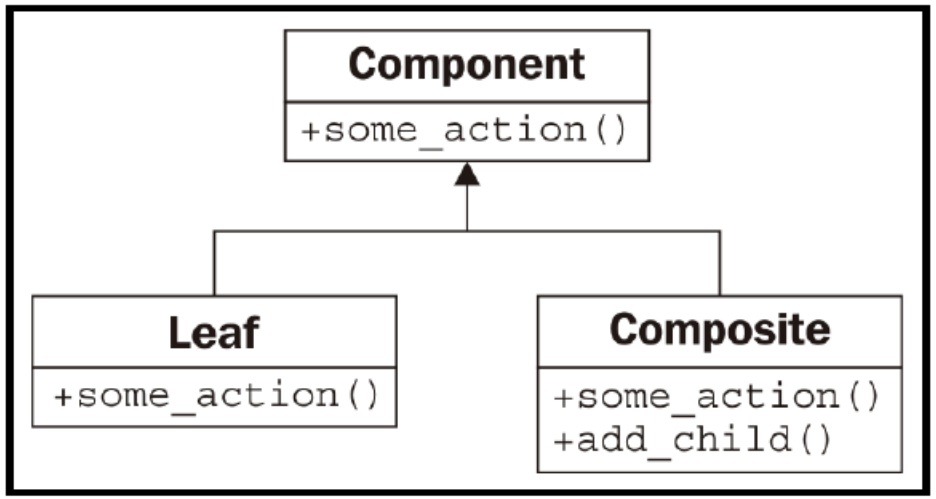
```
# Client code
def assemble_furniture(factory):
    chair = factory.create_chair()
    table = factory.create_table()

    print("Assembling furniture:")
    chair.sit_on()
    table.place_items()

modern_factory = ModernFurnitureFactory()
vintage_factory = VintageFurnitureFactory()

assemble_furniture(modern_factory)
assemble_furniture(vintage_factory)
```

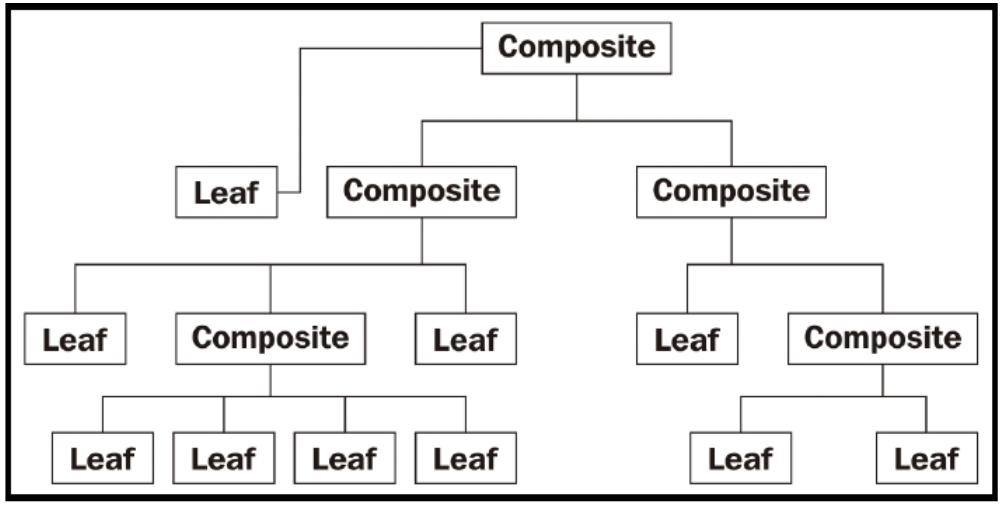
Program outputs

```
= RESTART: /Users/fren/Library/Cl
/CSIT121/Autumn_2023_Python/examp
Assembling furniture:
Sitting on a modern chair
Placing items on a modern table
Assembling furniture:
Sitting on a vintage chair
Placing items on a vintage table
```

Composite Pattern

- The composite pattern allows complex tree-like structures to be built from simple components called composite objects.
- Composite objects behave like a container and a variable, depending on whether they have child components.
- Traditionally, each component in a composite object must be either a leaf node (that cannot contain other objects) or a composite node. The key is that both composite and leaf nodes can have the same interface.

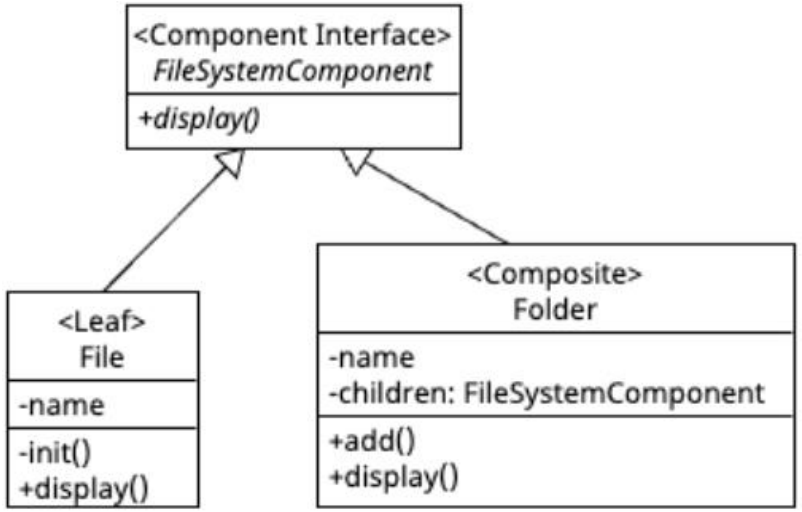




Composite Pattern Example

```
# Component interface
class FileSystemComponent(ABC):
    @abstractmethod
    def display(self):
        pass

# Leaf class
class File(FileSystemComponent):
    def __init__(self, name):
        self.name = name

    def display(self):
        print(f"File: {self.name}")

# Composite class
class Folder(FileSystemComponent):
    def __init__(self, name):
        self.name = name
        self.children = []

    def add(self, component):
        self.children.append(component)

    def display(self):
        print(f"Folder: {self.name}")
        for child in self.children:
            child.display()
```

```
# Client code
file1 = File("file1.txt")
file2 = File("file2.txt")
file3 = File("file3.txt")

subfolder1 = Folder("Subfolder 1")
subfolder1.add(file1)
subfolder1.add(file2)

root_folder = Folder("Root Folder")
root_folder.add(subfolder1)
root_folder.add(file3)

root_folder.display()
```



Program outputs

```
= RESTART: /Users/fren/Li
/CSIT121/Autumn_2023_Pyth
Folder: Root Folder
Folder: Subfolder 1
File: file1.txt
File: file2.txt
File: file3.txt
```

Suggested reading

Python 3 Object-Oriented Programming

- Chapter 9: The Iterator Pattern
- Chapter 10: Python Design Patterns I
- Chapter 11: Python Design Patterns II

Python

- https://www.python.org/
- https://docs.python.org/3/howto/regex.html