Programming and Design Patterns Lab

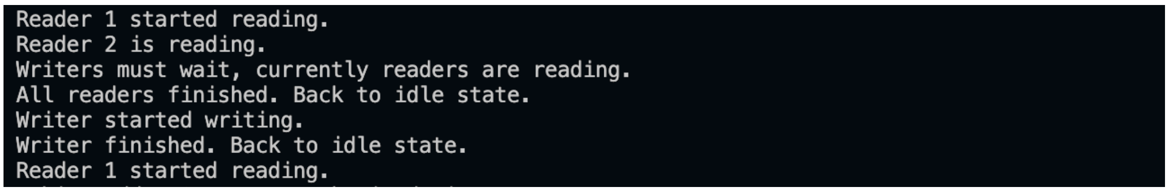
Ex 10 - Various patterns

Mohit R
IT-B

1)

```
from abc import ABC, abstractmethod

class State(ABC):
    @abstractmethod
    def read(self, context):
        pass

    @abstractmethod
    def write(self, context):
        pass

class ReadingState(State):
    def read(self, context):
        context.reader_count += 1
        print(f"Reader {context.reader_count} is reading.")

    def write(self, context):
        print("Writers must wait, currently readers are reading.")

class WritingState(State):
    def read(self, context):
        print("Readers must wait, a writer is writing.")

    def write(self, context):
        print("A writer is already writing.")

class IdleState(State):
    def read(self, context):
        context.set_state(ReadingState())
        context.reader_count += 1
        print(f"Reader {context.reader_count} started reading.")

    def write(self, context):
        context.set_state(WritingState())
        print("Writer started writing.")

class ReaderWriterContext:
    def __init__(self):
        self.state = IdleState()
        self.reader_count = 0

    def set_state(self, state: State):
        self.state = state
    def read(self):
        self.state.read(self)

    def write(self):
        self.state.write(self)

    def finish_reading(self):
        if self.reader_count > 0:
            self.reader_count -= 1
        if self.reader_count == 0:
            self.set_state(IdleState())
            print("All readers finished. Back to idle state.")

    def finish_writing(self):
        self.set_state(IdleState())
        print("Writer finished. Back to idle state.")

context = ReaderWriterContext()

context.read()
context.read()
context.write()
context.finish_reading()
context.finish_reading()
context.write()
context.finish_writing()
context.read()
```

OUTPUT :



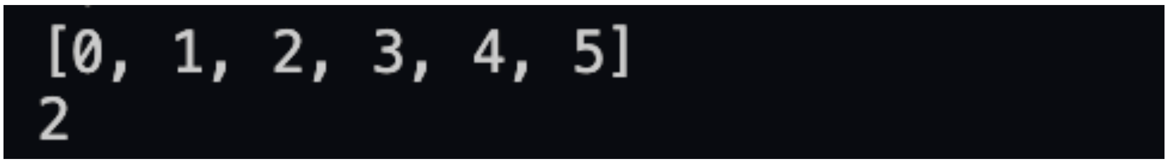
```
Reader 1 started reading.
Reader 2 is reading.
Writers must wait, currently readers are reading.
All readers finished. Back to idle state.
Writer started writing.
Writer finished. Back to idle state.
Reader 1 started reading.
```

2)

```
def sorting(lst):
    return sorted(lst)

def getval(lst, index):
    if index < len(lst):
        return lst[index]
    else:
        return None
```

```
class NDArrayAdapter:
    def __init__(self, ndarray):
        self.flat_list = self.flatten(ndarray)

    def flatten(self, ndarray):
        if isinstance(ndarray, (list, tuple)):
            flat_list = []
            for item in ndarray:
                flat_list.extend(self.flatten(item))
            return flat_list
        else:
            return [ndarray]

    def sort(self):
        self.flat_list = sorting(self.flat_list)

    def get_value(self, index):
        return getval(self.flat_list, index)

nd_array = [[3, 1], [4, 2],[5,0]]
adapter = NDArrayAdapter(nd_array)

# Flatten and sort
adapter.sort()
print(adapter.flat_list)  # Output: [1, 2, 3, 4]

# Get value at index
print(adapter.get_value(2))  # Output: 3
```

OUTPUT :



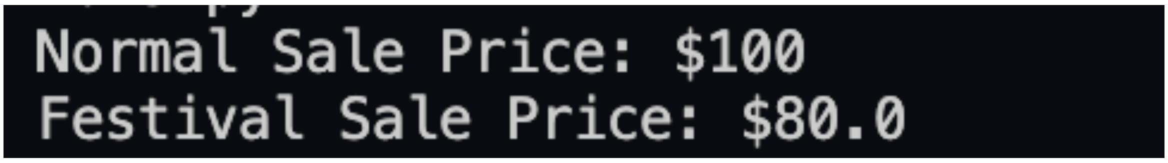
```
[0, 1, 2, 3, 4, 5]
2
```

3)

```
from abc import ABC, abstractmethod

class PricingStrategy(ABC):
    @abstractmethod
    def calculate_price(self, base_price):
        pass
```

```
class NormalSaleStrategy(PricingStrategy):
    def calculate_price(self, base_price):
        return base_price

class FestivalSaleStrategy(PricingStrategy):
    def calculate_price(self, base_price):
        discount = base_price * 0.20
        return base_price - discount

class SaleContext:
    def __init__(self, strategy: PricingStrategy):
        self.strategy = strategy

    def set_strategy(self, strategy: PricingStrategy):
        self.strategy = strategy

    def calculate_final_price(self, base_price):
        return self.strategy.calculate_price(base_price)



base_price = 100
normal_sale = NormalSaleStrategy()
sale_context = SaleContext(normal_sale)
print(f"Normal Sale Price: $
{sale_context.calculate_final_price(base_price)}")
festival_sale = FestivalSaleStrategy()
sale_context.set_strategy(festival_sale)
print(f"Festival Sale Price: $
{sale_context.calculate_final_price(base_price)}")
```

OUTPUT:

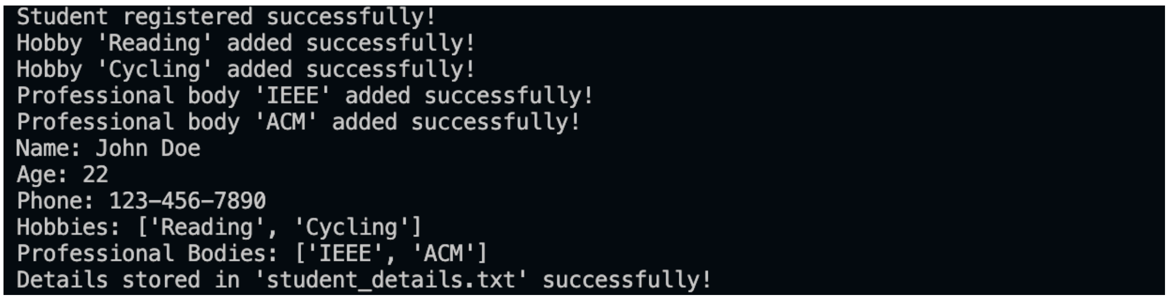


```
Normal Sale Price: $100
Festival Sale Price: $80.0
```

4)

```
class Student:
    def __init__(self, name, age, phone):
        self.name = name
        self.age = age
        self.phone = phone
        self.hobbies = []
        self.prof_bodies = []

    def add_hobby(self, hobby):
        self.hobbies.append(hobby)

    def add_prof_body(self, prof_body):
        self.prof_bodies.append(prof_body)

    def display(self):
        print(f"Name: {self.name}")
        print(f"Age: {self.age}")
        print(f"Phone: {self.phone}")
        print("Hobbies:", self.hobbies if self.hobbies else "None")
        print("Professional Bodies:", self.prof_bodies if self.prof_bodies else
"None")


class StudentRegistrationFacade:
    def __init__(self):
        self.student = None

    def register_student(self, name, age, phone):
        self.student = Student(name, age, phone)
        print("Student registered successfully!")

    def add_hobby(self, hobby):
        if self.student:
            self.student.add_hobby(hobby)
            print(f"Hobby '{hobby}' added successfully!")
        else:
            print("No student registered yet.")

    def add_prof_body(self, prof_body):
        if self.student:
            self.student.add_prof_body(prof_body)
            print(f"Professional body '{prof_body}' added successfully!")
        else:
            print("No student registered yet.")
    def store_in_file(self, filename):
        if self.student:
            with open(filename, 'w') as file:
                file.write(f"Name: {self.student.name}\n")
                file.write(f"Age: {self.student.age}\n")
                file.write(f"Phone: {self.student.phone}\n")
                file.write(f"Hobbies: {self.student.hobbies if self.student.hobbies
else 'None'}\n")
                file.write(f"Professional Bodies: {self.student.prof_bodies if
self.student.prof_bodies else 'None'}\n")
            print(f"Details stored in '{filename}' successfully!")
        else:
            print("No student registered to store details.")

    def display_student_details(self):
        if self.student:
            self.student.display()
        else:
            print("No student registered yet.")


# Usage Example
facade = StudentRegistrationFacade()
facade.register_student("John Doe", 22, "123-456-7890")
facade.add_hobby("Reading")
facade.add_hobby("Cycling")
facade.add_prof_body("IEEE")
facade.add_prof_body("ACM")
facade.display_student_details()
facade.store_in_file("student_details.txt")
```

OUTPUT :

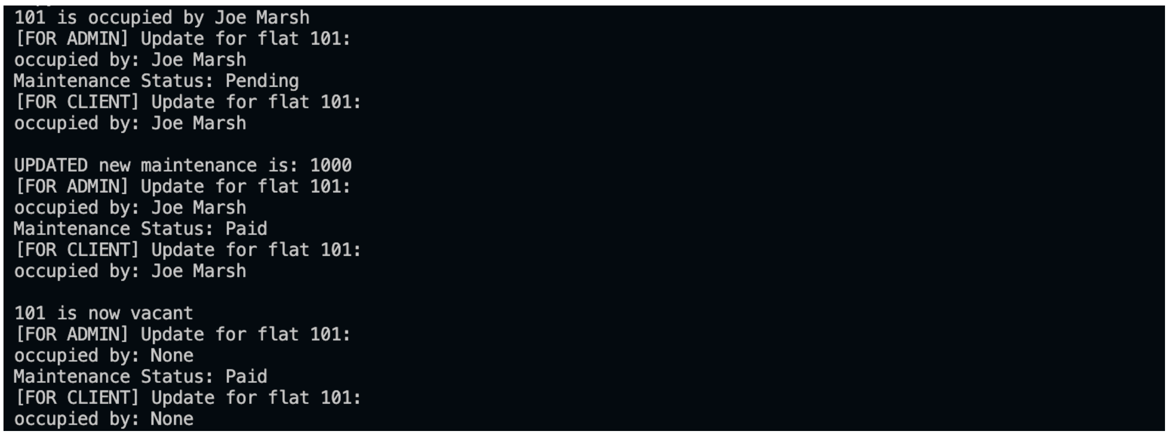


```
Student registered successfully!
Hobby 'Reading' added successfully!
Hobby 'Cycling' added successfully!
Professional body 'IEEE' added successfully!
Professional body 'ACM' added successfully!
Name: John Doe
Age: 22
Phone: 123-456-7890
Hobbies: ['Reading', 'Cycling']
Professional Bodies: ['IEEE', 'ACM']
Details stored in 'student_details.txt' successfully!
```

5)

```
from abc import ABC,abstractmethod

class Flat:
    def __init__(self,flatno,bhk,stat,detail):
        self.flatno=flatno
        self.bhk=bhk
        self.stat=stat
        self.detail=detail
        self.maint=0
        self.obs=[]

    def regisobs(self,obser):
        self.obs.append(obser)

    def remove_obs(self,obser):
        self.obs.remove(obser)

    def notifyobs(self):
        for observer in self.obs:
            observer.update(self)

    def updatemain(self,mainte):
        self.maint=mainte
        print("UPDATED new maintenance is:",self.maint)
        self.notifyobs()

    def occupy(self,det):
        self.stat="occupied"
        self.detail=det
        print(f"{self.flatno} is occupied by {self.detail}")
        self.notifyobs()

    def vacate(self):
        self.state="unoccupied"
        self.detail=None
        print(f"{self.flatno} is now vacant")
        self.notifyobs()

class observer( ABC):
    @abstractmethod
    def update(self,flat):
        pass

class admin(observer):
    def update(self,flat):
        print(f"[FOR ADMIN] Update for flat {flat.flatno}:")
        if flat.stat=="occupied":
            print("occupied by:",flat.detail)
        else:
            print("unoccupied flat")
        print(f"Maintenance Status: {'Paid' if flat.maint else 'Pending'}")

class client(observer):
    def update(self,flat):
        print(f"[FOR CLIENT] Update for flat {flat.flatno}:")
        if flat.stat=="occupied":
            print("occupied by:",flat.detail)
        else:
            print("unoccupied flat")

flat101=Flat(flatno=101,bhk=2,stat="unoccupied",detail=None)
admin_obs=admin()
client_obs=client()
flat101.regisobs(admin_obs)
flat101.regisobs(client_obs)
flat101.occupy("Joe Marsh")
print()
flat101.updatemain(1000)
print()
flat101.vacate()
```

OUTPUT:



```
101 is occupied by Joe Marsh
[FOR ADMIN] Update for flat 101:
occupied by: Joe Marsh
Maintenance Status: Pending
[FOR CLIENT] Update for flat 101:
occupied by: Joe Marsh

UPDATED new maintenance is: 1000
[FOR ADMIN] Update for flat 101:
occupied by: Joe Marsh
Maintenance Status: Paid
[FOR CLIENT] Update for flat 101:
occupied by: Joe Marsh

101 is now vacant
[FOR ADMIN] Update for flat 101:
occupied by: None
Maintenance Status: Paid
[FOR CLIENT] Update for flat 101:
occupied by: None
```