DAT055– Object Oriented Applications
Exam & Solution 2024-May-31

Examiner: Dr. Yehia Abd Alrahman

I will visit the exam hall at approximately one hour after the start, and one hour before the end.

Examination Rules:
The exam is 60 points and graded as U (0-23)/ 3 (24-35)/ 4 (36-47)/ 5 (48-60) (fail, pass, pass with merits, pass with distinction).

The exam consists of two parts:
Part A (questions 1-6) consists of questions covering broad knowledge expected to have been learned.

Part B (questions 7-8) covers advanced questions

Extra Aids: No extra aids allowed.

- Check that your exam paper is 8 pages.
- Answers must be given in English.
- Use page numbering on your pages.
- Start every question on a fresh page.
- Write clearly; unreadable = wrong!
- Unnecessarily complicated solutions are considered incorrect.
- Read all parts of the exam before starting to answer the first question.
- Indicate clearly when you make assumptions that are not given in the question.
- Insignificant syntax errors and similar will not be penalised.

Good luck!

PART A – Basic Knowledge

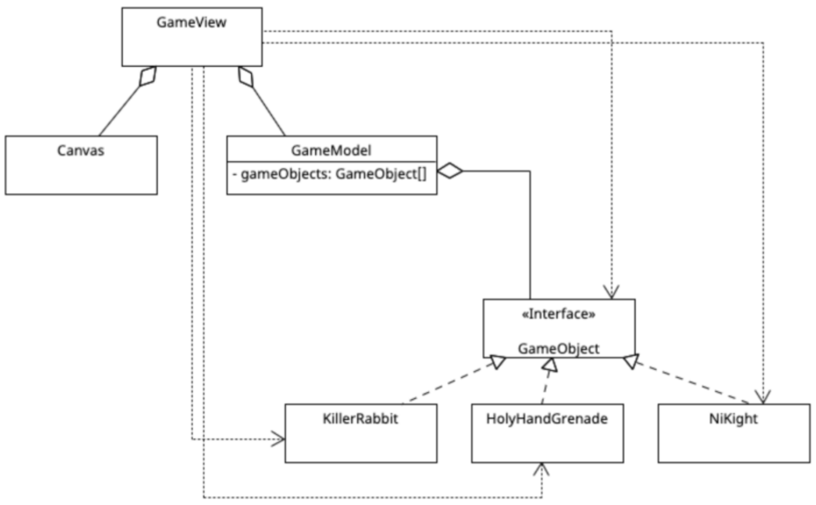
Question 1 – Object Oriented Design 6 Points

Study the code snippet below and answer the following questions.

```
public class GameView {
    private Canvas canvas;
    private GameModel model;

    public GameView(Canvas canvas, GameModel model) {

        ....
    }

    ...
    // This method is not thread-safe, but this is not to be considered a
    // defect in this context.
    public void draw() {
        canvas.clear();  // Clears the canvas before drawing.
        for(GameObject g : model.getGameObjects()) {
            if(g instanceof KillerRabbit) {
                drawRabbit(g);
            } else if(g instanceof HolyHandGrenade) {
                drawHandGrenade(g);
            } else if(g instanceof NiKnight) {
                drawKnight(g);
            } else {
                System.out.println(”No such monster!”);
            }
        }
    }
    ... // Draw-methods for the various types of GameObjects to the view’s canvas.
}

public interface GameObject {
    public Position getPosition();
}

public class GameModel {
    /** Returns all objects in the game. */
    public List<GameObject> getGameObjects() {
        ...
    }
}

public class KillerRabbit implements GameObject { ... }
public class HolyHandGrenade implements GameObject { ... }
public class NiKnight implements GameObject { ... }
```

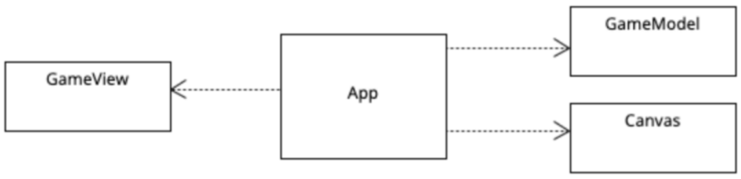
1. Plot the class diagram for the code above where you show all kinds of dependencies among the different classes.

   Solution:



2. If you are required to supply a driver class that uses the GameView class, called App, how should you modify your class diagram? (plot the change).

Solution:



Question 2 – Object Oriented Design 6 Points

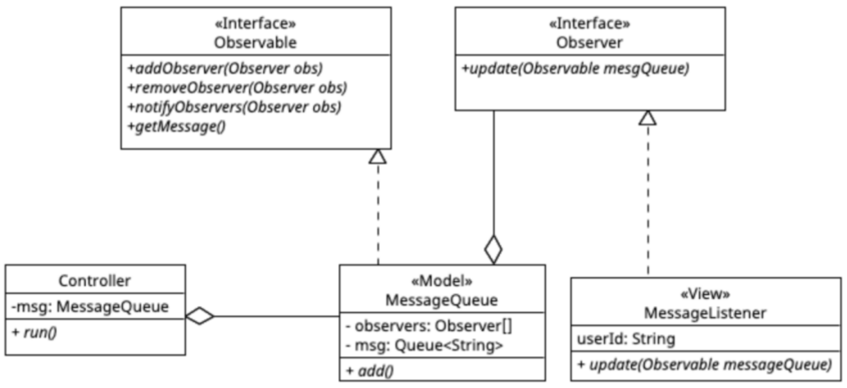


Fig.1: A CLI Chat Application

Consider the class diagram of a CLI Chat application in Fig.1 and answer the following questions:

a) You are required to generate stub classes that conform to Fig.1. (Hint: you are free to use the names of instance variables in your stub classes unless specified).

Solution:

```
public class Controller {
    private MessageQueue msg;
    public void run(){}
}
```

```
public class MessageListener implements Observer {
    private String userId;
    @Override
    public void update(Observable messageQueue) {
    }
}
```

```
public class MessageQueue implements Observable {
    private Set<Observer> observers = new HashSet<>();
    private Queue<String> message = new LinkedList<>();
    @Override
    public void addObserver(Observer obs) {}
    @Override
    public void removeObserver(Observer obs) {}
    public void add(String input) {
    }
    @Override
    public void notifyObservers() {}
    public String getMessage() {}
}

public interface Observer {
    public void update(Observable mesgQueue);
}
```

```
public interface Observable {
    public void addObserver(Observer obs);
    public void removeObserver(Observer obs);
    public void notifyObservers();
    public String getMessage();
}
```

b) Discuss the role of the Observer interface: is it needed? (Hint: If we remove this interface and allow the MessageQueue class to directly reference a set of MessageListener instead of Observer, what will happen?)

Solution: The is used to break the dependency between the Model and the View. Without it, adding new types of observers in the future other than MessageListener would break the code. So, Yes, it is needed.

Question 3 – Coupling and Cohesion 6 Points

The snippets below and answer the following questions:

```
public class View {
    private SQLDatabase sqlDatabase;
    public View()
    {
        sqlDatabase =  new SQLDatabase();
    }
    public void saveEmpId(String empId){
        sqlDatabase.saveEmpId(empId);
    }
}

public class SQLDatabase{
    public void saveEmpId(String empId){
        System.out.println("");
    }
}
```

a) Discuss whether the snippets above feature cohesiveness? If not refactor the code.

Solution: The code is cohesive.

b) Discuss whether the snippets above suffer from tight coupling? If yes, motive your answer by identifying a problem that may appear in the future and refactor your code accordingly.

Solution: The View class is tightly coupled to the SQL database. If in the future, we want to change this to another type of database, we must change the View class. The solution is to create an interface for a generic database, and we reference the interface in the View class.

```
Public interface Database{
    public void saveEmpId(String empId);
}
public class View {
    private Database database;
    public View(Database database)
    {
        This.database = database;
    }
    public void saveEmpId(String empId){
        database.saveEmpId(empId);
    }
}
```

```java
public class SQLDatabase implements Database{


    @Override
    public void saveEmpId(String empId){
        System.out.println("");
    }
}
```

Question 4 – Design Patterns 6 Points

Study the code snippet in the table below: The main class is the driver of this application, which is responsible of creating an iPhone15 instance, attach its charger, and put the iPhone on charge.

```java
public interface IPhone
{
    public void OnCharge();
}
```

```java
public interface Charger
{
    public void charge();
}
```

```java
public class Iphone15 implements
IPhone
{
    Charger charger;
    public Iphone15
                (Charger charger)
    {
        this.charger = charger;
    };

    @Override
    public void OnCharge()
    {
        charger.charge();
    }
}
```

```java
public class Iphone15charger
implements Charger
{
    Iphone15charger ()
    {

    }

    @Override
    public void charge()
    {
        System.out.println
                 ("is charging");
    }
}
```

```java
public class main
{
    public static void main(String args[])
    {
        Iphone15 iphone15 = new Iphone15(new Iphone15charger());
        iphone15.OnCharge();
    }
}
```

1. The problem is that you only have an **iPhone13** charger which is compatible with **iPhone15** in terms of power specification, but they have different connectors. Pick a design pattern that can help you in solving this problem without modifying classes (except for the main class). (Hint: **Iphone13Charger** class is the same as the **Iphone15Charger** class. You are not supposed to implement any code for

connectors. You only need to allow to call the charge method in **Iphone13Charger** when you put your **iPhone15** on charge. You may need to introduce new helping classes).

Solution: we can solve this problem by using an adapter. This means that in the main class, inside the main method, we want to be able to pass an adapter to an iphone13 charger instead of the original Iphone15charger. The adapter must be of type Charger as follows:

```
public class main
{
   public static void main(String args[])
   {
      Iphone15 iphone15 = new Iphone15(new Iphone13To15Adapter());
      iphone15.OnCharge();
   }
}
```

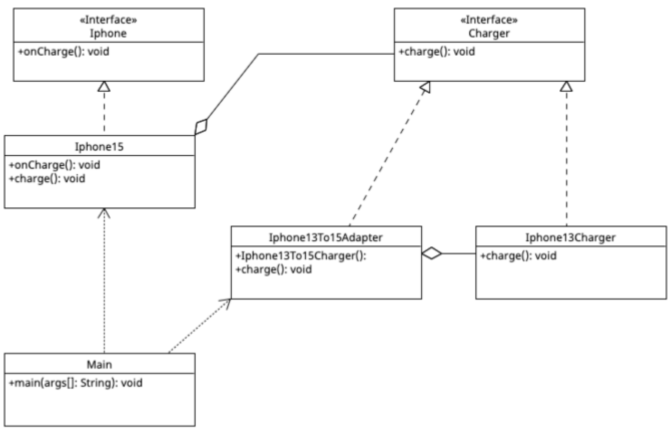
This is possible because an **Iphone15** accepts any object of type **Charger**. Now it remains to create the adapter class.

```
public class Iphone13To15Adapter implements Charger
{
    private Iphone13Charger iphone13Charger;

    Iphone13To15Adapter ()
    {
        iphone13Charger = new Iphone13Charger();
    }

    @Override
    public void charge()
    {
        iphone13Charger.charge();
    }
}
```

2. What design pattern have you used?

Solution: we use the Adapter design pattern.

3. Plot the class diagram of your design.
   Solution:



Question 5 – The Solid Principles 6 Points

Consider the code snippets in Question1 and answer the following questions:

1. Argue whether the **GameView** class violates any of the SOLID principle or not? If yes, refactor the code accordingly.

   Solution: It violates the OCP principle because the **draw()** method is tightly dependent on the different implementations of the **GameObject** interface. If a new implementation is introduced in the future, we need to change the body of this draw method. Thus, we need to break such tight dependencies.

```
public interface GameObject{
   public Position getPosition();
   public void draw(Canvas canvas);
}
```

We remove the specialized draw methods for various game objects (i.e., **drawRabbit(), drawHandGrenade(), drawKnight()**). Instead, we create a **draw()** method in the **GameObject** interface so that it is overridden by all implementations. We also pass the **Canvas** as a parameter to ensure that they all draw inside the same canvas when we execute **draw()** in the **GameView** class. Now the new **draw()** method is simply as follows:

```
public void draw()
{
    canvas.clear();
    for (GameObject g : model.getGameObjects()) {
        g.draw(canvas);

    }
}
```

Notice that **draw()** only depends on the interface **GameObject**.

2. Plot the class diagram for the refactored code only.

Solution: The only difference in the class diagram is that the dependencies from class **GameView** on the implementations of **GameObject** must be now removed. All other details stay the same.

Question 6 – The Solid Principles 6 Points

Consider the code snippet below and answer the following questions:

```java
public interface Animal{
    void swim();
    void fly();
    void walk();
}
```

```java
public class Duck implements Animal {

    @Override
    public void swim() {
        System.out.println("swim");
    }

    @Override
    public void fly() {
        System.out.println("fly");
    }

    @Override
    public void walk() {
    }

}
```

1. What principles does this code violate? Explain and give a scenario for each violation.

   Solution: it violates LSP (the Duck class does not provide an implementation for the overridden walk() method), ISP (the interface is bloated with methods that can be implemented by different objects), and SRP (Animal interface serves many purposes that are not shared by all animals).

2. Refactor this code so that it preserves all SOLID principles.

   Solution: break the Animal interface into 3 interfaces, e.g., WalkingAnimal, flyingAnimal, SwimmingAnimal, and Duck can implement only interfaces that supports as follows:

```java
public interface WalkingAnimal{
    void walk();
}
```

```java
public interface FlyingAnimal{
    void fly();
}
```

```java
public interface SwimmingAnimal{
    void swim();
}
```

```
public class Duck implements FlyingAnimal, SwimmingAnimal {

    @Override
    public void swim() {
        System.out.println("swim");
    }

    @Override
    public void fly() {
        System.out.println("fly");
    }
}
```

PART B – Advanced Questions

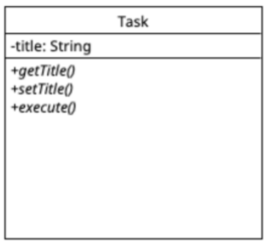
Question 7 – Design Patterns 12 Points

Imagine you are building a project management system that creates tasks for employees to perform. A Task is defined as follows:

- A Task has a title (String).
- Responsibilities are: getTitle(), setTitle(), and execute().

1. You are required to plot the class diagram for the Task following the specifications.

   Solution:



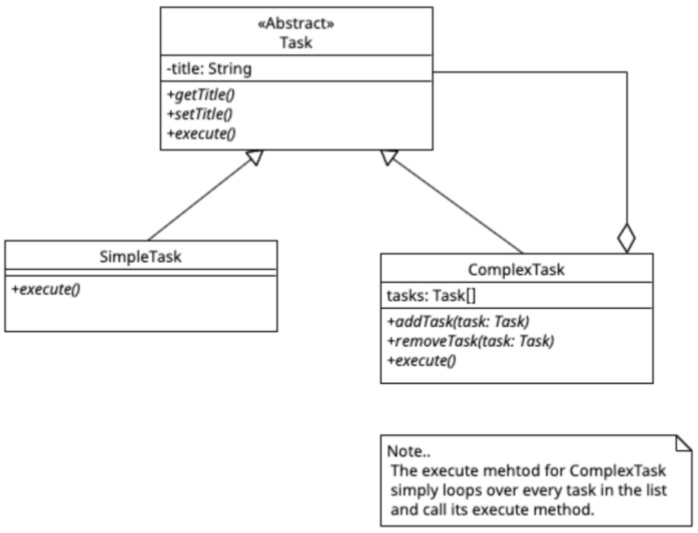
2. Assume that the structure in your company changed, where employees are structured into departments. Now, a task can be a SimpleTask (to be executed by a single employee) with specifications as before or a ComplexTask (to be executed by a department). The latter is a list of subtasks that can be assigned to a department. More precisely, you are required to change your class diagram only for Task (by choosing appropriate design pattern) according to the following specifications:

- A Task has a title and can be either a SimpleTask or a ComplexTask.
- Responsibilities for SimpleTask are: getTitle(), setTitle(), and execute().
- Responsibilities for ComplexTask are: addTask(Task task), removeTask(Task task), execute().
- A ComplexTask can contain a mixture of simple and/or complex subtasks.

- When a ComplexTask is executed, all its subtasks must be executed. The code of execute() for a simple task is to print “executed” while other methods are defined as usual.

(Note: no points if you re-invent the wheel (i.e., without a proper use of a design pattern)).

Solution: This is a typical example of a composite design pattern



3. Generate stub classes according to the change in the previous item.

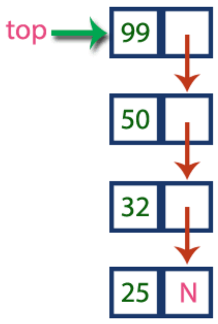
   Solution: the stub classes here are standard as before the only difference is that the execute method of the ComplexTask. Its body should call the bodies of in the task list.

Question 8 – Design by Contract 12 Points

Consider the following code that represents a stack implementation as a LinkedList. The main stack class (**LinkedStack**) implements the interface (**StackInterface**), and thus providing the standard methods supported by a stack as follows:

Name	Usage	Specifications
boolean isEmpty()	Returns true if the stack is empty and false otherwise.	Pre: True Post: stack is not changed
int size()	Returns an integer representing the number of elements in the stack	Pre: True Post: stack is not changed
void push(E item)	Adds an item with type E to the stack.	Pre: True Post: (1) stack is not empty (2) top points to the item (3) invariant still holds
E top()	Returns the item on the top of the stack.	Pre: stack is not empty (otherwise throws exception) Post: stack is not changed
void pop()	Removes the top item on the stack	Pre: Pre: stack is not empty (otherwise throws exception) Post: size is decremented

LinkedStack has two instance variables: (1) top of type **Cell** to refer to the **top** element in the stack. Notice that **Cell** is implemented as an inner class where it has an **item** of type **E** (representing data), and a pointer **next** of type **Cell** pointing to the next Cell (see example figure below with E=int); (2) **size** to store the current size of the stack.



(The specifications are left empty so that you fill them in as part of your solution)

```
public interface StackInterface<E> {
    public boolean isEmpty();

    public int size();

    public void push(E item);

    public E top();

    public void pop();
}
```

```
public class LinkedStack<E> implements StackInterface<E> {
    protected Cell top;
    protected int size;

    public class Cell {
        E item;
        Cell next;

        Cell(E item, Cell next) {
            this.item = item;
            this.next = next;
        }
    }
```

```
public LinkedStack() {
        top = null;
        size = 0;
    }
```

```
protected boolean invariant() {
        //fill invairant
    }
```

```
@Override
    public boolean isEmpty() {
        return this.size==0;
    }
```

```
@Override
    public void pop() {
        top=top.next;
        size--;
    }
```

```
@Override
    public void push(E item) {
        top = new Cell(item, top);
        size++;
    }
```

```
@Override
    public int size() {
        return this.size;
    }
```

1. Study the code above and fill all the specifications of all the **five** methods using the Design-by-Contract approach.

   Solution:
   See the red text in the table above.

2. Use assertions and directly implement preconditions and postconditions inside the code if needed.

```
public LinkedStack() {
        top = null;
        size = 0;
        assert invariant(); // invariant
    }
```

```
@Override
    public void pop() {
        int currentSize= this.size;
        assert !this.isEmpty(); // pre-condition
        top=top.next;
        size--;
        assert this.size==currentSize-1; // post-condition
    }
```

```
@Override
    public void push(E item) {
        top = new Cell(item, top);
        size++;
        assert !this.isEmpty(); // post-condition
        assert this.top() == item; // post-condition
        assert invariant(); // invariant
    }
```

```
@Override
    public E top() {
        assert !this.isEmpty(); // pre-condition
        return top.item;
    }
```

```
protected boolean invariant() {   // not part of this item.
        return (size >= 0) &&
               ((size == 0 && this.top == null)
                        || (size > 0 && this.top != null));
    }
```

```
No change for other methods.
```

3. Fill the method **invariant()** in the table above, which must specify the invariant on the class **LinkedStack**. Note: you need to adjust the specification of the constructor and other methods if required.

   Solution:

```
(size >= 0) &&
              ((size == 0 && this.top == null)
                     || (size > 0 && this.top != null));
```

End of Questions