DAT055– Object Oriented Applications
Re-Exam 2024-August-26

Examiner: Dr. Yehia Abd Alrahman
Phone: 031-772 60 54

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.

- Answers must be given in English.
- Use page numbering on your answer sheets.
- 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

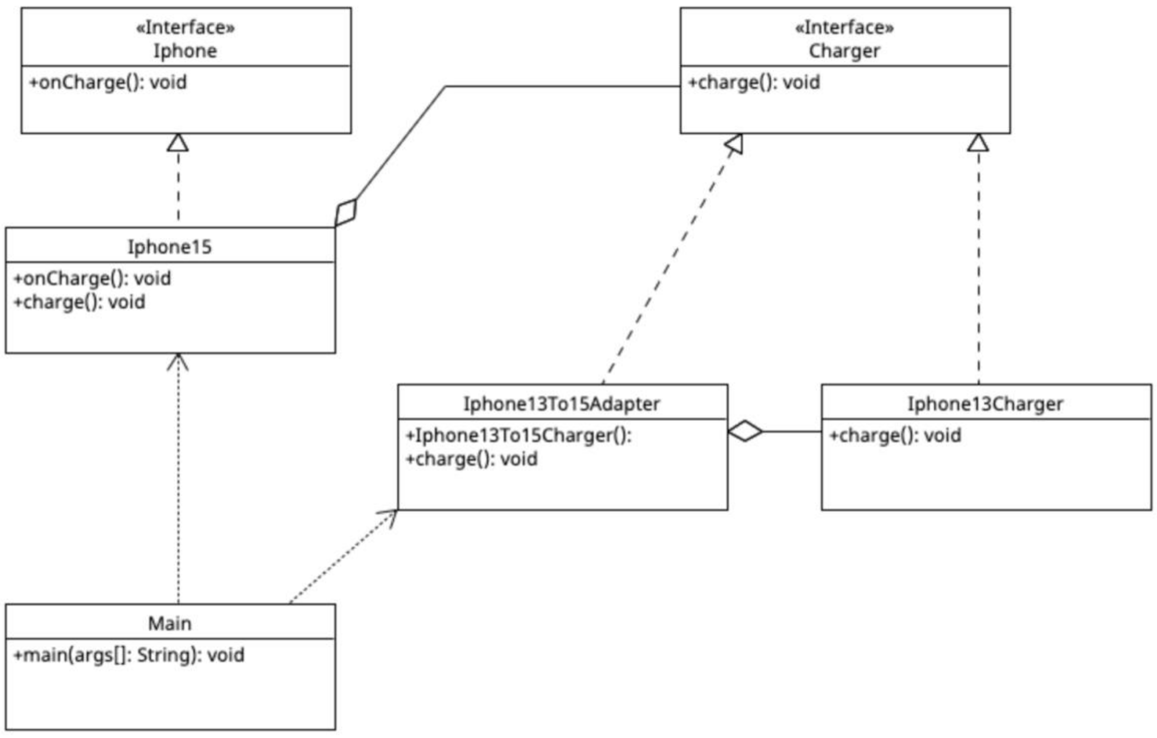
Question 1 – Object Oriented Design 6 Points

Study the code below and answer the following question.

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

public interface Charger
{
    public void charge();
}

public class Iphone13Charger implements Charger
{
    Iphone4sCharger(){};

    public void charge()
    {
        System.out.println("charging with 13 charger");
    }
}

public class Iphone13To15Adapter implements Charger
{
    Iphone13Charger iphone13Charger;

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

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


public class Iphone15 implements IPhone
{
    Charger Adapter;
    public Iphone15(Charger adapter)
    {
        this.Adapter = adapter;
    };

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



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

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

Solution:



Question 2 – Object Oriented Design 6 Points

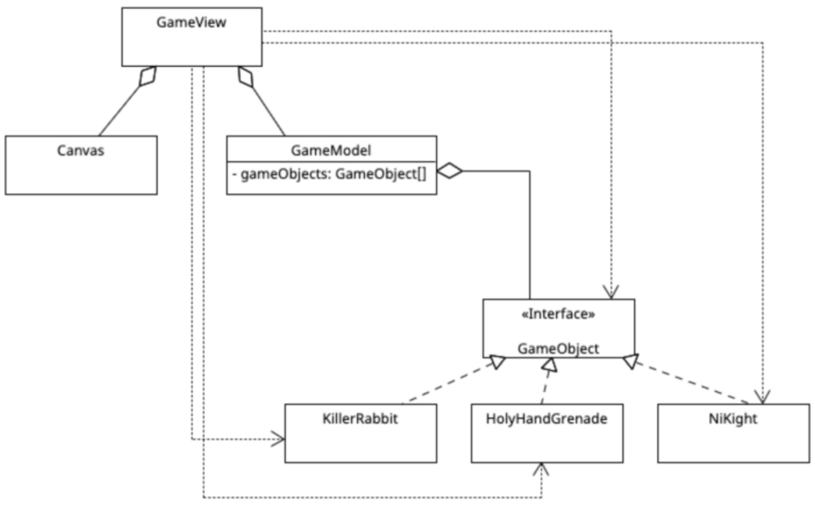


Fig.1: A Game Application

Consider the class diagram of a game application in Fig.1 and answer the following question:

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. Moreover, you are not supposed to implement the weakest relations in terms of coupling).

Solution:

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

    public GameView(Canvas canvas, GameModel model) {
        this.canvas = Objects.requireNonNull(canvas);
        this.model = Objects.requireNonNull(model);
    }
```

    …. // some code about the implementation of the weakest relations (not applicable here)

```
}

public interface GameObject {

}

public class GameModel {
    public List<GameObject> objects;
    public List<GameObject> getGameObjects() {
        ...
    }
}

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

b) Identify the weakest relations in Fig.1 in terms of coupling, and discuss the different alternative implementations in terms of code?

Solution:

The weakest relations are the dependency relations from GameView and the classes KillerRabbit, HolyHandGrenade, NiKnight.

They can be implemented either by creating an object of each class inside a method body of GameView, or by passing an object of any of these classes as a parameter in a method of GameView.

Question 3 – Coupling and Cohesion 6 Points

Consider the snippet below about a class that decides the distinction for different disciplines, and answer the following questions:

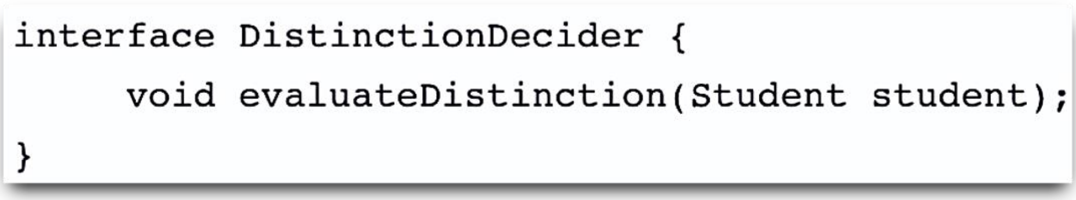
```
class DistinctionDecider {
List<String> science = Arrays.asList("Comp.Sc.","Physics")
List<String> arts = Arrays.asList("History","English");
 public void evaluateDistinction(Student student) {
   if (science.contains(student.department)) {
    if (student.score > 80) {
     System.out.println(student.regNumber+" has received
                        a distinction in science.");
     }
   }
   if (arts.contains(student.department)) {
    if (student.score > 70) {
     System.out.println(student.regNumber+" has received a
                        distinction in arts.");
               }
          }
     }
}
```

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

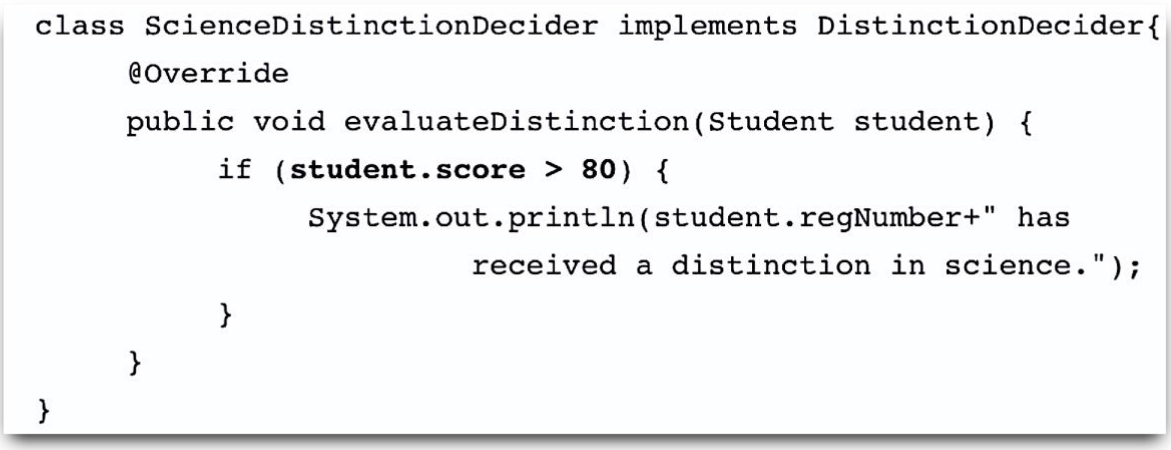
   Solution: the code is cohesive; it serves a single purpose.

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

   Solution: The code suffers from tight coupling on the lists: science and arts. The problem is that if a new discipline is added, the class needs to be modified, and a new list must be added. Moreover, the method of the class needs to be modified. A possible solution would be as follows.



```
interface DistinctionDecider {
      void evaluateDistinction(Student student);
}
```



```
class ScienceDistinctionDecider implements DistinctionDecider{
      @Override
      public void evaluateDistinction(Student student) {
            if (student.score > 80) {
                  System.out.println(student.regNumber+" has
                              received a distinction in science.");
            }
      }
}
```

The ArtDistinctionDecider is like the science one.

c) One argues that future updates to this code may incur an increase in its cyclomatic complexity, do you agree? If yes, how?

Solution: Yes, I agree, the cyclomatic complexity is a measure to the number of paths in a program. If we look at the evaluation distinction method, we can see that currently it has many paths due to multiple if-statements. If new disciplines are added then we would need to add more if-statements, and thus increase the cyclomatic complexity.

Question 4 – Model-View-Controller 6 Points

Consider an MVC CLI application according to the following requirements: The user types a message and aims to send it to all potential listeners (use **MessageListener** as class name). The input of the CLI is read by listening to the system standard input. All messages are stored in an object of the class **MessageQueue**. When the user types the message and hits Enter, the message is stored in the queue and is autonomously propagated to all listeners. Listeners pick up messages and display them on the system standard output.

1. Identify the Model , the View, and the Controller in this application.

   Solution:

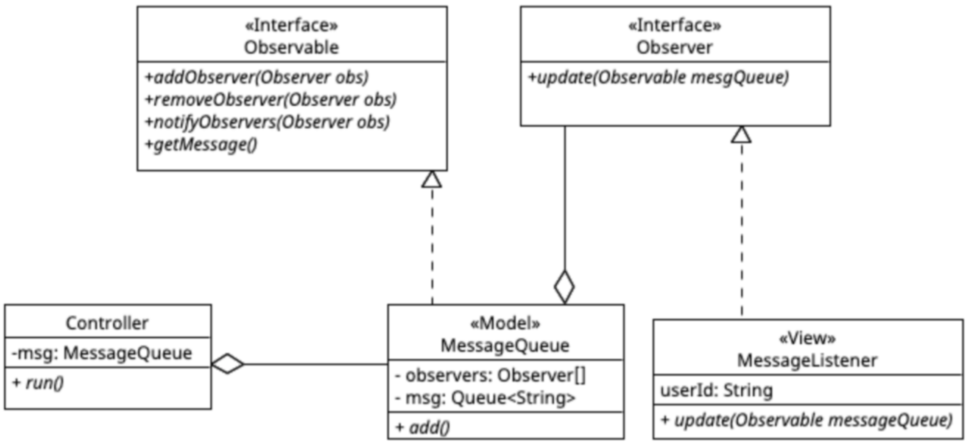
   Model: MessageQueue
   View: MessageListener
   Controller: a class that run the app. That is, it observe the standard input, query the model and mediate the interaction with MessageListners.

2. To permit autonomous delivery of the message for all listeners without asking, what design pattern is appropriate to realize this MVC architecture?

   Solution: Observer design pattern

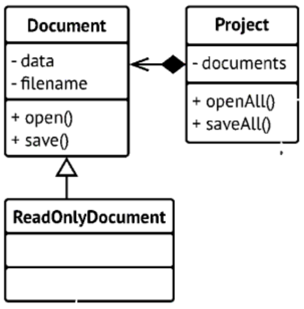
3. Plot the class diagram of your concrete design.

   Solution:



Question 5 – The Solid Principles 6 Points

Consider the class diagram below and answer the following question:



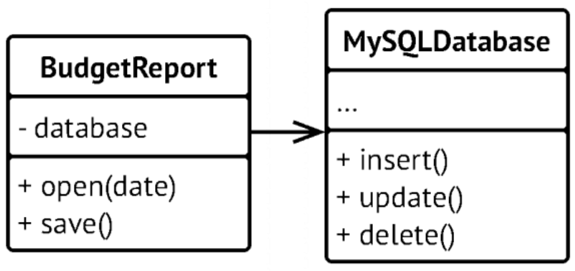
Argue whether the design violates any of the SOLID principle? If yes, explain why, which principles (give an example if something does not work), and propose a fix.

Solution: This design violates Liskov Substitution Principe. Note that the class ReadOnlyDocument cannot support the save() method that inherits from the Document class. In other words, ReadOnlyDocument does not satisfies the same properties that Document satisfies, and will sometimes fail when a save() method is called.

There are many ways to fix this design. One possible way is to change ReadOnlyDocument to WriteDocument class. Another way is to completely remove the ReadOnlyDocument class.

Question 6 – The Solid Principles 6 Points

Consider the class diagram below for a budget report that manipulates a database. Answer the following questions:

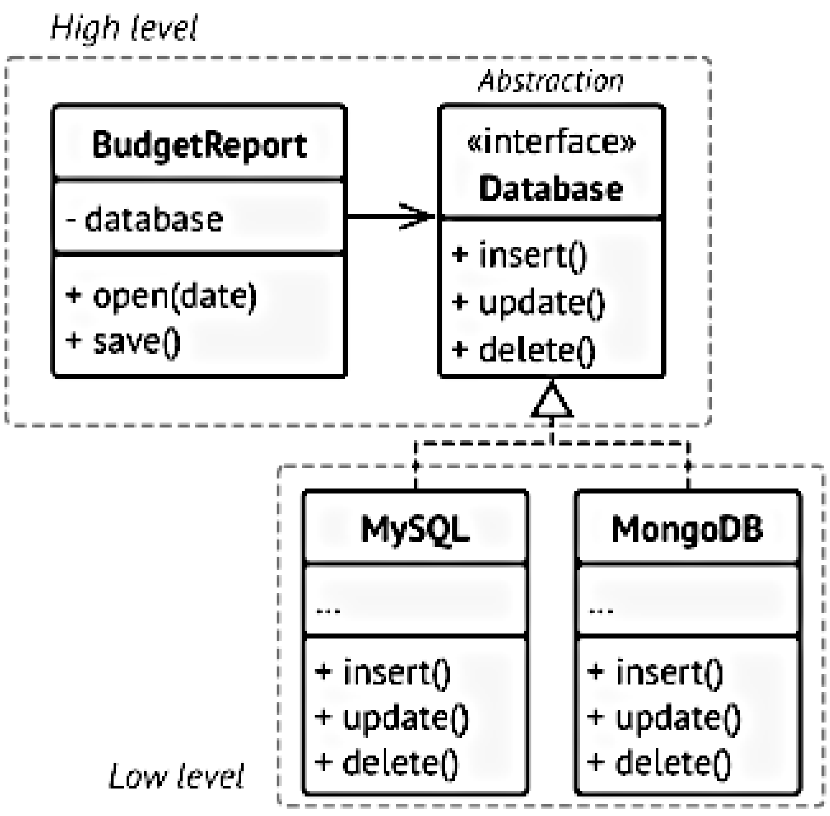


1. What principles does this code violate? Explain and give a scenario (also future update) for such violation.

   Solution: this design violates the Dependency Inverstion Principle. The BudgetReport is tightly coupled to MySQLDatabase. This means that MySQLDatabase should be developed before BudgetReport. What if we require a different database in the future?

2. Refactor this design so that it preserves the SOLID principles.

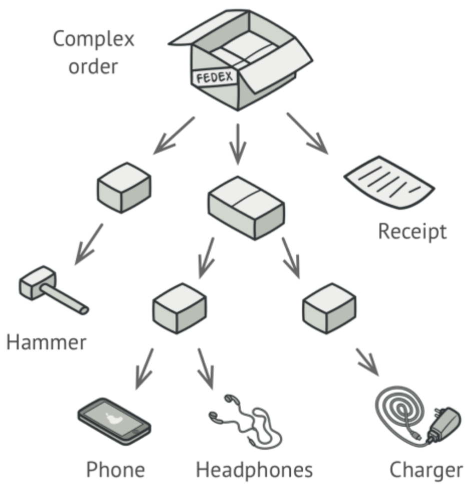
   Solution: A possible solution can be found below:



PART B – Advanced Questions

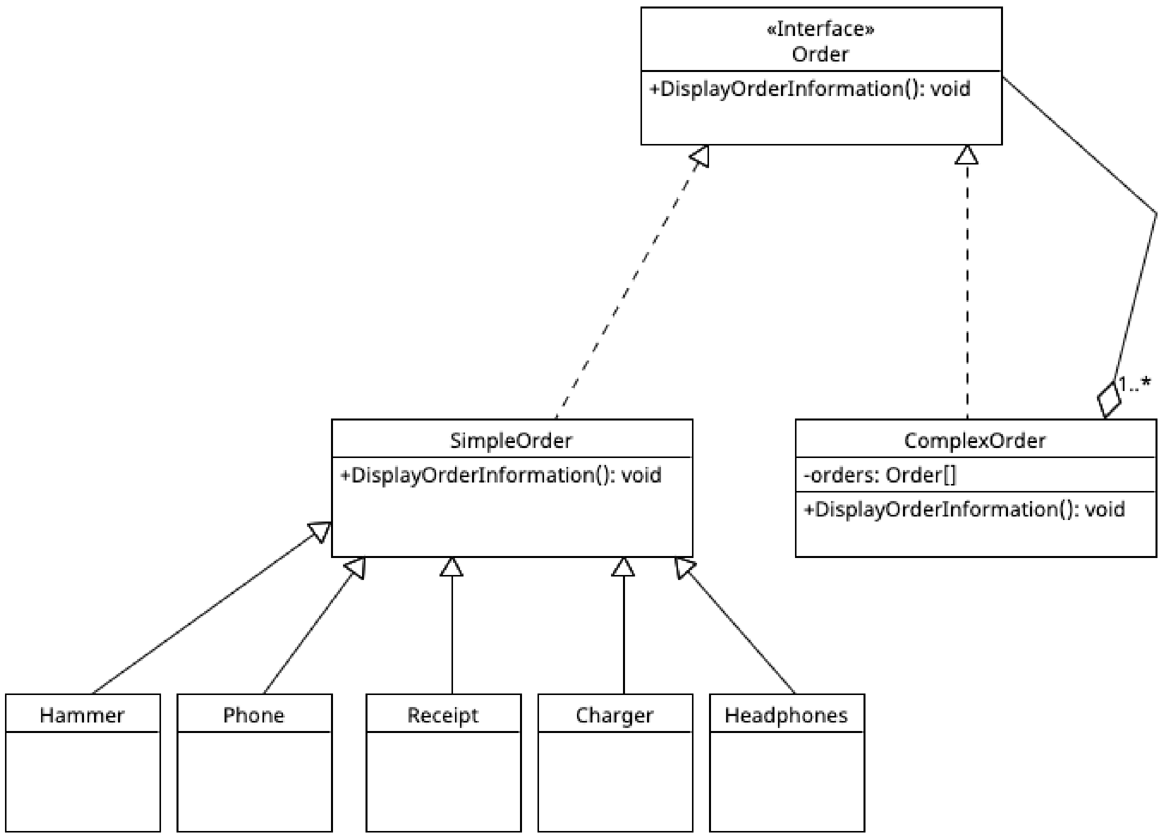
Question 7 – Design Patterns 12 Points

Imagine you are building an application to represent an order as shown in the figure below. An order can be a complex package that either contains a simple order such as a Hammer or a complex order such as a smaller package. Assume that any order has a single operation named **displayOrderInfomration().**



1. Pick an appropriate design pattern to implement this application and plot the class diagram.

   Solution: Composite Design Pattern


«Interface»
Order
+DisplayOrderInformation(): void
1..*
SimpleOrder
+DisplayOrderInformation(): void
ComplexOrder
-orders: Order[]
+DisplayOrderInformation(): void
Hammer
Phone
Receipt
Charger
Headphones

2. Show how the **displayOrderInfomration()** method can be implemented for both simple and complex orders.

Solution:

For simple order: it can be simple print statement, e.g., System.Out.Println("some message");

For complex order: it should loop over all its children as follows:

```
for (Order order : orders)
{
      order. displayOrderInfomration();
}
```

Note that the for loop is looping over the interface Order. This is because a complex order can compose either simple or complex orders.

Question 8 – Design by Contract 12 Points

Consider the following code that represents a java implementation of a size-limited integer set using array. The set does not accept duplicates, and has a limited size as defined below. Study the code below and answer the following questions.

```
public class LimitedIntSet {
    Private int capacity;    // maximum number of elements
    Private int[] arr;       // stores the elements of the set
    Private int size;        //current number of elements


    public LimitedIntSet (int limit){
        assert limit > 0
        capacity = limit;
        arr = new int[limit];
        size = 0;
        assert size == 0
        assert capacity == limit
        assert invariant()
    }
```

```
public boolean contain(int element)
{
  assert invariant()
  // some code
    }
```

```
public boolean invariant() {
     0 <= size < capacity &&
    capacity == arr.length()
    }
```

```
 public boolean add(int element)  {
  assert invariant()
  assert !this.contain(element)
  // some code

 }
```

```
public int find(int element) {
 /*  returns index of element if
   in the set, otherwise -1 */
 assert invariant()
 assert this.contain(element)
 // some code
}
```

```
}
```

1. Use assertions and directly implement preconditions and postconditions inside the code of the methods, and the constructor if needed.

   See the red part above.

2. Fill the method **invariant()** in the table above, which must specify the invariant on the class `LimitedIntSet`.
3.

   See the red part above.

   Note: If you use the math implementation or a code implementation both are acceptable. Here, I use the math representation for the sake of brevity.

End of Questions