DAT055– Object Oriented Applications
Exam date 18th MAR 2025

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

```
                    I N V O I C E

Sam's Small Appliances
100 Main Street
Anytown, CA 98765

Description                    Price  Qty  Total
Toaster                        29.95   3   89.85
Hair dryer                     24.95   1   24.95
Car vacuum                     19.99   2   39.98

AMOUNT DUE: $154.78
```

Fig.1: Invoice Form

Question 1 – Object Oriented Design 6 Points

Use the OOD approach to design a program to print out an invoice as in Fig.1 above. The program simply prints the billing address, all line items, and the amount due. Each line item contains the description and unit price of a product, the quantity ordered, and the total price.

Answer the following questions following the OO approach:

1. Given the description above, discover the names of the required classes in your application. (Note: some class names might not exist in the description, but maybe inferred)

Ans: Invoice, Address, LineItem, Product

2. Provide (final) CRC cards for each class you discover showing its responsibilities and collaborators.

Ans:

Invoice	
format the invoice	Address
add a product and quantity	LineItem
	Product

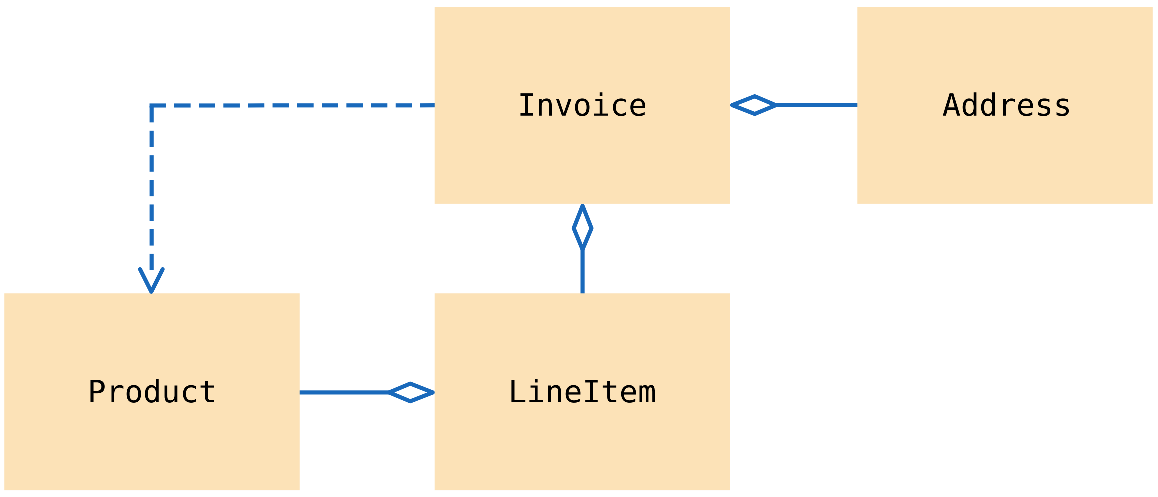
Product	
get description	
get unit price	

LineItem	
format the item	Product
get total price	

Address	
format the address	

3. Plot the class diagram for your design where you show all kinds of dependencies among the different classes.

Ans:



Question 2 – Object Oriented Design 6 Points

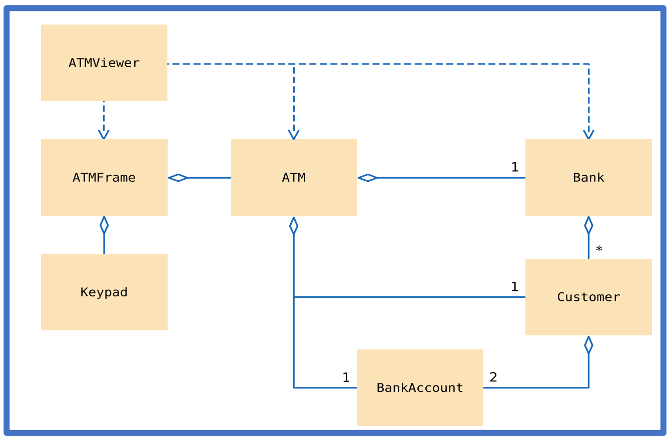


Fig.2: ATM Application

Consider the class diagram in Fig. 2 and answer the following questions:

a) The **ATMViewer** class is the driver of the ATM application that creates **ATM**, **Bank**, and **ATMFrame** objects and relates them according to the class diagram. You are required to generate stub classes that conform to Fig. 2. (Hint: you are free to use the names of instance variables in your stub classes)

Ans:

```
public class Customer
{
   private BankAccount checkingAccount;
   private BankAccount savingsAccount;
}

public class BankAccount
{
}
```

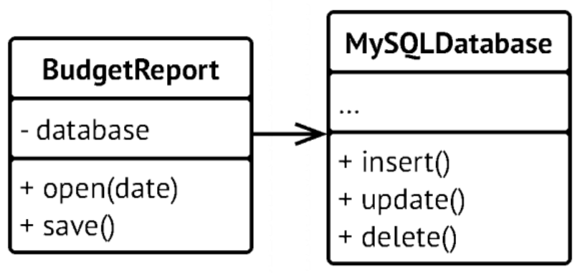
```
public class KeyPad
{
}

public class Bank
{
   private List<Customer> customers;
}

public class ATM
{
   private Customer currentCustomer;
   private BankAccount currentAccount;
   private Bank theBank;
}

public class ATMFrame
{
   private ATM theATM;
   private KeyPad pad;
}

public class ATMViewer
{
    Bank theBank = new Bank();
    ATM  theATM = new ATM(theBank);
    ATMFrame frame = new ATMFrame(theATM);

}
```

b) Consider the relation between the classes **Bank** and **Customer**, explain this relation and discuss if it is conceptually appropriate (if not, suggest an alternative while preserving the actual generated code).

Ans: Customer can be changed from aggregation to association. The reason is that a bank does not really aggregate clients, but rather clients can be associated to different banks. That is why association is more appropriate.

Question 3 – Coupling and Cohesion 6 Points

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

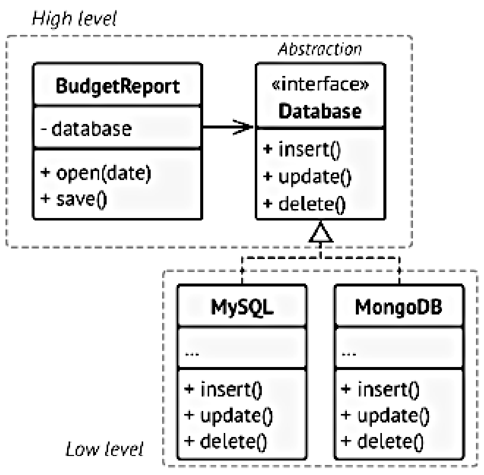


a) Discuss whether the classes above feature cohesiveness? If not refactor the design.

Ans: it is cohesive.

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

Ans: there is a tight coupling between the BudgetReport and the implementation MySQLDatabase. Class must only depend on abstraction. If we change the type of database in the future the BudgetReport needs to be changed as well. It can be refactored as follows:



Question 4 – Model-View-Controller 6 Points

Consider an MVC CLI application according to the following requirements: The user types a single line 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.

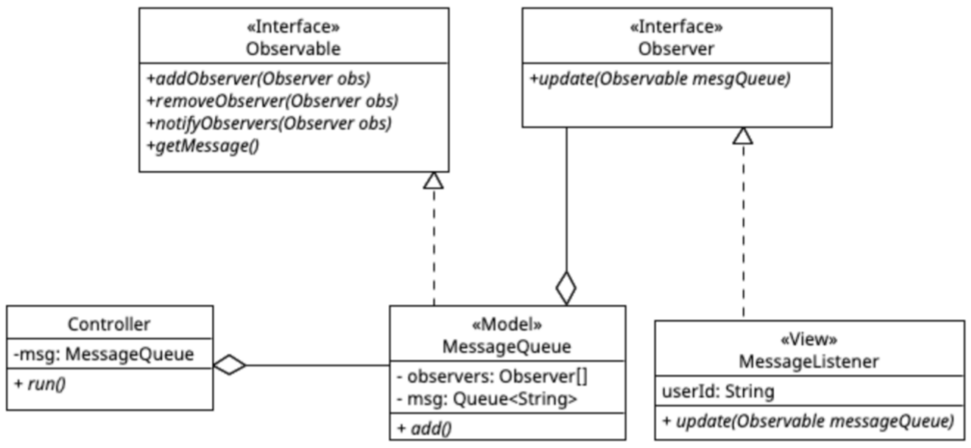
   Model: MessageQueue
   View: MessageListener
   Controller: a class that run the app. That is, it observes the standard input, query the model and mediate the interaction with MessageListners.

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

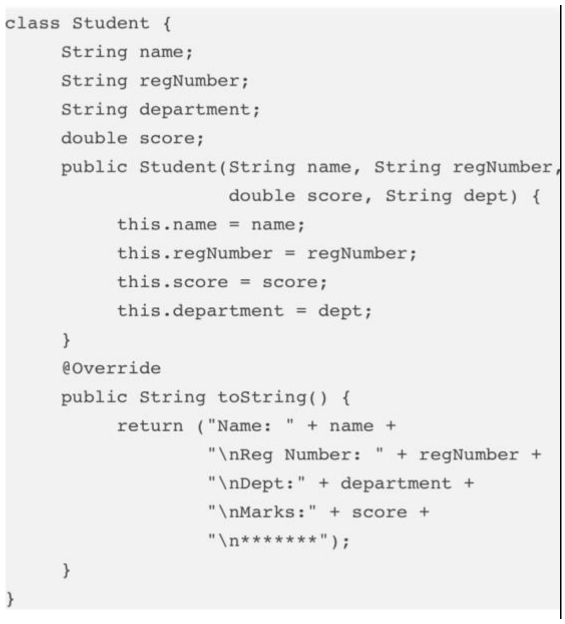
3. Plot the class diagram of your concrete design.

Solution:

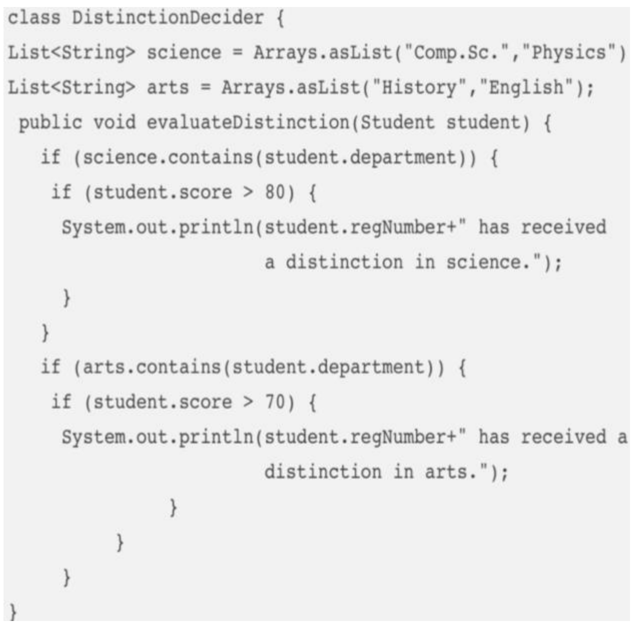


Question 5 – The Solid Principles 6 Points

As shown in the code snippets below, we consider a group of students taking an examination. We need to store the name, registration number, mark, and department (currently Science or Arts). We need to display the result of a student and evaluate their distinction. In Science department (e.g., Physics, CS), distinction is granted if mark is greater than 80 while in Arts (e.g., History, English), distinction is granted if mark is greater than 70. Observe the code and answer the following questions:



```
class Student {
     String name;
     String regNumber;
     String department;
     double score;
     public Student(String name, String regNumber,
                    double score, String dept) {
          this.name = name;
          this.regNumber = regNumber;
          this.score = score;
          this.department = dept;
     }
     @Override
     public String toString() {
          return ("Name: " + name +
                  "\nReg Number: " + regNumber +
                  "\nDept:" + department +
                  "\nMarks:" + score +
                  "\n*******");
     }
}
```



```
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.");
              }
          }
     }
}
```

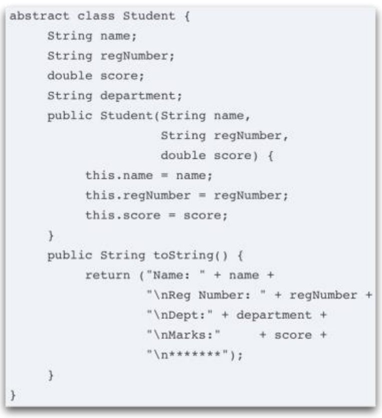
```
class Client {
     public static void main(String[] args) {
     List<Student> enrolledStudents = enrollStudents();
     // Display all results.
     System.out.println("===Results:===");
     for(Student student:enrolledStudents){
         System.out.println(student);
     }
     System.out.println("===Distinctions:===");
     DistinctionDecider distinctionDecider = new
     DistinctionDecider();
     // Evaluate distinctions.
     for(Student student:enrolledStudents){
       distinctionDecider.evaluateDistinction(student);
     }
}
```

1. Consider the Client class (the driver of the application) and check whether the code above satisfies the SRP and OCP principles. What about its cyclomatic complexity?

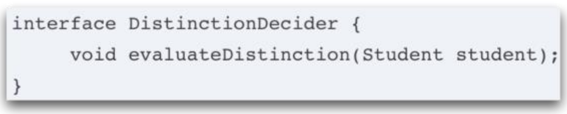
Ans: SRP is preserved, but OCP is not preserved. If a new department is introduced then the distinction decider class must be changed. This would also affect the cyclomatic complexity of this class as well. A new if statement would be added per department which would increase the cyclomatic complexity.

2. If the code violates any of the principles, you must refactor the code so that it satisfies both principles. Note, you can either refactor the code or plot the class diagram of the refactored code (Your choice).

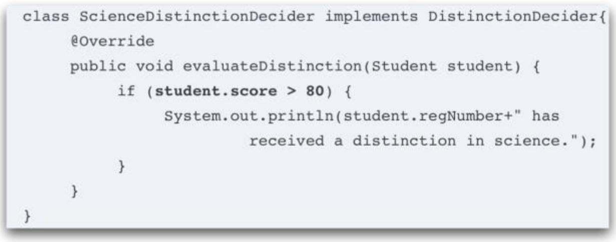
Ans: See below (a similar class of ArtsStudent is also required for ScienceStudent. ArtDistinctionDecider is similar to ScienceDistinctionDecider)



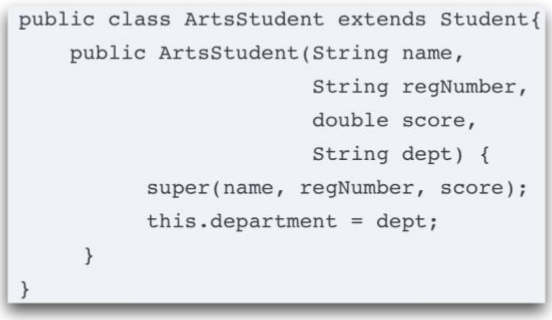
```
abstract class Student {
    String name;
    String regNumber;
    double score;
    String department;
    public Student(String name,
                   String regNumber,
                   double score) {
        this.name = name;
        this.regNumber = regNumber;
        this.score = score;
    }
    public String toString() {
        return ("Name: " + name +
                "\nReg Number: " + regNumber +
                "\nDept:" + department +
                "\nMarks:"     + score +
                "\n*******");
    }
}
```



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



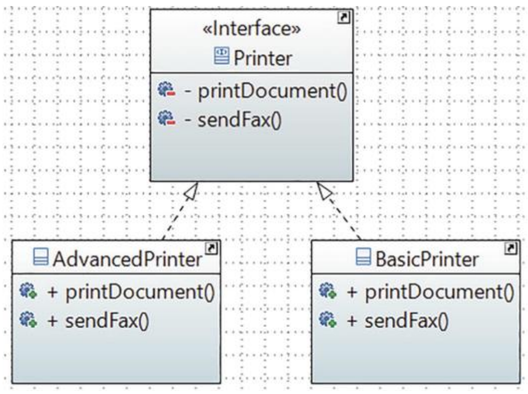
```
class ScienceDistinctionDecider implements DistinctionDecider{
    @Override
    public void evaluateDistinction(Student student) {
        if (student.score > 80) {
            System.out.println(student.regNumber+" has
                    received a distinction in science.");
        }
    }
}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



```
public class ArtsStudent extends Student{
    public ArtsStudent(String name,
                       String regNumber,
                       double score,
                       String dept) {
        super(name, regNumber, score);
        this.department = dept;
    }
}
```

Question 6 – The Solid Principles   6 Points

Consider the following class diagram and answer the following questions. Note that the BasicPrinter class can only print a document, and thus throws an exception when sendFax() is called.



1. What principles does this class diagram violate? Explain.

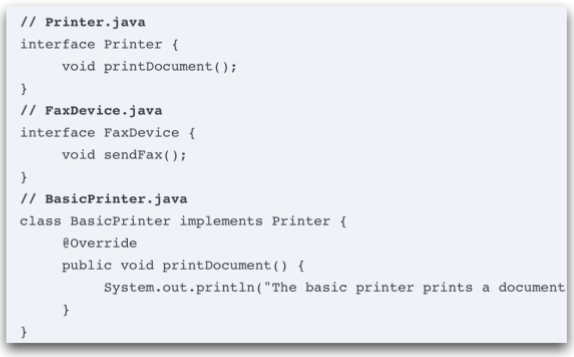
Ans: ISP because the interface is bloated. SRP because a printer does not send a fax. LSP because we cannot use the interface Printer in place of a BasicPrinter. This is because sendFax() throws an exception if called on BasicPrinter.

2. When I write this code:

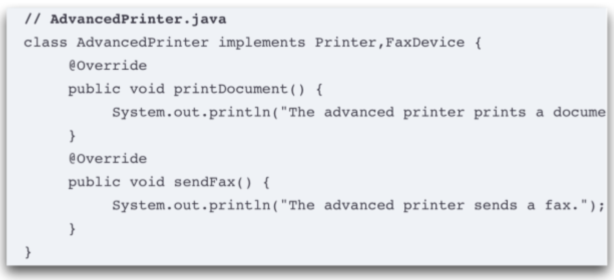
```
    Printer p = new BasicPrinter();
    p.sendFax();
```

I get the following error `Exception in thread "main" java.lang.UnsupportedOperationException`
How should I refactor the design so that this above code does not through an exception?
(Note: providing an empty implementation for sendFax() for BasicPrinter is wrong)

Ans: This is because the code does not support LSP and thus we need to refactor as follows:



```java
// Printer.java
interface Printer {
     void printDocument();
}
// FaxDevice.java
interface FaxDevice {
     void sendFax();
}
// BasicPrinter.java
class BasicPrinter implements Printer {
     @Override
     public void printDocument() {
          System.out.println("The basic printer prints a document
     }
}
```

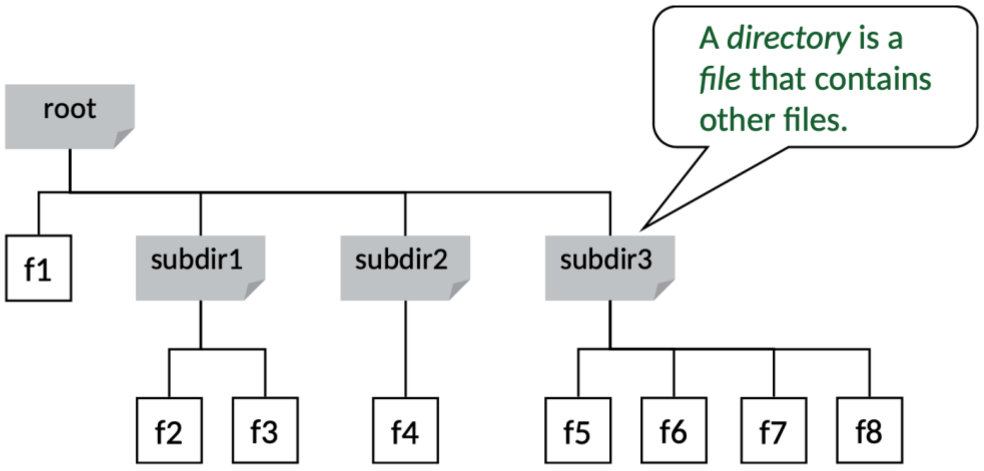


```java
// AdvancedPrinter.java
class AdvancedPrinter implements Printer,FaxDevice {
     @Override
     public void printDocument() {
          System.out.println("The advanced printer prints a docume
     }
     @Override
     public void sendFax() {
          System.out.println("The advanced printer sends a fax.");
     }
}
```

PART B – Advanced Questions

Question 7 – Design Patterns 12 Points

Consider the following representation of a file system. A directory can contain either a simple file (e.g., f1) or a mix of simple files and subdirectories:



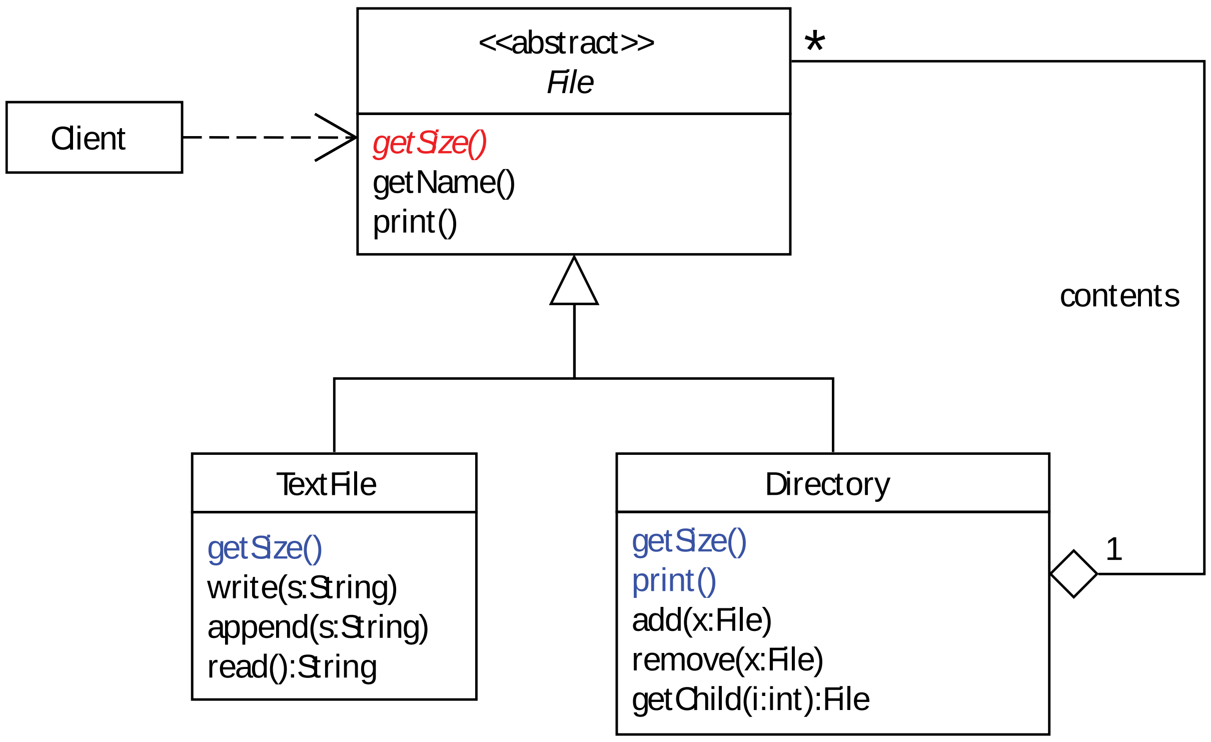
A file has a **size**, a **name** and must support the following methods: **getSize()** to return its size, **getName()** to return its name, and **print()** to print out its size and name. A directory must also be able to store a set of files and subdirectories, and in addition to the properties and methods of a file, it must supply methods to add and remove either a file or a subdirectory. Moreover, when a directory calls its print method, it should print out the details of each file or directory it contains. Note: you are not required to write any code for this question, just the class diagram with the mentioned details. Answer the following:

1. Pick an appropriate design pattern to represent this file system.

Ans: Composite Design Pattern

2. Plot the class diagram.

Ans:



Question 8 – Design by Contract 12 Points

Consider the following code that represents a java implementation of a forgetful queue that is implemented as a LinkedList. The queue has a limited capacity and follows the standard FIFO principle (First-in-First-out). However, if we try to add a new item to the queue when it is full, it forgets the first element and adds the new element as a last element (see enqueue method below). Study the code below and answer the following questions.

```java
public class ForgetfulQueue<T> {
   private final LinkedList<T> items;
   private final int capacity;

   public ForgetfulQueue(int capacity) {
      this.capacity = capacity;
      this.items = new LinkedList<>();
   }

   public void enqueue(T e) {
      if (this.items.size() == this.capacity) {
         this.items.removeFirst();
      }
      this.items.addLast(e);
   }

   public int size() {
      return this.items.size();
   }

   public T dequeue() {
      if (this.items.isEmpty()) {
         throw new NoSuchElementException(”Queue is empty.”);
      }
      return this.items.removeFirst();
   }
```

1. Use Java assertions and directly implement preconditions and postconditions for all the methods of the **ForgetfulQueue** class.

```
public class ForgetfulQueue<T> {
    Private final int capacity;
    Private final LinkedList<T> items;


    public ForgetfulQueue (int capacity){
        assert capacity > 0
        this.capacity = capacity;
        items = new LinkedList<>();
        assert items.size() == 0
        assert this.capacity == capacity
        assert invariant()
    }
```

```
public int size() {
 int size = items.size()
  return items.size();
  assert size == items.size()
    }
```

```
public boolean invariant() {
     return 0 <= capacity &&
    capacity == items.size();
    }
```

```
public void enqueue(T e)  {
  int size= items.size();

  // the code

assert (size == capacity) implies
items.size()== size

assert (size < capacity) implies
items.size()== size + 1


 }
```

```
public T dequeue() {
 /*  returns index of element if
   in the set, otherwise -1 */
 assert !items.isEmpty()

 // the code

Assert items.size()== size - 1
}
```

```
}
```

2. Supply an appropriate class invariant for **ForgetfulQueue.**

Ans: See the table above

End of Questions