Lab 6

Student Name		Student CSUSM ID	Contribution percentage
1	EJ Lilagan	lilag002	25%
2	Gabriela Rojas	rojas124	25%
3	Neo Argatides	argat001	25%
4	Benno Wiedner	wiedn002	25%

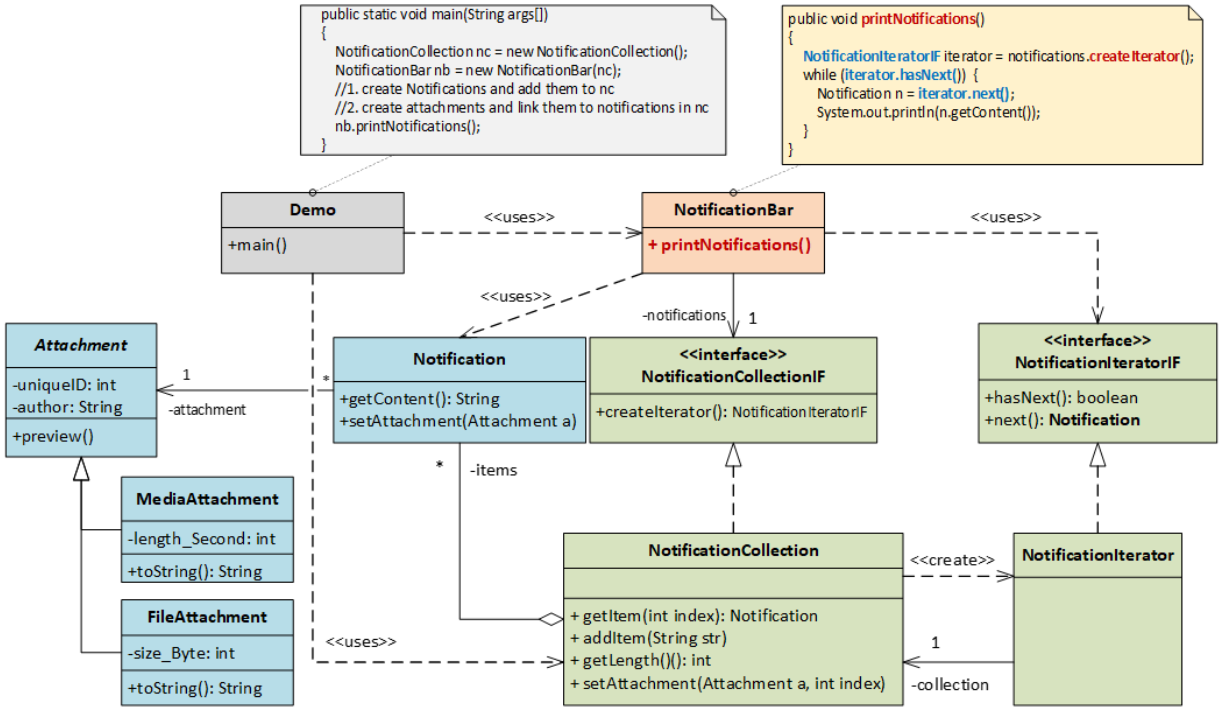
Grading Rubrics (for instructor only):

Criteria	1. Beginning	2. Developing	3. Proficient	4. Exemplary
Modeling	0-14	15-19	20-24	25-30
Program: functionality ***correctness***	0-9	10-14	15-19	20
Program: functionality ***Behavior Testing***	0-9	10-14	15-19	20
Program: quality -> ***Readability***	0-2	3-5	6-9	10
Program: quality -> ***Modularity***	0-2	3-5	6-9	10
Program: quality -> ***Simplicity***	0-2	3-5	6-9	10
Total Grade (100)				

SE 471 Software Architecture

Problems:

A system design in UML class diagram is shown below, where multiple notifications shown in a NotificationBar may share the same attached files or media objects. A client (Demo) may take two phases to initiate/use a NotificationCollection. In the first phase, the client may request the NotificationCollection 10 times, each time to add/create one Notification (with different string content). No attachment is associated with a Notification when it is created. In the second phase, the client may request the NotificationCollection to link the 1st, 2nd, and 3rd Notification objects to the same MediaAttachment object, and link 5th, 9th Notification objects to the same FileAttachment object, etc.



1. What design pattern is used for the classes (interfaces) in green? [10 points] (Module 8-1)

   Iterator design pattern. Given the green pattern gave away the name itself as indicating the word Iterator in the design pattern. As demonstrated in the notes, provide an example using Notification. Iterator is a collection independent of collection implementation, which makes it unique to consider the java classes that are involved to accumulate the necessary items, notifications, and collections used.

2. What design pattern is used for the classes in blue? [10 points] (Module 8-2)

Flyweight design pattern. As the objects to be shared are big enough, we know it utilizes the Flyweight pattern since it does not have to repeatedly use the same information inside each message object. This design pattern will minimize memory usage given that the usage of objects of a large number is a simple repeated representation that would use an unacceptable amount of memory.

3. Implement the design in Java. You may add more attributes or operations to a class if needed. Make sure all the relations are appropriately mapped into code. [80 points]

   ***Note*: Initially added folders to create files/media to be attached to the program (FileAttachmentFolder & MediaAttachmentFolder), but was not implemented since it was not mandatory to import files/media to the program.**

Screenshots for Lab 6

```
----------Attachment Process below----------
File Attachment: Fan_Syllabus.txt
Unique ID: 1001
Byte size: 100
Notification count 1: Pass

File Attachment: Zheng_Syllabus.txt
Unique ID: 1002
Byte size: 200
Notification count 2: Pass

Media Attachment: Heart.txt
Unique ID: 1003
Seconds: 1000
Notification count 3: Pass

Media Attachment: Turkey.txt
Unique ID: 1004
Seconds: 2300
Notification count 4: Pass

Media Attachment: Wolf.txt
Unique ID: 1005
Seconds: 3000
Notification count 5: Pass

Notification count 6: Pass
[Available to add attachment]

Notification count 7: Pass
[Available to add attachment]

Notification count 8: Pass
[Available to add attachment]

Notification count 9: Pass
[Available to add attachment]

Media Attachment: Wolfie
Unique ID: 1010
Seconds: 8000
Notification count 10: Pass

PS C:\Users\elaet\OneDrive\Desktop\Elaeth's Files\Spri
ng 2023\SE 471\Labs\Lab-6-Code>
```

Code.Blue.Attachment.java

```java
package Code.Blue;

public abstract class Attachment {
    /*
     * @brief variables that are used
     * @note specifies the name of the author
     * @note uniquely identify the attachment
     */
    private String author;
    private int uniqueID;
    /*
     * @brief Constructor
     * @param a : String for author
     * @param u : int for uniqueID
     */
    public Attachment(String a, int u){
        this.author = a;
        this.uniqueID = u;
    }
    /*
     * @brief get unique ID in attachment class
     * @return int, uniqueID
     * @note added function
     */
    public int getUniqueID(){
        return this.uniqueID;
    }
    /*
     * @brief get author in attachment class
     * @return String, author
     * @note added function
     */
    public String getAuthor(){
        return this.author;
    }
    /*
     * @brief Creating an output statement to display both the author and
uniqueID
     * @note made as void as mentioned in the uml
     */
    public void preview(){
        System.out.println("Attachment: " + author + "\nUnique ID: "+
uniqueID +"\n");
    }
}
```

Code.Blue.FileAttachment.java

```java
package Code.Blue;

/*
 * @brief FileAttachment class
 *
 * @note FileAttachment extends Attachment since the relationship is an
open arrow
 * @note size_Byte
 */
public class FileAttachment extends Attachment{
    private int size_Byte;
    /*
     * @brief constructor
     * @param 's1' for author initializations
     * @param 'id' for uniqueID initializations
     * @param 'size' for size_Byte initializations
     * @note used super to obtain attachment constructor
     */
    public FileAttachment(String s1, int id, int size){
        /*this.author = s1;
        this.uniqueID = id;*/
        super(s1, id);
        this.size_Byte = size;
    }
    /*
     * @brief translate size_Byte (int) into a String
     * @return string of byte_size
     */
    public String toString(){
        String byte_size = String.valueOf(size_Byte);
        //return String.format(preview() + " Byte Size: " + byte_size);
        return String.format("Byte size: " + byte_size);
    }
    /*
     * @brief output the necessary information of the fileAttachment
     * consisting author, uniqueID, and size_Byte
     * @note added function
     */
    public void preview(){
        System.out.println("File Attachment: " + super.getAuthor() +
        "\nUnique ID: " + super.getUniqueID() + "\n"
        + toString());
    }
}
```

Code.Blue.MediaAttachment.java

```java
package Code.Blue;

public class MediaAttachment extends Attachment{
    private int length_Second;
    /*
     * @brief Constructor
     * @param 'a' for author initializations
     * @param 'u' for uniqueID initializations
     * @param 'l' for length_Second initializations
     * @note used super to obtain attachment constructor
     */
    public MediaAttachment(String a, int u, int l){
        /*this.author = a;
        this.uniqueID = u;*/
        super(a, u);
        this.length_Second = l;
    }
    /*
     * @brief translate length_Second (int) to a String
     * @return String seconds
     */
    public String toString(){
        String seconds = String.valueOf(length_Second);
        return String.format("Seconds: " + seconds);
    }
    /*
     * @brief output the necessary information of the mediaAttachment
     * consisting author, uniqueID, and length_Second
     * @note added function
     */
    public void preview(){
        System.out.println("Media Attachment: " + super.getAuthor() +
        "\nUnique ID: " + super.getUniqueID() + "\n"
        + toString());
    }
}
```

Code.Blue.Notification.java

```
package Code.Blue;

public class Notification {
    /*
     * @brief variables used for optimization
     * @note set an attachment variable for attachments
     * @note set a message variable to set notifications
     */
    private Attachment attachment;
    private String message;
    /*
     * @brief constructor
     * @param m : String of message
     */
    public Notification(String m){
        this.message = m;
    }

    /*
     * @brief get preview of the content of the attachment
     * @return whole line of string
     *
     * @note initially used ternary to use preview()
     * @note made two conditions to indicate the attachment process,
     * a flaw is having a void preview(), which can be satisfied for
ternary
     * @note ended up doing if statement to show the process
     */
    public String getContent(){

        if(attachment == null){
            return String.format(message + "\n[Available to add
attachment]\n");
        }else{
            attachment.preview();
            return String.format(message + "\n");
        }
        //return message + "\n" + ((attachment == null) ? "[Empty]\n" :
attachment.preview());
    }
    /*
     * @brief set the attachment with passed parameter
     * @param 'a' variable of Attachment object
     */
    public void setAttachment(Attachment a){
        attachment = a;
    }
}
```

Code.Green.NotificationCollection.java

```java
package Code.Green;

import Code.Blue.Attachment; //to set attachment
import Code.Blue.Notification; //to access notification object
import java.util.ArrayList; //data structure used to store items
/*
 * @brief NotificationCollection.java
 * @note has an implementation arrow that connects to
NotificationCollectionIF
 */
public class NotificationCollection implements NotificationCollectionIF{
    ArrayList<Notification> items;
    /*
     * @brief constructor
     */
    public NotificationCollection(){
        items = new ArrayList<>();
    }
    /*
     * @brief getItem func given an index
     * @param index : integer
     * @return get func
     */
    public Notification getItem(int index){
        return items.get(index);
    }
    /*
     * @brief adds an item into the collection (array)
     * @param str : String of added item from main
     */
    public void addItem(String str){
        items.add(new Notification(str));
    }
    /*
     * @brief gets the total length/size of the array list
     * @return size
     */
    public int getLength(){
        return items.size();
    }
    /*
     * @brief sets the attachment for the notification collection
     * @note rather than creating new variables, it is much easier
     * to recall back with using the getItem func in this java class
     * and obtaining setAttachment from the Attachment.java
     */
    public void setAttachment(Attachment a, int index){
        this.getItem(index).setAttachment(a);
    }
    /*
```

```
     * @brief gets an iterator from the Notification Collection
     * @return NotificationIterator
     */
    public NotificationIteratorIF createIterator(){
        return new NotificationIterator(this);
    }
}
```

Code.Green.NotificationCollectionIF.java

```
package Code.Green;
/*
 * @brief optimization from NotificationBar.java
 */
public interface NotificationCollectionIF {
    public NotificationIteratorIF createIterator();
}
```

Code.Green.NotificationIterator.java

```
package Code.Green;
import Code.Blue.Notification;
/*
 * @brief NotificationIterator has a relationship with
NotificationIteratorIF which
 * implements its interface, and must involve notification as it will be
used
 *
 * @note named collection for uml purposes
 * @note index to indicate the location of the pointer
 */
public class NotificationIterator implements NotificationIteratorIF{
    private NotificationCollection collection;
    public int index;

    /*
     * @brief constructor
     * @param c : NotificationCollection of collection
     */
    NotificationIterator(NotificationCollection c){
        this.index = 0;
        this.collection = c;
    }

    /*
     * @brief if iterator has more objects versus the index that has not
been iterated
     * @return true with more objects, else object is at the end
     */
    public boolean hasNext(){
        return index < collection.getLength();
    }
```

```
    /*
     * @brief get the next notification in the collection
     * @return NotificationCollection
     */
    public Notification next(){
        return collection.getItem(index++);
    }
}
```

Code.Green.NotificationIteratorIF.java

```
package Code.Green;

import Code.Blue.Notification; //needed since Notification object is used

/*
 * @brief creating a Notification Iterator interface to create
 * hasNext() and next()
 */
public interface NotificationIteratorIF {
    public boolean hasNext();
    public Notification next();
}
```

Code.Demo.java

```
package Code;

import Code.Green.NotificationCollection; //Notification Collection
object, and for number 1
import Code.Blue.Attachment; //for number 2
import Code.Blue.FileAttachment; //for attach files feature
import Code.Blue.MediaAttachment; //for media files feature

public class Main {
    public static void main(String[] args) {
        NotificationCollection nc = new NotificationCollection();
        NotificationBar nb = new NotificationBar(nc);

        //1. create Notifications and add them to nc
            //nc.addItem("string?");
        /*nc.addItem("Big Bad Wolf");
        nc.addItem("Green Eggs and Turkey");
        nc.addItem("Eye Heart Radio");
        nc.addItem("");*/

        //generate ten new notification items to be added in the
collection
        for(int i = 1; i <= 10; i++){
            nc.addItem("Notification count " + i + ": Pass");
        }
        //System.out.println("Complete!\n\n");
```

```
        //nc.printNotifications();
        //2. create attachments and link them to notifications in nc
            //nc.getItem(index).attach(name of attachment);
        System.out.println("-----------Attachment Process
below----------");
        Attachment text1 = new FileAttachment("Fan_Syllabus.txt", 1001,
100);
        Attachment text2 = new FileAttachment("Zheng_Syllabus.txt", 1002,
200);
        Attachment heart = new MediaAttachment("Heart.txt", 1003, 1000);
        Attachment turkey = new MediaAttachment("Turkey.txt", 1004, 2300);
        Attachment wolf = new MediaAttachment("Wolf.txt", 1005, 3000);
        Attachment wolf2 = new MediaAttachment("Wolfie", 1010, 8000);

        /*
         * cont. of part 2, set the attachment corresponding to the vars
above
         */
        nc.setAttachment(text1, 0);
        nc.setAttachment(text2, 1);
        nc.setAttachment(heart, 2);
        nc.setAttachment(turkey, 3);
        nc.setAttachment(wolf, 4);
        nc.setAttachment(wolf2, 9);

        /*
         * call the notification bar object to print the rest of the
attachments
         */
        nb.printNotifications();
    }
}
```

Code.NotificationBar.java

```
package Code;

import Code.Blue.Notification; //notification object
import Code.Green.NotificationCollectionIF; //notifications variable
import Code.Green.NotificationIteratorIF; //for utilizing the print func

/*
 * @brief
 * @note given notifications object, it must be redeclared to use the
createIterator() as
 * shown on the yellow pop up in the lab
 *
 */
public class NotificationBar{
    private NotificationCollectionIF notifications;

    /*
```

```
     * @brief constructor
     * @param nb : NotificationCollectionIF
     */
    NotificationBar(NotificationCollectionIF nb){
        this.notifications = nb;
    }
    /*
     * @brief output function to print notifications
     * @note copied the whole format given from the lab
     */
    public void printNotifications(){
        NotificationIteratorIF iterator = notifications.createIterator();
        while(iterator.hasNext()){
            Notification n = iterator.next();
            System.out.println(n.getContent());
        }
    }
}
```

Solution:

- First, remember to zip the src folder of your project and submit the zip file to the ungraded assignment named "Lab6CodeSubmission". One submission from each team.
- Paste a screenshot of a run of your program here.
- Also paste all you source code here.
- Save this report in PDF, then submit the pdf report to the graded assignment named "Lab6ReportSubmission". One submission from each team.