**Lab 04**

*In this lab, you will:*

* *Experience, through your own alterations, how to implement ‘separation of concerns’ with regard to a software architecture.*
* *This will occur through an alteration of the code, specifically separating certain aspects of a controller class, which are them moved into a Repository, implementing the Repository pattern.*
* *Finally, building on the notion of software architecture you will be guided to implement a Data Access Object pattern to help abstract the data persistence seen in the code, and in previous examples.*

*… which will result in the following learning outcomes:*

* An appreciation of the practical steps required to implement architectural changes.
* *A practical appreciation of the following concept from the lecture:*
  + ***“Separation of concerns****: a design principle which encourages the development of software units that focus on single aspects of the overall program functionality, and overlap with each other as little as possible…….”*

Table of Contents

[1.1 Preliminary 3](#_Toc503276106)

[1.2 Exercise 1 3](#_Toc503276107)

[1.3 Exercise 2 4](#_Toc503276108)

[1.4 Exercise 3 5](#_Toc503276109)

[1.5 Exercise 4 6](#_Toc503276110)

[1.6 Task 6](#_Toc503276111)

[1.6.1 Subtask 6](#_Toc503276112)

[1.7 Task 6](#_Toc503276113)

[1.7.1 Subtask 6](#_Toc503276114)

Figure 1: Lab4\_CustomerProjectsAppLab4Starter 3

## Preliminary

* Download the lab 4 zip file <**Lab4\_CustomerProjectsAppStarter.zip**> from GCULearn and unzip
* Open the **Lab4\_CustomerProjectsAppStarter** project in NetBeans:

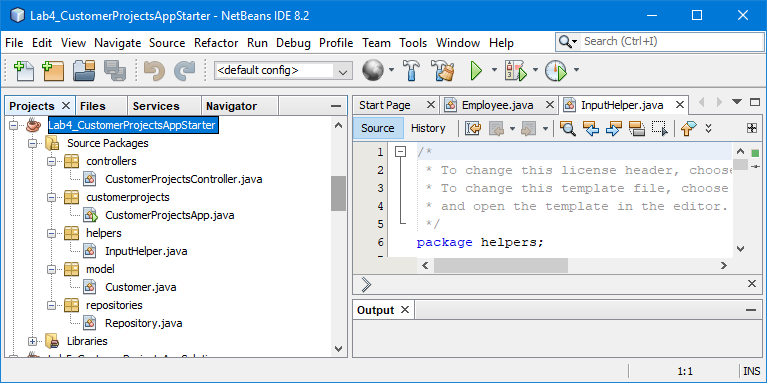


Figure : Lab4\_CustomerProjectsAppStarter

## Exercise 1

1. Open the **Customer** class, in the model folder, and note the attributes: **customerId**, **customerName**, and **customerProjects**. Constructor, getter and setter methods are defined as well as add, and remove project methods and the overridden method **toString()**.
2. Now open the *customers.txt* file and note the format. Ensure you understand how this might be loaded into a collection using the ideas in labs 2 & 3.

**1,"Martin",2,"MLGProj1","MLGProj2"**

**2,"Lynn",1,"LPKProj1"**

**3,"Ciara",3,"CODProj1","CODProj2","CODProj3"**

Open the controller class: a constructor method requests a file name from the user and passes the filename into a load method. Ensure you understand the **load()** private method:

**public ArrayList<Customer> load(String filename) {**

**ArrayList<Customer> repository = new ArrayList<>();**

**try (BufferedReader br = new BufferedReader(**

**new FileReader(filename))) {**

**String[] temp;**

**String line = br.readLine();**

**while(line!=null){**

**temp=line.split(Character.toString(DELIMITER));**

**int customerId = Integer.parseInt(temp[0]);**

**String customerName = stripQuotes(temp[1]);**

**Customer customer =**

**new Customer(customerId, customerName);**

**int noProjects = Integer.parseInt(temp[2]);**

**for (int i=0; i<noProjects; i++) {**

**String project = stripQuotes(temp[i+3]);**

**customer.addProjectToCustomer(project);**

**}**

**repository.add(customer);**

**line = br.readLine();**

**}**

**br.close();**

**} catch (IOException ex) { … }**

**return repository;**

**}**

1. Run the **CustomerProjectsApp** class and test that the customer projects can be successfully loaded and displayed. Add and remove projects from a specified customer and store them back to a text file. Ensure you can reload the file correctly.

## Exercise 2

We are now going to modify the architecture of the app to implement a repository pattern to manage the **customers** collection.

1. Open the **CustomerProjectsController** class and amend the model definition from an **ArrayList<Customer>** to a **Repository**:

**private final Repository repository;**

Add the import for the **Repository** class, adjust the lines which create the **Repository** object and remove the **load()**, **store()** & **stripQuotes()** methods - this functionality is now implemented in the **Repository** class.

1. View the **Repository** class in the repositories folder and ensure you understand its attributes and methods.
2. Amend the **addProjectToCustomer()** and **removeProjectFromCustomer()** methods by replacing the **for** loop with the commented statement:

//requiredCustomer = repository.getItem(customerId);

**for (Customer customer:this.repository) {**

**if (customer.getCustomerId() == customerId)**

**requiredCustomer = customer;**

**}**

1. Adjust the *Finish* menu option to call the **store()** method of the **repository** object.
2. Run the **CustomerProjectsApp** class and test that the customer projects can be successfully loaded and displayed. Add and remove projects from a specified customer and store them back to a text file. Ensure you can reload the file correctly.
3. Modify the **Repository** class to utilise a **LinkedList** implementation rather than an **ArrayList** implementation. Make the changes required to the **Repository** class and then run the **CustomerProjectsApp** as before.

## Exercise 3

We are now going to abstract the code which handles persistence i.e. File I-O from the controller class to allow it to focus on the functionality of the app. This approach uses a **Data Access Object** (**DAO**) pattern with dedicated classes for File I-O of model objects.

1. Create a **daos** folder within the project and an interface, **DAOInterface**, to specify the methods a DAO object must have i.e. **load()** and **store()**. The interface cannot be instantiated but defines methods signatures, including parameters, which an implementation must implement.

**public interface DAOInterface {**

**public Repository load(String filename);**

**public void store(String filename, Repository repository);**

**}**

Note the **load()** method expects a filename, as **String**, and returns a **Repository** object; while the **store()** method expects the **String** filename and a **Repository** object to persist.

1. As we cannot instantiate an interface object, and, anyway, it doesn’t actually contain code to implement the **load()** and **store()** methods, we need to have implementation classes.

Initially, we will focus on a text file implementation. Create a **DAOTextImpl** class in the **daos** folder to implement the **load()** and **store()** methods using delimited text files.

**public class DAOTextImpl implements DAOInterface {**

**static final char DELIMITER=',';**

**@Override**

**public Repository load(String filename) {**

**…**

Move the load and store code from the **Repository** class and make any changes required.

1. Adjust the **Repository** constructor and **store()** methodsto create a **dao** object and execute its **load()** and **store()** methods respectively:

**DAOTextImpl dao = new DAOTextImpl();**

**this.items = dao.load(filename).getItems();**

and:

**DAOTextImpl dao = new DAOTextImpl();**

**dao.store(filename, this);**

1. Run the **CustomerProjectsApp** as before.

## Exercise 4

1. Create a class to implement the **DAOInterface** using object files.
2. Run the **CustomerProjectsApp** as before.