# 1 Application Description

An online book shop company has been very successful in selling books in the past years. The

company management plans to open the online shop as a marketplace to other products and thirdparty

companies so that they can also sell their products using the online platform.

The company needs to update the backend software system of the online shop with the following

requirements:

1. Customer Data Access.

The customer data is retrieved from a large database by using the

customer email address only. The main company has a lots of legacy systems that work with

the existing interfaces to get the customer data.

The company engineers plan to change this system. In the new online shop (marketplace), all

components including external companies have to retrieve customer data using email address

and phone number. The company plans to implement the new system in a later phase. At

the current stage, engineers like to expose only the new API to external companies. Consider

that we have the following two APIs:

// Old System

interface CustomerDataOld {

Customer getCustomer ( String email );

}

The new system has the following API to access customer data using customer ids.

// New System

interface CustomerDataManagementSystem {

Customer getCustomer ( String email , String phoneNumber );

}

Customer data requests should include both customer email and phone number. The customer

data storage system returns customer data item as an object encapsulating all customer data

attribute/values.

As engineers have not yet developed the new system, all of the requests have to be converted

to the old API and use the old system. The old system is only using the email address.

To convert the requests to the old API, customer email address should be forwarded and

customer phone number be dropped.

2. Customer Data Processing.

The customer data has to be processed in different steps. for

example like the following steps, 1. EMail address verification 2. Background Checks 3.

Welcome email generation and send out 4. Rejection email generation, and others.

Depending on the situation, the company may need to run some of these data processing steps

and needs the flexibility to add or remove them. Design your implementation to provide this

flexibility.

3. Order Processing. (OPTIONAL Requirement)

Orders processing is very complex and

includes multiple sub-processes. The order processing includes multiple sub-process, like

the following steps, 1. product pick-ups, 2. packaging (if a gift then use gift wrap) and 3.

package delivery. Consider that these 3 steps are very complex and include multiple subprocesses.

The main company requires to provide a base implementation of these 3 processes and allow

external companies reuse the basic implementation of these, or have the possibility to jump

in specific places of the process execution and changed it as they needed.

You should provide a software design for the implementation of these processes that allows

such flexibility.

# 1.1 Implementation Details

Your implementation should include the following functionalities:

• Create specific implementation of the given interfaces. (You can create some Mockup objects

for customer objects )

• Create concrete implementations and test executions as demonstration of your implementation.

Note: In this assignment, you should apply “at least one correct" design pattern. We do not

mention which one of the design patterns is suitable for this scenario. This would be your task as a

software engineer to find it out.

Note: The given application scenario can be implemented in different ways and with different

implementation details. There is not only one single correct design and implementation for

the above application scenario. Each software developer might consider different assumptions and

design the software based on them. In this assignment you are free to have your own detail assumptions

and implement the details in your own way based on your own ideas. You should document

your assumptions very well in your README.md file of your project and consider them in your

UML diagrams.

# 2 Assignment Tasks

## 2.1 Task 1 : Implementation Description. (2 points )

Describe how would you implement this application considering software design principles. Describe

what are your main software design concepts regarding this application.

For example describe:

### • How is the flexibility, of your implementation, e.g., how you add or remove in future new

### types?

The customerdata package is flexible in that the adapter pattern is applied. The concept of this

assignment was that the Customer class had been originally (as indicated by the CustomerDataOld interface)

only tracking and looking up Customers by email. The "new system" as denoted by

the CustomerDataManagement interface wants to track and lookup customers by email and phone number.

The idea behind the CustomerAdapter is to give backwards compatibility to the Old System

and the flexibility for the New System to continue to develop as requirements develop. In this

implementation, classes aren't added or removed - rather refactored. There is an implicit jump

from the old system to the new system in my reading of the context provided in the assignment

description that attributes and methods are simply added to the Customer class from Old System

to New System and that the Adapter pattern is needed to handle data stored in a database or on

disk somewhere in the old format (I have only simulated this by creating objects in the old

and new formats in tests, not through file reading or sql queries as I believe the functionality

displayed in tests implies that reading Objects into memory to either format and using the

Adapter on these read objects follows from the tests shown).

-The dataprocessing package is flexible in that data processing algorithms are free to be

created, deleted and flexible in how they are added and removed from the processing queue.

This design is completely agnostic of order although one might imagine there is an implicit

logic to which processing functions might be performed (background check before welcome email etc.)

The system however is flexible enough to enforce such ordering if that became a requirement

but is explicitly designed in it's current state to be completely agnostic in this regard.

The orderprocessing package is also fairly flexible since states all implement the same

interface and are not dependent on one another in any way. They can be created and deleted

or expanded upon as needed.

The notifications package is there such that orderprocessing/Orders which HAVE-A state

are placed by Customers who can, through this basic modified Observer Pattern (in that

Publishers/Orders have a one to one relationship with Subscribers/Customers). This implementation

is also flexible and easily modifiable to meet changing future requirements.

### • How is the simplicity and understandability of your implementation?

Although there are four layers of patterns (Adapter, Template, State and Observer) in this implementation,

it remains fairly a fairly simple implementation and quite a lot of functionality is accomplished

with a relatively small amount of code. Further, simplicity seems high as well and contextually

each pattern reflects real world, tangible examples of: modifying/adapting old system data to

a new model, having states for orders and restrictions on how these states transition from one

to another in ways that are intuitive and translate to the real world, and notifications that

also reflect real world applications such as Amazon notifying when a delivery has been packed,

shipped and delivered.

• How you avoided duplicated code?

- Simply by incorporating all of the requirements and ensuring these are met by the layering

of these three patterns on top of one another. One exception here is that, by nature of the

pattern, the Adapter pattern does have some duplicated code, although it is not redundancy,

rather a reframing on the code to allow one system to be re-interpreted by another through

an Adapter interface and Class. Otherwise, following the "text book" versions of these patterns

closely eliminates the need for duplicated code.

In case that you have used any design patterns, describe which design pattern you have applied

and why.

Write your description in a README.md file, use MarkDown format https://spec.commonmark.

org/current/ and add the README.md file to the root folder of your project.

We recommend writing this description after you finished the other tasks in this assignment.

# 2.2 UML Diagrams

. Class Diagram also available [here](https://lucid.app/lucidchart/d8b93ebb-9962-42db-8353-b94ed5e71330/edit?page=0_0&invitationId=inv_2773542b-f812-49e5-a8b1-c9dd1c2fdb1d%23)

Diagram

Description automatically generated

. State Transition Diagram for orderprocessing package also available [here](https://lucid.app/lucidchart/1c8f6c35-84af-44ef-a785-6991616b0b00/edit?beaconFlowId=C1805BF390DAF84C&invitationId=inv_aa57a925-da9e-439b-a22e-18c8cefd22f5&page=0_0%23)

Diagram

Description automatically generated

# 2.3 Task 4 Implementation of Your Solution in Java

<https://github.com/metcs/met-cs665-assignment-4-MichaelKramerGuitar>