SW2 – Project Evaluation Form

- **Each team must submit the following Documentation that contains:**
  - **Project Description in detail.**
  - **Class Diagram. And Database Schema.**
- **Each team must submit the project via GitHub:**
  - **Source Code.**
  - **Video Demo for running (2 – 5 Minutes).**
  - **Documentation and Evaluation Form.**
- **The Evaluation will start with giving all teams 30 marks then check the following criteria:**

Violation Level	Full	Medium	Small	Grade
Documentation	-5	-2	-1	
Not Apply MVC (it does not Separate Business logic from GUI). Example of violation: write an implementation for a method such as an inset item into the database inside the Button Action method	-6	- 3	-1	
Violate clean code – Variables	-2	-1	-.05	
Violate clean code – Functions	-2	-1	-.05	
Violate Single-responsibility Principle	-2	-1	-.05	
Violate Open-closed Principle	-2	-1	-.05	
Violate the Liskov Substitution Principle	-2	-1	-.05	
Violate Interface Segregation Principle	-2	-1	-.05	
Violate Dependency Inversion Principle	-2	-1	-.05	
Not Upload code to GitHub	-1			
Only One Branch Without Merge (GitHub)	-2			
Only One Contribution (GitHub)	-2			
Total Minus from Grade				

Design Pattern Bounce	+4	
Bounce on Overall Work	+2	
Total Team Grade / 30		

Full violated –They violate the principle in all cases in the Code.
Medium Violated – They apply the principle in some cases and violate it in some cases (in total the student applies the principle on average).
Small violated – They Apply the principle in most cases except for a very few cases, no more than one or two.

Name (Arabic)	ID	Individual Bounce +2	Grade
احمد عزت نصر محمد	201900062		
احمد ماهر عبد القادر سيد	201900079		
خالد رزق علي فرج	201900279		
مريم يحى صلاح مرزوق	201900806		
وليد شعبان عبد الحليم رضوان	201900950		
عادل رجب فتح الله علي حرب	201900392		

Electricity Billing System

Contents

Functional requirements 3
Non-Functional requirements.....4
Use Case.....5
Use Case Description.....9
Use Case Scenario 14
Activity Diagrams.....18
Sequence Diagrams 22
Communication.....38
Diagrams 38
Package Diagram 47
System architecture.....49
ERD Diagram.....51
Design Patterns 54
Class Diagrams 62

Project Description

Electricity Billing System is a software-based application. The project aims at serving the department of electricity by computerizing the billing system. It mainly focuses on the calculation of Units consumed during the specified time and the money to be paid to electricity offices. This computerized system will make the overall billing system easy, accessible, comfortable and effective for consumers.

Requirements

Functional requirements

1. **Old Customer Module**:
    a. Enables customer to view all bills with meter code.
    b. Enables customer to enter monthly reading of meter.
    c. Enables customer to complain about bill with meter code.

2. **New Customer Module**:
    a. Enables customer to fill out the contract.

3. **Operator Module**:
    a. Enables operator to view all bills and collect payments from customer.
    b. Enables operator to view all bills of a specific region.
    c. Enables operator to activate/deactivate meter or cancel subscription for customer.

4. **Admin Module**:
    a. Enables admin to view all bills of specific region.
    b. Enables the admin to view the total payments collected.
    c. Enables admin to make consumption statistics for specific region.
    d. Enables admin to add / update / delete users with different roles.

5. **System Module**:
    a. System will send email automatically to notify customer when meter is ready.
    b. System will send email automatically if a customer didn't pay within 3 months.
    c. The system validates the entered reading with the real consumption.
    d. The system defines tariff for customer.

Non-Functional requirements

1. **System should be performance efficient**:

    System shouldn't take very large space in memory.

2. **System must be secure for user's data privacy**:

    Must prevent unauthorized access of data, no customer can see data of another customer.

3. **System must be extendable for future features**:

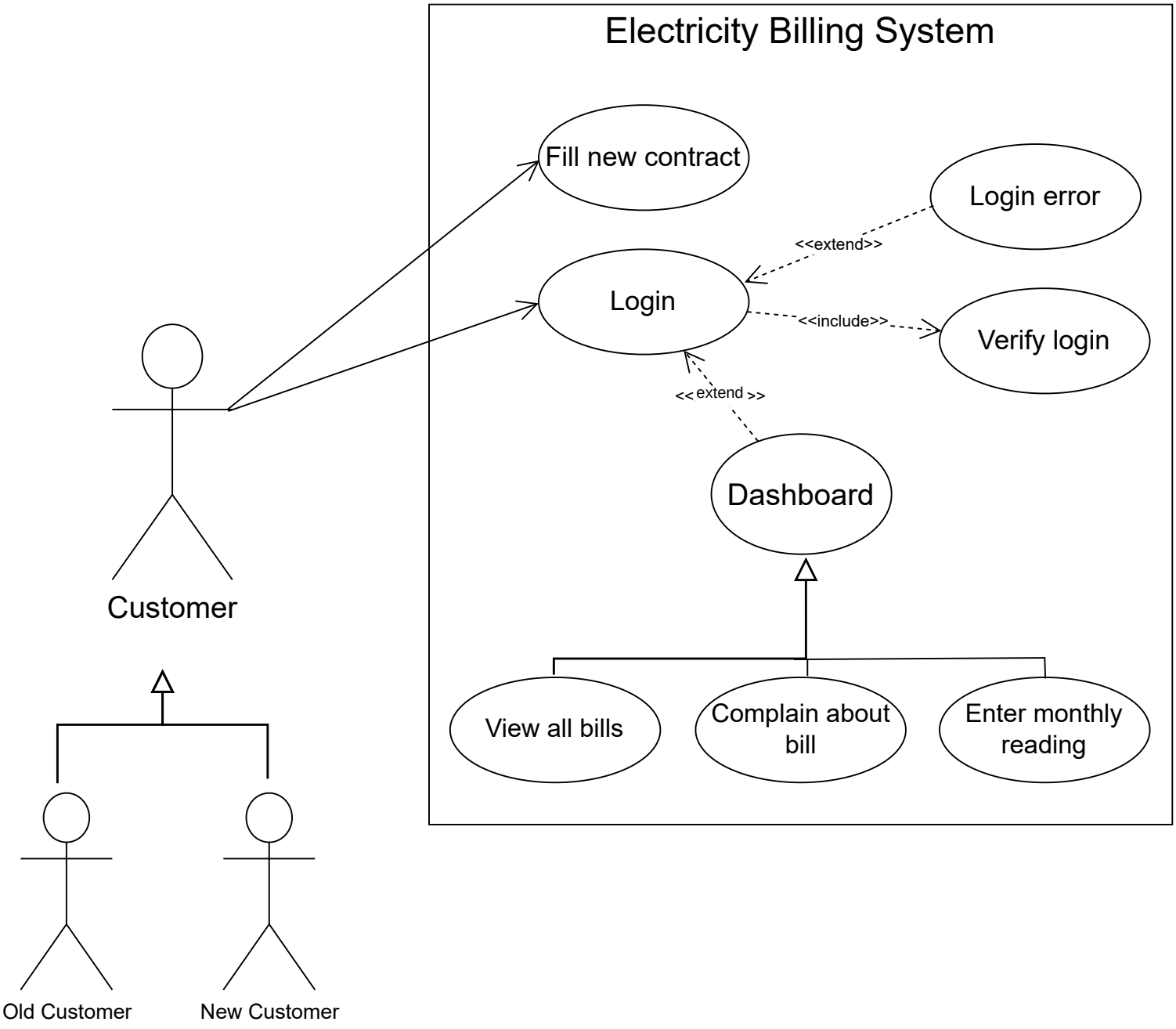
    System must be extendable for growth of company or changes of company policy.

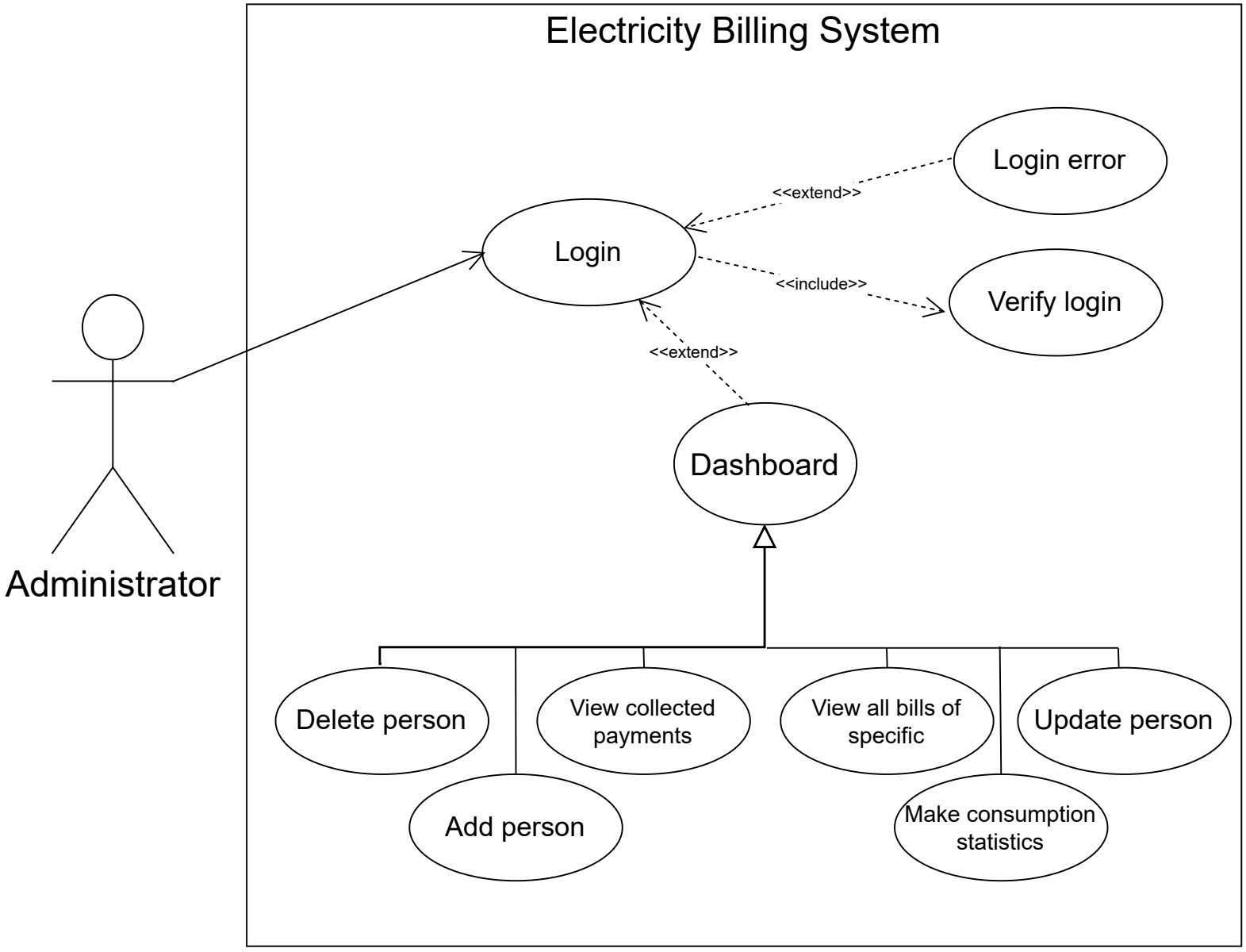
4. **System must be usable**:

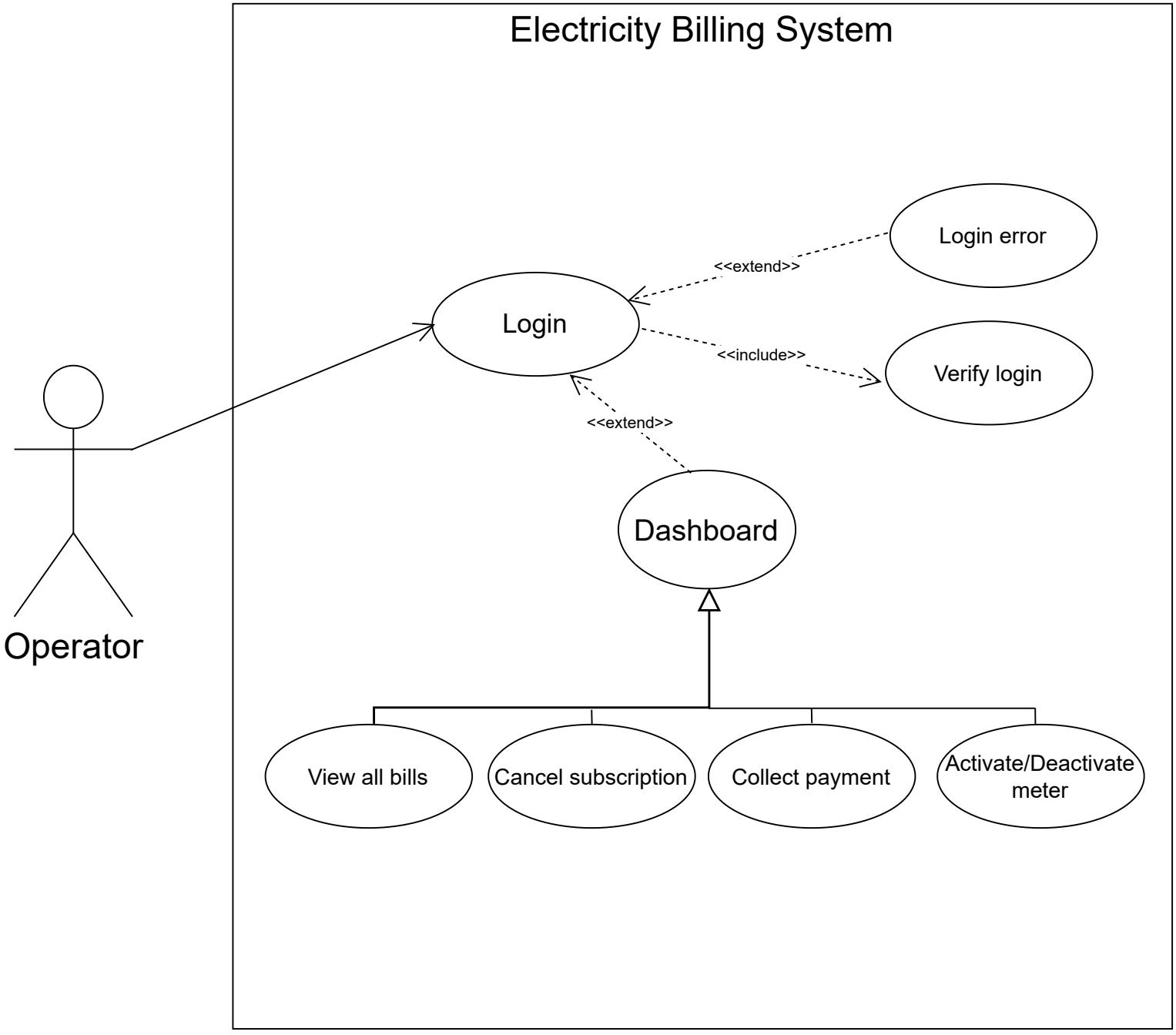
    System must be usable by people who know some basic of computers without a lot of training.

Use Case

Use case Diagrams




Electricity Billing System
Login error
<<extend>>
Login
<<include>>
Verify login
<<extend>>
Dashboard
Administrator
Delete person
View collected payments
View all bills of specific
Update person
Add person
Make consumption statistics


Electricity Billing System
Login error
<<extend>>
Login
<<include>>
Verify login
<<extend>>
Dashboard
Operator
View all bills
Cancel subscription
Collect payment
Activate/Deactivate meter

Use Case Description

Use Case ID	UC_1
Use Case Name	New Customer
Description	This use case allows new customer to fill new contract request. To fill contract there is several information must be provided like national id, name, attach copy of property ownership contract, email, phone number, address, photo, etc.
Primary Actor	New customer
Secondary Actor	None
Preconditions	None
Postconditions	1. The system must generate the meter code and the password and send it to the customer via email if the request is accepted.

Use Case ID	UC_2
Use Case Name	Old customer dashboard
Description	This use case allows old customer to do variance things such as enter monthly reading, view bills and complain about bill.
Primary Actor	Old customer
Secondary Actor	None
Preconditions	1. Old customer must have a valid meter code.
Postconditions	None

Use Case ID	UC_3
Use Case Name	Login
Description	This use case allows user to login into the system to access the relevant functions according to the user's role. The various user roles are customer, administrator and operator. To login to the system, administrator and operator must enter their unique id and password, customer must enter his meter code and password. After successful login, the system will display the relevant user's dashboard.
Primary Actor	Admin, Operator, Customer
Secondary Actor	None
Preconditions	1. Admin, Operator and Customer must have a valid account.
Postconditions	1. The system displays the relevant dashboard of each user.

Use Case ID	UC_4
Use Case Name	Administrator dashboard
Description	This use case allows Administrator to do variance things such delete (administrator, operator, customer), update (administrator, operator, customer) and add (administrator, operator, customer) to system.
Primary Actor	Administrator
Secondary Actor	None
Preconditions	1. Administrator must have a valid account to access his dashboard.
Postconditions	None

Use Case ID	UC_5
Use Case Name	Operator dashboard
Description	This use case allows operator to do variance things such as cancel subscription of customer and stop customer meter code, activate the meter code and collect payment from customers.
Primary Actor	Operator
Secondary Actor	None
Preconditions	1. Operator must have a valid account to access his dashboard.
Postconditions	None

Use Case Scenario

Use Case Name	New Customer
Action Actor	**System Reactions**
1. Pressing new customer button. 3. Fill new contract request. 4. Pressing the Request button.	2. Redirect to Fill contract page. 5. System verifies the entered data. 6. If successful registration. System must generate the meter code and password and send it to the customer via email. 7. Redirect the customer to home page to login.

Use Case Name	Customer Login
Action Actor	**System Reactions**
1. Pressing old customer button. 3. *Enter meter code and password then press login button.*	2. Redirect to customer login dashboard. 4. Redirect to customer dashboard.
1. *Pressing forget password button.*	2. Send new password via email.

Use Case Name	Administrator Login
Action Actor	**System Reactions**
1. Pressing Administrator button. 3. *Enter ID and password then press login button.*	2. Redirect to Administrator login dashboard. 4. Check if entered ID is Admin ID redirect to admin dashboard else if operator ID redirect to operator dashboard.
1. *Pressing forget password button.*	2. Send new password via email.

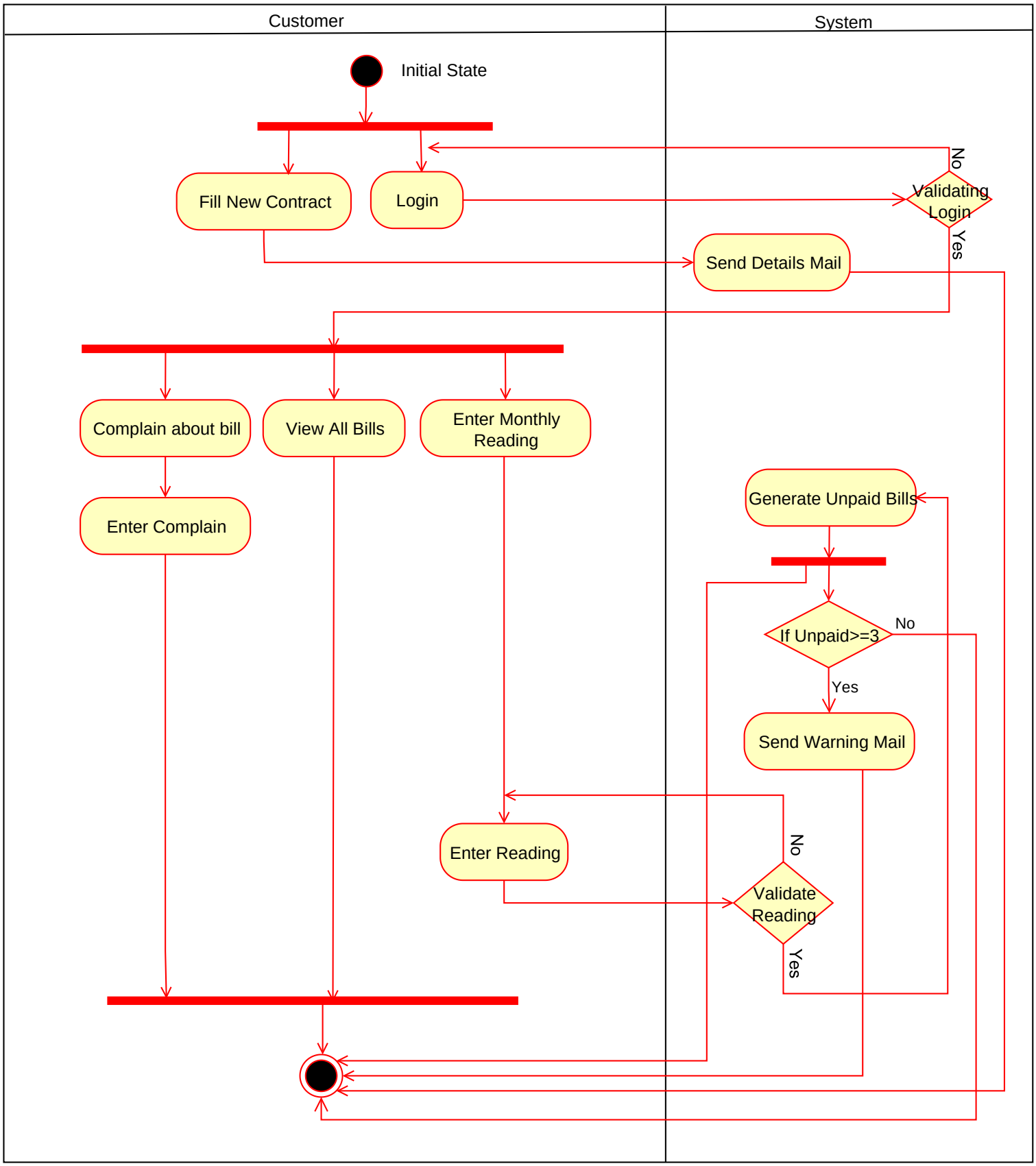
Use Case Name	Old customer dashboard
Action Actor	**System Reactions**
1. Pressing Enter monthly reading button. 3. Enter monthly Reading and the date of bill then submit reading.	2. If account is active redirect to enter monthly reading page. If inactive Enter monthly reading button is disabled and user cannot press it. 4. Redirect to customer dashboard.
1. Pressing Complain about bill button. 3. Select bill and write the complaint massage then submit.	2. Redirect to complain about bill page. 4. Redirect to customer dashboard.
1. *Pressing view all bills.*	2. View bills table upon the selected status if unpaid bills view unpaid bills else view paid bills.

Use Case Name	Admin dashboard
Action Actor	**System Reactions**
1. Pressing make consumption button.	2. Redirect to make consumption page and view consumption of selected region.
1. Pressing view total collected money button.	2. View total collected money.
1. *Pressing view all bills of specific region.*	2. Redirect to view all bills of specific region page and view bills of selected region.
1. *Pressing Add administrator.* Fill 3. new administrator information then submit.	2. Redirect to add administrator page. 4. If successful registration. System must generate the new administrator ID based on his role and password and send it via email.
1. ssing the new customer button. 3. new contract request. sing the 4. Pres uest button.	2. Redirect to Fill contract page. 5. If successful registration. System must generate the meter code and password and send it to the customer via email. 6. Redirect to Admin dashboard.
1. *Pressing update administrator. Enter* 2. *administrator ID.*	3. Redirect to update administrator page.
1. *Pressing update customer. Enter* 2. *customer meter code.*	3. Redirect to update customer page.
1. *Pressing delete administrator. Enter* 2. *administrator ID.*	3. System checks the number of administrators in the system. Then delete entered administrator if the number of administrators more than one. otherwise, alert admin that he cannot delete himself.
1. *Pressing delete customer.* 2. *Enter customer meter code.*	3. Redirect to delete customer page.

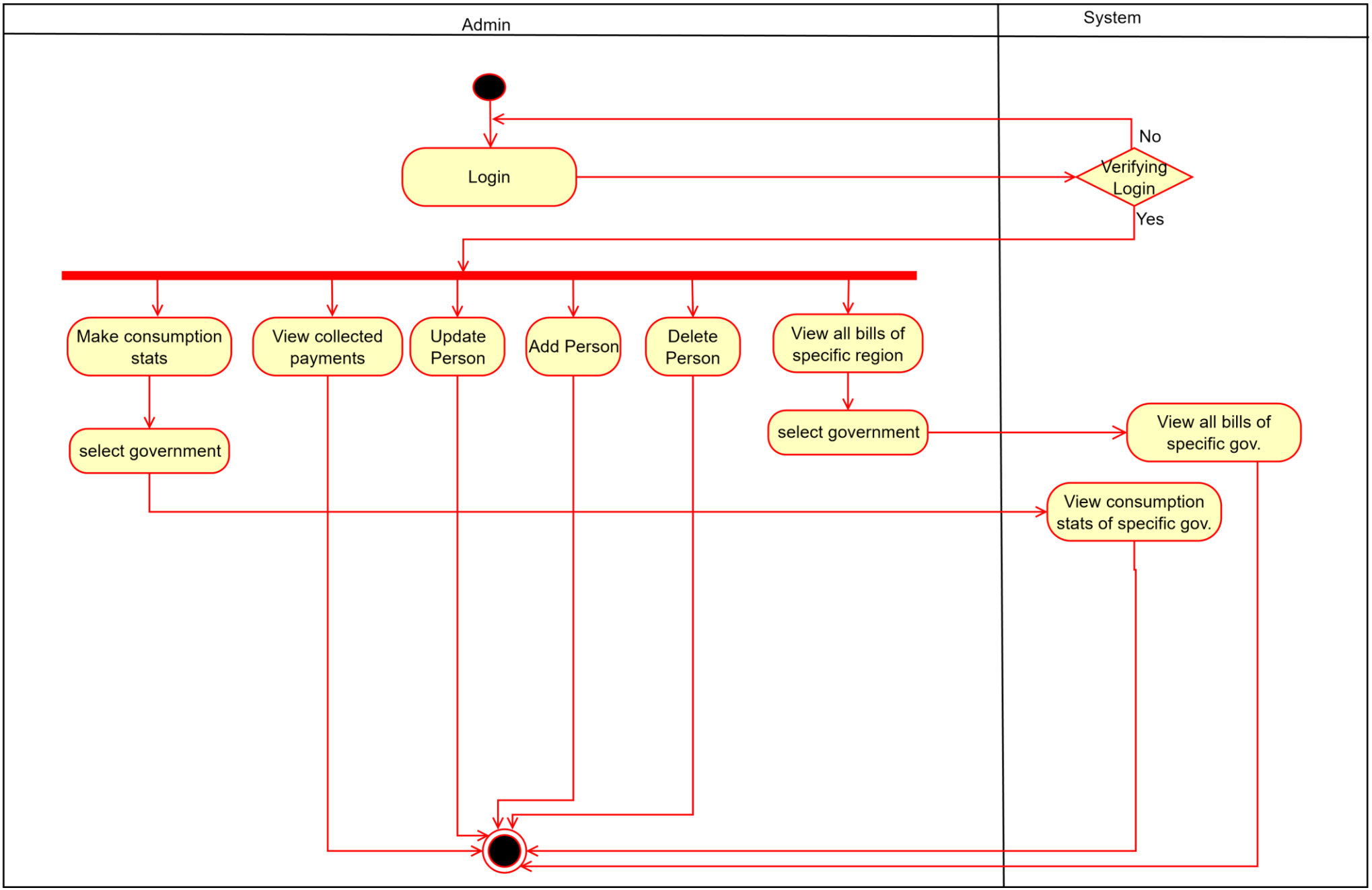
Use Case Name	Operator dashboard
Action Actor	**System Reactions**
1. *Pressing view all bills of specific region button.*	2. Redirect to view all bills of specific region page and view bills of selected region.
1. *Pressing view all bills and collect payment button.* 2. *Enter customer meter code.* 4. *Pressing pay bill button.* 6. *Pressing pay this bill button.*	3. *Redirect to view all bills and collect payment page.* 5. *Redirect to pay bill page in which operator can pay oldest unpaid bill.* 7. System mark bill as paid bill and redirect operator back to updated view all bills page.
1. *Pressing activation button.* 2. *Enter customer meter code.* 4. *Pressing toggle activation state button.*	3. *Redirect to activation page.* 5. *System detects activation state and toggle it.*
1. *Pressing cancel subscription.* 2. *Enter customer meter code.* 4. *Pressing delete button.*	3. Redirect to delete customer page. 5. Delete customer if there is no unpaid bills else alert operator that we cannot delete this meter code.

Activity Diagrams

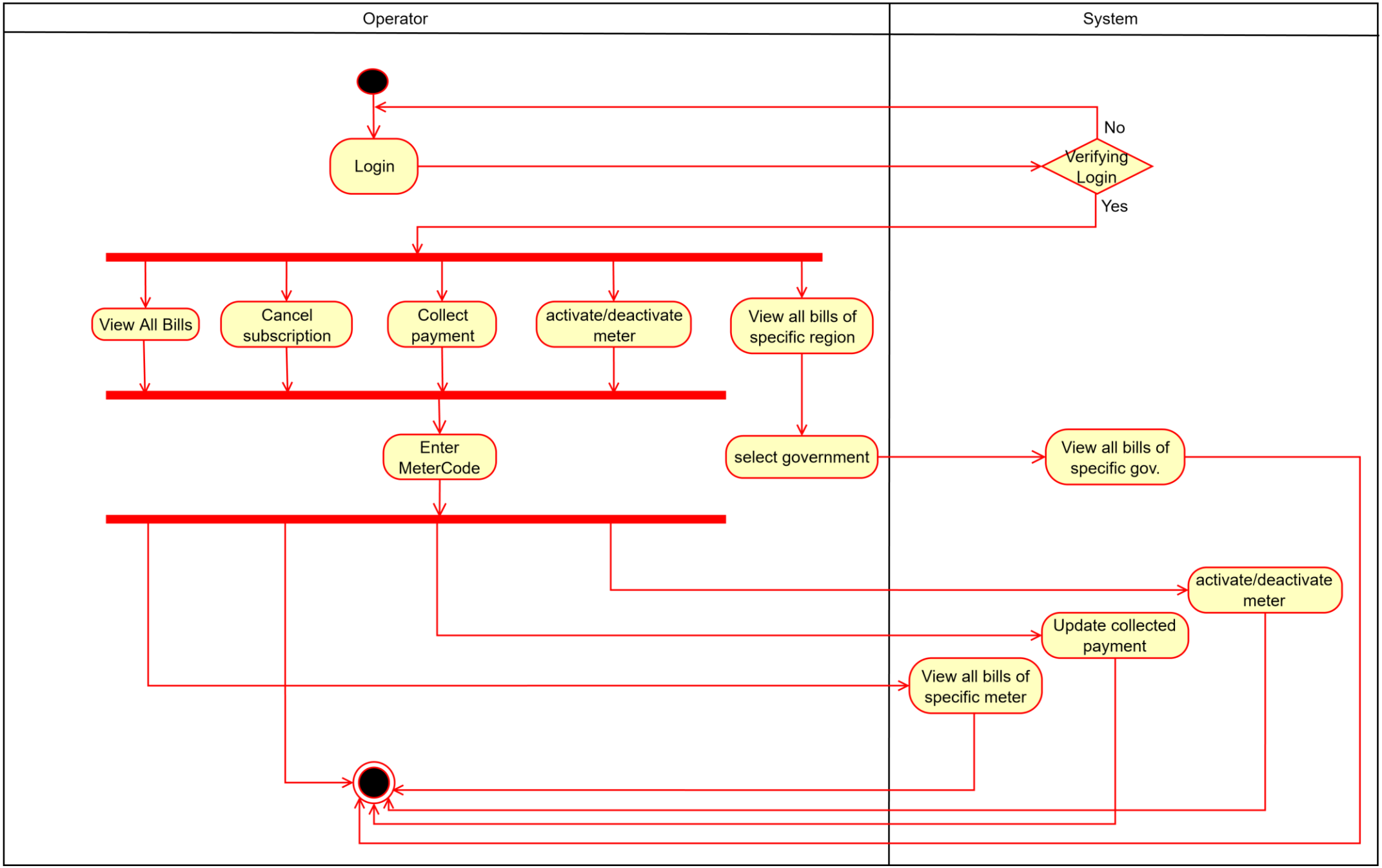
(Old& New) Customer Activity Diagram



Admin Activity Diagram



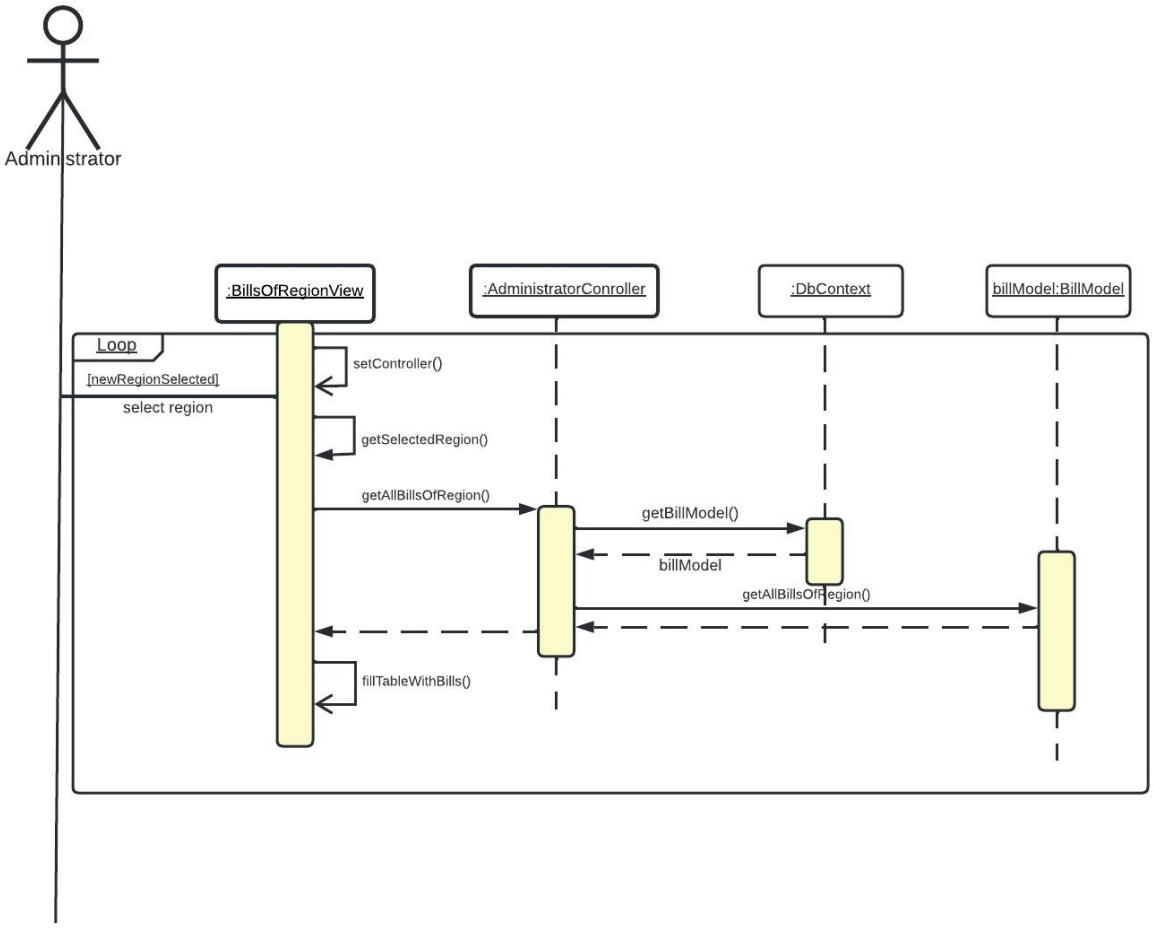
Operator Activity Diagram



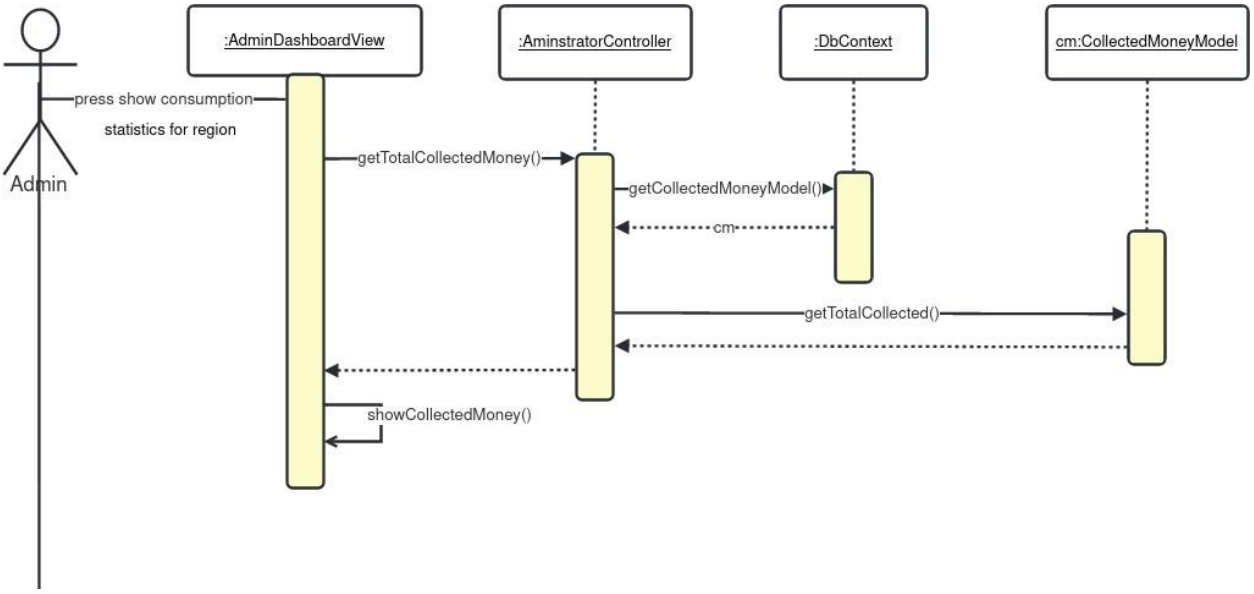
Sequence Diagrams

Admin module

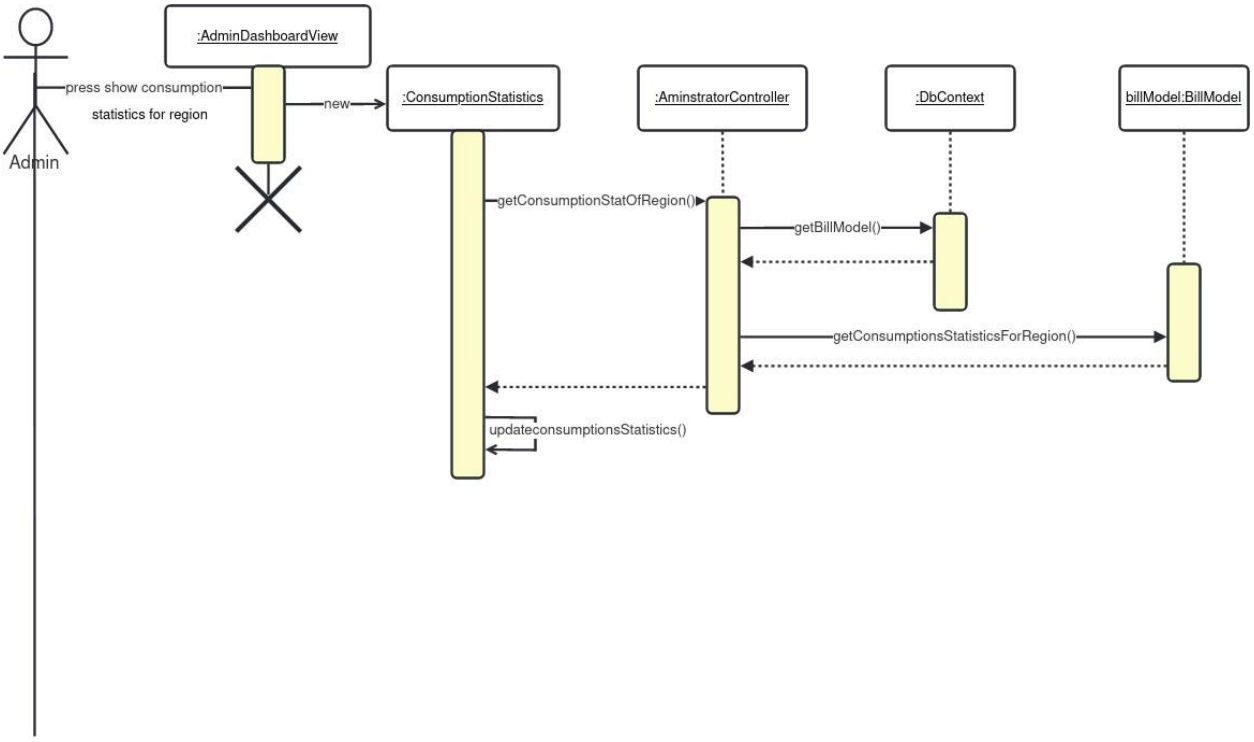
Enables admin to view all bills of specific region.



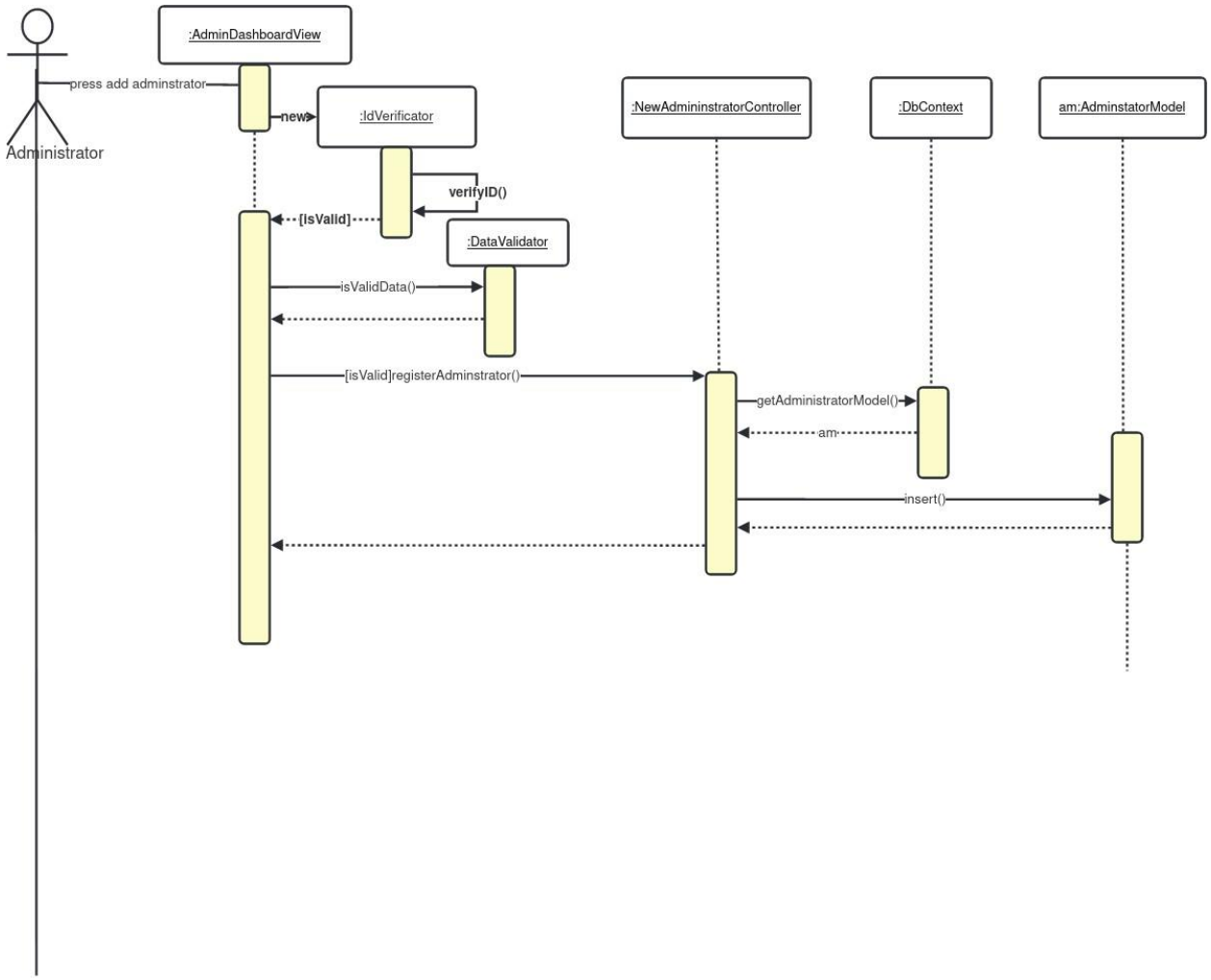
Enables the admin to view the total payments collected.

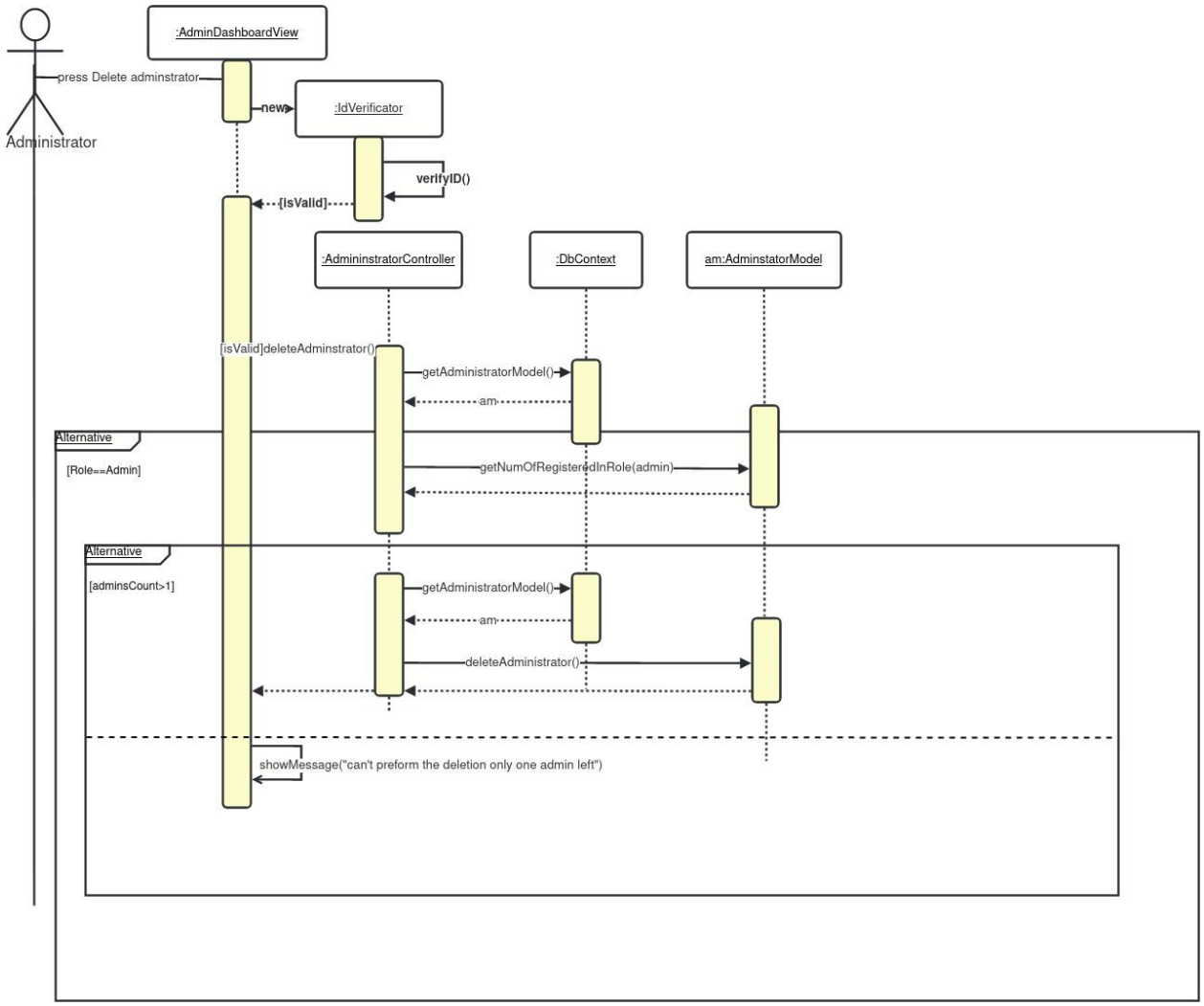


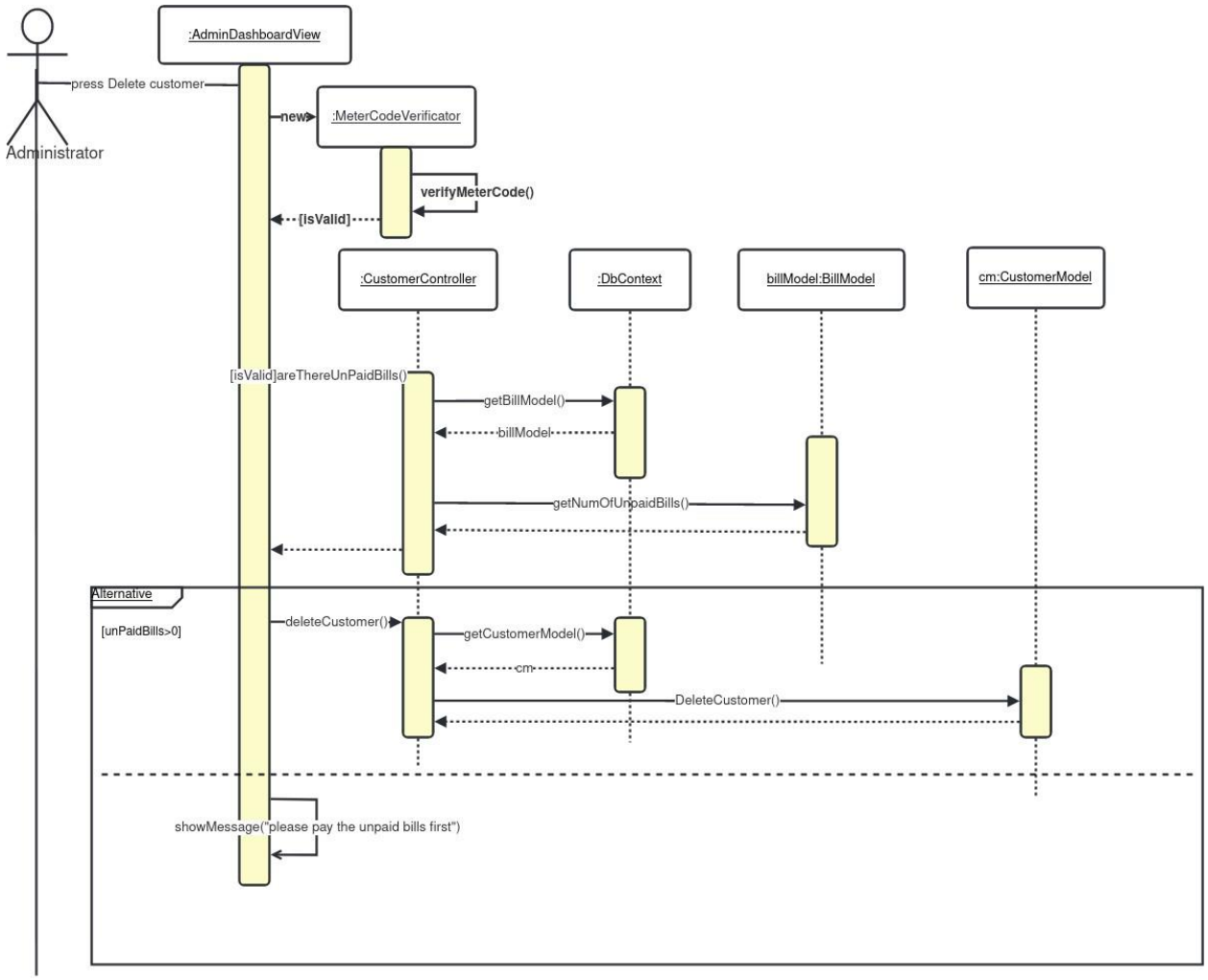
Enables admin to make consumption statistics for specific region.

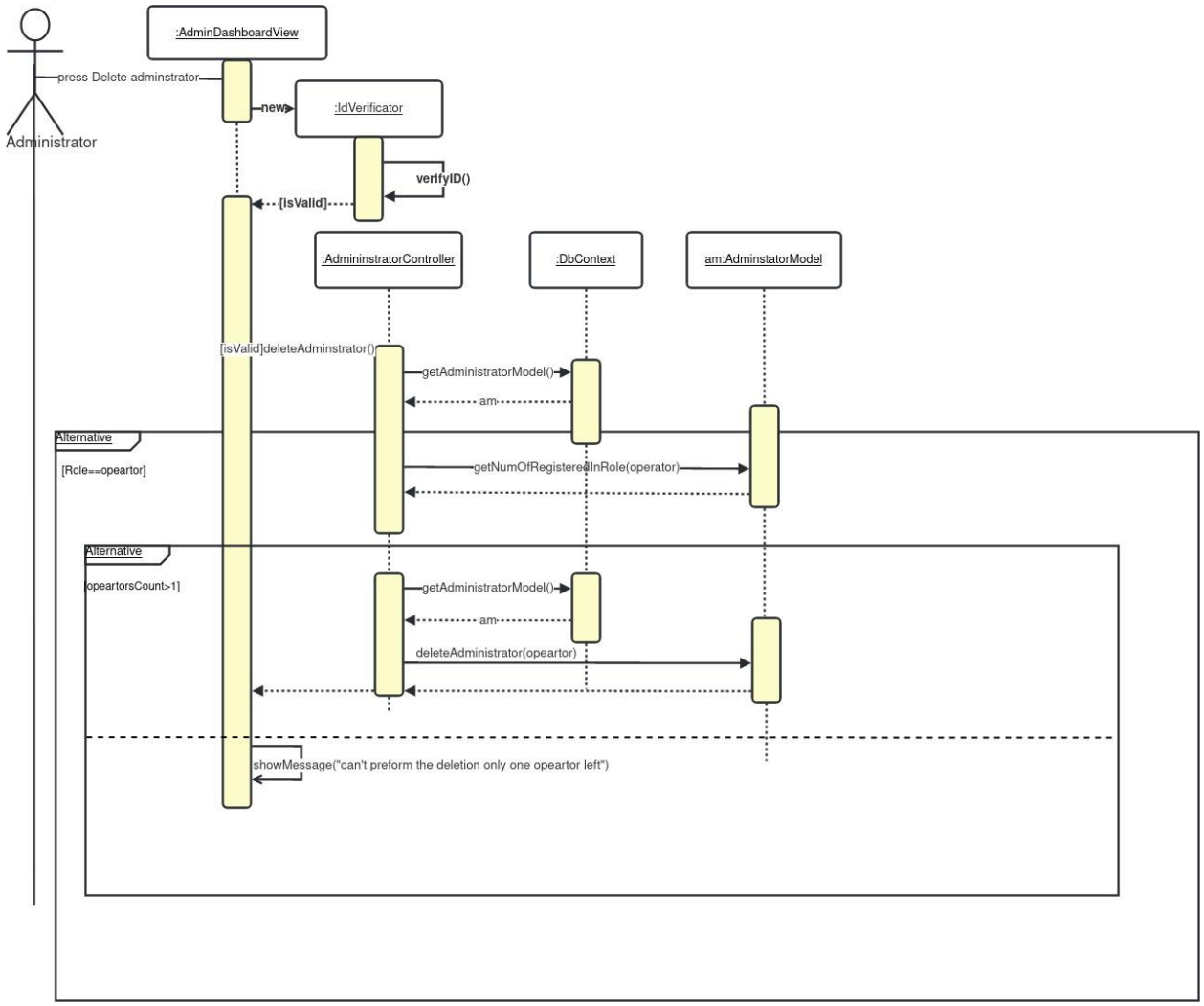


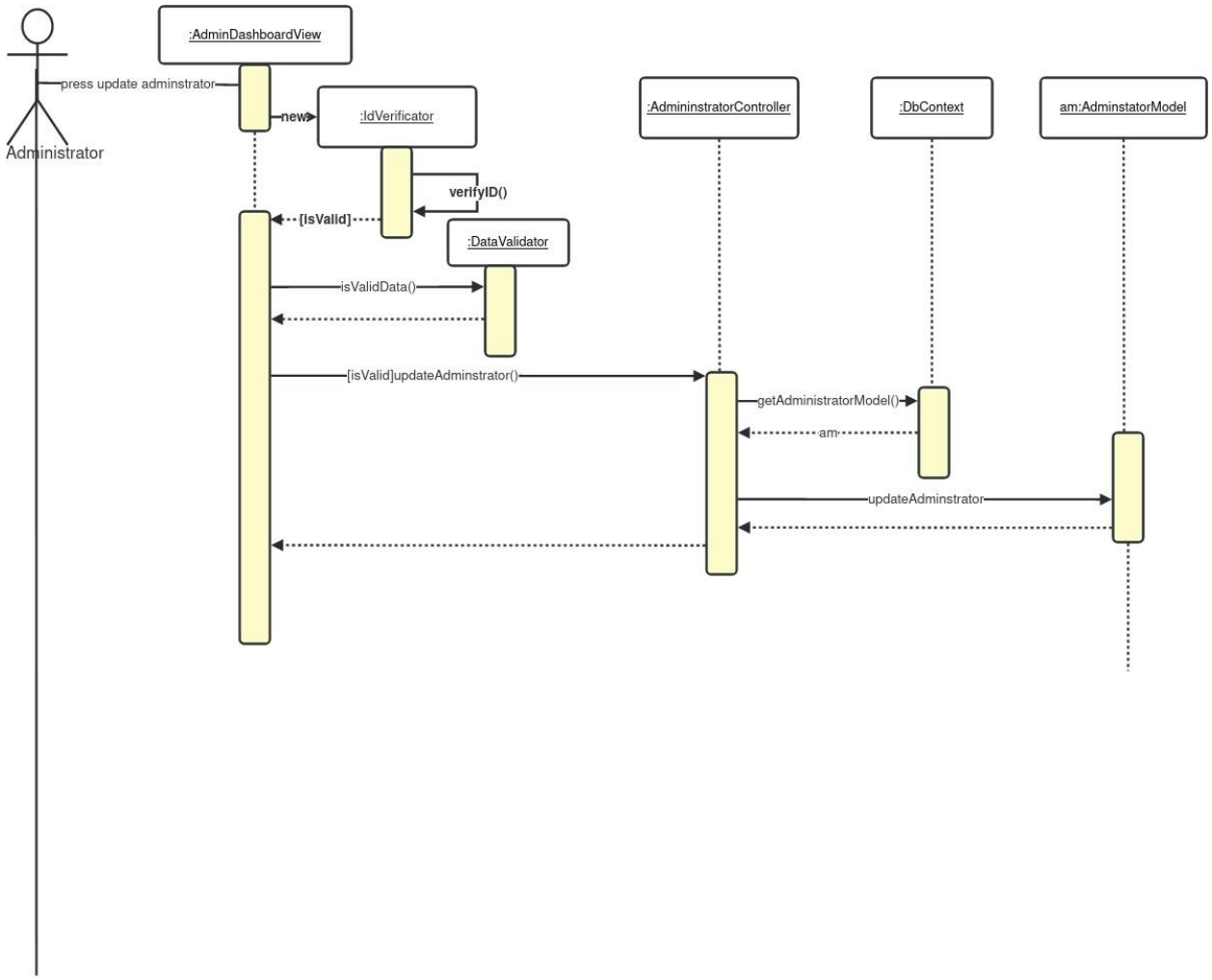
Enables admin to add / update / delete users with different roles

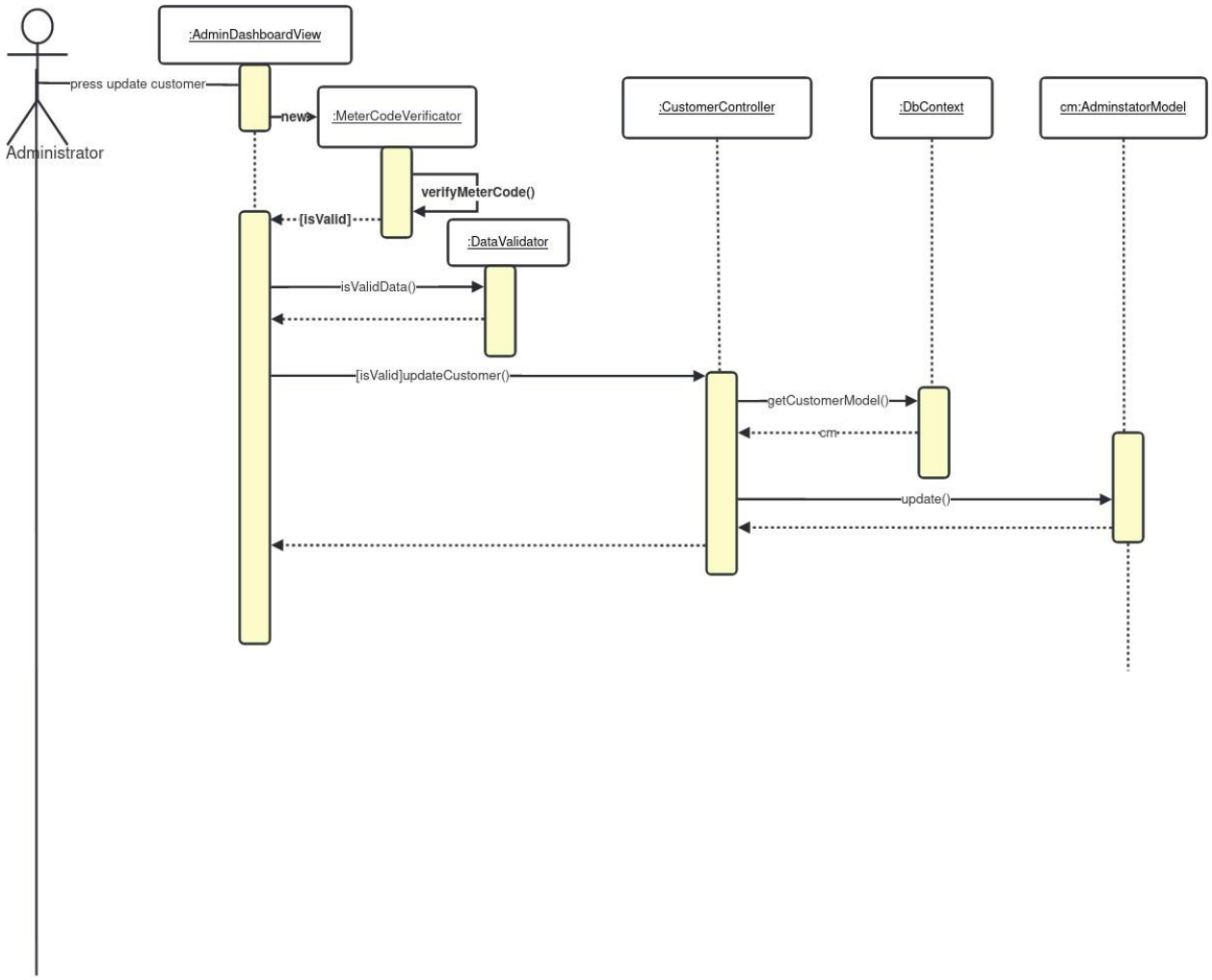



Administrator
:AdminDashboardView
press Delete adminstrator
new
:IdVerificator
verifyID()
[IsValid]
:AdmininstratorController
:DbContext
am:AdminstatorModel
[isValid]deleteAdminstrator()
getAdministratorModel()
am
Alternative
[Role==Admin]
getNumOfRegisteredInRole(admin)
Alternative
[adminsCount>1]
getAdministratorModel()
am
deleteAdministrator()
showMessage("can't preform the deletion only one admin left")


Administrator
:AdminDashboardView
press Delete customer
new
:MeterCodeVerificator
verifyMeterCode()
[isValid]
:CustomerController
:DbContext
billModel:BillModel
cm:CustomerModel
[isValid]areThereUnPaidBills()
getBillModel()
billModel
getNumOfUnpaidBills()
Alternative
[unPaidBills>0]
deleteCustomer()
getCustomerModel()
cm
DeleteCustomer()
showMessage("please pay the unpaid bills first")

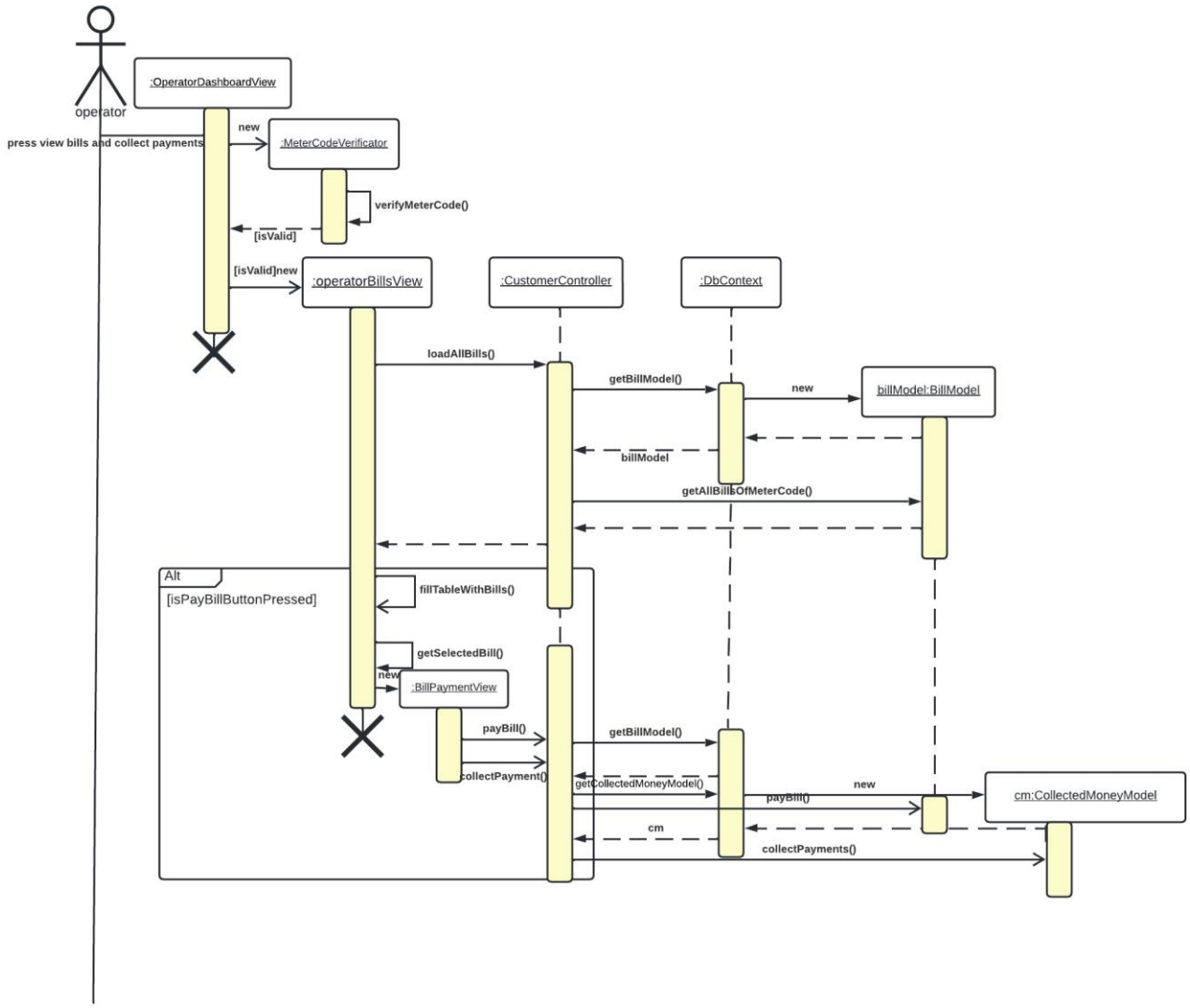

Administrator
:AdminDashboardView
press Delete adminstrator
new
:IdVerificator
verifyID()
[IsValid]
:AdmininstratorController
:DbContext
am:AdminstatorModel
[isValid]deleteAdminstrator()
getAdministratorModel()
am
Alternative
[Role==opeartor]
getNumOfRegisteredInRole(operator)
Alternative
[opeartorsCount>1]
getAdministratorModel()
am
deleteAdministrator(opeartor)
showMessage("can't preform the deletion only one opeartor left")


Administrator
:AdminDashboardView
press update adminstrator
new
:IdVerificator
verifyID()
[isValid]
:DataValidator
isValidData()
:AdmininstratorController
[isValid]updateAdminstrator()
:DbContext
getAdministratorModel()
am
am:AdminstatorModel
updateAdminstrator

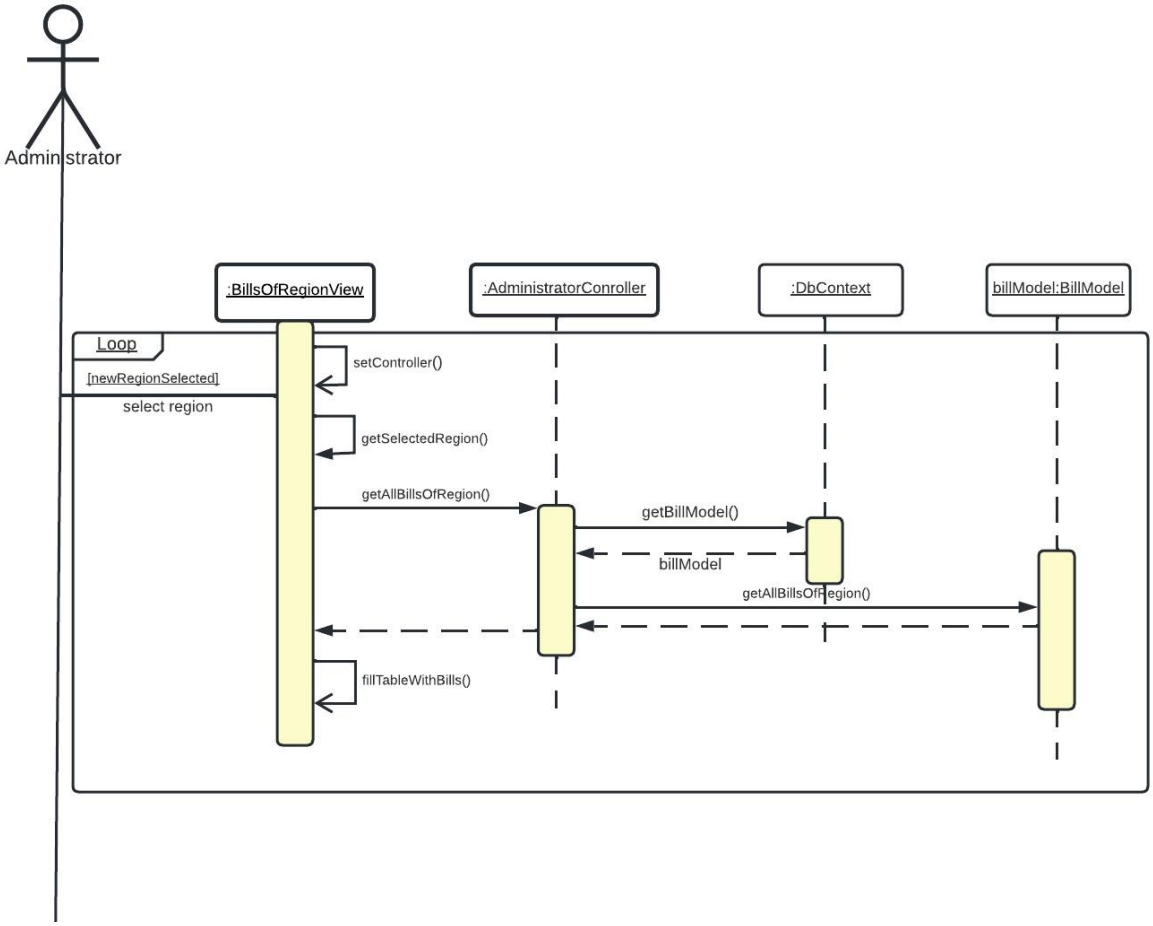

Administrator
:AdminDashboardView
press update customer
new
:MeterCodeVerificator
verifyMeterCode()
[isValid]
:DataValidator
isValidData()
:CustomerController
[isValid]updateCustomer()
:DbContext
getCustomerModel()
cm
cm:AdminstatorModel
update()

Operator Module:

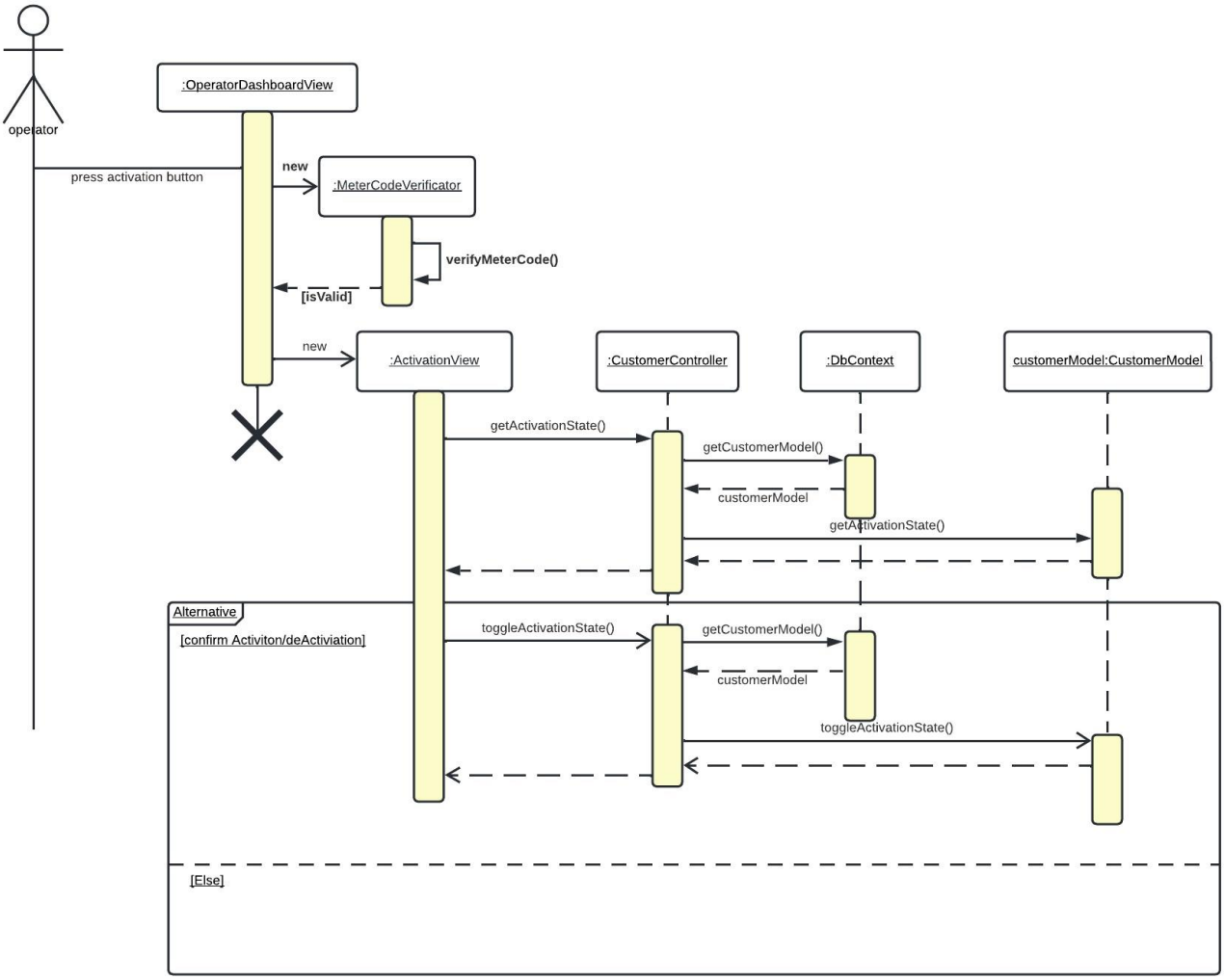
Enables operator to view all bills and collect payments from customer.



Enables operator to view all bills of a specific region

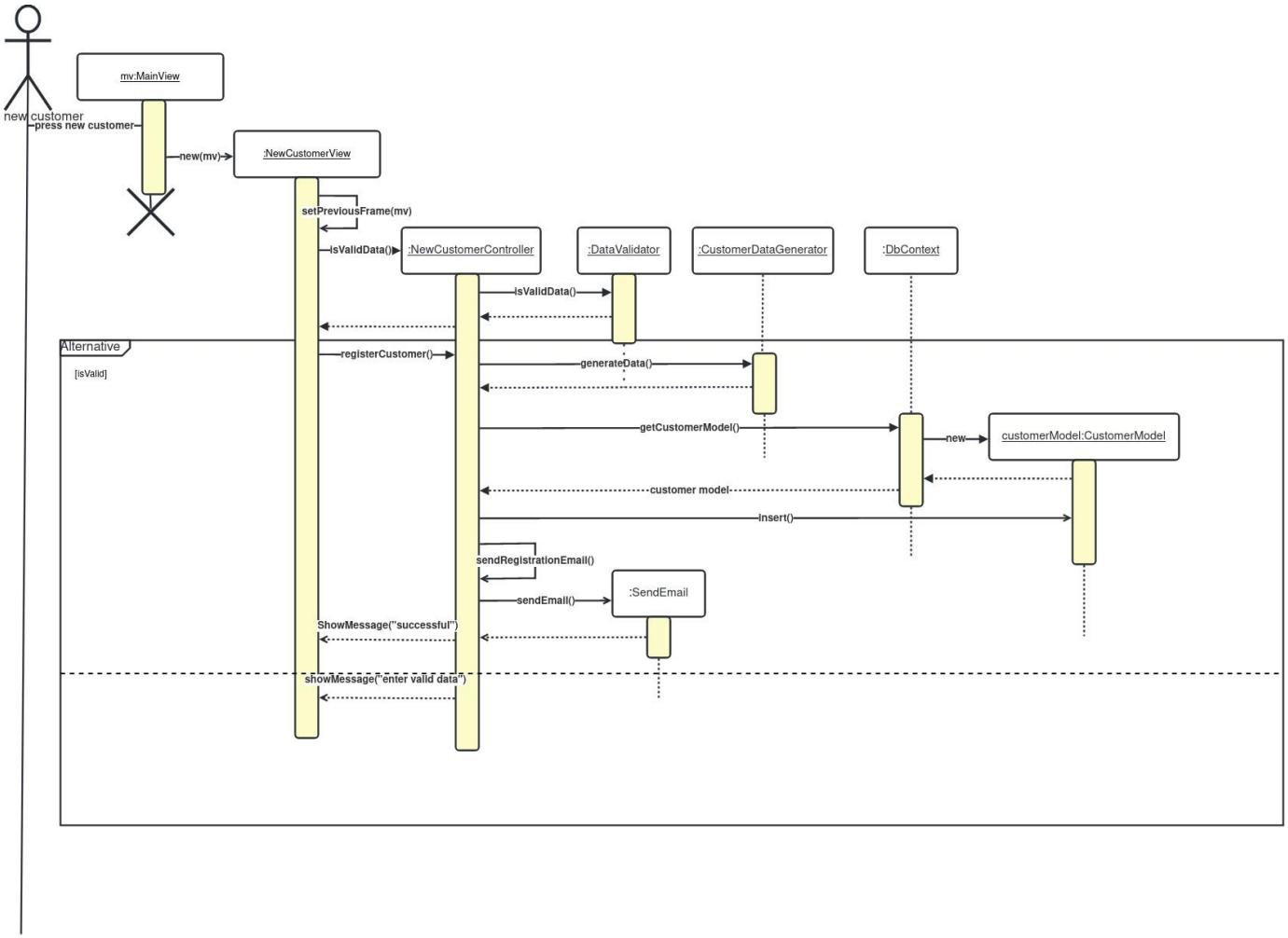


Enables operator to activate/deactivate meter or cancel subscription for customer.



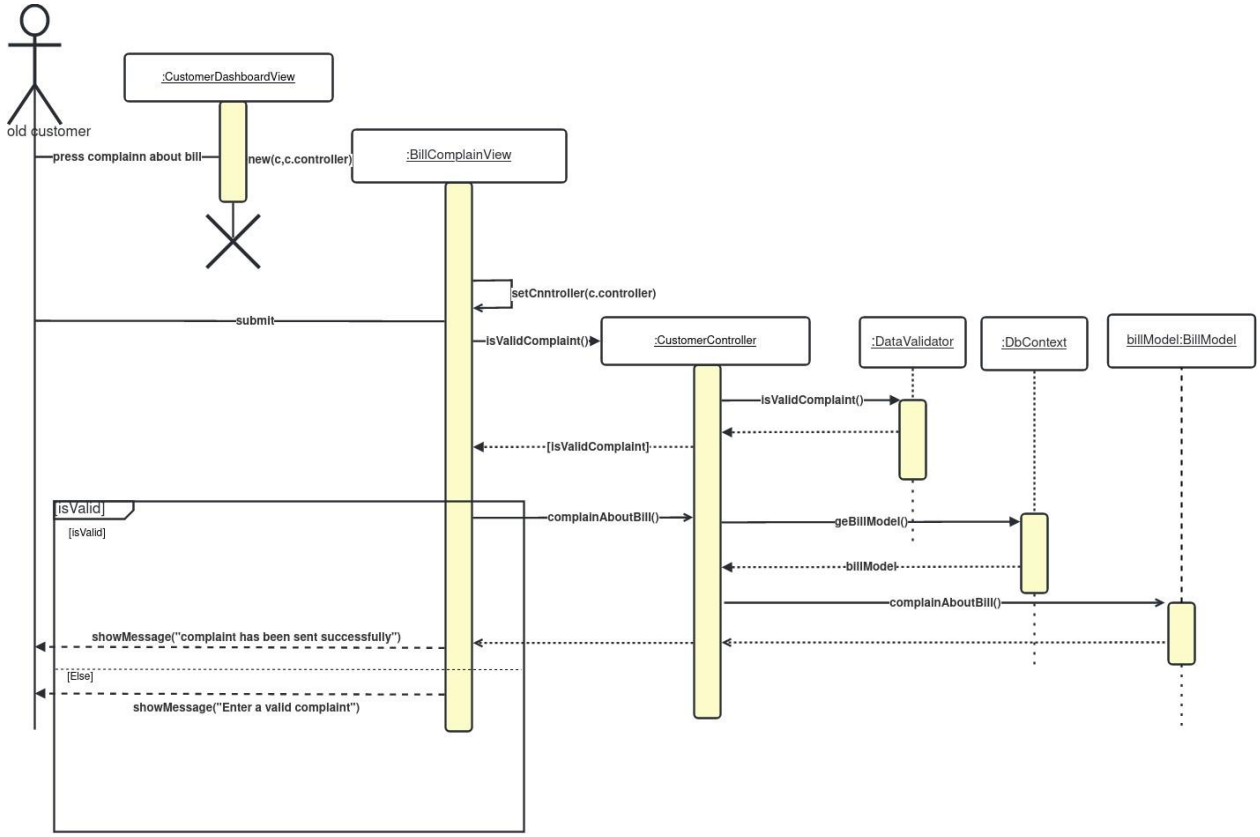
New Customer Module:

Enables customer to fill out the contract.

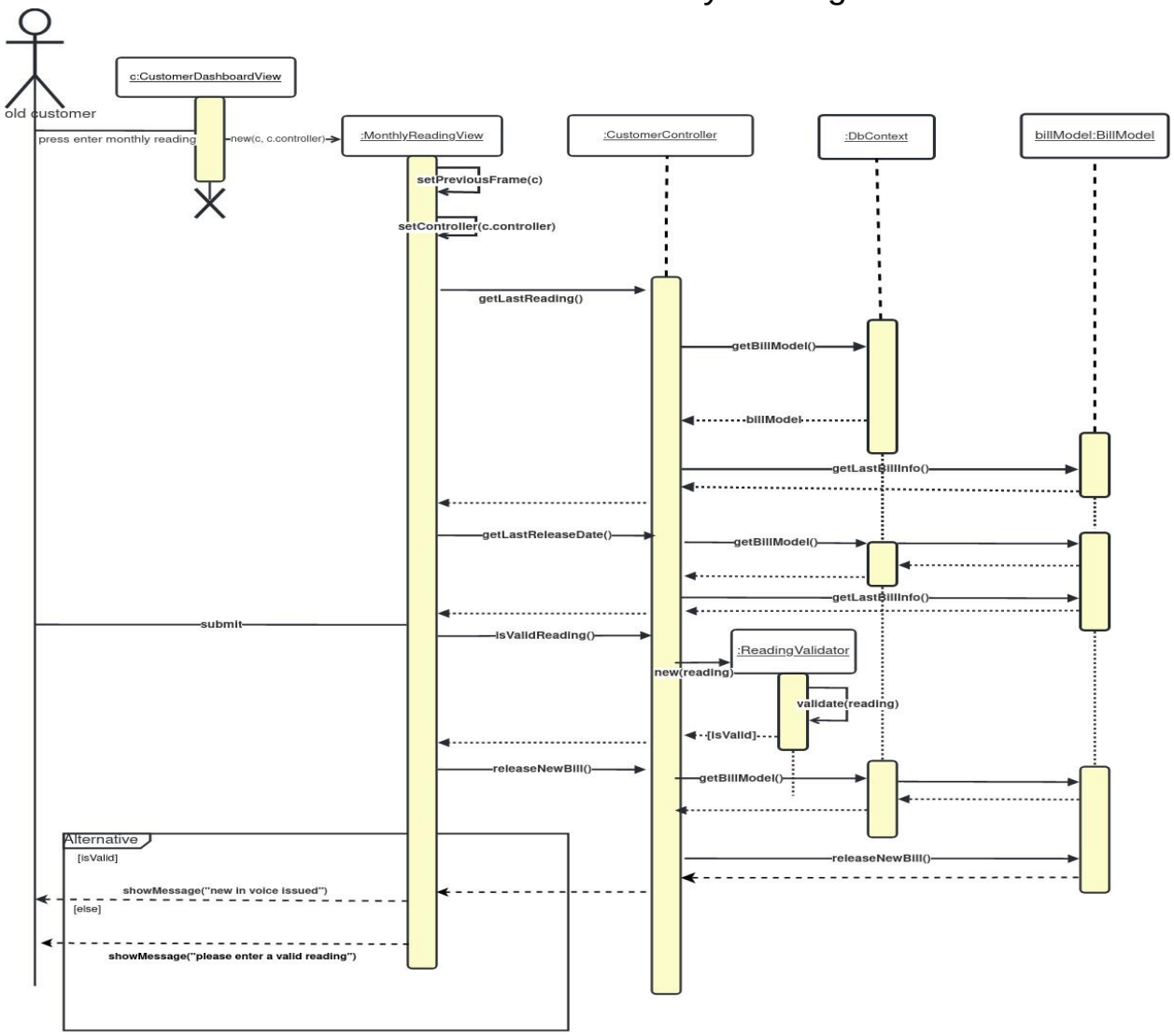


Old Customer Module:

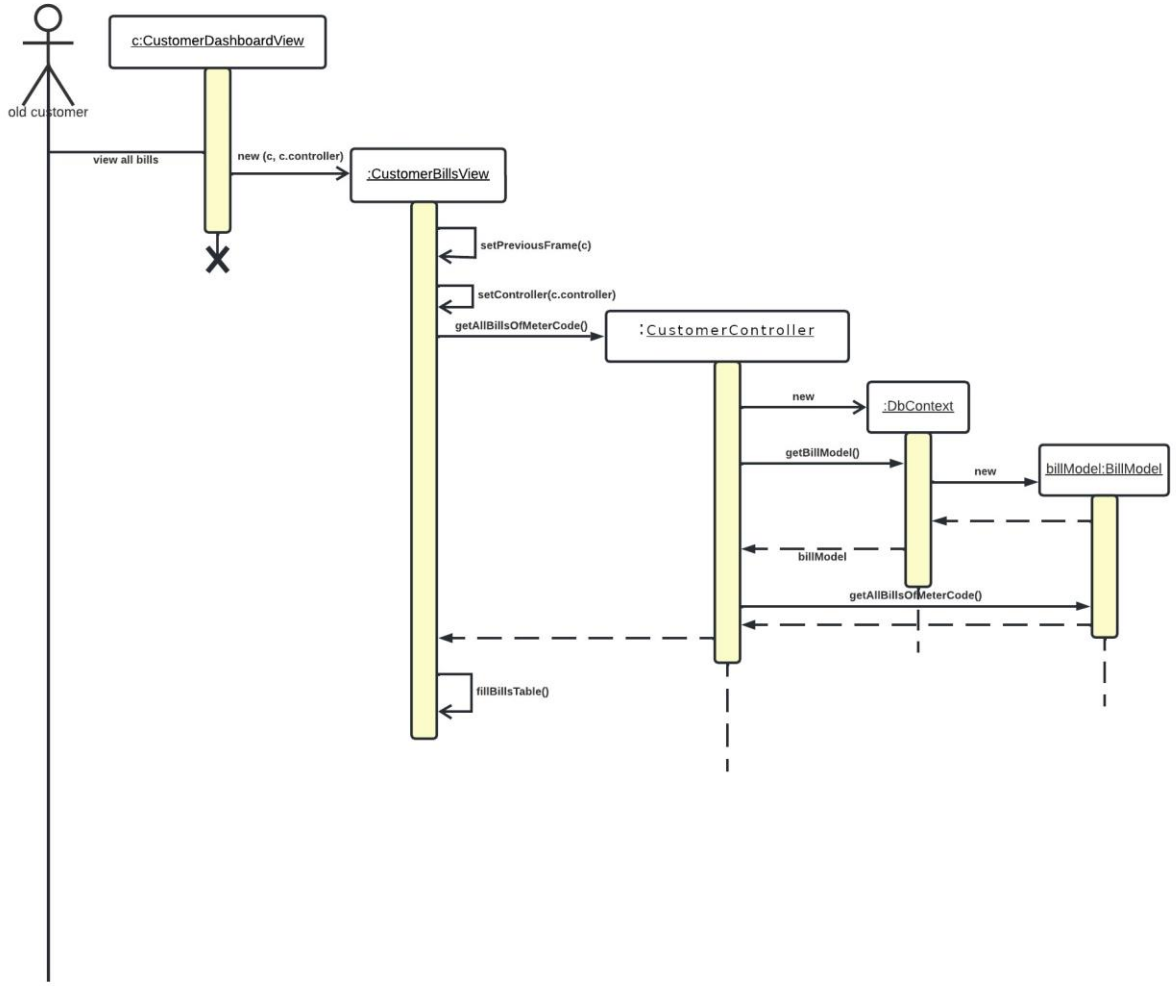
Enables customer to complain about bill with meter code.



Enables customer to enter monthly reading of meter.



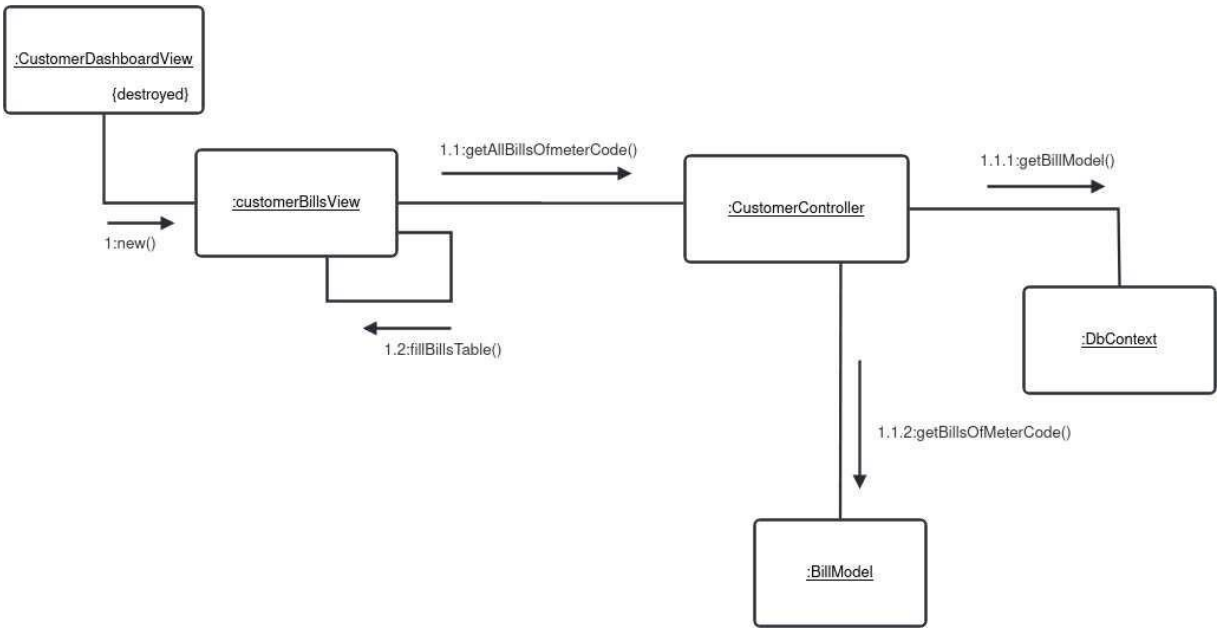
Enables customer to view all bills with meter code.



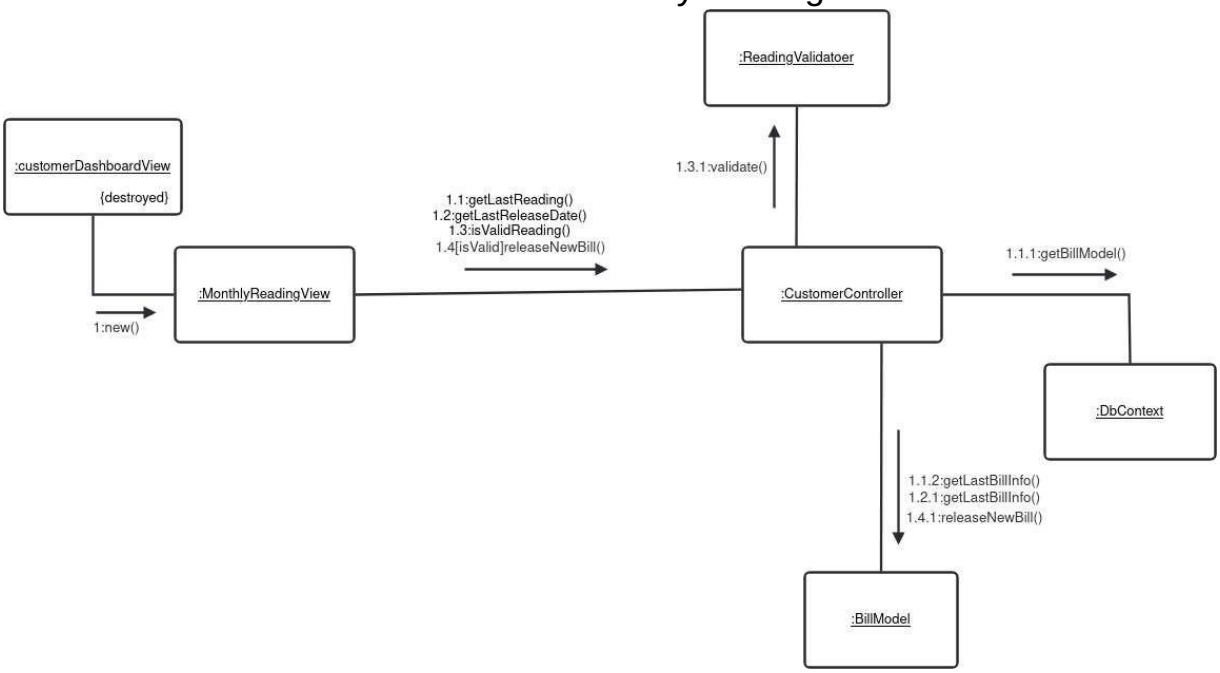
Communication Diagrams

1. **Old Customer Module**:

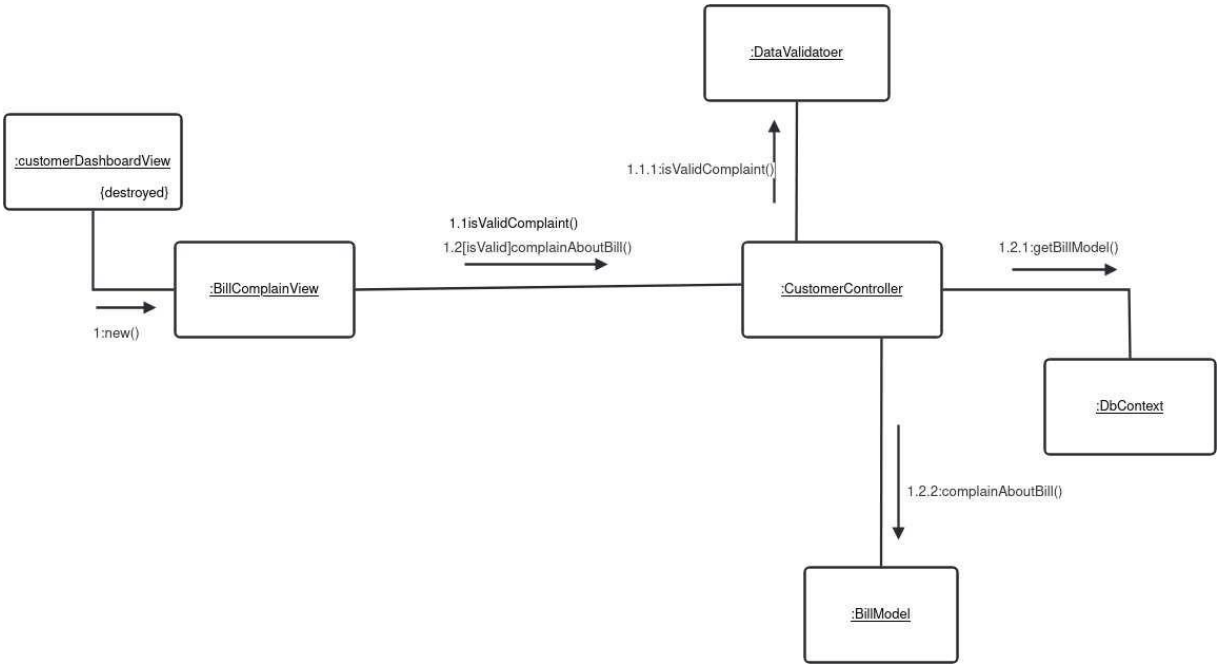
a. Enables customer to view all bills with meter code.



b. Enables customer to enter monthly reading of meter.

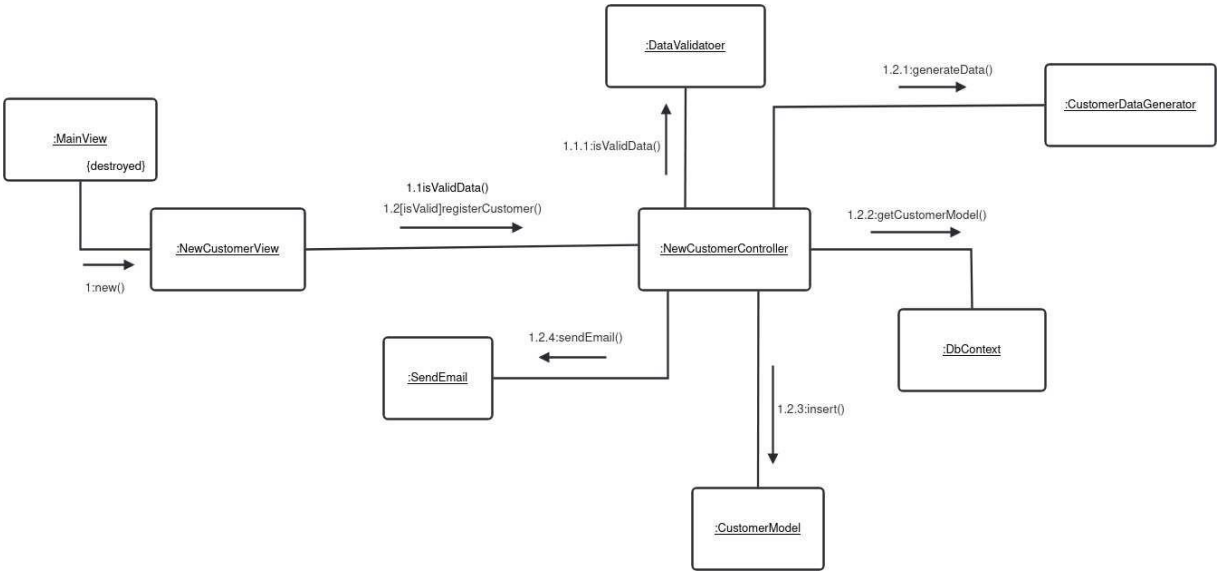


c. Enables customer to complain about bill with meter code.



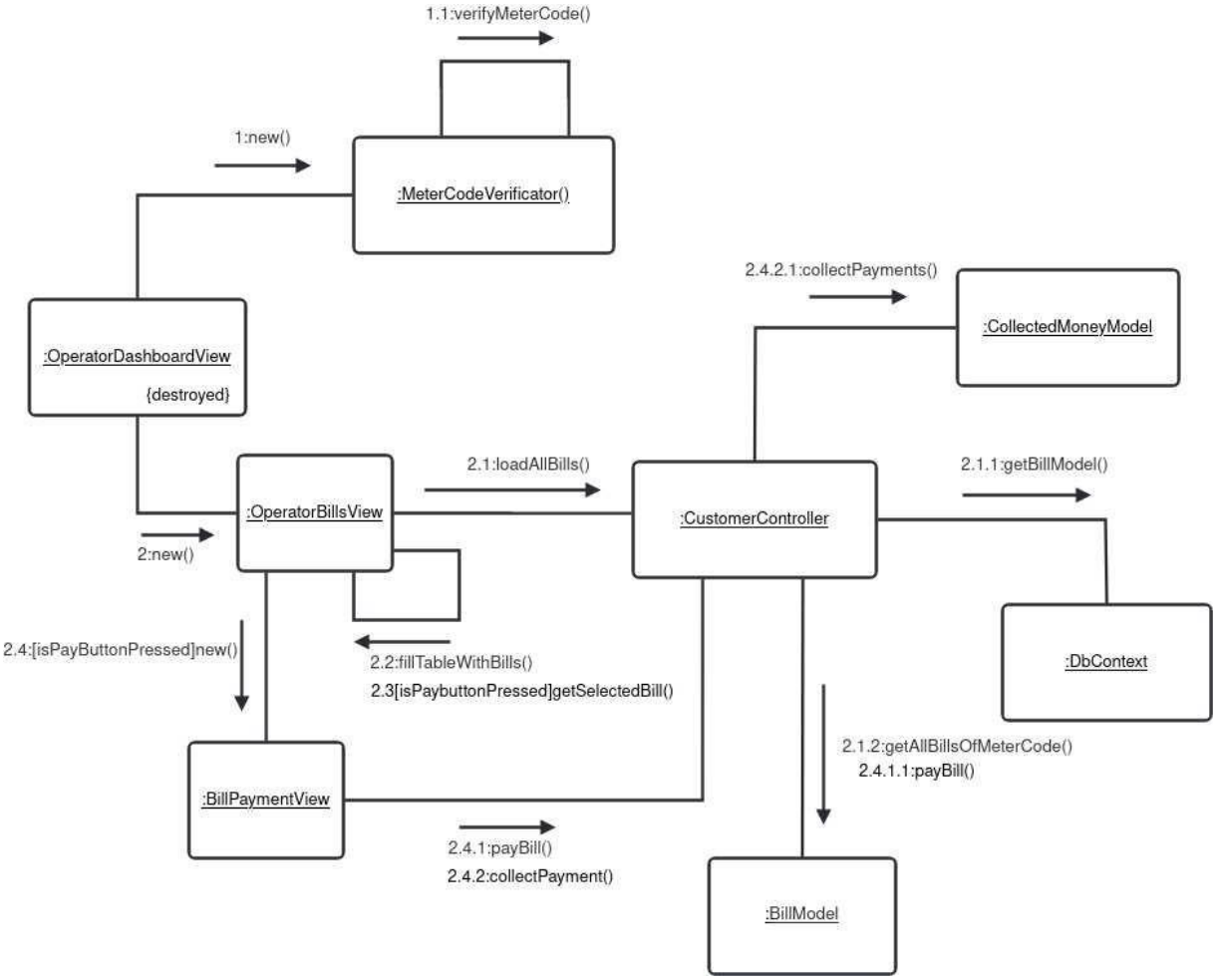
2. **New Customer Module**:

a. Enables customer to fill out the contract.

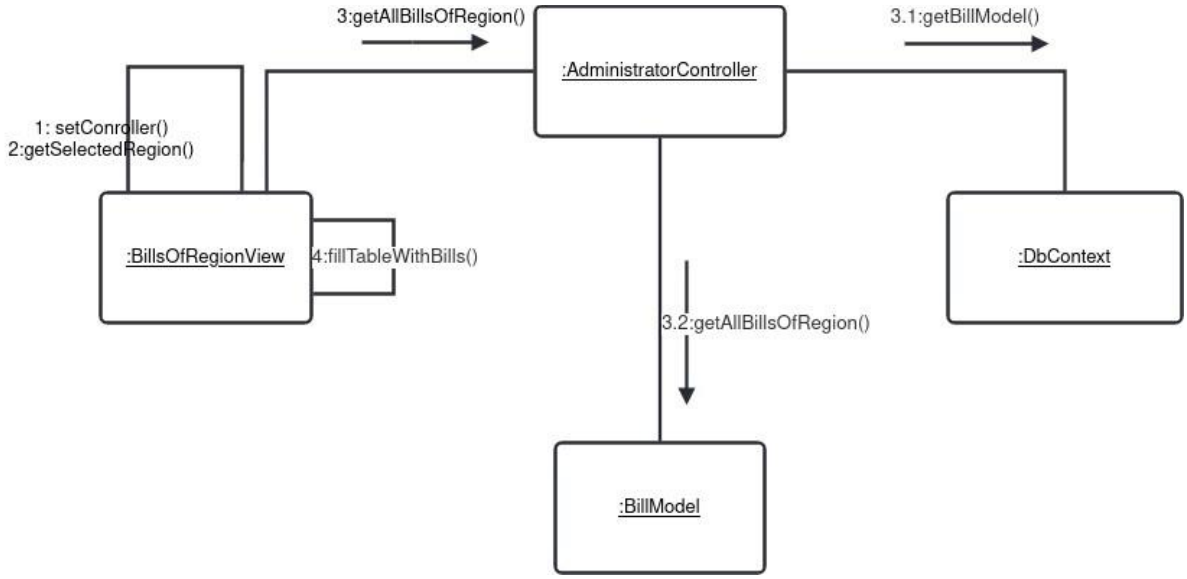


3. **Operator Module**:

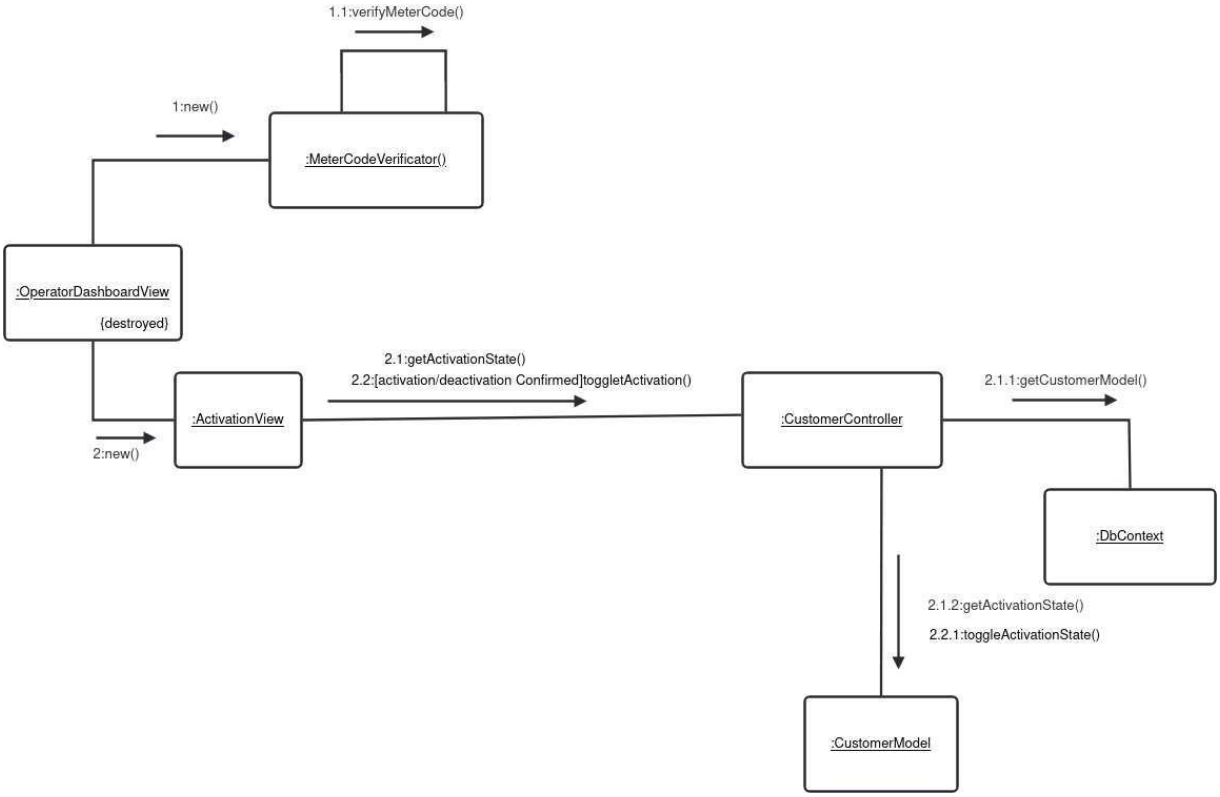
a. Enables operator to view all bills and collect payments from customer.



b. Enables operator to view all bills of a specific region.

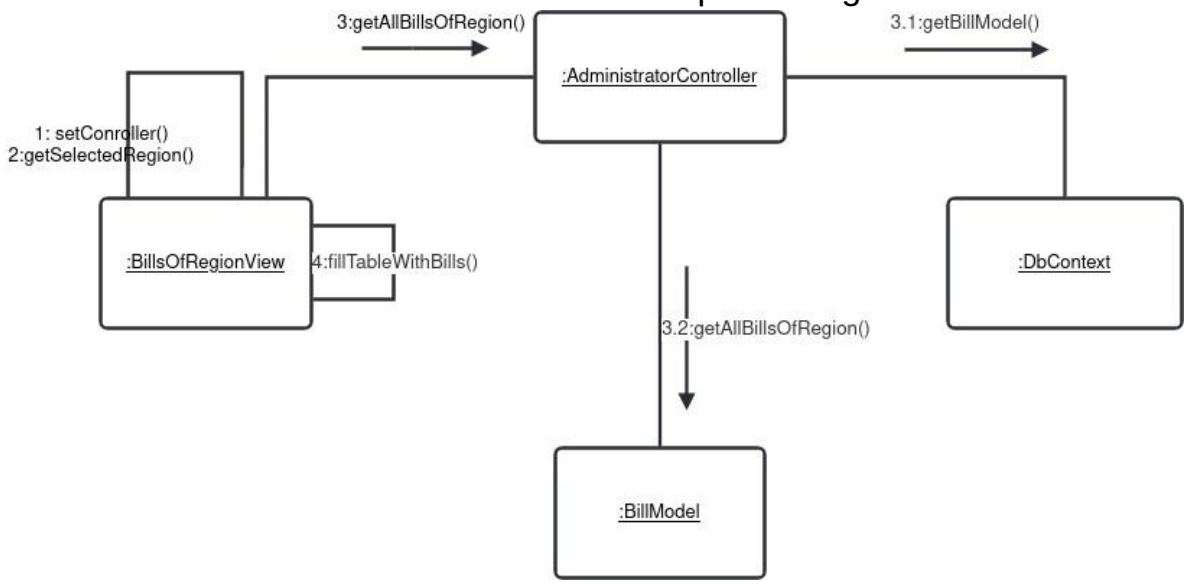


c. Enables operator to activate/deactivate meter or cancel subscription for customer.

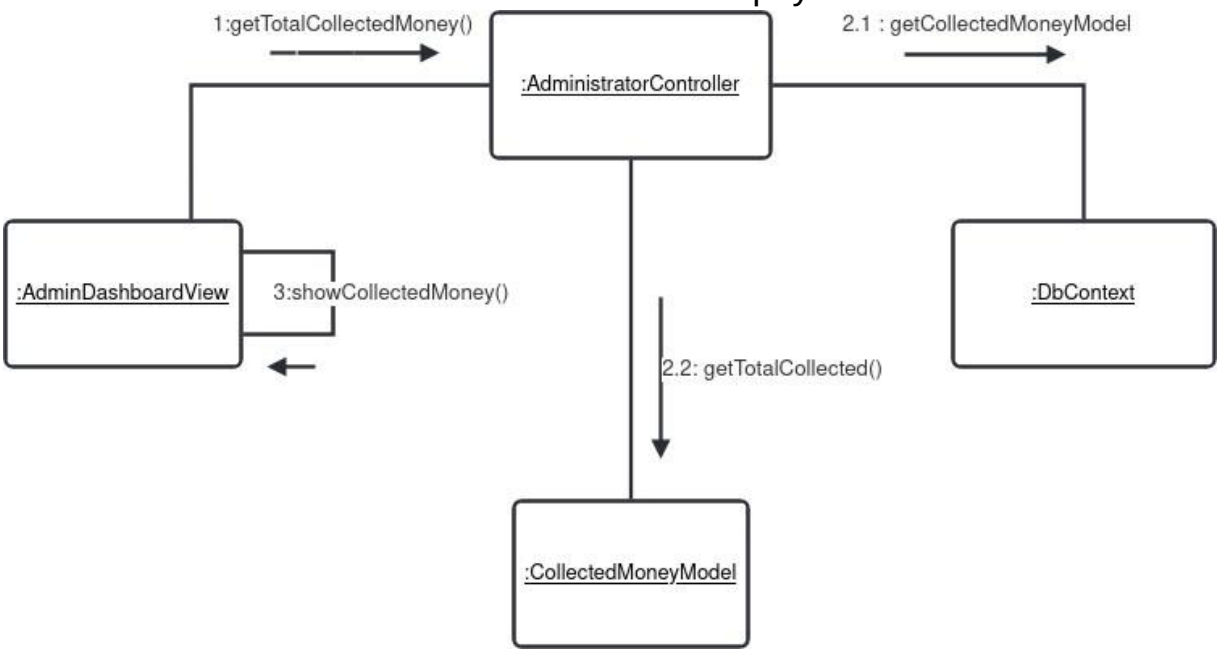


4. **Admin Module**:

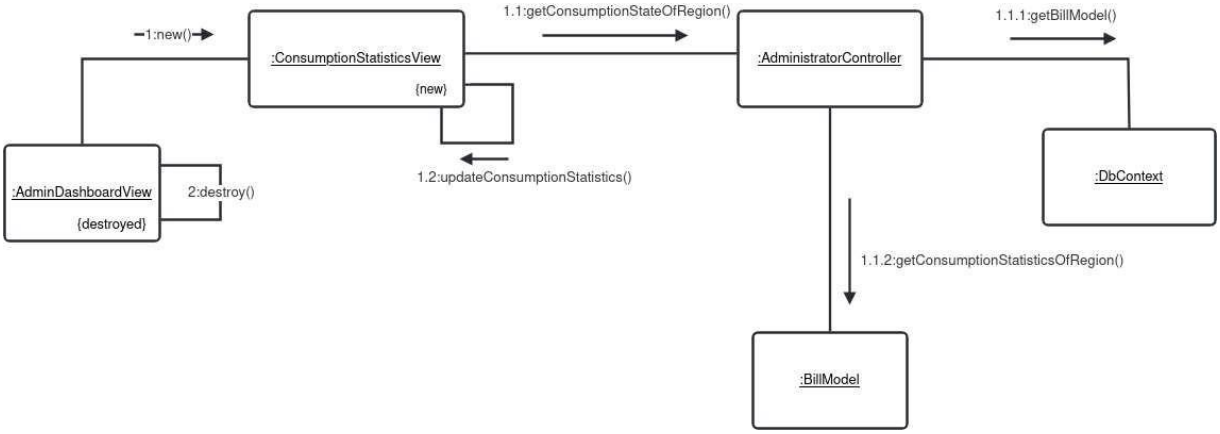
a. Enables admin to view all bills of specific region.



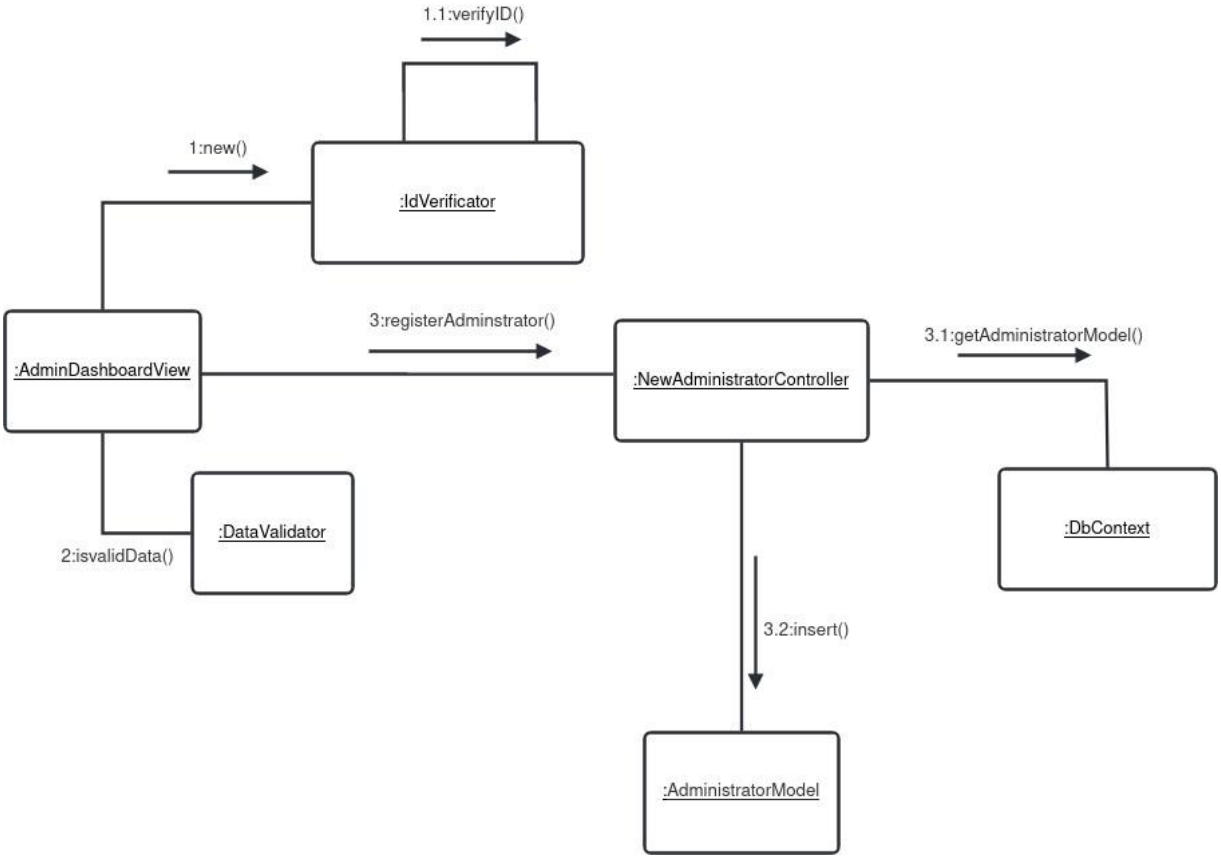
b. Enables the admin to view the total payments collected.

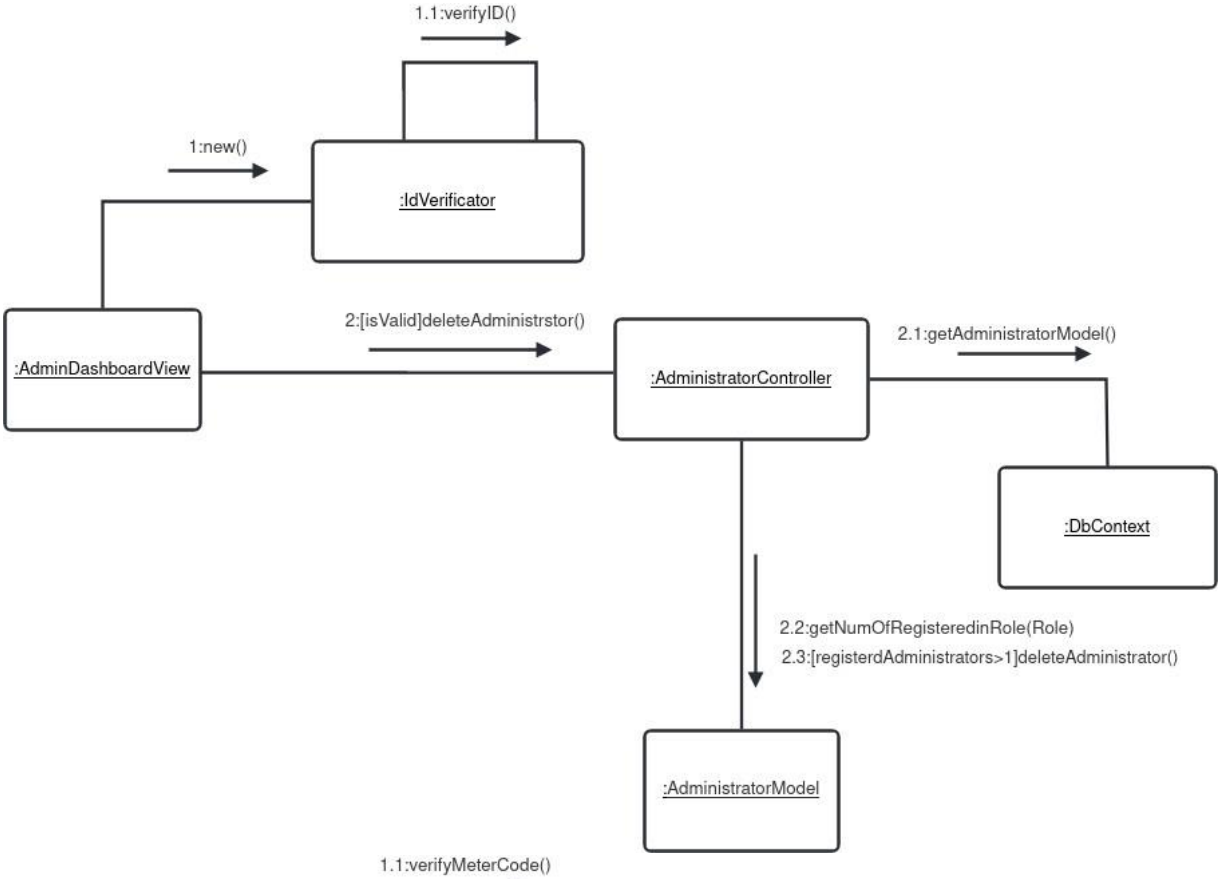


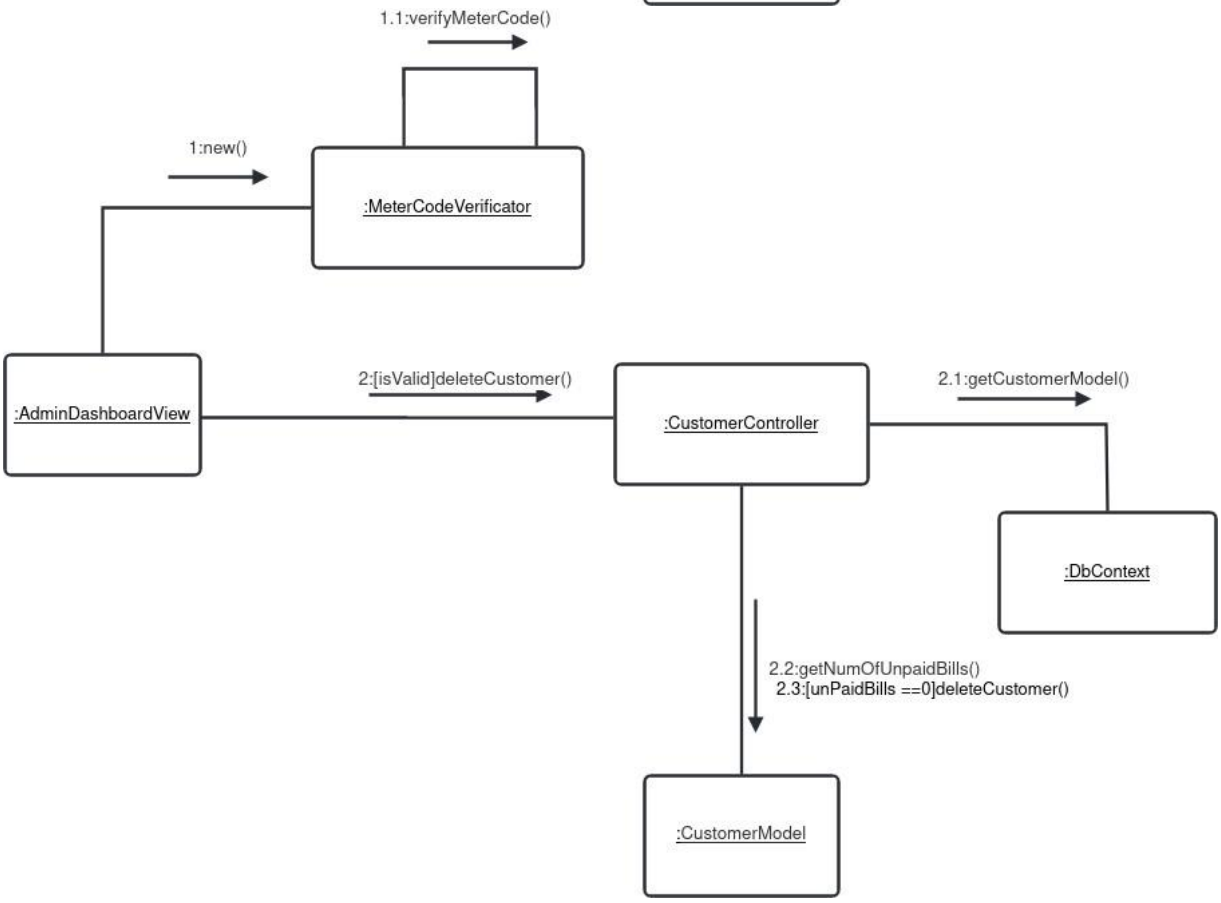
c. Enables admin to make consumption statistics for specific region.

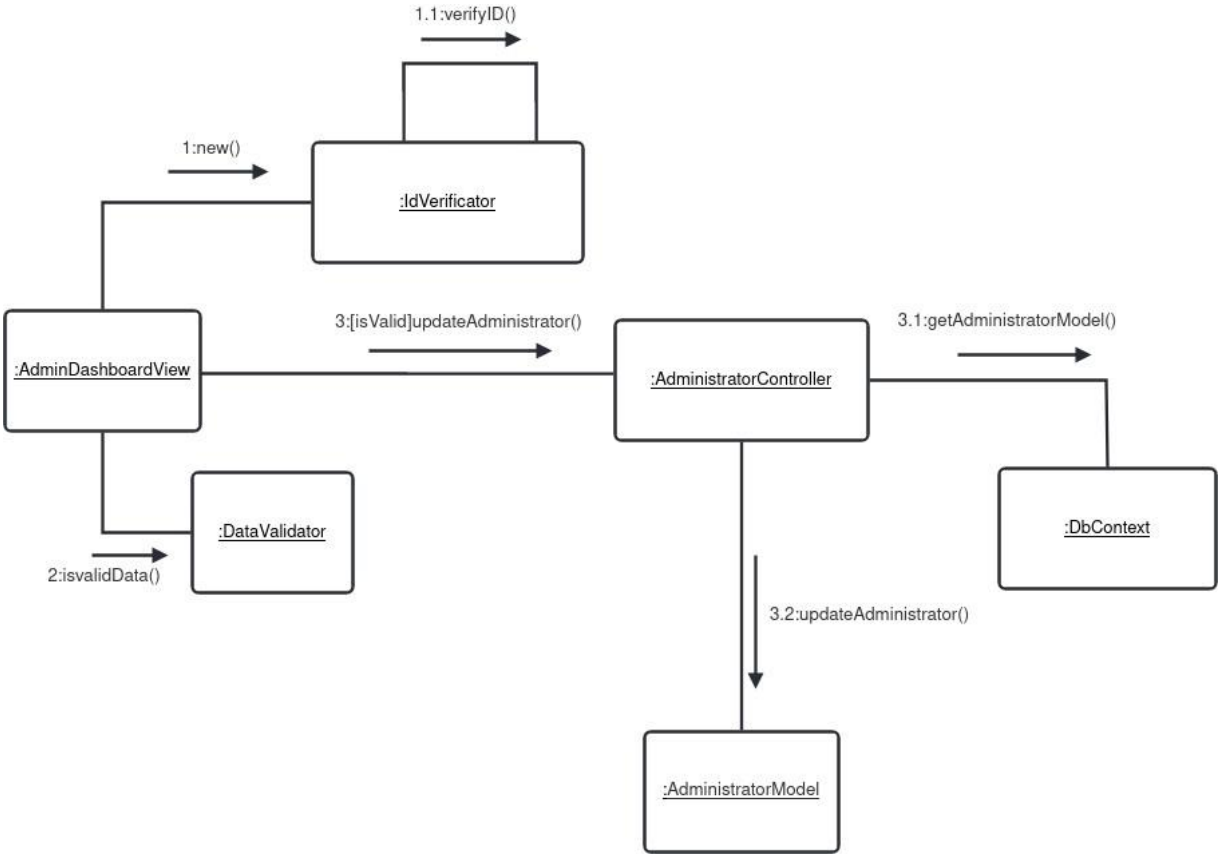


d. Enables admin to add / update / delete users with different roles.



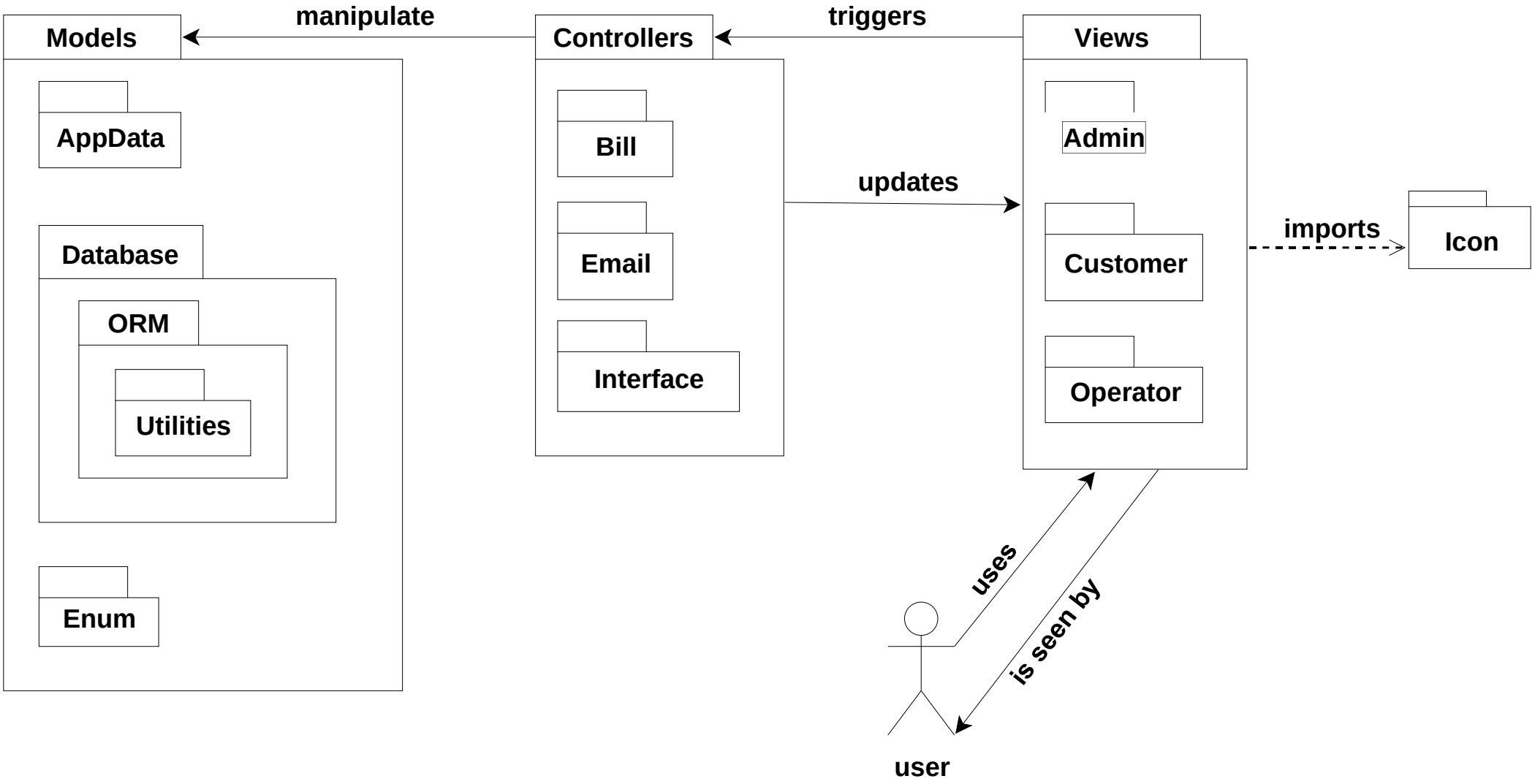

1.1:verifyID()
1:new()
:IdVerificator
:AdminDashboardView
2:[isValid]deleteAdministrstor()
:AdministratorController
2.1:getAdministratorModel()
:DbContext
2.2:getNumOfRegisteredinRole(Role)
2.3:[registerdAdministrators>1]deleteAdministrator()
:AdministratorModel


1.1:verifyMeterCode()
1:new()
:MeterCodeVerificator
:AdminDashboardView
2:[isValid]deleteCustomer()
:CustomerController
2.1:getCustomerModel()
:DbContext
2.2:getNumOfUnpaidBills()
2.3:[unPaidBills ==0]deleteCustomer()
:CustomerModel

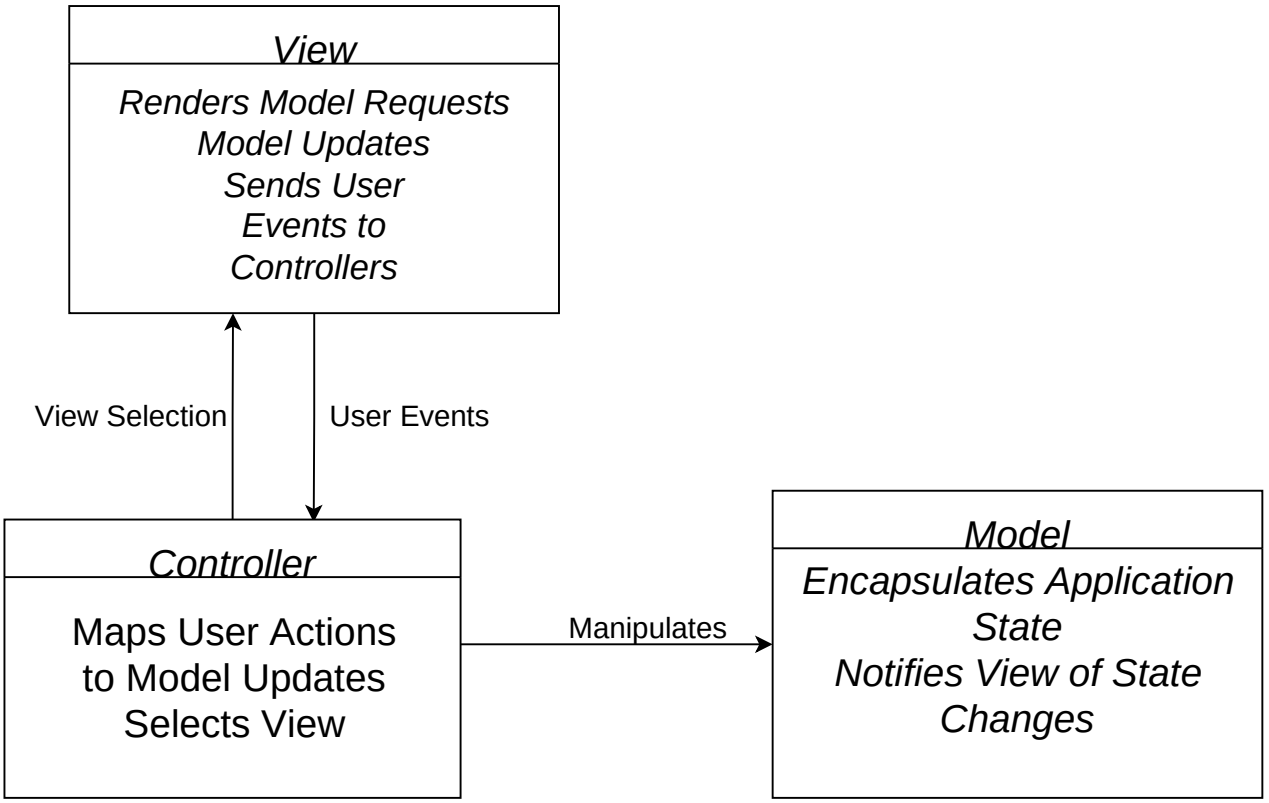

1.1:verifyID()
1:new()
:IdVerificator
:AdminDashboardView
3:[isValid]updateAdministrator()
:AdministratorController
3.1:getAdministratorModel()
:DataValidator
2:isvalidData()
:DbContext
3.2:updateAdministrator()
:AdministratorModel

Package Diagram

Package Diagram

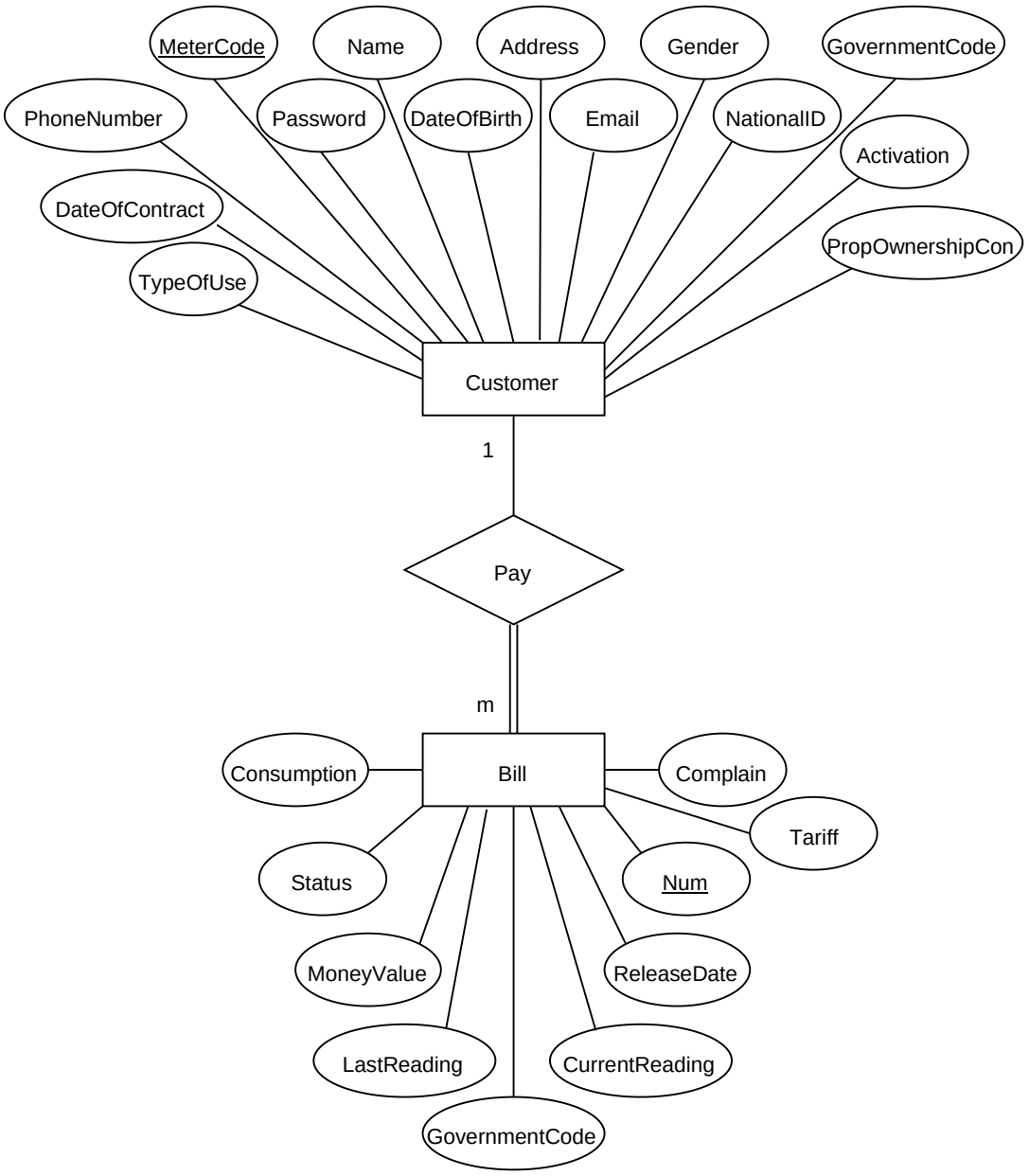


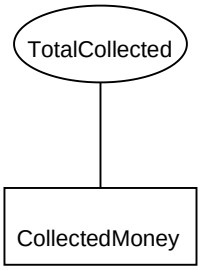
System architecture Diagram

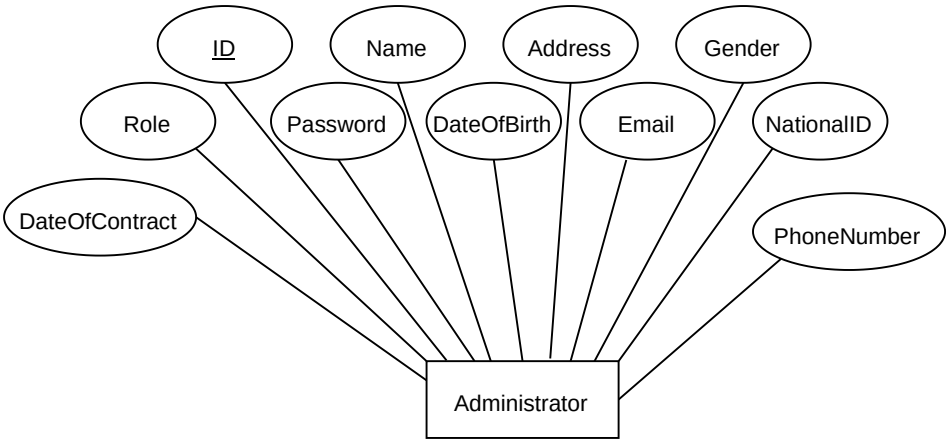

View
Renders Model Requests
Model Updates
Sends User
Events to
Controllers
View Selection
User Events
Controller
Maps User Actions
to Model Updates
Selects View
Manipulates
Model
Encapsulates Application
State
Notifies View of State
Changes

ERD Diagram

ERD Model


MeterCode
Name
Address
Gender
GovernmentCode
PhoneNumber
Password
DateOfBirth
Email
NationalID
Activation
DateOfContract
PropOwnershipCon
TypeOfUse
Customer
1
Pay
m
Consumption
Bill
Complain
Tariff
Status
Num
MoneyValue
ReleaseDate
LastReading
CurrentReading
GovernmentCode


TotalCollected
CollectedMoney


ID
Name
Address
Gender
Role
Password
DateOfBirth
Email
NationalID
DateOfContract
PhoneNumber
Administrator

Design Patterns

Design Patterns

- Singleton

The **singleton pattern** is a **creational** software design pattern that restricts the instantiation of a class to one "single" instance. This is useful when exactly one object is needed to coordinate actions across the system.

Overview:

The singleton design pattern solves problems by allowing it to

- Ensure that a class only has one instance
- Easily access the sole instance of a class
- Control its instantiation
- Restrict the number of instances
- Access a global variable

The singleton design pattern describes how to solve such problems:

- Hide the constructors of the class.
- Define a public static operation (`getInstance()`) that returns the sole instance of the class.

In essence, the singleton pattern forces it to be responsible for ensuring that it is only instantiated once. A hidden constructor - declared `private` or `protected` - ensures that the class can never be instantiated from outside the class. The public static operation can be accessed by using the class name and operation name, `Singleton.getInstance()` e.g.,.

Implementation:

Implementations of the singleton pattern must:

- Ensure that *only one instance* of the singleton class ever exists; and ● Provide *global access* to that instance.

Typically, this is done by:

- Declaring all constructors of the class to be private; and
- Providing a static method that returns a reference to the instance.

The instance is usually stored as a private static variable.

- Model–view–controller:

Example from our code:

There must be one single alive connection to the database, so we created a `DatabaseConnection` constructor private to ensure that no other class will be able to create an instance of the `DatabaseConnection` class. Also we have defined a public static method, the first time this method is called, it creates a single instance of the class `DatabaseConnection` and stores a connection from the `DatabaseConnection` class.

Code:

```
// Singleton pattern
private static Connection uniqueInstance;

private DatabaseConnection(){

}

static Connection getInstance(ConnectionString connStrings){

    try {

        if(uniqueInstance == null || uniqueInstance.isClosed())
            uniqueInstance = new DatabaseConnection().getConnection(connStrings);

    } catch (SQLException ex) {
        System.out.println(ex);
    }

    return uniqueInstance;
}
```

Model–view–controller (**MVC**) is an **architectural** software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is presented to and accepted from the user.

Components:

- **Model:**

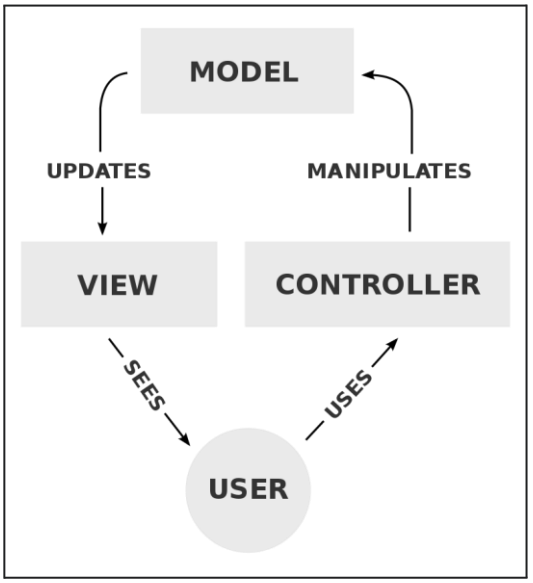
The central component of the pattern. It is the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the controller.

- **View**

  Any representation of information such as a chart, diagram or table. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

- **Controller**

  Accepts input and converts it to commands for the model or view. Also it’s responsible for the logic and rules of the application.



In addition to dividing the application into these components, the model–view–controller design defines the interactions between them.

- The model is responsible for managing the data of the application. It receives user input from the controller.
- The view renders the presentation of the model in a particular format.
- The controller responds to the user input and performs interactions on the data model objects. The controller receives the input, optionally validates it and then passes the input to the model.

As with other software patterns, MVC expresses the "core of the solution" to a problem while allowing it to be adapted for each system.Particular MVC designs can vary significantly from the traditional description here.

<u>Example from our code:</u>

- **Model (Data)**

The model's job is to simply manage the data. Whether the data is from a database or a JSON file, the model is responsible for managing it.

In the EBS (Electricity Billing System) application, the model object contains a database with all the customers and admins information (data) needed for the app. It also contains a JSON reader for all the information needed for the database and email configurations.

- **View**

The view's job is to decide what the user will see on their screen, and how. The EBS (Electricity Billing System) has a lot of views like : AdministratorLoginView, BillsOfRegionView .. etc.

All of the views have critical functions that define what each view wants to do. These functions are where the app decides what the user will see and how.

- **Controller**

The controller's responsibility is to pull, modify, and provide data to the user. Essentially, the controller is the link between the view and model.

Through getter and setter functions, the controller pulls data from the model and initializes the views.If there are any updates from the views, it modifies the data with a setter function.

Also it's responsible for the logic and rules of the application as above we used it to validate the name.

```
public NewCustomerControllerImp(View view, DbContext dbContext){

    this.view = view;
    this.dbContext = dbContext;

    this.generator = new CustomerDataGenerator();
    this.validator = new DataValidator();

    this.registerInView();
}

@Override
public final void registerInView(){

    if(this.view != null)
        this.view.setController(this);
}

@Override
public boolean isValidName(String name) {

    return this.validator.isValidName(name);
}
```

- Builder:

The **builder pattern** is a **creational** design pattern designed to provide a flexible solution to various object creation problems in object-oriented programming. The intent of the Builder design pattern is to separate the construction of a complex object from its representation.

The intent of the Builder design pattern is to separate the construction of a complex object from its representation. By doing so, the same construction process can create different representations.

Overview:

The Builder design pattern is one of the *GoF design patterns* that describe how to solve recurring design problems in object-oriented software.

The Builder design pattern solves problems like:

- How can a class (the same construction process) create different representations of a complex object?
- How can a class that includes creating a complex object be simplified?

Creating and assembling the parts of a complex object directly within a class is inflexible. It commits the class to creating a particular representation of the complex object and makes it impossible to change the representation later independently from (without having to change) the class.

The Builder design pattern describes how to solve such problems:

- Encapsulate creating and assembling the parts of a complex object in a separate `Builder` object.
- A class delegates object creation to a `Builder` object instead of creating the objects directly.

A class (the same construction process) can delegate to different `Builder` objects to create different representations of a complex object.

Advantages:

Advantages of the Builder pattern include:

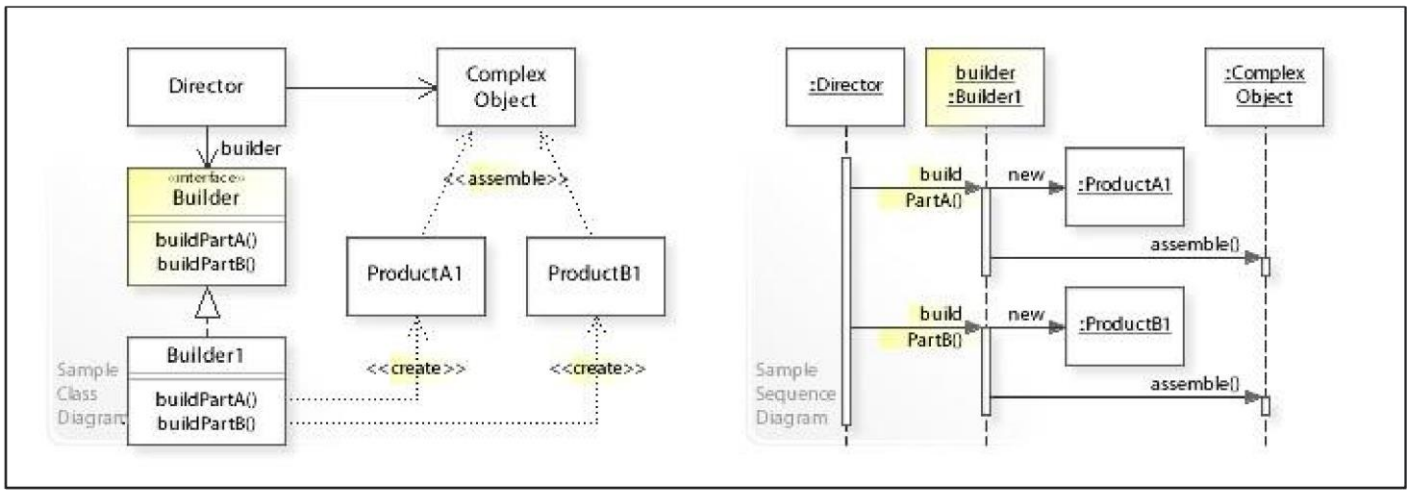
- Allows you to vary a product's internal representation.
- Encapsulates code for construction and representation.
- Provides control over steps of the construction process.

Disadvantages:

Disadvantages of the Builder pattern include:

- A distinct ConcreteBuilder must be created for each type of product.
- Builder classes must be mutable.
- May hamper/complicate dependency injection.

Structure:



In the above UML class diagram, the `Director` class doesn't create and assemble the `ProductA1` and `ProductB1` objects directly. Instead, the `Director` refers to the `Builder` interface for building (creating and assembling) the parts of a complex object, which makes the `Director` independent of which concrete classes are instantiated (which representation is created). The `Builder1` class implements the `Builder` interface by creating and assembling the `ProductA1` and `ProductB1` objects.

The UML sequence diagram shows the run-time interactions: The `Director` object calls `buildPartA()` on the `Builder1` object, which creates and assembles the `ProductA1` object. Thereafter, the `Director` calls `buildPartB()` on `Builder1`, which creates and assembles the `ProductB1` object.

Example from our code:

Consider the construction of a `Select` statement. `Select Query` is the final end product (object) that is to be returned as the output of the construction process. It will have many steps like `tableName` construction, `fields` construction, and so on `whereCondition` construction.

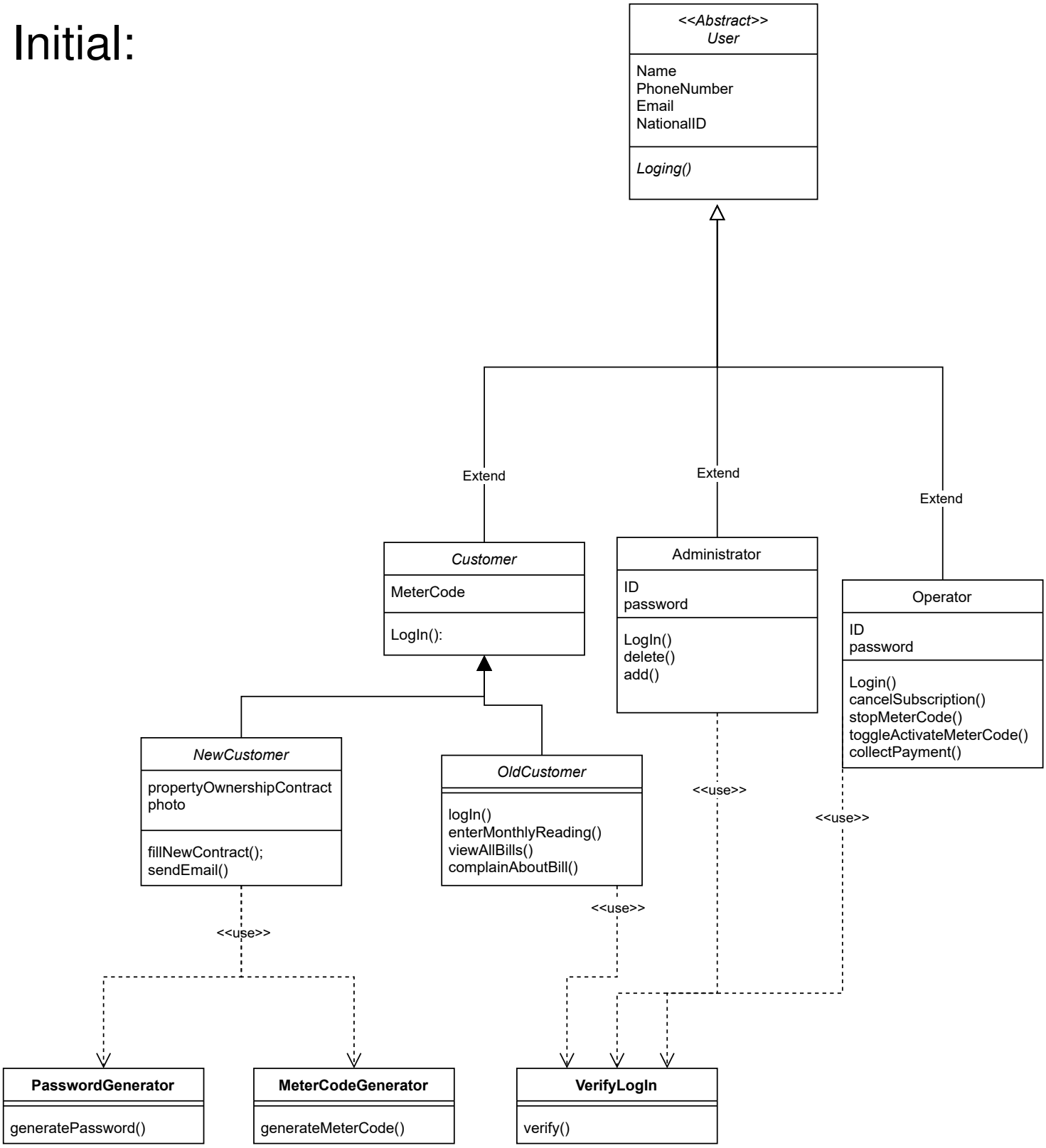
Finally, the whole `Select` object is returned.

Here using the same process you can build `Select` queries with different properties.

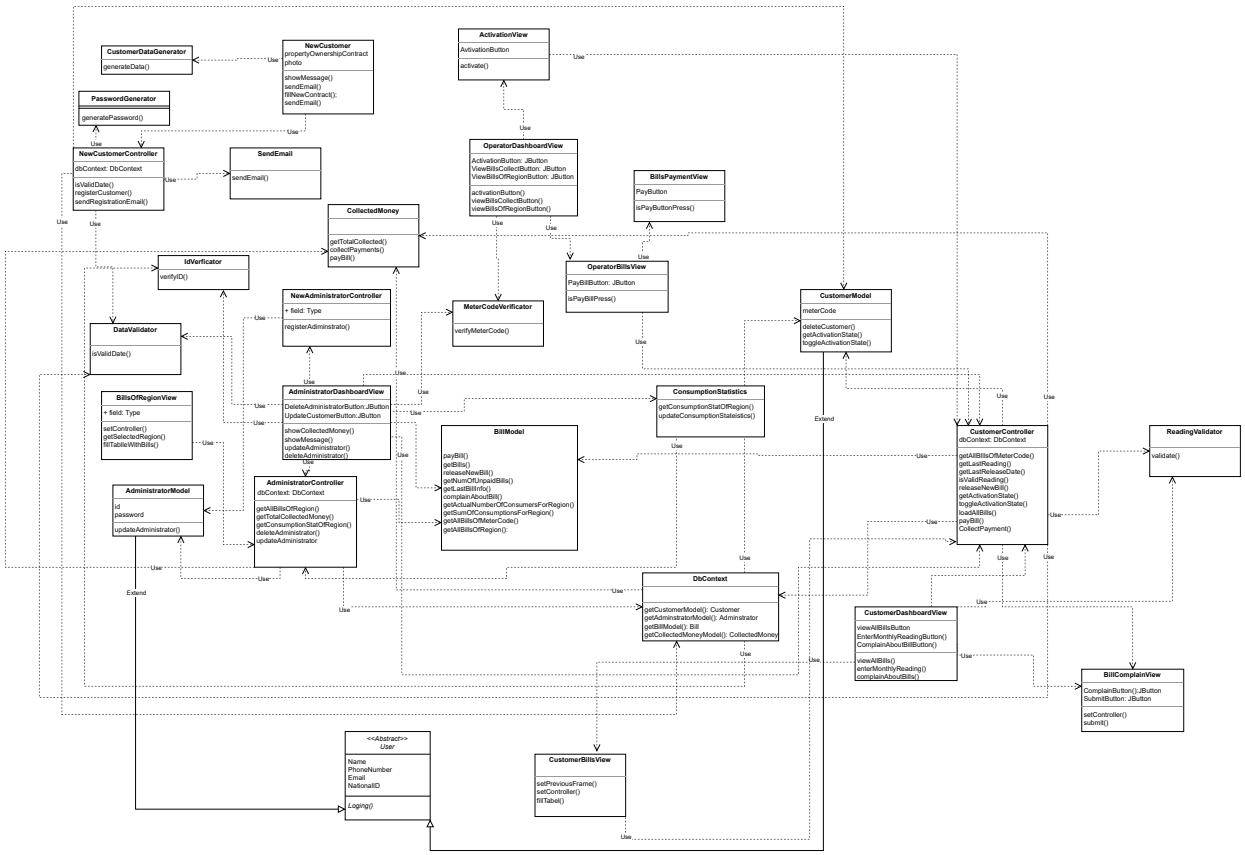
```
public class SelectBuilder extends DML {

    private final String table;
    private final List<Object> fields;
    private String where, orderBy, limit;

    public SelectBuilder(List<Object> fields, Enum table){

        this.table = table.name();
        this.fields = fields;
        this.where = "";
        this.orderBy = "";
        this.limit = "";
    }



    public SelectBuilder where(Enum field, String operator, Object value){

        this.where += super.Where(field, operator, value);
        return this;
    }
```

Class Diagrams

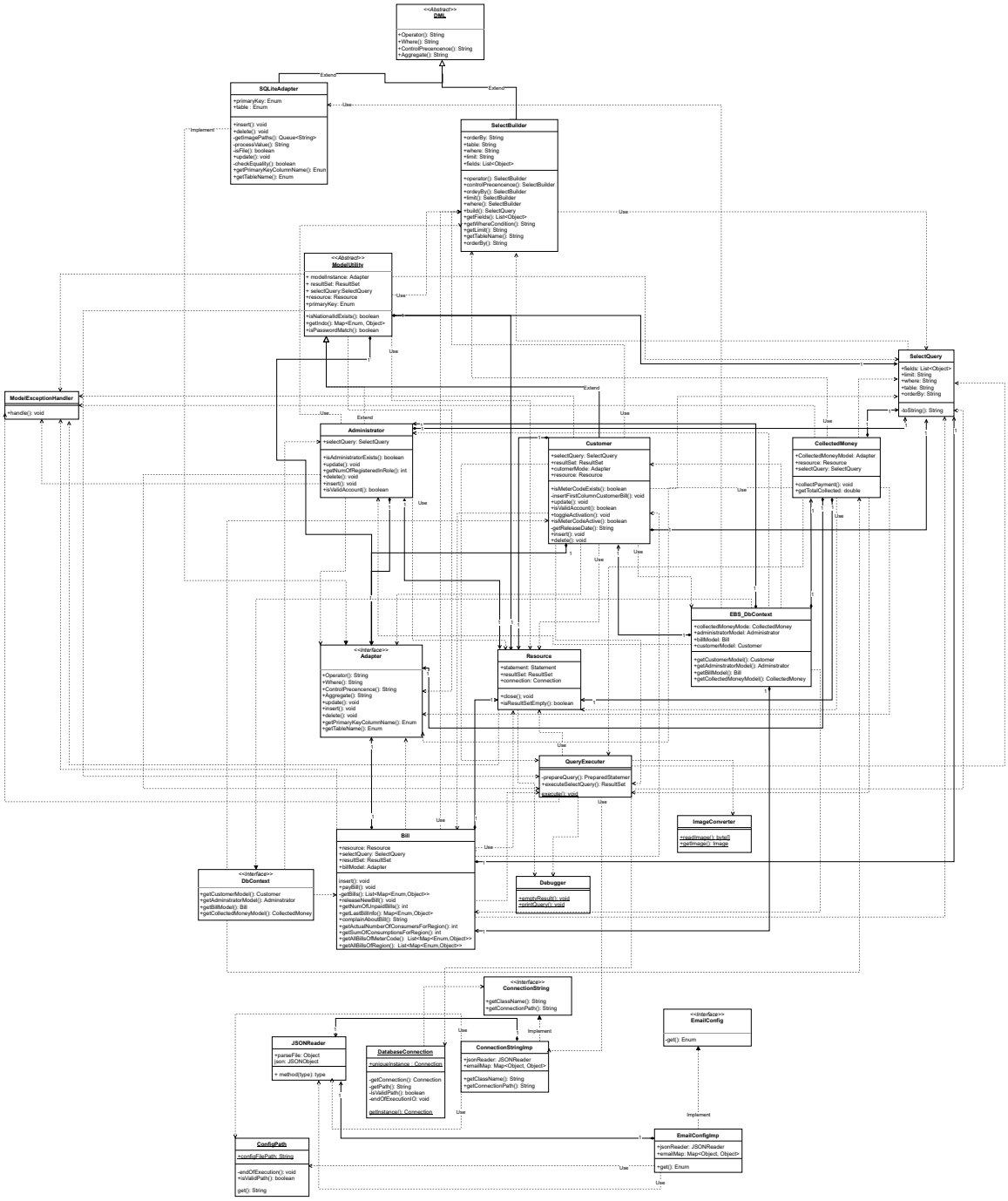
Initial:


<<Abstract>>
User
Name
PhoneNumber
Email
NationalID
Loging()
Extend
Extend
Extend
Customer
MeterCode
LogIn():
Administrator
ID
password
LogIn()
delete()
add()
Operator
ID
password
Login()
cancelSubscription()
stopMeterCode()
toggleActivateMeterCode()
collectPayment()
NewCustomer
propertyOwnershipContract
photo
fillNewContract();
sendEmail()
OldCustomer
logIn()
enterMonthlyReading()
viewAllBills()
complainAboutBill()
<<use>>
<<use>>
<<use>>
<<use>>
PasswordGenerator
generatePassword()
MeterCodeGenerator
generateMeterCode()
VerifyLogIn
verify()

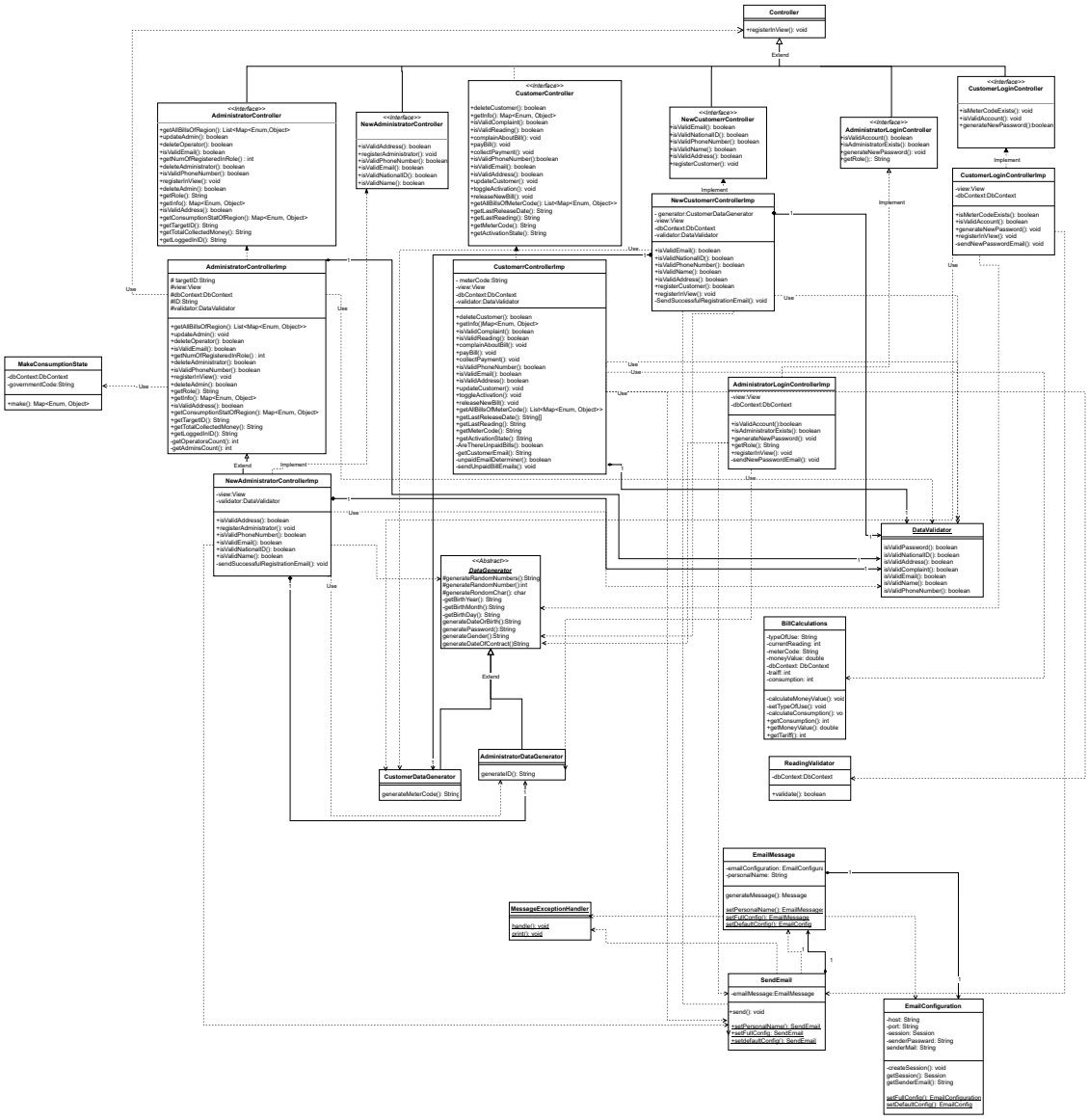
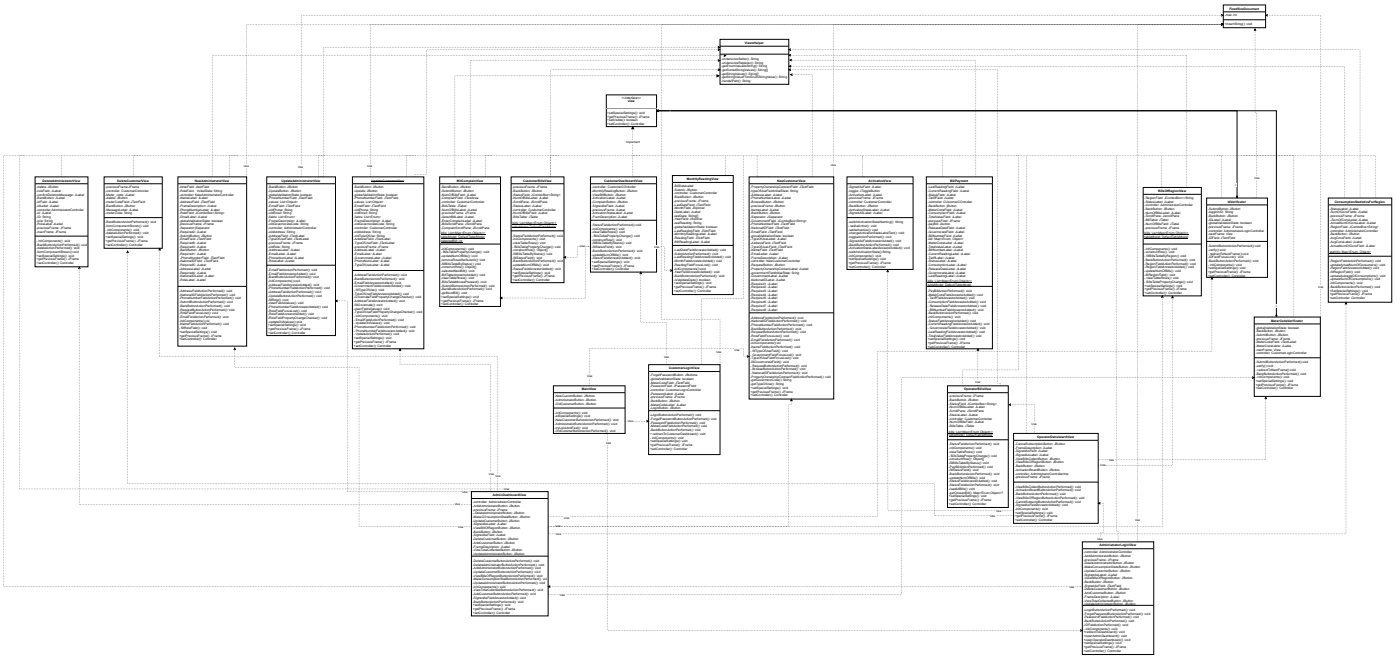
Intermediate:



Models:


<<Abstract>>
DML
+Operator(): String
+Where(): String
+ControlPrecenceence(): String
+Aggregate(): String
Extend
Extend
SQLiteAdapter
+primaryKey: Enum
+table : Enum
+insert(): void
+delete(): void
-getImagePaths(): Queue<String>
-processValue(): String
-isFile(): boolean
+update(): void
-checkEquality(): boolean
+getPrimaryKeyColumnName(): Enun
+getTableName(): Enum
Use
Implement
SelectBuilder
+orderBy: String
+table: String
+where: String
+limit: String
+fields: List<Object>
+operator(): SelectBuilder
+controlPrecenceence(): SelectBuilder
+orderBy(): SelectBuilder
+limit(): SelectBuilder
+where(): SelectBuilder
+build(): SelectQuery
-getFields(): List<Object>
+getWhereCondition(): String
-getLimit(): String
+getTableName(): String
+orderBy(): String
Use
<<Abstract>>
ModelUtility
+ modelInstance: Adapter
+ resultSet: ResultSet
+ selectQuery:SelectQuery
+resource: Resource
+primaryKey: Enum
+isNationalIdExists(): boolean
+getIndo(): Map<Enum, Object>
+isPasswordMatch(): boolean
Use
Use
SelectQuery
+fields: List<Object>
+limit: String
+where: String
+table: String
+orderBy: String
+toString(): String
ModelExceptionHandler
+handle(): void
Use
Extend
Extend
Use
Use
Administrator
+selectQuery: SelectQuery
+isAdministratorExists(): boolean
+update(): void
+getNumOfRegisteredInRole(): int
+delete(): void
+insert(): void
+isValidAccount(): boolean
Use
Customer
+selectQuery: SelectQuery
+resultSet: ResultSet
+customerMode: Adapter
+resource: Resource
+isMeterCodeExists(): boolean
-insertFirstColumnCustomerBill(): void
+update(): void
+isValidAccount(): boolean
+toggleActivation(): void
+isMeterCodeActive(): boolean
-getReleaseDate(): String
+insert(): void
+delete(): void
Use
Use
CollectedMoney
+CollectedMoneyModel: Adapter
+resource: Resource
+selectQuery: SelectQuery
+collectPayment(): void
+getTotalCollected: double
Use
Use
Use
EBS_DbContext
+collectedMoneyMode: CollectedMoney
+administratorModel: Administrator
+billModel: Bill
+customerModel: Customer
+getCustomerModel(): Customer
+getAdministratorModel(): Administrator
+getBillModel(): Bill
+getCollectedMoneyModel(): CollectedMoney
<<Interface>>
Adapter
+Operator(): String
+Where(): String
+ControlPrecenceence(): String
+Aggregate(): String
+update(): void
+insert(): void
+delete(): void
+getPrimaryKeyColumnName(): Enum
+getTableName(): Enum
Resource
+statement: Statement
+resultSet: ResultSet
+connection: Connection
+close(): void
+isResultSetEmpty(): boolean
Use
QueryExecuter
-prepareQuery(): PreparedStatemer
+executeSelectQuery(): ResultSet
+execute(): void
Use
Use
ImageConverter
+readImage(): byte[]
+getImage(): Image
Bill
+resource: Resource
+selectQuery: SelectQuery
+resultSet: ResultSet
+billModel: Adapter
insert(): void
+payBill(): void
-getBills(): List<Map<Enum,Object>>
+releaseNewBill(): void
+getNumOfUnpaidBills(): int
-getLastBillInfo(): Map<Enum,Object>
+complainAboutBill(): String
+getActualNumberOfConsumersForRegion(): int
+getSumOfConsumptionsForRegion(): int
+getAllBillsOfMeterCode(): List<Map<Enum,Object>>
+getAllBillsOfRegion(): List<Map<Enum,Object>>
Use
Use
<<Interface>>
DbContext
+getCustomerModel(): Customer
+getAdministratorModel(): Administrator
+getBillModel(): Bill
+getCollectedMoneyModel(): CollectedMoney
Debugger
+emptyResult(): void
+printQuery(): void
<<Interface>>
ConnectionString
+getClassName(): String
+getConnectionPath(): String
Use
Implement
<<Interface>>
EmailConfig
-get(): Enum
JSONReader
+parseFile: Object
json: JSONObject
+ method(type): type
DatabaseConnection
+uniqueInstance : Connection
-getConnection(): Connection
-getPath(): String
-isValidPath(): boolean
-endOfExecutionIO: void
getInstance(): Connection
ConnectionStringImp
+jsonReader: JSONReader
+emailMap: Map<Object, Object>
+getClassName(): String
+getConnectionPath(): String
Use
Implement
ConfigPath
+configFilePath: String
-endOfExecution(): void
+isValidPath(): boolean
get(): String
EmailConfigImp
+jsonReader: JSONReader
+emailMap: Map<Object, Object>
+get(): Enum
Use
Use

Controller:


Controller
+registerInView(): void
Extend
<<Interface>>
AdministratorController
+getAllBillsOfRegion(): List<Map<Enum,Object>
+updateAdmin(): boolean
+deleteOperator(): boolean
+isValidEmail(): boolean
+getNumOfRegisteredInRole() : int
+deleteAdministrator(): boolean
+isValidPhoneNumber(): boolean
+registerInView(): void
+deleteAdmin(): boolean
+getRole(): String
+getInfo(): Map<Enum, Object>
+isValidAddress(): boolean
+getConsumptionStatOfRegion(): Map<Enum, Object>
+getTargetID(): String
+getTotalCollectedMoney(): String
+getLoggedInID(): String
<<Interface>>
NewAdministratorController
+isValidAddress(): boolean
+registerAdministrator(): void
+isValidPhoneNumber(): boolean
+isValidEmail(): boolean
+isValidNationalID(): boolean
+isValidName(): boolean
<<Interface>>
CustomerController
+deleteCustomer(): boolean
+getInfo(): Map<Enum, Object>
+isValidComplaint(): boolean
+isValidReading(): boolean
+complainAboutBill(): void
+payBill(): void
+collectPayment(): void
+isValidPhoneNumber():boolean
+isValidEmail(): boolean
+isValidAddress(): boolean
+updateCustomer(): void
+toggleActivation(): void
+releaseNewBill(): void
+getAllBillsOfMeterCode(): List<Map<Enum, Object>>
+getLastReleaseDate(): String
+getLastReading(): String
+getMeterCode(): String
+getActivationState(): String
<<Interface>>
NewCustomerController
+isValidEmail(): boolean
+isValidNationalID(): boolean
+isValidPhoneNumber(): boolean
+isValidName(): boolean
+isValidAddress(): boolean
+registerCustomer(): void
<<Interface>>
AdministratorLoginController
+isValidAccount(): boolean
+isAdministratorExists(): boolean
+generateNewPassword(): void
+getRole(): String
<<Interface>>
CustomerLoginController
+isMeterCodeExists(): void
+isValidAccount(): void
+generateNewPassword():boolean
Implement
CustomerLoginControllerImp
-view:View
-dbContext:DbContext
+isMeterCodeExists(): boolean
+isValidAccount(): boolean
+generateNewPassword(): void
+registerInView(): void
-sendNewPasswordEmail(): void
Use
Implement
NewCustomerControllerImp
- generator:CustomerDataGenerator
-view:View
-dbContext:DbContext
-validator:DataValidator
+isValidEmail(): boolean
+isValidNationalID(): boolean
+isValidPhoneNumber(): boolean
+isValidName(): boolean
+isValidAddress(): boolean
+registerCustomer(): boolean
+registerInView(): void
-SendSuccessfulRegistrationEmail(): void
AdministratorControllerImp
targetID:String
#view:View
#dbContext:DbContext
#ID:String
#validator:DataValidator
+getAllBillsOfRegion(): List<Map<Enum, Object>>
+updateAdmin(): void
+deleteOperator(): boolean
+isValidEmail(): boolean
+getNumOfRegisteredInRole() : int
+deleteAdministrator(): boolean
+isValidPhoneNumber(): boolean
+registerInView(): void
+deleteAdmin(): boolean
+getRole(): String
+getInfo(): Map<Enum, Object>
+isValidAddress(): boolean
+getConsumptionStatOfRegion(): Map<Enum, Object>
+getTargetID(): String
+getTotalCollectedMoney(): String
+getLoggedInID(): String
-getOperatorsCount(): int
-getAdminsCount(): int
CustomerControllerImp
- meterCode:String
-view:View
-dbContext:DbContext
-validator:DataValidator
+deleteCustomer(): boolean
+getInfo(:Map<Enum, Object>
+isValidComplaint(): boolean
+isValidReading(): boolean
+complainAboutBill(): void
+payBill(): void
+collectPayment(): void
+isValidPhoneNumber(): boolean
+isValidEmail(): boolean
+isValidAddress(): boolean
+updateCustomer(): void
+toggleActivation(): void
+releaseNewBill(): void
+getAllBillsOfMeterCode(): List<Map<Enum, Object>>
+getLastReleaseDate(): String[]
+getLastReading(): String
+getMeterCode(): String
+getActivationState(): String
-AreThereUnpaidBills(): boolean
-getCustomerEmail(): String
-unpaidEmailDeterminer(): boolean
-sendUnpaidBillEmails() :void
AdministratorLoginControllerImp
-view:View
-dbContext:DbContext
+isValidAccount():boolean
+isAdministratorExists(): boolean
+generateNewPassword(): void
+getRole(): String
+registerInView(): void
-sendNewPasswordEmail(): void
MakeConsumptionState
-dbContext:DbContext
-governmentCode:String
+make(): Map<Enum, Object>
NewAdministratorControllerImp
-view:View
-validator:DataValidator
+isValidAddress(): boolean
+registerAdministrator(): void
+isValidPhoneNumber(): boolean
+isValidEmail(): boolean
+isValidNationalID(): boolean
+isValidName(): boolean
-sendSuccessfulRegistrationEmail(): void
DataValidator
isValidPassword(): boolean
isValidNationalID(): boolean
isValidAddress(): boolean
isValidComplaint(): boolean
isValidEmail(): boolean
isValidName(): boolean
isValidPhoneNumber(): boolean
<<Abstract>>
DataGenerator
#generateRandomNumbers():String
#generateRandomNumber():int
#generateRondomChar(): char
-getBirthYear(): String
-getBirthMonth():String
-getBirthDay(): String
generateDateOfBirth():String
generatePassword():String
generateGender():String
generateDateOfContract():String
BillCalculations
-typeOfUse: String
-currentReading: int
-meterCode: String
-moneyValue: double
-dbContext: DbContext
-tariff: int
-consumption: int
-calculateMoneyValue(): void
-setTypeOfUse(): void
-calculateConsumption(): vo
+getConsumption(): int
+getMoneyValue(): double
+getTariff(): int
Extend
CustomerDataGenerator
generateMeterCode(): String
AdministratorDataGenerator
generateID(): String
ReadingValidator
-dbContext:DbContext
+validate(): boolean
EmailMessage
-emailConfiguration: EmailConfigura
-personalName: String
generateMessage(): Message
setPersonalName(): EmailMessage
setFullConfig(): EmailMessage
setDefaultConfig(): EmailConfig
MessageExceptionHandler
handle(): void
print(): void
SendEmail
-emailMessage:EmailMessage
+send(): void
-setPersonalName(): SentEmail
+setFullConfig: SendEmail
-setDefaultConfig(): SentEmail
EmailConfiguration
-host: String
-port: String
-session: Session
-senderPassward: String
senderMail: String
-createSession(): void
getSession(): Session
getSenderEmail(): String
setFullConfig(): EmailConfiguration
setDefaultConfig(): EmailConfig

View: