Electrical Billing System - Design & Implementation

Will Humphlett, Swati Baskiyar, DJ Harris, Jonathon Porco, Michael Duvall

Analysis Phase

Domain Analysis

Domain Class Model

Identified Classes

Customer - A contributor to the payment of the electricity usage of one household. The same person contributing to the payment of a different household is considered a different customer.
Employee - An employee is a user hired by the electrical company that is allowed to enable/disable service of households as well as change the cost per power unit for each household. Employees are not allowed to edit user information or view user payment information beyond payments made and transactions charged.
Admin - An admin is a user hired by the electrical company that is allowed to view and edit any and all information associated with the classes defined. As a security measure, all changes made by an admin are logged.
Household - A household is a property being supplied power and is being charged for the usage of this power. Whether the property is residential or commercial is not of consequence. Multiple customers may be tied to one household as multiple customers may be responsible for the payment of electricity for one household.
PowerUsage - Power usage is the total units of power consumed over a given timeframe subject to a given price. This is recorded as power is consumed to enable charges to automatically be levied against a household as well as usage statistics to be generated for each household.
CreditCard - This is a credit card attached to a customer that enables them to make payments. All details necessary to make payments are saved.
BankAccount - This is a bank account attached to a customer that enables them to make payments. All details necessary to make payments are saved.
Payment - a record of a payment against the balance of a household made by a customer. Details such as how much, by whom, with what method, and when are saved for the purposes of calculating balance and creating payment summaries.
Charge - a record of a charge levied against a household for power units consumed or fees incurred. Charges for units are automatically generated every month as well as late fees if a balance is not paid in sum by a set deadline.

ContactMethod - includes a method to send push notifications to a customer in the event of account updates, late payments, or any other information that is needed to send to a customer immediately. Whether or not a method is the most preferred method is saved as well.

Identified Associations

Households are made of multiple Customers
Transactions are ascribed to one Household
Payments are made by one Customer
Charges are a collection of PowerUsages
Multiple PaymentMethods can be used by one Customer
One PaymentMethod is used per Payment
Admins and Employees are not Customers

Attributes
User
- username
- password

Admin
Employee
Customer
- household

Household
- address
- pricePerUnit

PowerUsage
- household
- pricePerUnit
- units
- datetime

PaymentMethod
- customer
- isDefault

CreditCard
- address
- name
- number
- ccv

BankAccount
- accountNumber
- routingNumber

Transaction

- household

Charge

- amount
- pricePerUnit
- datetime

Payment

- customer
- amount
- datetime
- paymentMethod

ContactMethod

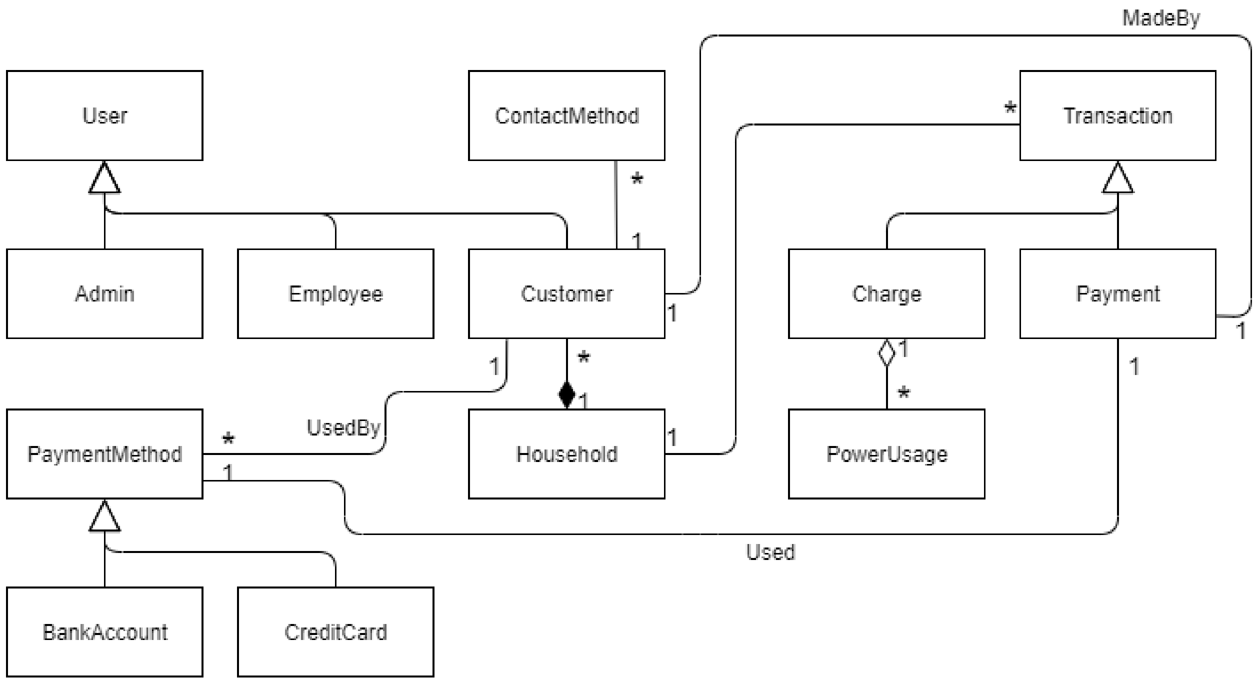
- customer
- type
- contact
- isDefault

Organized Classes

Admins, Employees, and Customers can be defined as an extension of Users
Charges and Payments are both Transactions
CreditCards and BankAccounts are both PaymentMethods

These generalizations allow for common attributes to be stored in the parent class and improve readability of the class diagram

Domain Class Diagram



Domain State Model

Identified Domain Classes

Household
A Household’s electrical services begin when the customer creates the account and inputs valid payment information, with the current state of pending. The account is verified by an administrator or employee and the state is set to enabled. The customer can disable and reenable the household’s services at any time as well as cancel the services for the household. If payment is overdue, the services are disabled and locked until payment is received.

Charge
When a charge is made on a household, the charge is set to an awaiting payment state. When the payment information is received, the state is changed to pending. If the payment information is not received within an allotted amount of time, the state is changed to overdue. The payment information is verified by the bank. If the bank verifies the information, the payment is processed and the charge is set to completed. If the information is denied, the charge is set back to awaiting payment.

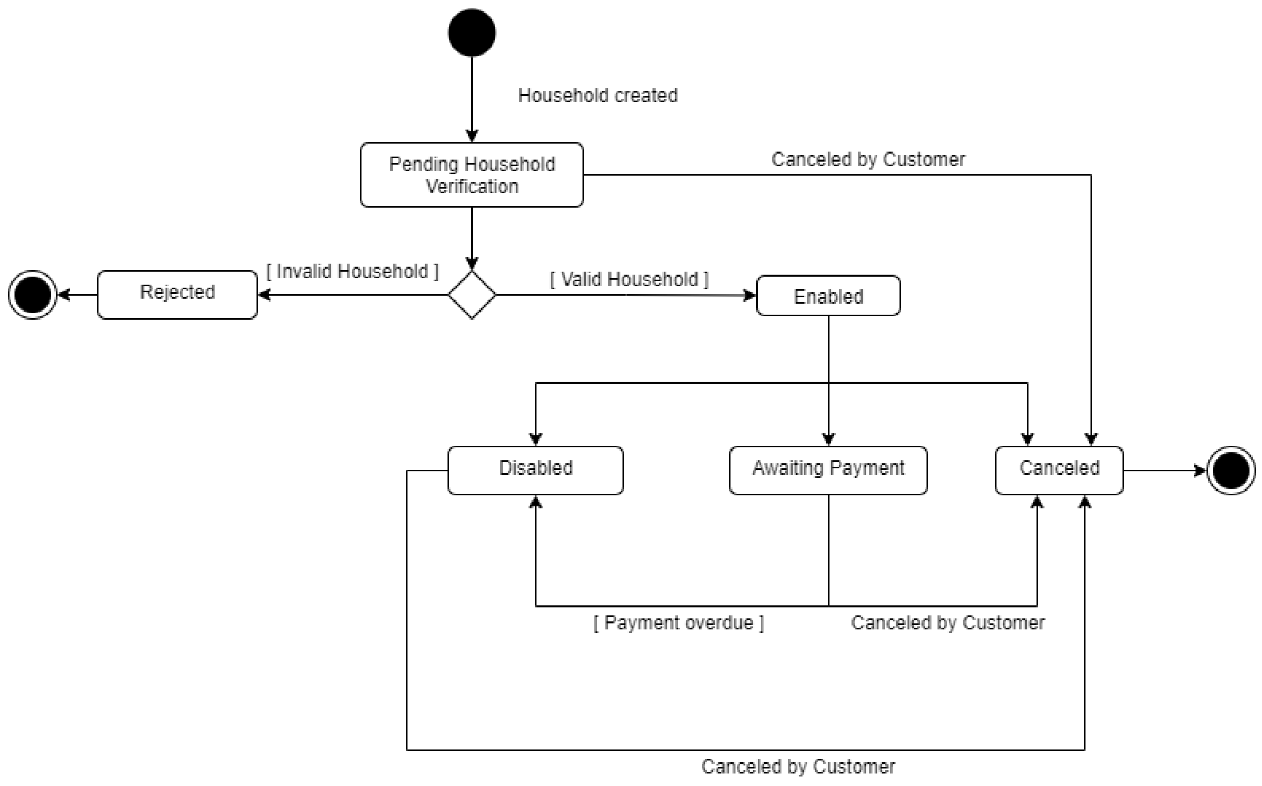
Customer
When a customer creates an account, the account is pending verification. At any time the customer can edit account information, and the account is set to pending approval. When the account is approved, the customer is active. The customer can disable the account or leave the

household, which disables the account, at any time. The customer can request a household transfer which sets the state to pending verification. The customer can cancel the account at any time.

Identified Significant Events

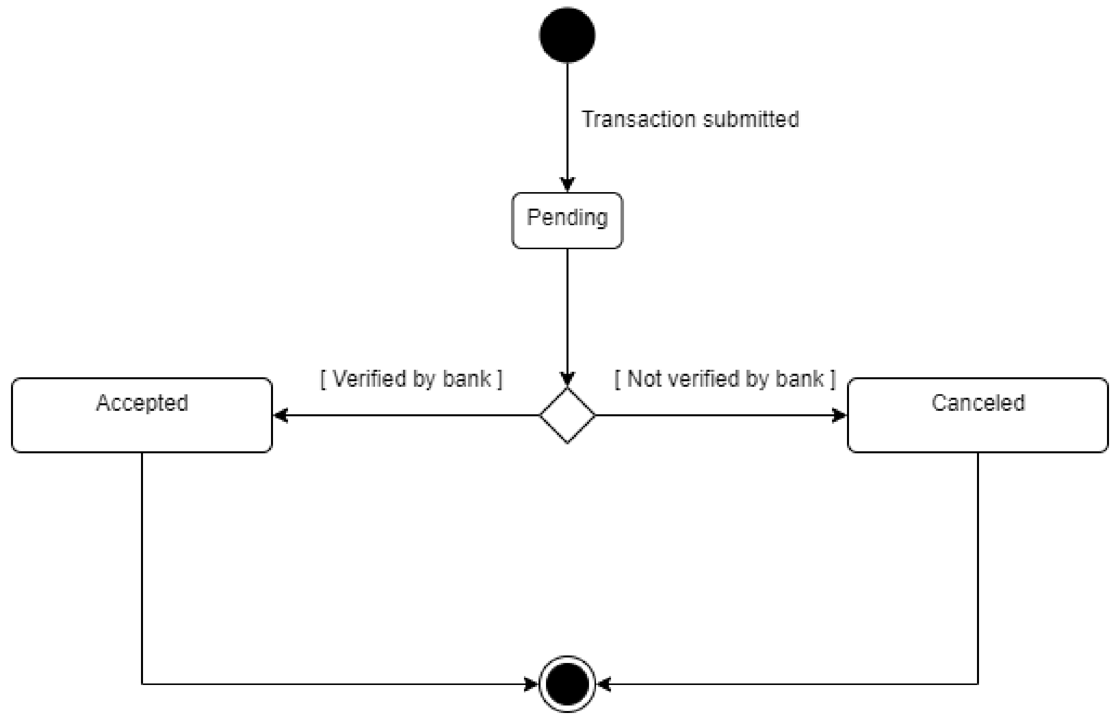
Household

- Pending Verification
- Rejected
- Enabled
- Disabled
- Awaiting payment
- Canceled



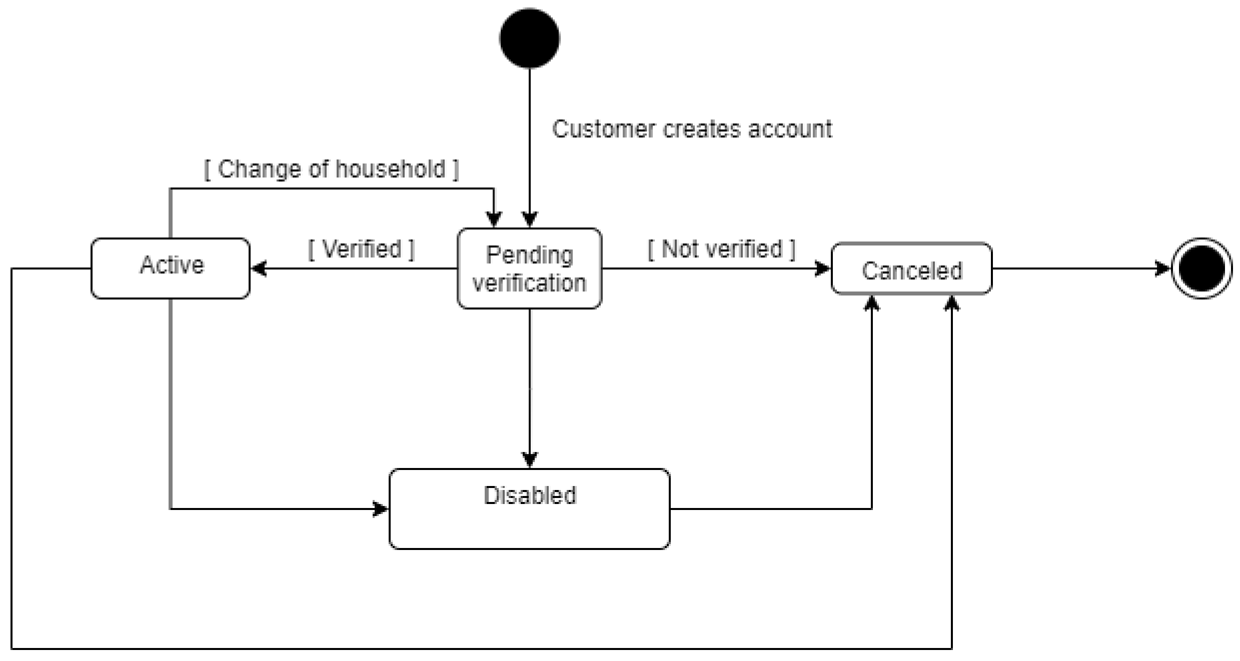
Transaction

- Pending
- Accepted [verified by bank]
- Canceled [not verified by bank]



Customer

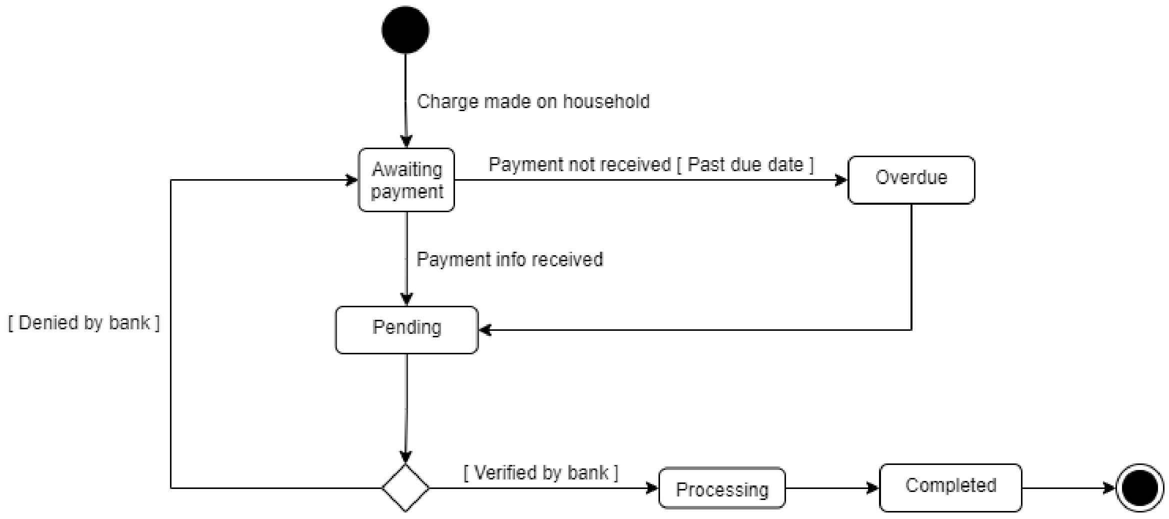
- Pending verification
- Canceled
- Disabled
- Active



Charge

- Awaiting payment

- Pending (Payment information received)
- Overdue (Payment information not received [Past due date])
- Processing [verified by bank]
- [denied by bank]
- Completed



Application Analysis

Application Interaction Model

Identified Use Cases

New User Registration
A user creates an account themselves and the data they input can be validated automatically by the system. The employee can also create an account for a customer. This creates an account that the customer can access to view billing schedules, payment needed, detailed summaries, and other tools that seek to better inform the user without requiring manual input.

Make Payments
When a billing statement is issued, the customer will pay online with no employee aid using their preferred method of payment. Built into this payment system is a calculation of late fees, overdue balances, and any other pricing concerns needed to calculate a final amount due per customer per statement.

Query Bills
Customers would be able to view billing statements, past, present, and future, to allow them to see what they have paid and what they will pay in the future. This data is stored in the system database. If an employee needs to adjust details about a customer's billing statements, they can

edit the database entries as needed (e.g. address, payment method for the customer view and amount due, power consumed for the statement view)

Turn On/Off service

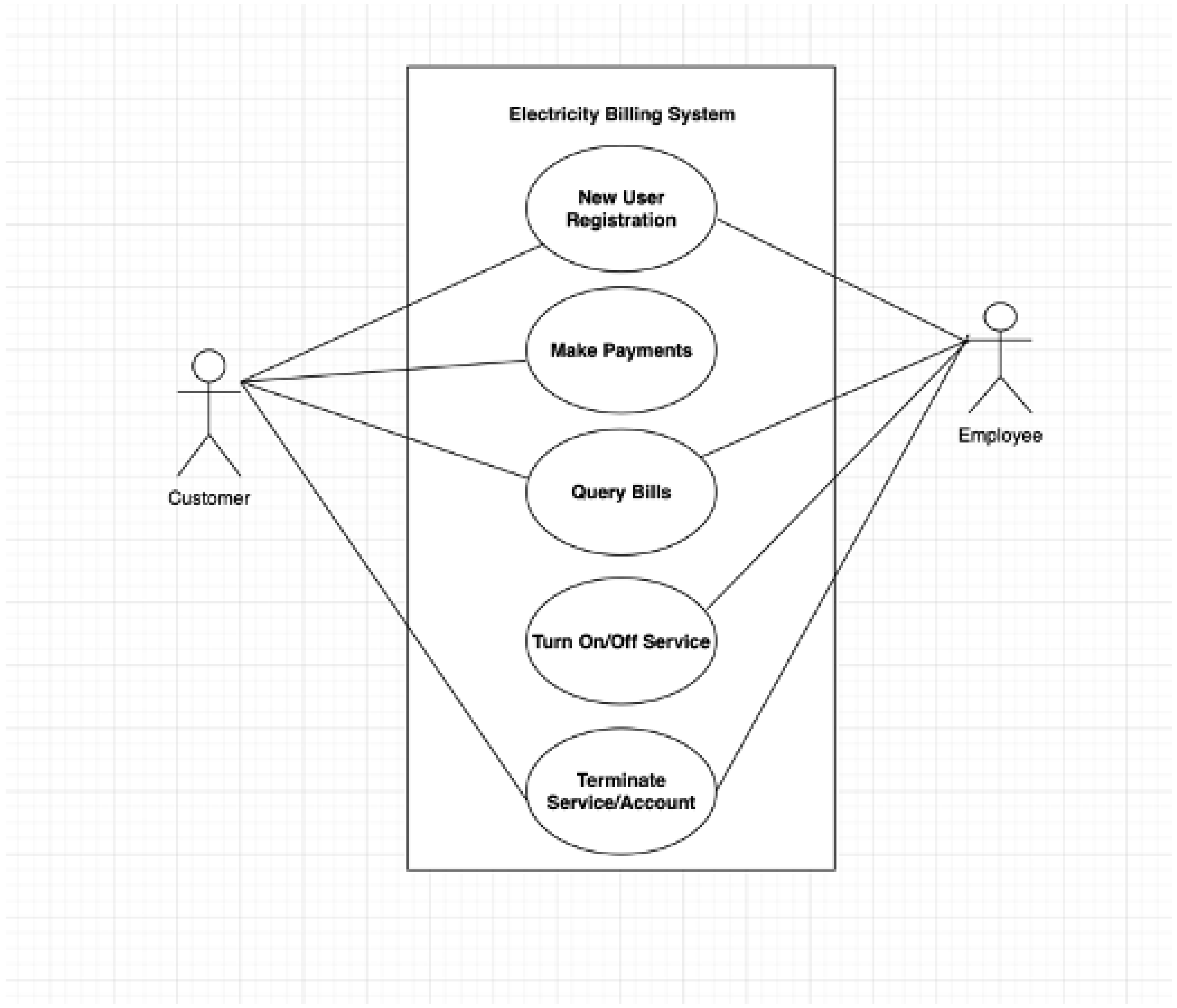
When a customer registers an account, assuming they are in good financial standing, their service can be automatically enabled. Conversely, if they don't make payments, their service can automatically be disabled, until the customer pays the bills. Notifications of each of these state changes can be sent to the customer via the push notification system.

Terminate Account and Services

A customer can request to terminate their services. An employee must approve the request. After the request is approved, the service will be turned off. The customer will no longer have access to the account and the customer's information will be stored in the system's database.

Identified Actors

Actors: Customer and Employee



Prepared Scenarios

New User Registration

Normal:
customer enters personal information into system
System verifies customer information
System adds account to database
System sends confirmation to customer

Exception:
customer enters personal information into system
System determines customer information is incorrect
System informs customer that information is incorrect
System requests customer for correct information

Make Payment

Normal:
Customer enters username and password
System verifies password
Customer enters credit card number
Customer submits payment information
System verifies credit card
System confirms payment

Exception:
Customer enters username and password
System verifies password
Customer enters credit card number
Customer submits payment information
System rejects credit card
System informs customer that payment is unsuccessful

Query Bills

Normal:
Customer enters username and password
System verifies password
Customer requests billing information
System delivers billing information

Exception:
Customer enters username and password
System determines that password is incorrect
System requests customer to reenter password

Turn On/Off Service

Normal:
System notifies employee of customers with overdue bills
Employee requests system to turn off customers' service
System notifies customer that service is being terminated
System turns off service

Exception:
System notifies employee of customers with overdue bills
Employee requests system to ignore overdue bills
System keeps service on

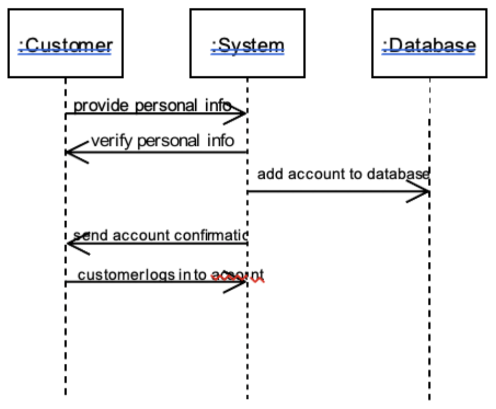
Terminate Service

Normal:
Customer requests system to terminate service
System notifies employee of request
Employee approves request
System terminates service and account
System stores account in database

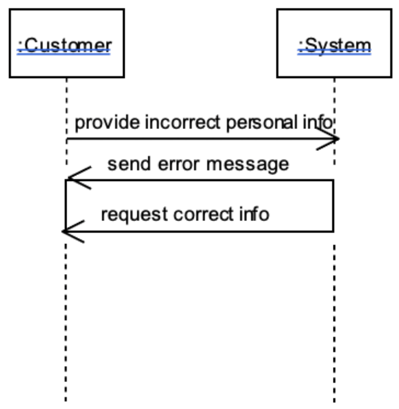
Exception:
Customer requests system to terminate service
System notifies employee of request
Employee does not approve request
System notifies customer of failure of approval

Application Interaction Model - Activity Diagrams - Swati Baskiyar

New User Registration

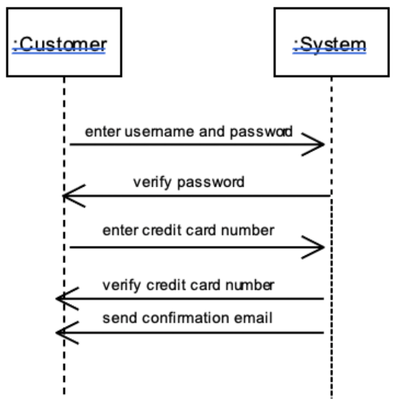


Normal

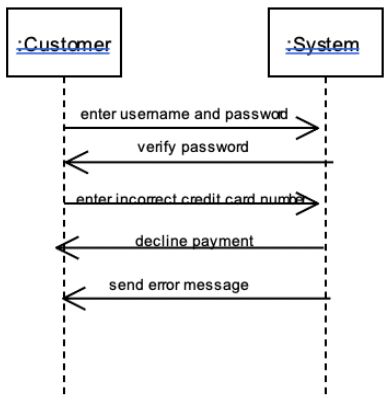


Exception

Make Payment

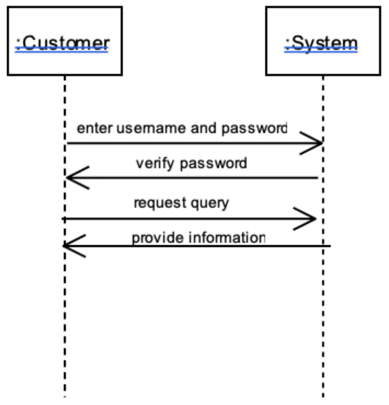


Normal

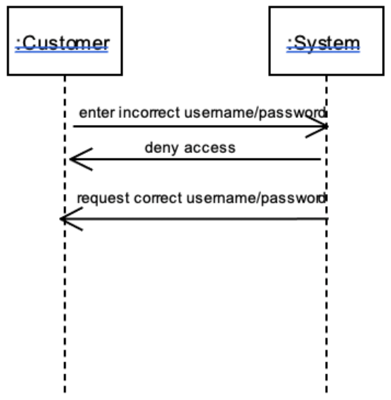


Exception

Query Billing System



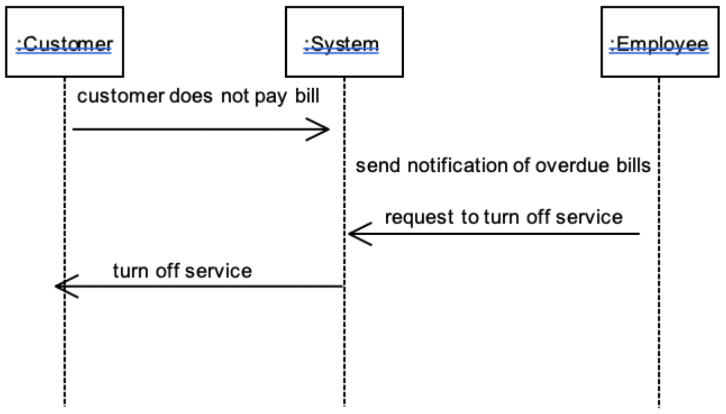
Normal



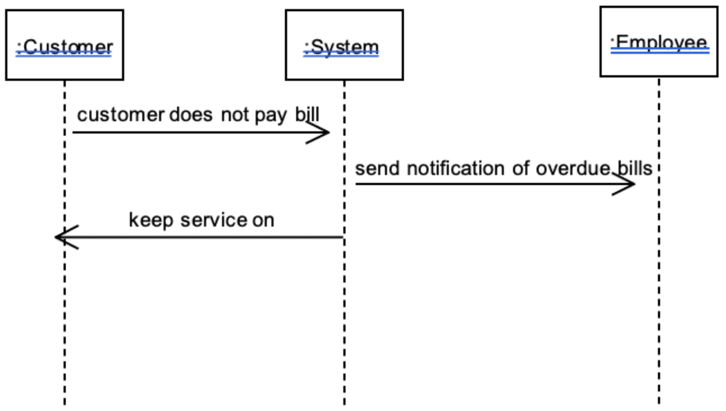
:Customer
:System
enter incorrect username/password
deny access
request correct username/password

Exception

Turn on/off Service

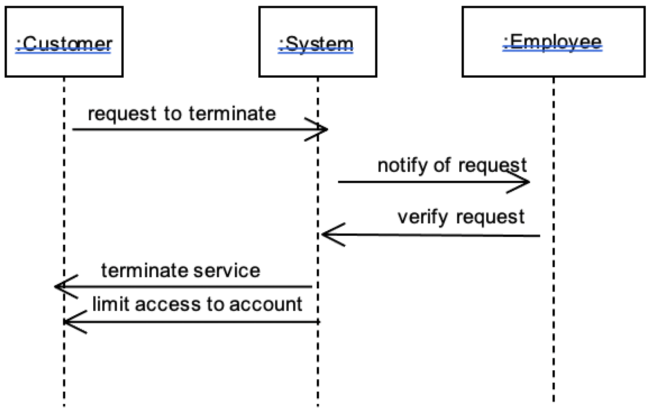

:Customer
:System
:Employee
customer does not pay bill
send notification of overdue bills
request to turn off service
turn off service

Normal

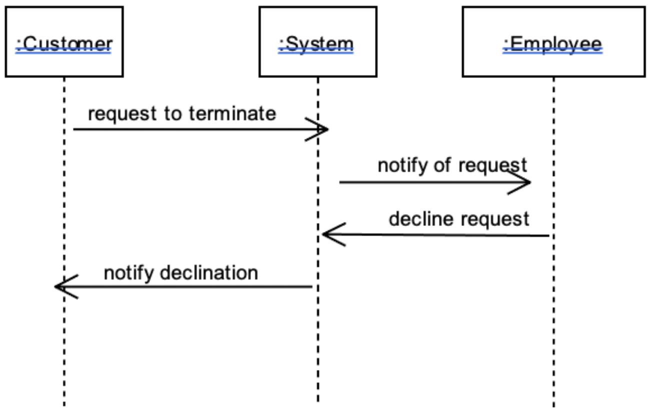

:Customer
:System
:Employee
customer does not pay bill
send notification of overdue bills
keep service on

Exception

Terminate Service



Normal



Exception

Application Class Model

Identified Application Types

Interface
Transaction is an interface class. The transaction class would be a menu that the user is sent to when they want to pay their bill. The menu would have an option to use the payment method that they already provided when creating their account, or if they would like to input any new payment methods that they would like to use. If they choose the second option, then they are taken to another screen where they would input information such as card number, cv number, billing address, current address, and name of the card owner. After inputting that information, the screen would return to the previous menu with the new information. After information confirmation, the user would then confirm the payment.

Boundary
Customer is a boundary class. The Customer has an account that would separate all of the inner workings of the program that are operating so that the exterior interface is working smoothly and properly. The Customer (or account) acts as a barrier for the user that is currently operating the interface so that they only see the exterior extent of the program. When they input a command or fill in any information, they are saving that information in the system to be used at a later point in time.

Controller
Household is a controller class. A controller responds to and/or creates events. The household in our project creates a charge based on the power consumption of the users living inside it. Once this charge is read, the bill can be generated at the start of the next month.

Identified Application Associations

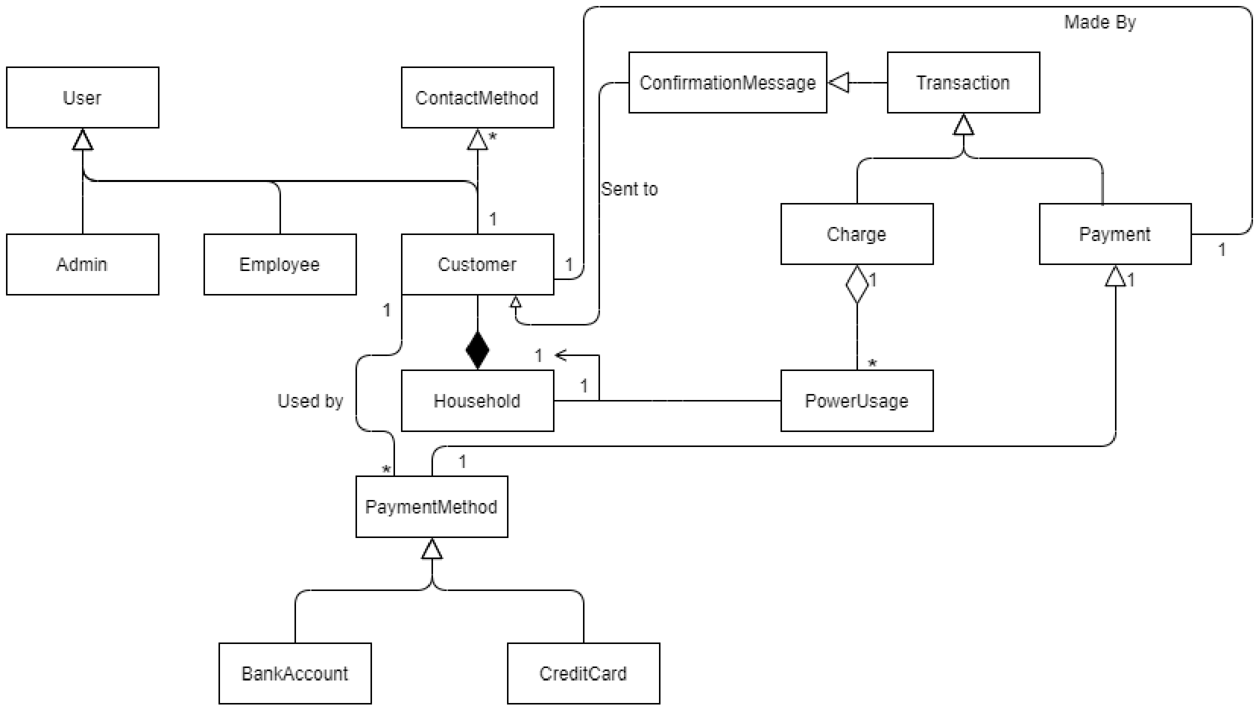
Associations

When a transaction is complete, the customer will get a Confirmation Message

Attributes

confirmationMessage

Updated Domain Class Diagram



Application State Model

Identified Temporal Application Class

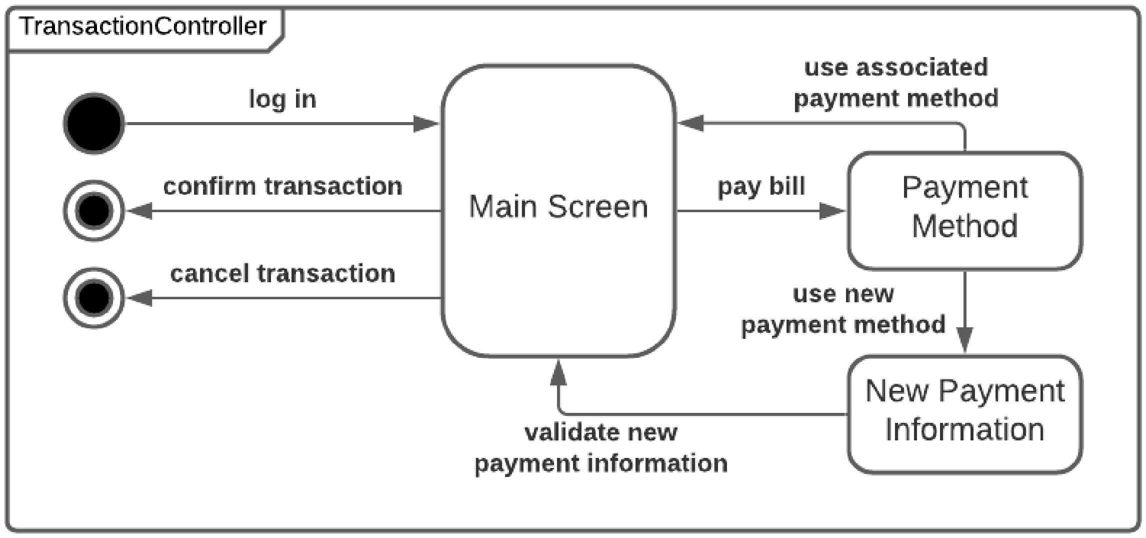
TransactionController

- TransactionController is a controller type application class with significant temporal behavior. The user is brought to the main screen after they log in. The user selects to pay the bill and is asked for a payment method. If the user uses the payment information associated with their account, they're brought back to the main screen. If the user uses a new payment method they input the new payment information and after it is validated, they're brought back to the main screen. From the main screen the user can confirm or cancel the transaction.

Significant Events

- pay bill
- select payment method
- input new payment information
- confirm/cancel transaction

State Diagram



Detailed Design

Interaction Design

System Operations

Make Payment Operations
- calculateBalance
- createBillSummary

New User Registration
- createProfile
- verifyPassword

Operation Contracts

Operation: calculateBalance(previousBalance: float, currentCharges: float, paymentsReceived: float, fees:float)
Cross References: Use case - make payment
Preconditions: customer has an account
Postconditions:
- value of currentBalance is updated in bill summary

Operation: createBillSummary(accountName: String, accountPassword: String)
Cross References: Use case - createBillSummary
Preconditions: customer requests bill summary or monthly bill is due,

Post conditions:

- new instance BillSummary is created
- BillSummary is associated with the customer

Operation: createProfile(accountName: String, accountPassword: String, zipCode: float)
Cross References: use case - new user registration
Precondition: customer requests new account
Post conditions:

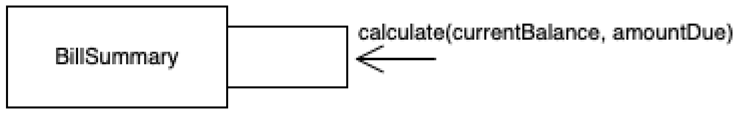
- new instance of customer created
- Customer associated with Payment and PaymentMethod

Operation: verifyPassword(accountName: String, accountPassword: String)
Cross Reference: use case - new user registration
Precondition: customer has created account, customer attempts to login
Post conditions:

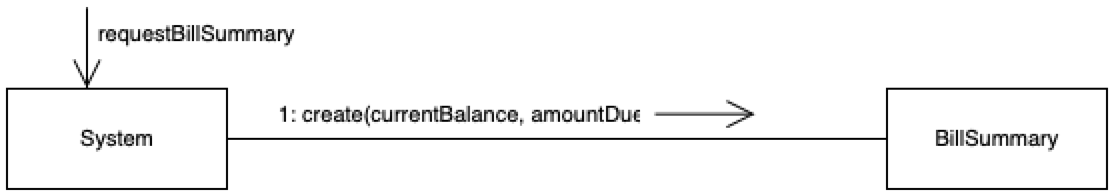
- customer is verified access to account information
- if password is incorrect, customer receives error message "incorrect password"
- if accountName does not exist, customer receives error message "incorrect accountName"

Collaboration Diagrams

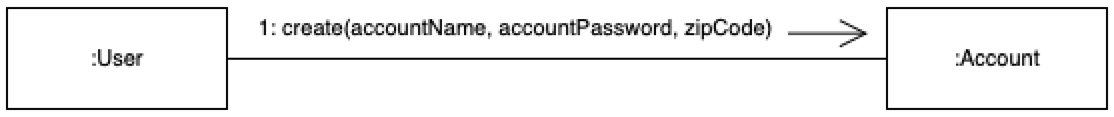
calculateBalance



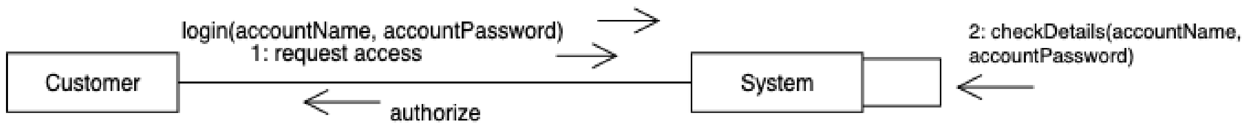
createBillSummary



createProfile



verifyPassword


login(accountName, accountPassword)
1: request access
Customer
authorize
System
2: checkDetails(accountName, accountPassword)

Design Patterns

calculateBalance:
For the calculateBalance collaboration diagram, the information expert pattern was used to assign responsibilities to the BillSummary object. The BillSummary object contains all the information needed to calculate the balance. The high cohesion pattern was also used, shown by the clear, well focused responsibility of the BillSummary object.

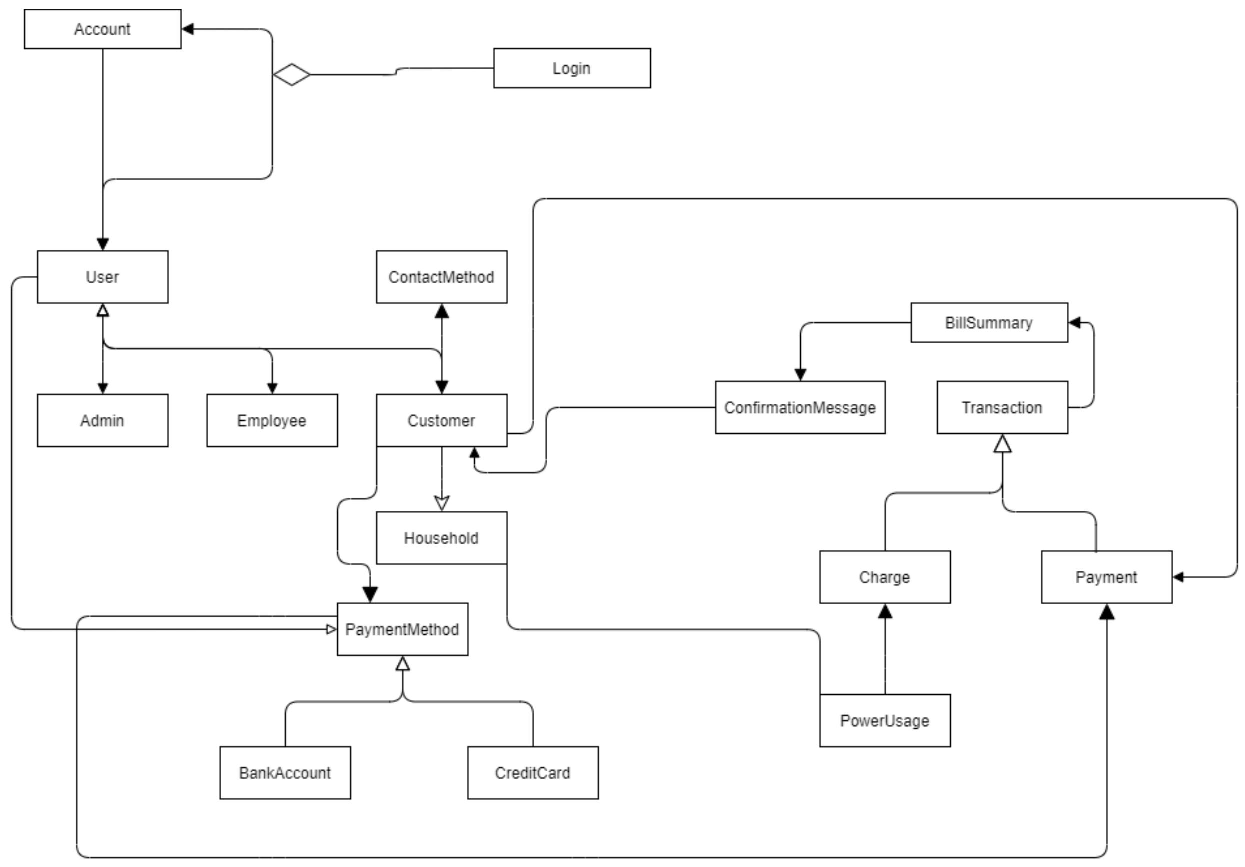
createBillSummary:
For the createBillSummary collaboration diagram, the controller pattern was used to assign responsibilities to the System object. The System object represents the system and is the assigned controller for requestBillSummary. The high cohesion pattern was also used, shown by the clear, well focused responsibilities of the System and BillSummary objects.

createProfile:
For the createProfile collaboration diagram, the creator pattern was used to assign responsibilities to the User object. The User object contains the Account object so it has the responsibility of creating Account instances. The high cohesion pattern was also used, shown by the clear, well focused responsibilities of the User and Account objects.

verifyPassword:
For the verifyPassword collaboration diagram, the controller pattern was used to assign responsibilities to the System object. The System object represents the system and is the assigned controller for account detail checking and login authorization. The high cohesion pattern was also used, shown by the clear, well focused responsibilities of the Customer and System objects.

Class Design

Design Class Diagram



One pattern that is used here is Facade. This pattern is used to connect an interface that is being used to a subsystem that is running underneath. The way this pattern is used is because the interface that is being used communicates with the system underneath, where all the data and operations are stored and completed.

Behavioral Design

State-dependent classes:

- Household
- Charge

Pre/Post Conditions

Household
Method: makePayment
Pre-conditions: paymentCharged - there is a pending charge on the household; householdStatus - the household is active and not on hold; paymentDeadline - date which the payment is due.

Post-conditions: paymentCompleted - the charge has been completed and verified; householdStatus - the household is active and not on hold; powerOn - the household's power is on/off; paymentDeadline - day which the payment is due

Method: togglePower
Pre-conditions: householdStatus - the household is active/on hold, powerOn - the household's power is on/off
Post-conditions: householdStatus - the household is active/on hold, powerOn - the household's power is on/off

Method: terminateHousehold
Pre-conditions: householdStatus - the household is active/on hold, powerOn - the household's power is on/off
Post-conditions: householdStatus - the household is active/on hold, powerOn - the household's power is on/off

Method: getStatement
Pre-conditions: dateRange - specified date range for statement; householdStatus - the household is active/on hold, powerOn - the household's power is on/off
Post-conditions: dateRange - specified date range for statement; householdStatus - the household is active/on hold, powerOn - the household's power is on/off

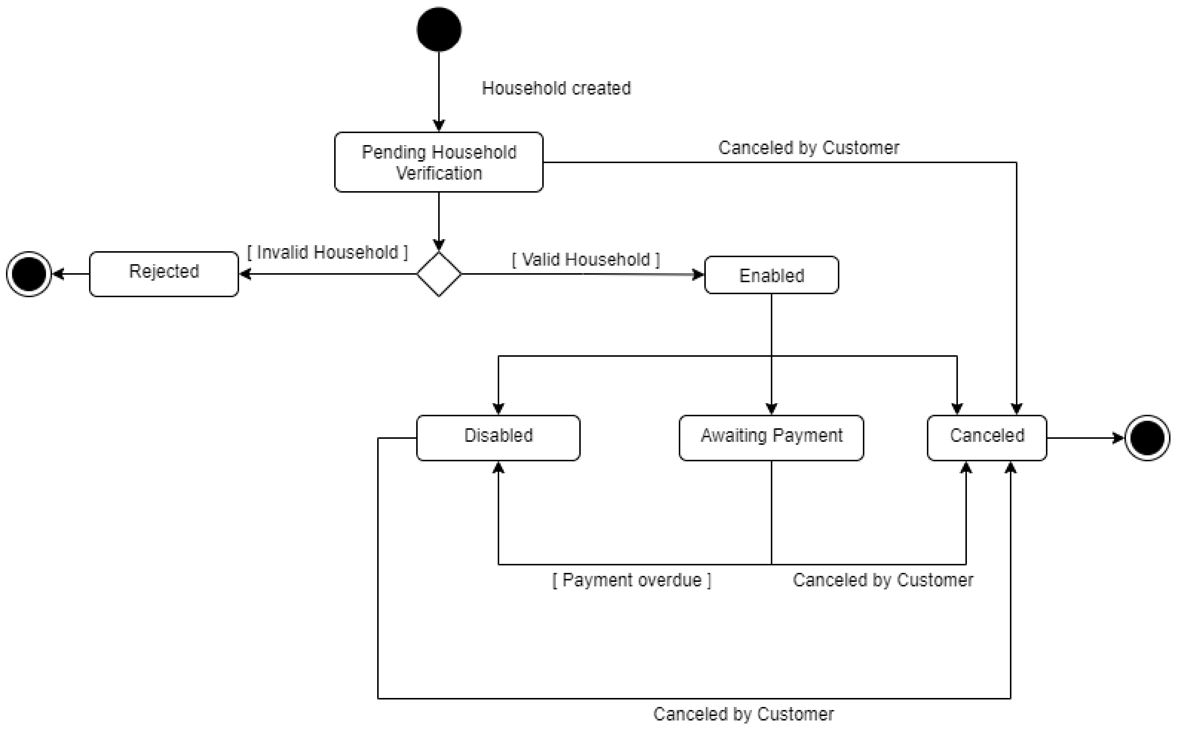
Charge
Method: verifyInfo
Pre-conditions: paymentInfo - amount being charged; paymentDeadline - due date of payment; cardInfo - card information; amountDue - amount that is due; householdInfo - household ID
Post-conditions: paymentVerified - amount being charged verified; deadlineVerified - due date of payment verified; cardVerified - card info verified; amountVerified - amount due verified; householdVerified - household ID verified; infoVerified - true if info verified

Method: verifyPayment
Pre-conditions: cardInfo - card information; bankId - bank information; routingNum - routing number; payment - payment amount
Post-conditions: paymentStatus - if payment was completed and verified

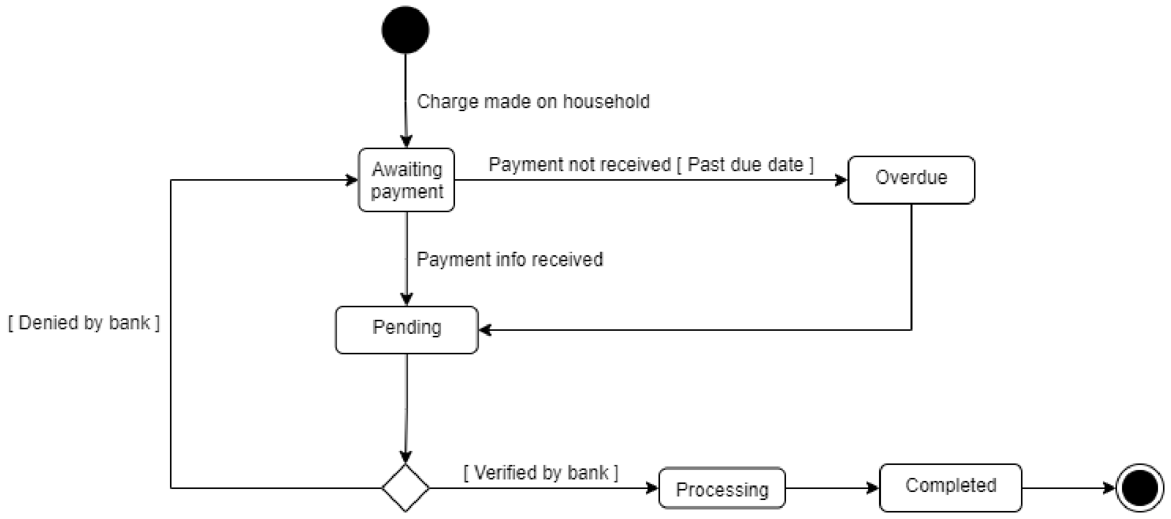
Method: makeCharge
Pre-conditions: cardInfo- card information; bankId- bank information; payment - payment amount; verifyInfo - verify all information; verifyPayment - verify payment info
Post-conditions: paymentStatus - if payment was completed

State Chart

Household:



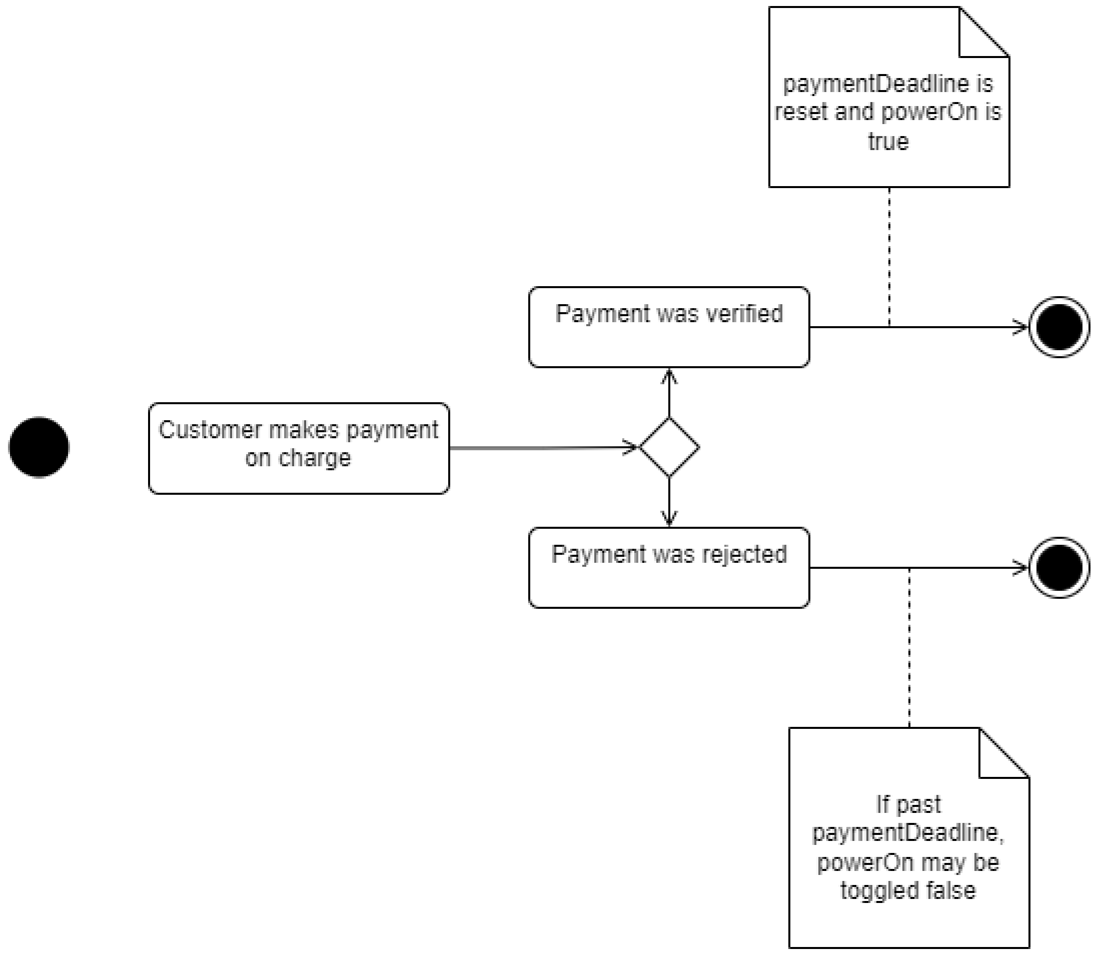
Charge:



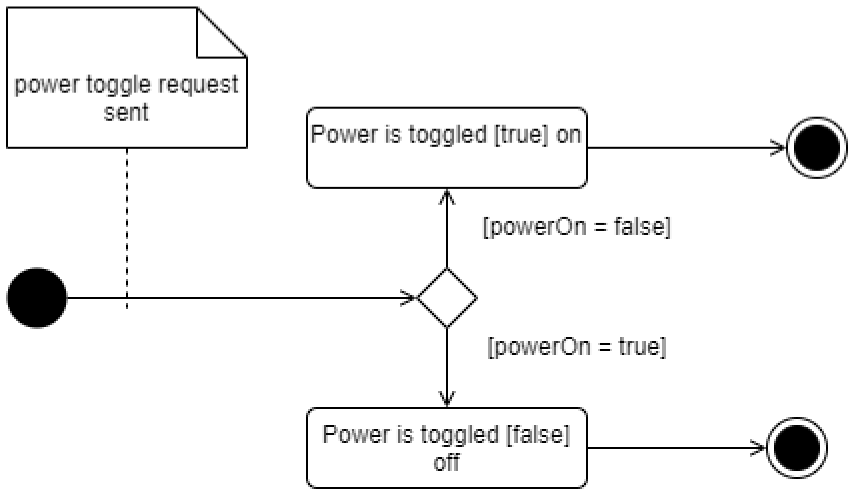
Activity Diagram

Household:

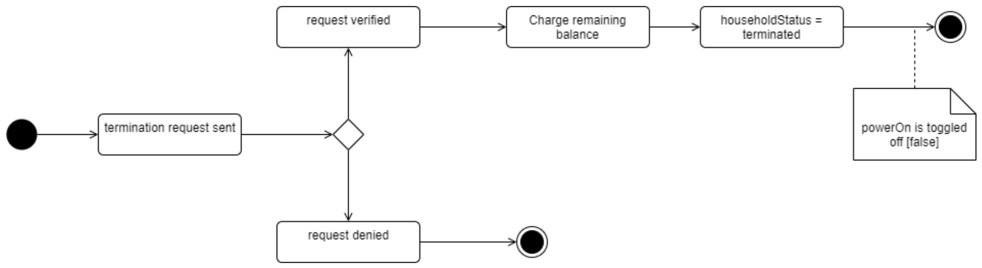
Method: makePayment


paymentDeadline is reset and powerOn is true
Payment was verified
Customer makes payment on charge
Payment was rejected
If past paymentDeadline, powerOn may be toggled false

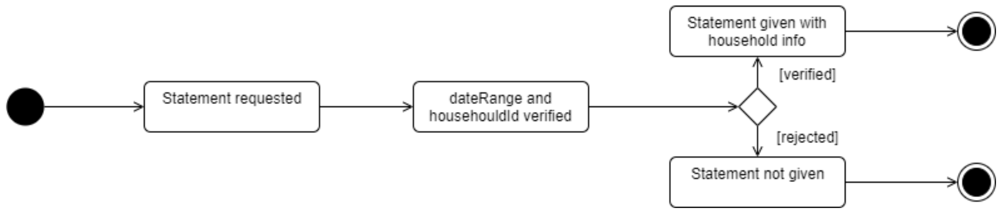
Method: togglePower


power toggle request sent
Power is toggled [true] on
[powerOn = false]
[powerOn = true]
Power is toggled [false] off

Method: terminateHousehold

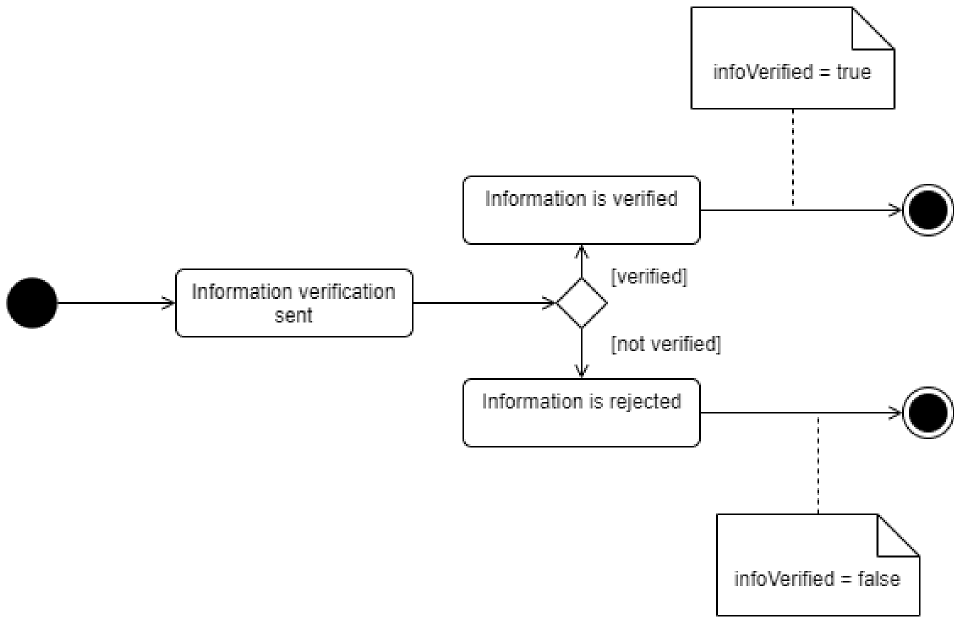


Method: getStatement

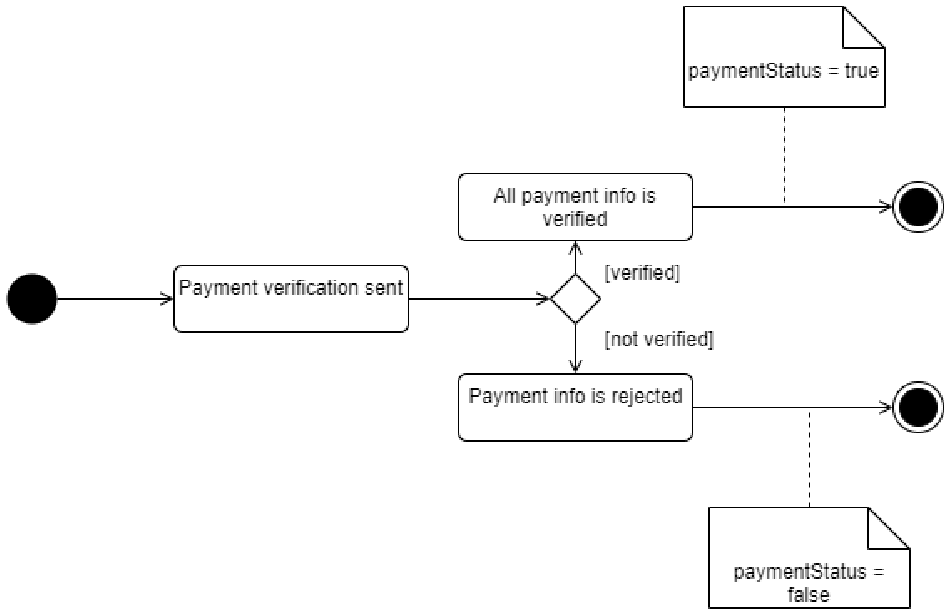

Statement given with household info
[verified]
Statement requested
dateRange and househouldId verified
[rejected]
Statement not given

Charge

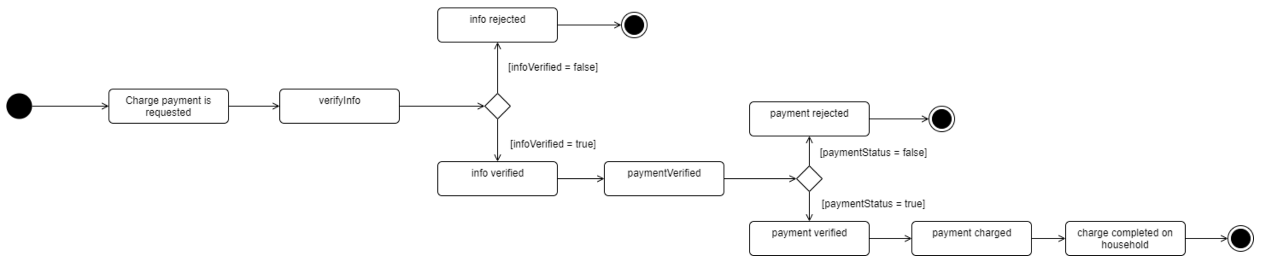
Method: verifyInfo



Method: verifyPayment



Method: makeCharge



Implementation

Implementation of the designs discussed can be found at https://github.com/wumphlett/COMP-3700. This was used as a central git repository for collaborative work on the project.

Documentation

Most of the project is contained in the git repository under the “elec_billing” directory. Inside that is the “billing_system” directory which contains a few files of note.

- models.py - This contains the direct translation of the described classes to models to be stored and manipulated in the database.
- views.py - This contains the views needed to serve our Model-View-Controller architecture pattern.

Instructions for how to set-up your local environment to execute our code can be found at https://github.com/wumphlett/COMP-3700/blob/main/README.md. This includes the commands necessary to set-up your local environment and run a copy of the website.