Capstone Design Program Report

CECS 343

Group: 4

Ben Kweon, Bharath V Kakarlapudi,

Yuwei Du, Hoang-Uyen Tran

Due: 28th June 2022

Table of Contents

Problem statement and Requirements **4**
Purpose: 4
Overall Description: 4
Working Environment: 4
Project Link: 4
Problem Statement **5**
Vision Statement 6
Use Case and Scenarios: **7**
Use Case Diagram: **7**
Use case documentation 7
1. Log in the system 7
2. Start the program 8
3. Add new tenant 8
4. Input a rental payment 9
5. Display the Tenant List 9
6. Display the Rent Record 9
7. Display the Expense Record 10
8. Display the Annual Summary 10
Requirement Modeling **11**
Class diagram: **11**
Class Diagram Description: 11
Describes-Overall Class Diagram 11
Attributes & Operation-Overall Class Diagram 12
Class-Responsibility-Collaborator (CRC) Modeling **17**
Activity diagram **21**
Overall Activity Diagram: 21
A detailed Activity Diagram: 23
Sequence diagram: **24**
State diagram: **28**
Collaboration diagram: **29**
Component Diagram **29**
Detail Component Detail: Add Tenant 30
User Interface Design **30**
Deployment Design **32**

Installation Steps **33**

Coding **33**
AnnualReport.java 33
DBconnect.java 36
Expense.java 44
ExpenseInputScreen.java 46
ExpenseRecord.java 47
Main.java 48
OutputScreen.java 49
RentRecord.java 50
RentRow.java 52
RentRowInputScreen.java 53
Tenant.java 55
TenantInputScreen.java 56
TenantList.java 58
UserInterface.java 60

Unit Testing **66**

Sample Run-time output: **67**
Display Tenant 67

Summary: **68**

Problem statement and Requirements

Purpose:

Capstone design is the system that will organize and keep track of all the tenants' information and expenses. This system allows the landlord to keep track of his/her tenant information such as: rent expenses, expenses record, tenants' information(name and aptNo).

Overall Description:

Landlord will be the main customer who will be using this system. Starting the system, the menu of options will display to serve the landlord purpose such as: Insert(tenants' information and rental payment). Display(annual summary, expense record, rent record, and tenant list). Quit(quit the program).

Depending on the landlord chosen which option, the system will save all the data input and display the information based on the input data.

Working Environment:

1. Computer/Laptop
2. Operation System: Windows 10,11
3. Terminal

Project Link:

- https://github.com/ben9543/CECS-343-Capstone-Project

Problem Statement

Purpose:

Before a system design is implemented, specified in detail to make analysis and design possible.

Requirement:

There are some requirements that help break down the complexity of the project. Techniques can be break down to

- Identify the problem
- Affects
- The impact
- Successful solution

Problem Statement	
The problem of	Excess paperwork to maintain the records of payments received from tenants
Affects	John nguyen
The impact of which is	Less time consuming and loss track of data
a successful solution would be	A program can keep track of large amount of datas, can readjust anytime and paperless

Vision Statement

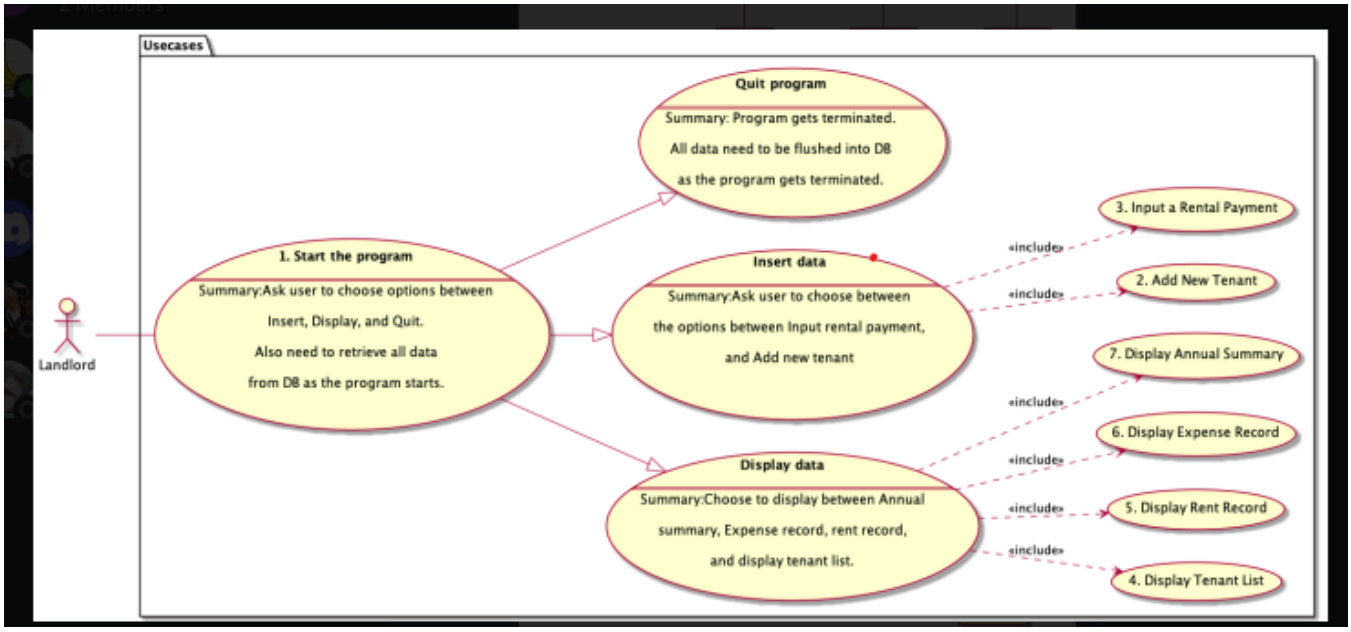
Purpose:

More details of the breakdown part of the system.

Vision Statement	
For	John Nguyen who is the landlord
Who	Record and manage 20 units' tenant information and financial status
The (product name)	Building Complex
That	Will help management team easily record tenant information, rental income and building expense Find past record faster Easier to use and keep records
Unlike	Maintaining a handwritten financial record of the tenant payments
Our product	Enables John Nguyen to better track the financial record of the tenant payments by making the process digitally automated. Easy to update new information, keep all records organize, can store large amount of data without loss track

Use Case and Scenarios:

Use Case Diagram:



Use case documentation

1. Log in the system

- Use case number: 1
- Use case name: Log in the system
- Actor: The landlord
- Precondition: The user needs to have a valid account
- Summary: Authenticate user before starting the program so that unauthenticated users cannot use the program.
- Scenario:
    1. Ask input
        - Input username
        - Input password
    2. Validation
        - Does username exist
        - Does the stored password and the given password match
- Exception:

    - Scenario 2 => When user credential does not match
- NonFunctional:
    - Interaction with database is needed to save userdata
    - Passwords should be saved and encrypted.
    - Input validation needed for both username and password(no whitespace, … etc)

2. Start the program

- Use case number: 2
- Use case name: Start the program
- Actor: Log in the system (usecase)
- Summary: Ask user to choose options between insert, display, and quit
- Scenario:
    1. Display the menu
    2. Ask user input
    3. Actor would choose either insert, display or quit
- Exception
    - Scenario 3 => If the userinput is neither ‘i’, ‘d’, or ‘q’, ask input again
- Non Functional
    - Input validation needs to cause no error (otherwise the program will crash)
    - Use one variable to store the current userinput

3. Add new tenant

- Use case number: 3
- Use case name: Add new tenant
- Summary: Adding a new tenant to the system
- Actor: The landlord
- Precondition: The tenant is new here and the apt is empty.
- Scenario:
    1. Enter input data surface
    2. Enter to add tenant surface
    3. Enter tenant’s name
    4. Enter tenant’s apt number
    5. Back to input data surface
- Exception
    - The tenant has existed in the system or apt is occupied.
- Postcondition: New tenant added to the tenant list
- NonFunctional: interact with input surface

4. Input a rental payment

- Use case number: 4
- Use case name: Input a rental payment
- Summary: Enter the rental payment amount
- Actor: The landlord
- Precondition: The tenant is in the system, and is currently living in the apt
- Scenario:
    1. Enter ‘r’ to enter record rent payment page
    2. Enter tenant’s name to enter his/her page
    3. Enter amount paid
    4. Enter the month for the payment
    5. Back to main page
- Exception:
    - The tenant is not in the system, cannot exceed the payment it should be paid.
- Postcondition: input rental payment data to record
- NonFunctional: interact with paymentRecord

5. Display the Tenant List

- Use case number: 5
- Use case name: Display the Tenant List
- Actor: The landlord
- Summary: display tenant’s list, name and apt number
- Scenario:
    1. Enter ‘t’ to display tenant’s list
    2. Access to tenant Record
    3. Display tenant’s list
- Exception:
    - tenantList is empty
- Postcondition: Display the tenantList
- NonFunctional: interact with tenant record

6. Display the Rent Record

- Use case number: 6
- Use case name: Display the Rent Record
- Actor: The Landlord
- Summary: Ask user to choose options between insert, display, and quit
- Scenario:

1. Enter ‘r’ to display rent record
2. Display each RentRow stored in rent record

- Exception:
  - Scenario 2 => If the rent record does not contain any RentRow, print out a warning message and go back to the ask user input stage.
- NonFunctional:
  - We implement a caching system for display so that we don’t have to use the for loop everytime we call the display function.

7. Display the Expense Record

- Use case number: 7
- Use case name: Display Expense Record
- Summary: Display all the finances fees according to the tenants
- Actor: The landlord
- Precondition: Need to specify the budget category to which payments to be display
- Scenario:
  1. Display general information
  2. budget category with according date, month
- NonFunctional:
  - tenant’s name should be documentation

8. Display the Annual Summary

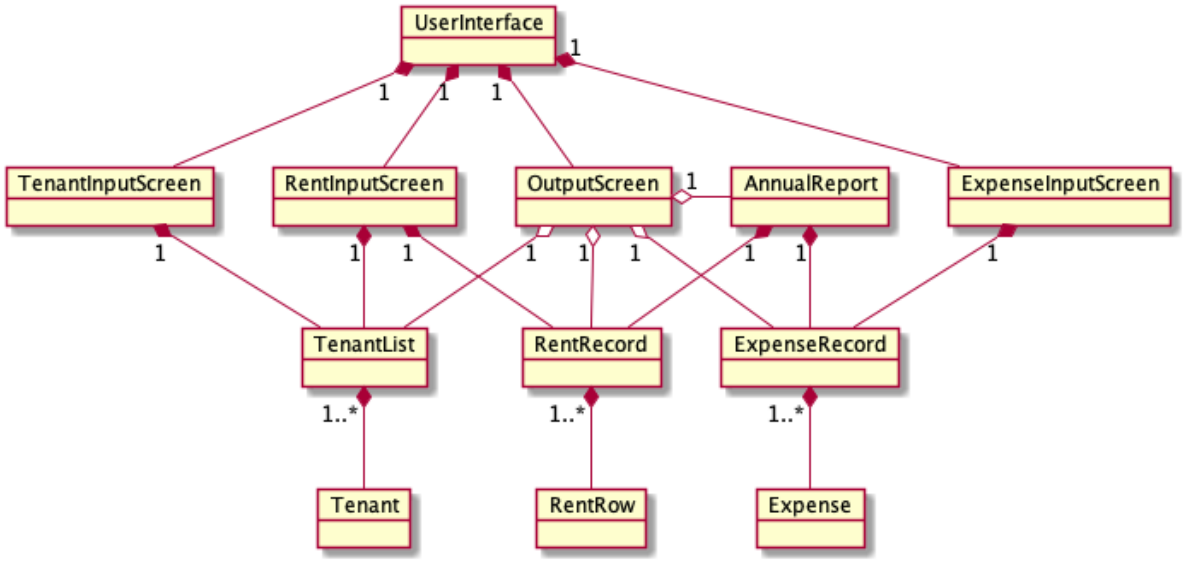
- Use case number: 8
- Use case name: Display Annual Summary
- Summary: Record annual expenses
- Actor: annual report
- Precondition: need datas from rent income and expense record
- Scenario:
  1. Display table
     - display the amount of summed rents
  2. Display budget
     a. Total of expenses for each budget category
  3. Display balance
     a. Display profit/ loss for the year to date

Requirement Modeling

Class diagram:

The class diagram uses a UML to show the relationship between the landlord and program system. Here is an overall class diagram showing the relationships and multiplicity between each class.

Class Diagram Description:



Describes-Overall Class Diagram

UserInterface attributes & contain composition relationship

- TenantInputScreen
- RentInputScreen
- OutputScreen
- ExpenseInputScreen

Display

- TenantList
- RentRecord
- ExpenseRecord
- AnnualReport

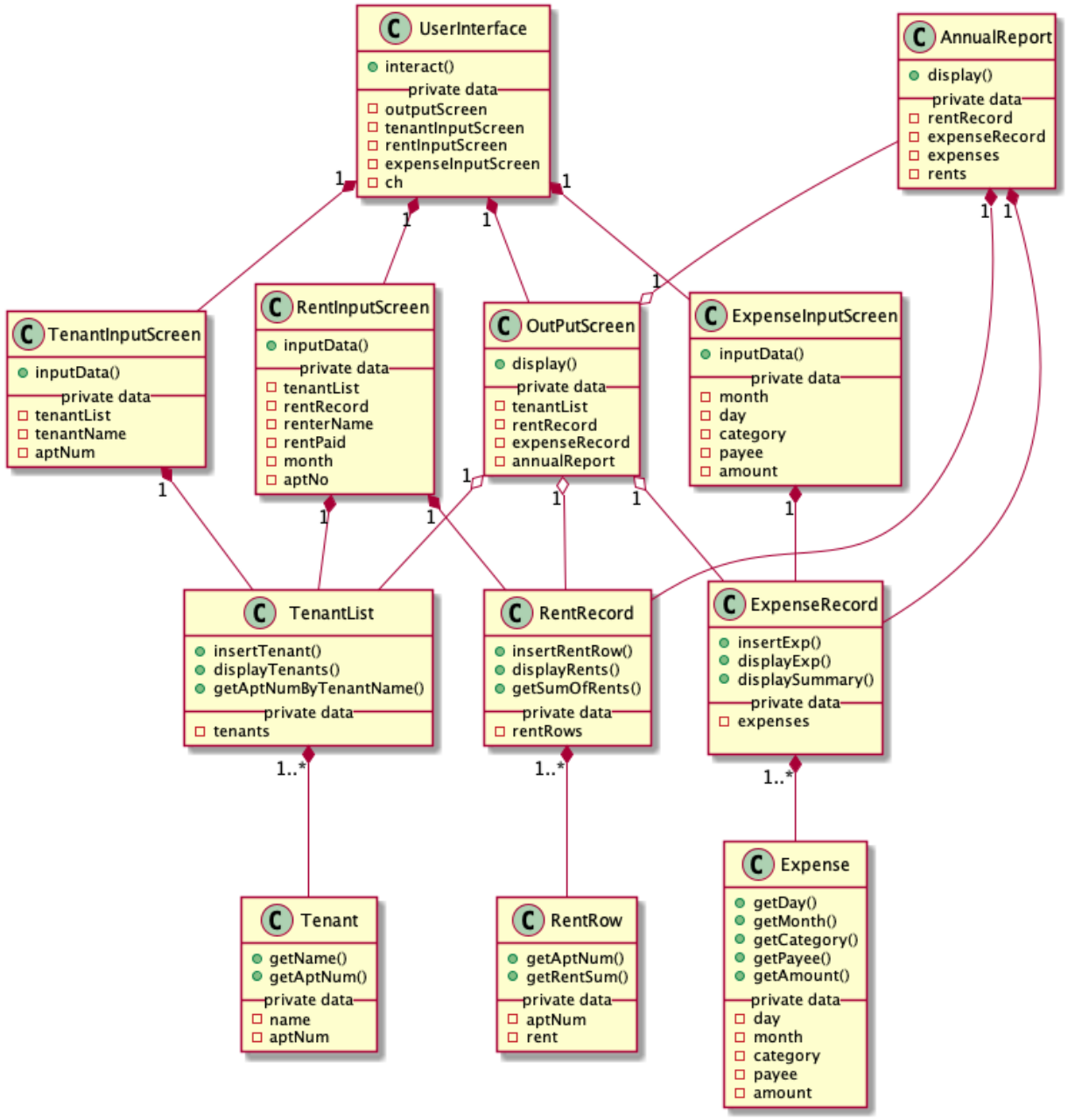
Display All

- OutputScreen

Classes with one-to-many relationship

- tenantList with Tenant class
- RentRecord with Record class
- ExpenseRecord with Expense class

Attributes & Operation-Overall Class Diagram



Class: Interface
Attributes:
- outputScreen: display all input screens
- tenantInputScreen: display tenant
- rentInputScreen: display amount of rent
- expenseInputScreen: display all the expense of tenant
- Ch:store string

Operation:
- interact()- start the program

Class: AnnualReport
Attributes:
- rentRecord: save rent amount
- expenseRecord: save expense record
- Expenses:input expense data
- Rents:input rent data

Operation:
- display(): display all the expenses amount

Class: TenantInputScreen
Attributes:
- tenantList:store tenant name and aptNum into an array
- tenantName: tenant's name
- aptNum: tenant's aptNum

Operation:
- inputData():input tenant name and aptNum

Class: RentInputScreen
Attributes:
- tenantList:list contain tenant's name and aptNo
- rentRecord:record rent amount
- renterName:tenant's name
- rentPaid:amount rent paid
- Month:month of the year
- aptNo:aptNo of the tenant

Operation:

- inputData():input rent information

Class: DBConnect

Attributes:

- tenantList: contain tenant's name and aptNo
- rentRecord: rent amount record
- expenseRecord: expense amount record

Operation:

- saveAll(): save all the information of tenant's name, aptNo, rent, and expense

Class: OutPutScreen

Attributes:

- tenantList:list contain tenant's name and aptNo
- rentRecord:record rent amount
- expenseRecord:expense amount record
- annualReport:record of all expenses

Operation:

- display():display all the data

Class: ExpenseInputScreen

Attributes:

- Month: month of the year
- Day: day of the year
- Category: define each category
- Payee:person who pay
- Amount:total amount of payment

Operation:

- inputData(): take input data

Class: TenantList

Attributes:

- Tenants: tenant's name

Operation:

- insertTenant(): input tenant's name
- displayTenant(): display tenant's name
- getAptNumByTenantName(): return aptNo by tenant's name

Class: RentRecord

Attributes:

- rentRows: number of row for total of tenants

Operation:

- insertRentRow():get total of row for each tenant
- displayRent():display the amount of rent
- getSumOfRents():add the sum of rent

Class: ExpenseRecord

Attributes:

- Expenses: expense total

Operations:

- insertExp(): get amount of expense
- displayExp(): display the expense amount
- displaySummary(): display the total amount

Class: Tenant

Attributes:

- Name: tenant's name
- aptNum: tenant's aptNum

Operations:

- getName(): return tenant's name
- getAptNum(): return tenant's aptNum

Class: RentRow

Attributes:

- aptNum: apartment number
- Rent: rent amount

Operation:

- getAptNum(): return apartment number
- getRentSum(): return total rent

Class: Expense

Attributes:

- Day: day of the year
- Month: month of the year
- Category: define each category
- Payee:person who pay
- amount:total amount of payment

Operation:

- getDay(): return day
- getMonth(): return month
- getCategory(): return category
- getPayee(): return payee
- getAmount(): return amount

Class-Responsibility-Collaborator (CRC) Modeling

A Class Responsibility Collaborator (CRC) model is a collection of standard index cards that have been divided into three section tables: class name, its responsibilities, and collaborations. A class represents a collection of objects. Responsibility of a class is the functionalities of the class; what the class knows and what it does. A collaborator is another class interacting or depending on to fulfill its responsibilities. Each CRC card represents a class, and each class has its own set of responsibilities (tasks), with some responsibilities needing to collaborate with other classes for those responsibilities to go underway.

1. Tenant	
Responsibility	Collaboration
Name	
Apt number	

2. RentRow	
Responsibility	Collaboration
Apt number	
Rent (Capacity:12)	
Return sum of all rents for 12 months	

3. Expense	
Responsibilities	Collaborations
Month	
Day	
Category	
Payee	
Amount	

4. TenantInputScreen	
Responsibility	Collaborator
Name	Tenant
Apt Number	Tenant

5. RentInputScreen	
Responsibility	Collaboration
tenant name	tenantList RentRecord
rent paid	tenantList RentRecord
month	RentRecord
apt number	tenantList RentRecord

6. ExpenseInputScreen	
Responsibilities	Collaborations
ExpenseRecord	ExpenseRecord

7. RentRecord	
Responsibilities	Collaborations
RentRows	RentRow
Ask user input for the rent record (one tenant and one month)	

8. ExpenseRecord	
Responsibilities	Collaborations
Expenses	Expense

9. TenantList	
Responsibilities	Collaborations
Tenants	Tenant
Insert tenants	Tenant
Display tenant list	Tenant

12. UserInterface	
Relationship	Collaboration
tenantInputScreen	TenantList
rentInputScreen	TenanList RentRecord
expenseInputScreen	ExpenseRecord
annual report	RentRecord Expense record
ouputScreen	OutputScreen

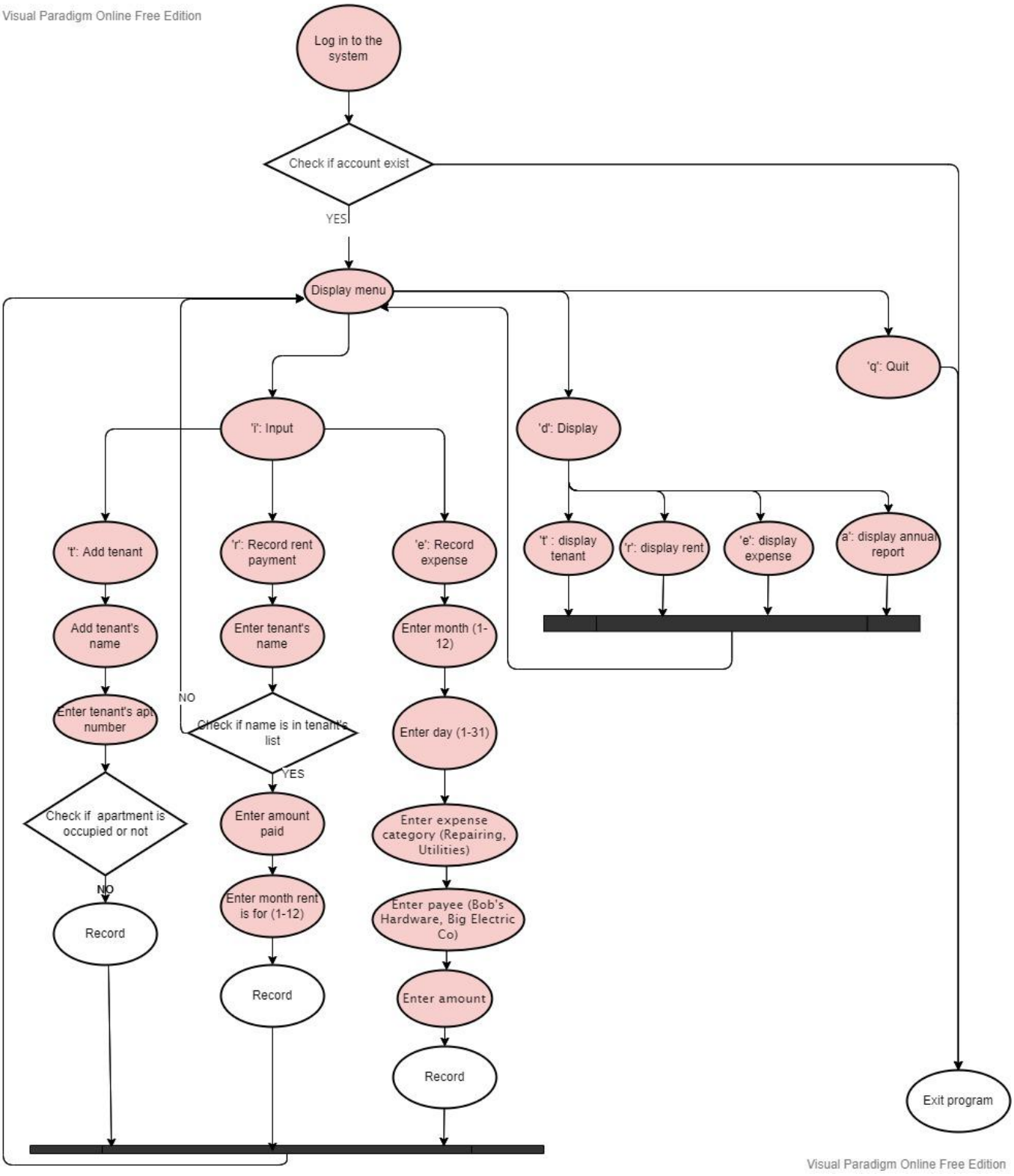
Activity diagram

The activity diagram, also known as a swim-lane diagram or cross-functional flowchart, describes how a set of activities are coordinated to provide a service. The three main benefits of activity diagrams are:

- Far less complicated than UML diagrams, making them easier for analysis and stakeholders to fully comprehend
- Allow an analyst to display multiple conditions and actors within a workflow
- Ability to describe the steps performed in a UML use-case.

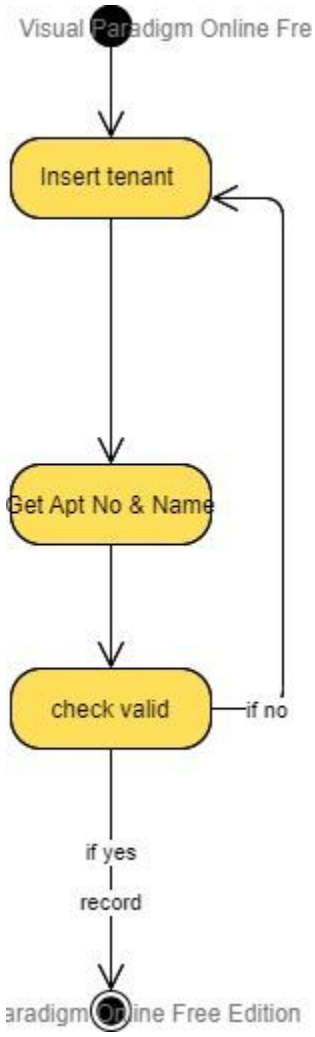
Overall Activity Diagram:

The Activity Diagram below shows the initial state of the Landlord Program. Once the landlord begins to use the program, the first activity has begun. The black circle represents the start of the program. The program prompts the landlord to enter the system, enter ‘i’ to input data, enter ‘d’ to display a report, enter ‘q’ to quit the program. After entering ‘i’, then the landlord can choose ‘Add tenant’, ‘Record Rent Payment’, or ‘Record Expense’. For ‘Add Tenant’, the program will prompt the user to enter the tenant's name and the apt number, the system then will check if the apt is occupied or not, if it is valid input, then the data will be recorded. For ‘Record rent payment’, the program asks the user to enter the tenant's name, and check if the name is in the system or not. If it is a valid name then it will ask the landlord to enter the amount and month for the payment and record it. For ‘Record Expense’, the program asks users to enter month, day, category, payee and amount, then record to the database.


Visual Paradigm Online Free Edition
Log in to the system
Check if account exist
YES
Display menu
'q': Quit
'i': Input
'd': Display
't': Add tenant
'r': Record rent payment
'e': Record expense
't' : display tenant
'r': display rent
'e': display expense
'a': display annual report
Add tenant's name
Enter tenant's name
Enter month (1-12)
Enter tenant's apt number
NO
Check if name is in tenant's list
Enter day (1-31)
YES
Check if apartment is occupied or not
Enter amount paid
Enter expense category (Repairing, Utilities)
NO
Record
Enter month rent is for (1-12)
Enter payee (Bob's Hardware, Big Electric Co)
Record
Enter amount
Record
Exit program
Visual Paradigm Online Free Edition

A detailed Activity Diagram:

Diagram shown below is a detailed activity diagram showing how inserting a tenant works. After the user chooses ‘Add Tenant’, the program prompts the user to ask a name and the Apt number, then the system will check with the database, if the Apt is not occupied then it will record the input.

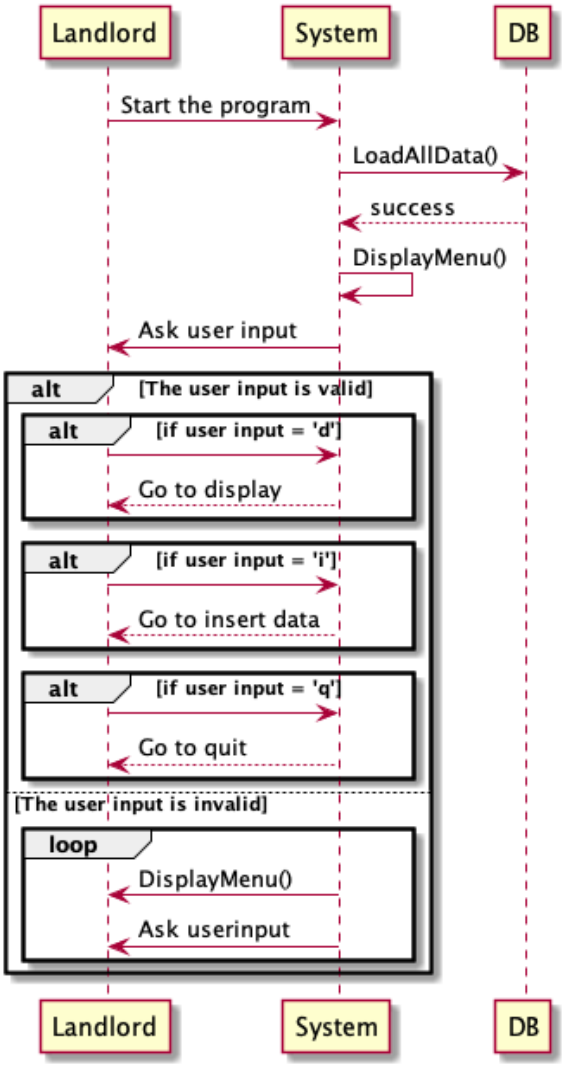


Sequence diagram:

In each of our sequence diagrams we will show how our classes interact with each other in a timed sequence for each of the functionality included in our Landlord system (input, display). Each of our diagrams all begins with the same sequence of events. First, to start the program, after opening the program, the program will automatically load data from the database and then call displayMenu() to display the menu to the user. And then it will prompt the user to display the menu interface.

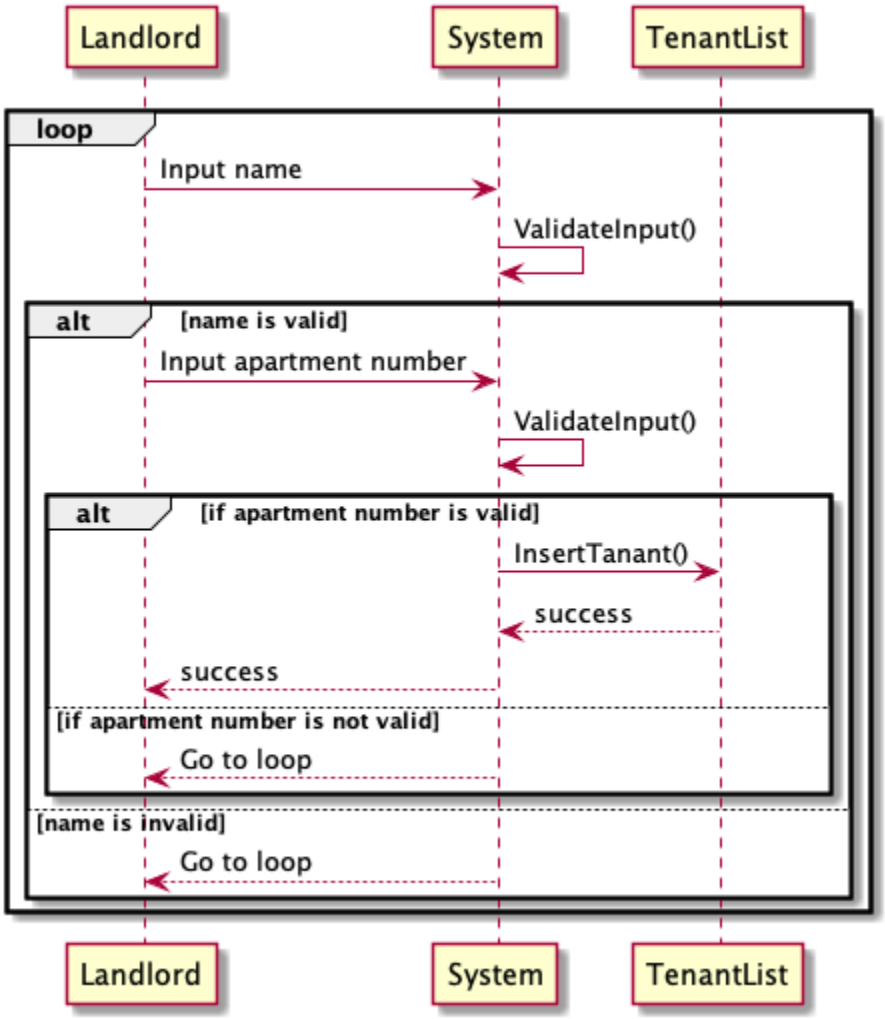
1. Start the program:

First, we will show how to start the program. After loading data from the database, the program will call displayMenu() to display the menu to the user. The user has three options to enter the next interface, ‘d’ to display data, ‘i’ to input data, ‘q’ to quit the program.



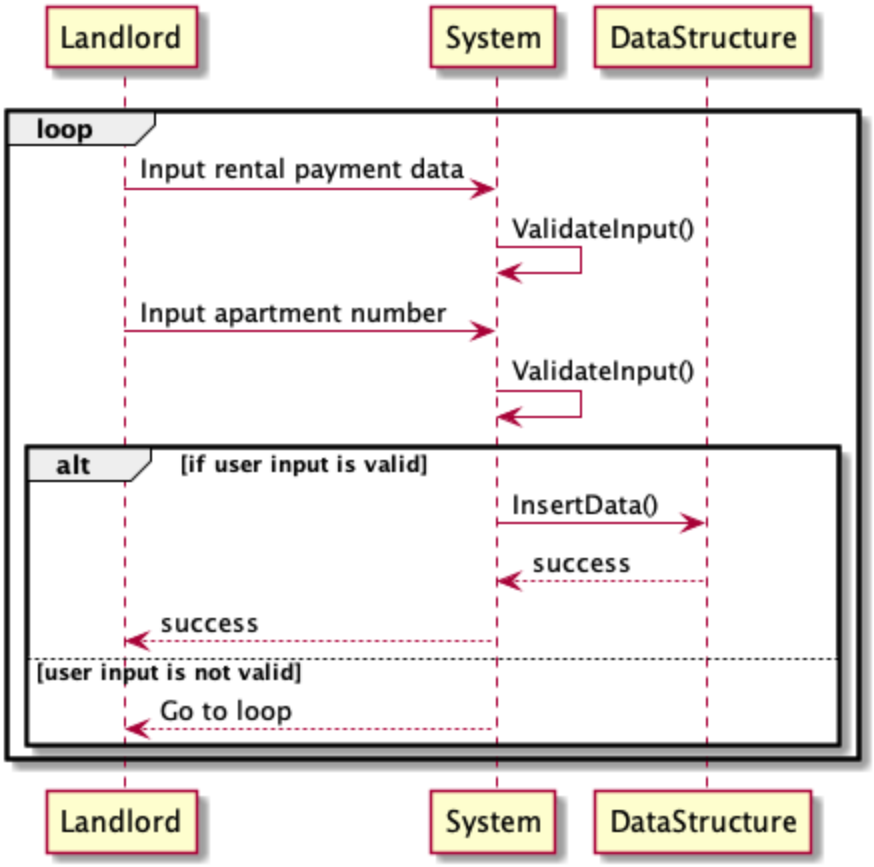
2. Add new tenant

The program will validate the input and if the input is not valid, it will keep up with the loop. Else, the program will insert a new Tenant object to TenantList.



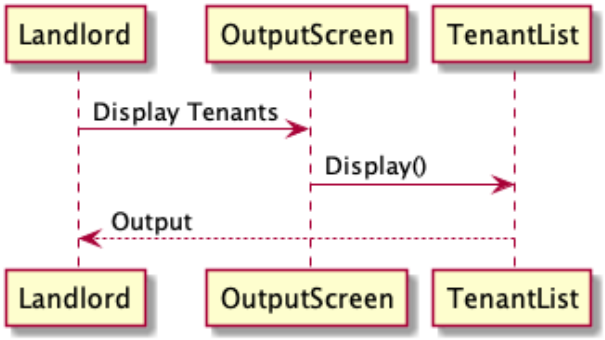
3. Input a rental payment

For input rental payment, the program will ask the user to enter a tenant name, the system will check with the database to see if the name exists in it, if the name is valid, then the user can enter the payment amount and the month for the payment, and the program calls insertData() to store data to database.



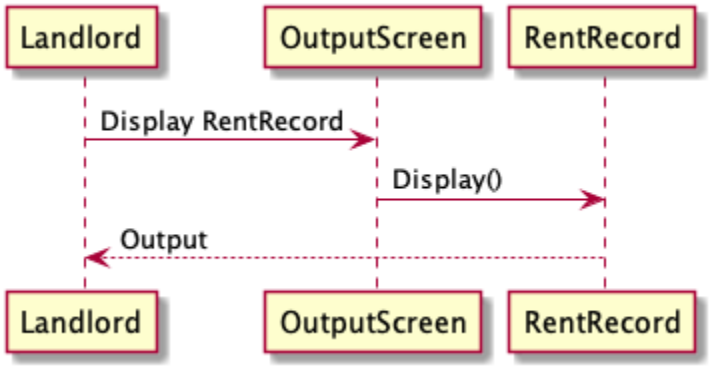
4. Display TenantList

As the landlord selects to display the TenantList, the program will call display() in TenantList so that it can display all Tenant objects in the TenantList.

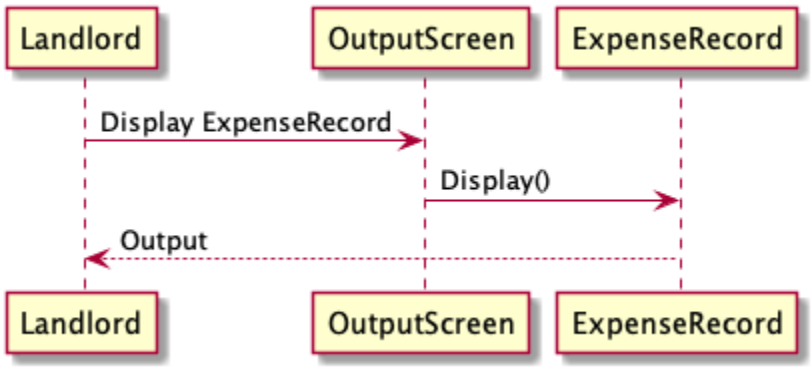


5. Display RentRecord

To display a rental record, the program will call display() in the RentRecord class and the display() function will access the database to display the data stored inside.



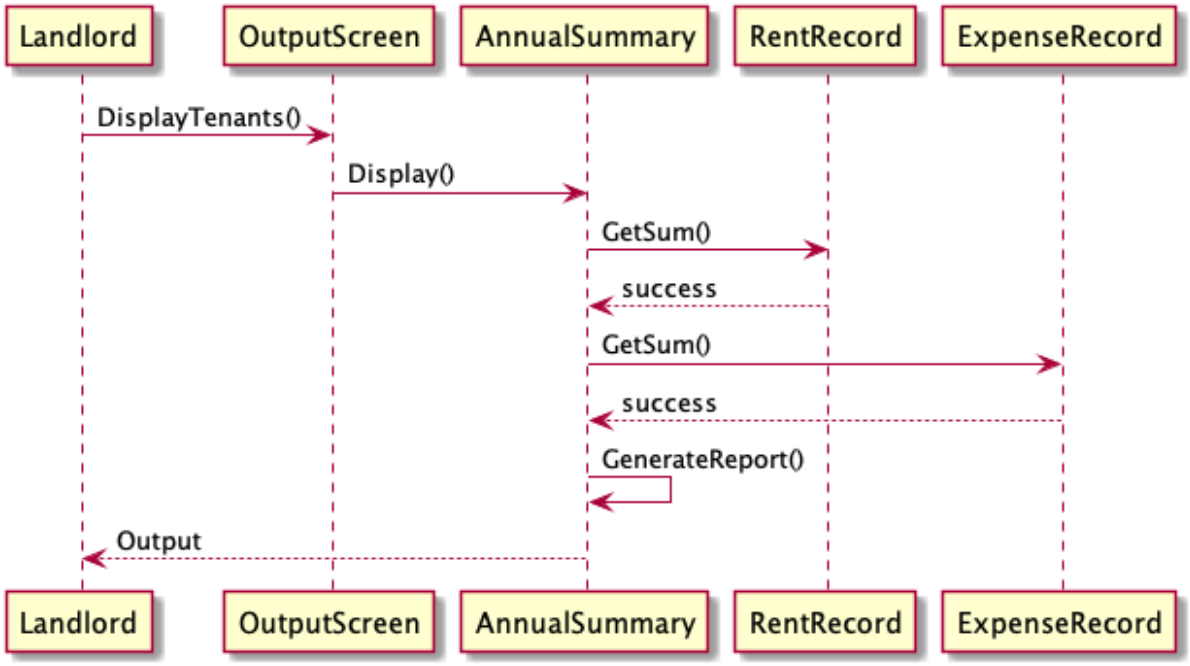
6. Display ExpenseRecord



As the landlord selects to display the ExpenseRecord, the program will call display() in ExpenseRecordso that it can display all Expense objects in the ExpenseRecord.

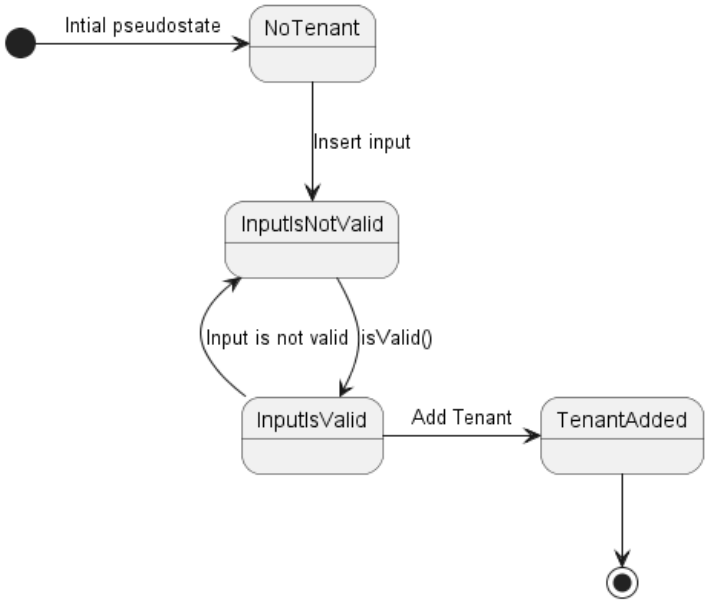
7. Display the Annual Summary

To display an annual summary, the system will call displayTenants() and it calls display() in the Annual class to display all data. In AnnualSummary, it calls GetSum() to get rent overall amount and overall expense amount. After getSum() is done, the generateRepoort() will generate a report to display all the data.



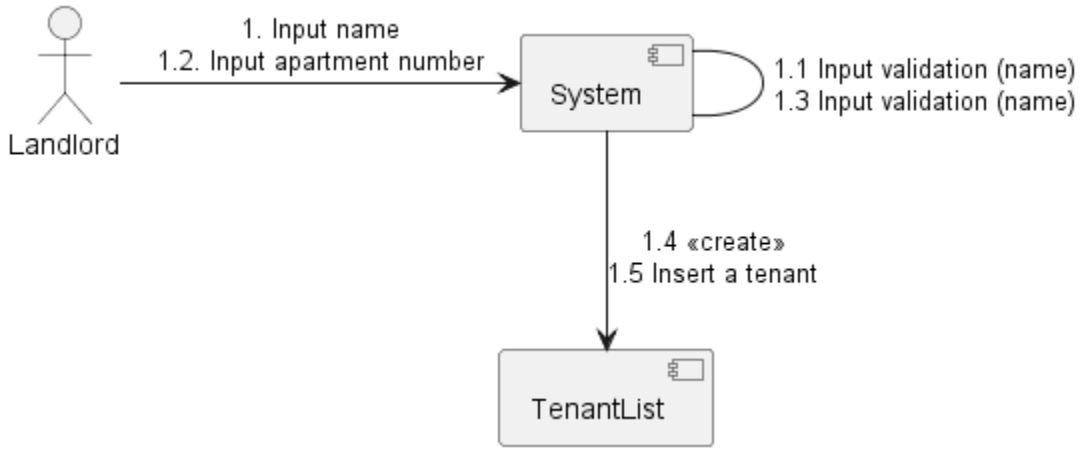
State diagram:

This state diagram shows the behavior of our Tenant object within each of our functionalities. Begin with no tenant in the tenantList, and the user enters the name and the apt number, the program will check if the inputs are valid and then if it is valid, it will call insertTenant() to record the data into database.



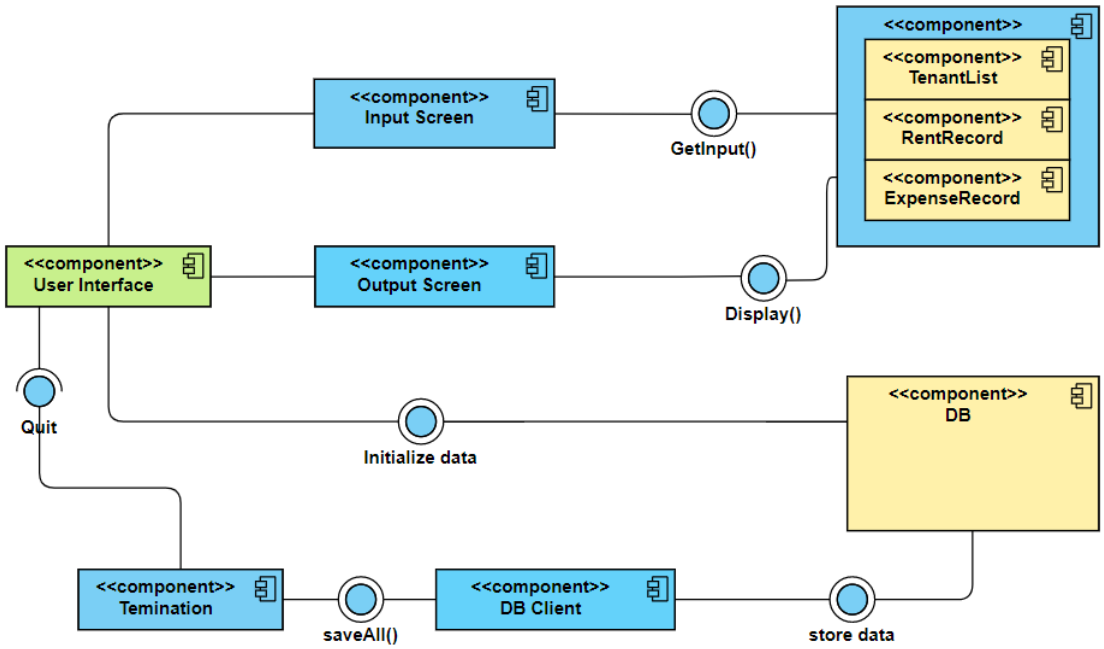
Collaboration diagram:

The diagram below is a collaboration diagram showing addTenant. It extends from the sequence diagrams; it shows the same time sequence but shows how each of the actions relates to one another based on when the methods were called. The diagram is a more simplified way of showing how our classes interact with one another.

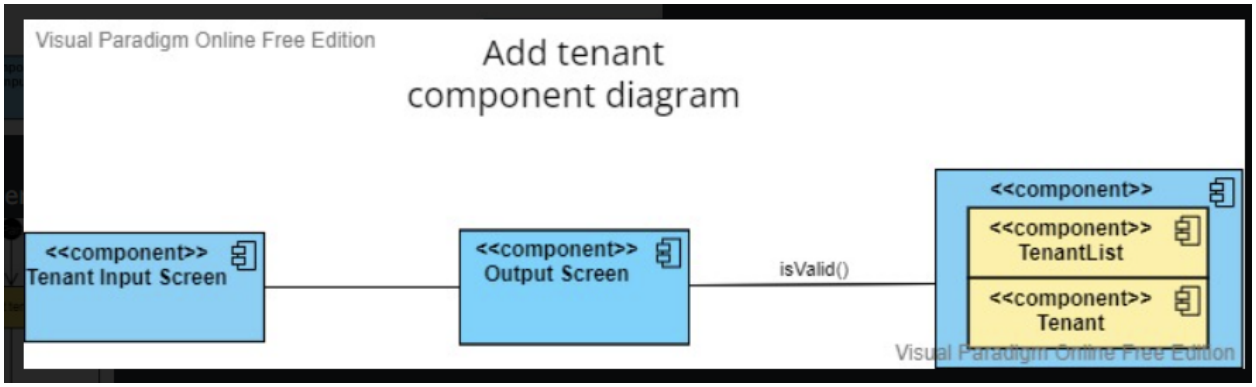


Component Diagram

Collaboration diagram provides a view of the interactions or structural relationships that occur between objects and object-like entities in the current model. To demonstrate this close relationship, we will translate the sequence diagram into a collaboration diagram.



Detail Component Detail: Add Tenant



User Interface Design

For the program, all the input and outputs are displayed in the terminal.

Main menu

```
============================
Enter
'i' to input data,
'd' to display a report,
'q' to quit program:
============================
```

Insert data

```
============================
Enter
't' to add tenant,
'r' to record rent payment,
'e' to record expense:
============================
```

Display TenantList

```
Name                APT
-----------------------
Ben                 101
Sam                 102
Jasmine             103
Hina                104
benji               105
```

Display RentRecord

```
APT     Jan     Feb     Mar     Apr     May     Jun     Jul     Aug     Sep     Oct     Nov     Dec
-------------------------------------------------------------------------------------------------
101     0.0     600.0   700.0   0.0     0.0     0.0     600.0   0.0     0.0     0.0     0.0     0.0
103     0.0     0.0     0.0     0.0     600.0   0.0     0.0     0.0     0.0     0.0     0.0     0.0
```

Display ExpenseRecord

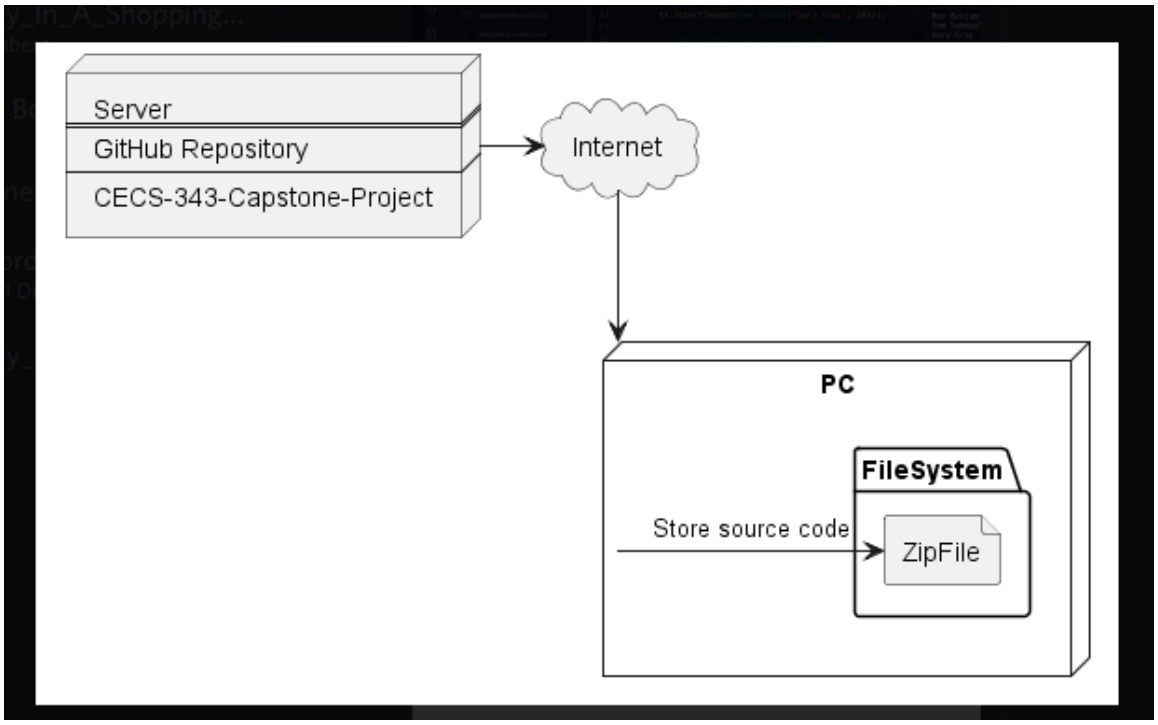
```
Date            Payee           Amount          Category
-------------------------------------------------------
12/21           Marbrisa Apt    1140.0          Utitlity Fees
```

Display AnnualReport

```
Annual Summary
----------------------
Income Rent: 2500.0
Income Expenses: 1140.0
```

Deployment Design

The Deployment Diagram shows how the program is deployed. In this project, the program is stored at a GitHub repository “CECS-343-Capstone-Project”. The internet node is a must since it connects the user device to the GitHub server. With the internet, the user device (computer, laptop) with Windows or Linux Operating system, connects to the GitHub server and clones the given repository. The user can now run the program in his/her device by compiling all the .java files and running it in the terminal.



Installation Steps

Step 1: Install JDE
Step 2: Download the start.jar file from cloud
Step 3: Save the start.jar file on any device customer using
Step 4: Run the file with commands

Coding

AnnualReport.java

```
Class
AnnualReport {
                    private RentRecord rentRecord;
                    private ExpenseRecord expenseRecord;
                    private double expenses = 0;
                    private double rents = 0;


                      AnnualReport(RentRecord rentRecord, ExpenseRecord
                    expenseRecord){
                        this.rentRecord = rentRecord;
                        this.expenseRecord = expenseRecord;
                      }
```

```
public void display () {

    // Calculate the report
    getAllSum();
    getAmountExpense();

    // Start print
    System.out.println();
    System.out.println (String.format("Annual Summary"));
    System.out.println ("----------------------");
    System.out.println ("Income Rent: " + this.rents);
    System.out.println ("Income Expenses: " + this.expenses);

    /* This can be expense by catetories

    for(Expense e: expenseRecord.getExpenseRecords()){
        System.out.println ();
    }

    */
}
```

```
    private void getAllSum(){
        for(RentRow r: rentRecord.getRentRecords()) {
            rents+=r.getSumOfRow();
        }
    }


    private void getAmountExpense() {
        for(Expense e: expenseRecord.getExpenseRecords()){
            expenses += e.getAmount();
        }
    }
}
```

DBconnect.java

```
import
java.util.
*;
        import java.io.*;


        class DBconnect {


          // Member variables
          private TenantList tl;
          private RentRecord rr;
```

```
private ExpenseRecord er;


// File names
static final String tenantFile = "tenantList.txt";
static final String rentRecordFile = "rentRecord.txt";
static final String expenseRecordFile = "expenseRecord.txt";


// Constructors
DBconnect(){}
DBconnect(TenantList tl, RentRecord rr, ExpenseRecord er){
   this.tl = tl;
   this.rr = rr;
   this.er = er;
   System.out.println("File creating ...");
   try {
      File tenant = new File(tenantFile);
      File rentRecord = new File(rentRecordFile);
      File expenseRecord = new File(expenseRecordFile);
      createNewFile(tenant);
      createNewFile(rentRecord);
      createNewFile(expenseRecord);
   } catch (IOException e) {
      System.out.println("An error occurred.");
      e.printStackTrace();
```

```
    }
    loadTenantList();
    loadRentRecord();
    loadExpenseRecord();
};


private void createNewFile(File f)throws IOException{
    if (f.createNewFile()) {
        System.out.println("File created: " + f.getName());
    } else {
        System.out.println("File " + f.getName() + " already exists.");
    }
}


private void writeToFile(String fileName, String row){
    try {
        FileWriter myWriter = new FileWriter(fileName, true);
        BufferedWriter out = new BufferedWriter(myWriter);
        out.write(row);
        out.newLine();
        out.close();
        // System.out.println(fileName + ": " + "Successfully wrote to the file.");
    } catch (IOException e) {
        System.out.println("An error occurred.");
```

```
            e.printStackTrace();
        }
    }

    // Save
    public void saveAll() throws IOException {
        try{
            File tenant = new File(tenantFile);
            File rentRecord = new File(rentRecordFile);
            File expenseRecord = new File(expenseRecordFile);
            PrintWriter writer;

            // 1. Cleaning up textfiles
            writer = new PrintWriter(tenant);
            writer.print("");
            writer.close();
            writer = new PrintWriter(rentRecord);
            writer.print("");
            writer.close();
            writer = new PrintWriter(expenseRecord);
            writer.print("");
            writer.close();

            // 2. Saving data
            saveTenantList();
```

```
            saveRentRecords();
            saveExpenseRecords();

        }catch(IOException e){
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
    private void saveTenantList(){
        for (Tenant x : tl.getTenants()){
            saveTenant(x);
        }
    }
    private void saveRentRecords(){
        for (RentRow x : rr.getRentRecords()){
            saveRentRow(x);
        }
    }
    private void saveExpenseRecords(){
        for (Expense x : er.getExpenseRecords()){
            saveExpense(x);
        }
    }
    private void saveTenant(Tenant t){
        String row = t.getName() + "," + t.getAptNum();
```

```
        writeToFile(tenantFile, row);
    }
    private void saveRentRow(RentRow r){
        int aptNum = r.getAptNum();
        double rents[] = r.getRents();
        String row = String.valueOf(aptNum)+",";
        for (int i = 0; i < 12; i++){
            if (i == 11)
                row += rents[i];
            else
                row += (rents[i] + ",");
        }
        writeToFile(rentRecordFile, row);
    }
    private void saveExpense(Expense e){
        // Date Payee Amount Category
        String s[] = {String.valueOf(e.getMonth()), String.valueOf(e.getDay()),
e.getPayee(), String.valueOf(e.getAmount()), e.getCategory()};
        String row = String.join(",", s);
        writeToFile(expenseRecordFile, row);
    }


    // Read (in progress)
    private Tenant readTenant(String row){
        String[] l = row.split(",", 3);
```

```
    String name = l[0];
    int aptNum = Integer.parseInt(l[1]);
    return new Tenant(name, aptNum);
}
private RentRow readRentRow(String row){
    String[] l = row.split(",", 13);
    double rent[] = new double[12];
    int aptNum = Integer.parseInt(l[0]);
    for (int i = 1; i < l.length; i++){
        rent[i-1] = Double.parseDouble(l[i]);
    }
    return new RentRow(aptNum, rent);
}
private Expense readExpense(String row){
    // Month Day Payee Amount Category
    String[] l = row.split(",", 5);
    int month = Integer.parseInt(l[0]);
    int day = Integer.parseInt(l[1]);
    String payee = l[2];
    double amount = Double.parseDouble(l[3]);
    String category = l[4];
    return new Expense(month, day, payee, amount, category);
}


// Load (in progress)
```

```
private void loadTenantList(){
    try {
        File myObj = new File(tenantFile);
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            Tenant t = readTenant(data);
            tl.insertTenant(t);
            // System.out.println(data);
        }
        myReader.close();
    } catch (FileNotFoundException e) {
        System.out.println("An error occurred.");
        e.printStackTrace();
    }
}
private void loadRentRecord(){
    try {
        File myObj = new File(rentRecordFile);
        Scanner myReader = new Scanner(myObj);
        while (myReader.hasNextLine()) {
            String data = myReader.nextLine();
            RentRow r = readRentRow(data);
            rr.insertRentRow(r);
        }
```

```
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
    private void loadExpenseRecord(){
        try {
            File myObj = new File(expenseRecordFile);
            Scanner myReader = new Scanner(myObj);
            while (myReader.hasNextLine()) {
                String data = myReader.nextLine();
                Expense e = readExpense(data);
                er.insertExpense(e);
                // System.out.println(data);
            }
            myReader.close();
        } catch (FileNotFoundException e) {
            System.out.println("An error occurred.");
            e.printStackTrace();
        }
    }
}
```

Expense.java

```
class Expense
{
    private int month;
    private int day;
    private String category;
    private String payee;
    private double amount;
    Expense(){}

    // More validation with month&day in ExpenseInputScreen
    Expense(int month, int day, String payee, double amount, String
category){
        this.month = month;
        this.day = day;
        this.payee = payee;
        this.amount = amount;
        this.category = category;
    }

    public String toString(){

        // 1/2 City Water 978 Utilities
        String date = this.month + "/" + this.day;
        String s[] = {date, payee, String.valueOf(amount), category};
        String r = "";
```

```
        for (String ss: s)
            r += String.format("%-15s", ss);
        return r;
    }


    // Getters
    public int getMonth(){return month;}
    public int getDay(){return day;}
    public String getCategory(){return category;}
    public String getPayee(){return payee;}
    public double getAmount(){return amount;}
}
```

ExpenseInputScreen.java

```
import java.util.Scanner;


                class ExpenseInputScreen {
                 private int month, day;
                 private String category, payee;
                 private double amount;
                 private ExpenseRecord exRc;


                 ExpenseInputScreen(ExpenseRecord exRc){
                    this.exRc = exRc;
```

```
}

public void getInput(){
  Scanner in = new Scanner( System.in );
  System.out.println("Enter month (1-12):");
  month = in.nextInt();//check range?

  System.out.println("Enter day (1-31):");
  day = in.nextInt();//check range?

  System.out.println("Enter expense category :");
  category = in.nextLine();
  System.out.println("Enter payee:");
  payee = in.nextLine();
  System.out.println("Enter amount:");
  amount = in.nextDouble();

  Expense e = new Expense(month,day, payee, amount,
category);
  if(isValid())exRc.insertExpense(e);
  else System.out.println("Inserted invalid month / day");


  }
  private boolean isValid(){
     return (month<=12 && month >=1 && day<=31 &&
day>=1);
```

```
    }
  }
```

ExpenseRecord.java

```
import
java.util.ArrayList;
                class ExpenseRecord {
                  private ArrayList<Expense>expenseRecords = new
ArrayList<Expense>();
                  public ArrayList<Expense> getExpenseRecords(){return
expenseRecords;}
                  ExpenseRecord(){}


                  // month, day, amount validation in inputScreen
                  public void insertExpense(Expense e){
                     expenseRecords.add(e);
                  }
                  public void display(){
                     String r = "";
                     String l [] = {"Date", "Payee", "Amount", "Category"};
                     for (String s : l){
                        r+=String.format("%-15s", s);
                     }
                     System.out.println();
                     System.out.println(r);
```

```
System.out.println("------------------------------------------------------------
");
        for(Expense e: expenseRecords){
            System.out.println(e);
        }
        System.out.println();
    }
}
```

Main.java

```
import
java.io.IOException;
                class Main {
                  public static void main(String[] args) throws IOException {
                    UserInterface ui = new UserInterface();
                    while(true){
                      ui.interact();
                    }
                  }
                }
```

OutputScreen.java

```
class
OutputScreen {
            private TenantList tl;
```

```
  private RentRecord rr;
  private ExpenseRecord er;
  private AnnualReport ar;

  OutputScreen(TenantList tl, RentRecord rr, ExpenseRecord er,
AnnualReport ar){
    this.tl = tl;
    this.rr = rr;
    this.er = er;
    this.ar = ar;
  }

  void display(char ch){
    if (ch == 't') tl.display();
    else if(ch == 'r') rr.display();
    else if(ch == 'e') er.display();
    else if(ch == 'a') ar.display();
    else {// Error needs to be handled before char input
      System.out.println("ERROR: Invalid input");
    }
  }
}
```

RentRecord.java

```
import
java.util.Arra
yList;
                class RentRecord {
                  private ArrayList<RentRow>rentRecords = new
                ArrayList<RentRow>(20);
                  public ArrayList<RentRow> getRentRecords(){return rentRecords;}
                  public void insertRentRow(RentRow r){
                     rentRecords.add(r);
                  }


                  // More validation in RentInputScreen
                  public void insertRent(int aptNum, int month, double rent){
                     boolean doesExist = false;
                     for (RentRow x : rentRecords){
                        if(x.getAptNum() == aptNum){
                           doesExist = true;
                           x.setRent(month, rent);
                           break;
                        }
                     }
                     if (!doesExist){
                        RentRow r = new RentRow(aptNum);
                        r.setRent(month, rent);
                        rentRecords.add(r);
                     }
```

```
  }
  public void display(){
      String l[] = {"APT", "Jan", "Feb", "Mar", "Apr", "May", "Jun", "Jul",
"Aug", "Sep", "Oct", "Nov", "Dec"};
      System.out.println();
      for (String s: l)System.out.format("%-8s", s);
      System.out.println();

System.out.println("----------------------------------------------------------------
-----------------------------------");
      for (RentRow x : rentRecords)System.out.printf("%s\n", x);
      System.out.println();
  }
}
```

RentRow.java

```
import java.util.*;


     class RentRow {
        private int aptNum;
        private double rents[] = new double[12];
        RentRow(int aptNum){
           this.aptNum = aptNum;
        }
        RentRow(int aptNum, double rents[]){
           this.aptNum = aptNum;
```

```
    this.rents = rents;
}
public void setRent(int month, double rent){
    this.rents[month-1] = rent;
}


public double getSumOfRow(){
    double sum = 0;
    for (double d : rents){
        sum+=d;
    }
    return sum;
}


// Getters
public int getAptNum(){return this.aptNum;}
public double[] getRents(){return this.rents;}
public String toString(){
    String r = String.format("%-8s", String.valueOf(aptNum));
    for (double d: rents){
        r += String.format("%-8s", d);
    }
    return r;
}
```

```
}
```

RentRowInputScreen.java

```
import
java.util.Scanner;
                        import java.util.HashMap;


                        class RentRowInputScreen {
                          private int aptNum;
                          private double rent;
                          private String name;
                          private int month;
                          private TenantList tl;
                          private RentRecord rr;
                          HashMap<String,Integer> tenant;

                          RentRowInputScreen(TenantList tl, RentRecord rr){
                             this.tl = tl;
                             this.rr = rr;
                             tenant = new HashMap<String,Integer>();
                          }

                          private boolean isValidName(){
                            for (Tenant t:tl.getTenants()){
                             tenant.put(t.getName(),t.getAptNum());
```

```
    }
    if(tenant.containsKey(name)){ //check if name is in the
tenantList
     return true;
    }
     return false;
   }
   private boolean isValidMonth(){
     return (month<=12 && month >0);
   }

   public void getInput(){
    Scanner in = new Scanner(System.in);
    System.out.println("Enter tenant's' name:");
    name = in.nextLine();
    if(isValidName()){
       System.out.println("Inserted invalid name.");
    } else {
       return;
    }
    System.out.println("Enter amount paid:");
    rent = in.nextDouble();
    System.out.println("Enter month rent is for(1-12):");
    month = in.nextInt();
    if (isValidMonth()){
       System.out.println("Inserted invalid month.");
```

```
            } else {
                return;
            }
            if(isValidName()){
                rr.insertRent(aptNum, month, rent);
            }
            // in.close();
        }
    }
```

Tenant.java

```
class Tenant {
    private String name;
    private int aptNum; // three digits
    Tenant(){}
    Tenant(String name, int aptNum){
        this.name = name;
        this.aptNum = aptNum;
    }

    // Getters
    public String getName(){return this.name;}
    public int getAptNum(){return this.aptNum;}

    public String toString(){
        return String.format("%-15s%-15s", this.getName(), this.getAptNum());
    }
}
```

TenantInputScreen.java

```
import
java.util.Scanner;
                        import java.util.HashMap;


                        class TenantInputScreen {

                          private TenantList tl;
                          private String name;
                          private int aptNum;


                          HashMap<String,Integer> tenant = new
                        HashMap<String,Integer>();


                          TenantInputScreen(TenantList tl){
                             this.tl = tl;
                          }

                          public void getInput(){
                            System.out.println("Enter tenant's name:");//Ask for name
                            Scanner in = new Scanner( System.in );
                            name = in.nextLine();
                            if(!isValidName()){
                               System.out.println("The name is already in the tenant list.");
                               return;
                            }
```

```
    System.out.println("Enter the apartment number:");//Ask for
apt#
    aptNum = in.nextInt();
    if(!isValidAptNum()){
        System.out.println("The apartment is already occupied.");
        return;
    }
    Tenant t = new Tenant(name,aptNum);
    tl.insertTenant(t);

    // in.close();//close scanner
  }

  private boolean isValidName(){
      for (Tenant t:tl.getTenants()){
          if(t.getName().equals(this.name)){
              return false;
          }
      }//end isValid
      return true;
  }

  private boolean isValidAptNum(){
      for (Tenant t:tl.getTenants()){
          if(t.getAptNum() == this.aptNum){
```

```
            return false;
          }
        }//end isValid
        return true;
      }
    }//end class
```

TenantList.java

```
import
java.util.ArrayList;


  class TenantList {
    private ArrayList<Tenant>tenants = new
  ArrayList<Tenant>(20);
    public ArrayList<Tenant> getTenants() {return tenants;}

    // Return aptNum by name
    public int getAptNum(String name){
      for (Tenant x : tenants){
        if (x.getName().equals(name)){
          return x.getAptNum();
        }
      }
      return 0;
    }
```

```
public void insertTenant(Tenant t){
    boolean isValid = true;
    for (Tenant x : tenants){
      if (!inputValidation(x, t)){
         isValid = false;
         break;
      }
    }
    if (isValid) {
       tenants.add(t);
    };
}
// TenantInputScreen will do more validation
private boolean inputValidation(Tenant t1, Tenant t2){
    if (t1.getName().equals(t2.getName())){
       System.out.println("The tenant's name has to be unique.");
       return false;
    } else if(t1.getAptNum() == t2.getAptNum()) {
       System.out.println("The unit is already occupied.");
       return false;
    } else {
       return true;
    }
}
public void display(){
```

```
            System.out.println();
            System.out.println(String.format("%-15s%-15s", "Name",
        "APT"));
            System.out.println("----------------------");
            for (Tenant x : tenants)System.out.println(x);
            System.out.println();
          }
        }
```

UserInterface.java

```
import java
util.Scanner
;
                import java.io.IOException;
                class UserInterface {

                  /* Data models */
                  private TenantList tl;
                  private RentRecord rr;
                  private ExpenseRecord er;
                  private AnnualReport ar;
                  private DBconnect db;

                  /* Input screens */
                  private TenantInputScreen ti;
```

```
private RentRowInputScreen ri;
private ExpenseInputScreen ei;

/* Member variables */
private OutputScreen output;
private char ch = ' ';
private Scanner sc = new Scanner(System.in);

UserInterface(){
    this.tl = new TenantList();
    this.rr = new RentRecord();
    this.er = new ExpenseRecord();
    this.ti = new TenantInputScreen(this.tl);
    this.ri = new RentRowInputScreen(this.tl, this.rr);
    this.ei = new ExpenseInputScreen(this.er);
    this.ar = new AnnualReport(rr, er);
    this.output = new OutputScreen(tl, rr, er, ar);
    this.db = new DBconnect(tl, rr, er);
}

/* Public */
public void interact() throws IOException {
    getUserInputForMenu();
    if (ch == 'd'){
```

```
            // Get user input
            getUserInputForOutput();

            // Using OutputScreen to display
            output.display(this.ch);

        } else if (ch=='i'){

            // Get user input
            getUserInputForInsert();

            // Insert data
            insertData();

        } else {
            System.out.println("\nBye!\n");
            saveAlltoDB();
            sc.close();
            System.exit(0);
        }
    }
```

```
/* Private */
private void saveAlltoDB() throws IOException {
    this.db.saveAll();
}


private void insertData(){
    if (this.ch == 't'){
        this.ti.getInput();
    } else if(this.ch=='r'){
        this.ri.getInput();
    } else if(this.ch=='e'){
        this.ei.getInput();
        //System.out.println("In progress");
    }
}

private void getUserInputForMenu(){
    char temp = ' ';
    while (!isValidMenuInput(temp)){
        printStartMenu();
        temp = sc.next().charAt(0);
    }
    this.ch = temp;
}
private void getUserInputForOutput(){
```

```
    char temp = ' ';
    while (!isValidOutputInput(temp)){
        printOutputMenu();
        temp = sc.next().charAt(0);
    }
    this.ch = temp;
}
private void getUserInputForInsert(){
    char temp = ' ';
    while (!isValidInsertInput(temp)){
        printInsertMenu();
        temp = sc.next().charAt(0);
    }
    this.ch = temp;
}

private boolean isValidMenuInput(char ch){
    return (ch=='d' || ch=='i' || ch=='q');
}
private boolean isValidOutputInput(char ch){
    return (ch=='t' || ch=='r' || ch=='e' || ch=='a');
}
private boolean isValidInsertInput(char ch){
    return (ch=='t' || ch=='r' || ch=='e');
}
```

```
  // Printing
  private void printStartMenu(){
      System.out.println("\n============================");
      System.out.println("Enter\n'i' to input data,\n'd' to display a report,\n'q' to
quit program:");
      System.out.println("============================\n");
  }

  private void printOutputMenu(){
      System.out.println("\n============================");
      System.out.println("Enter\n't' to display tenants,\n'r' to display rents,\n'e'
to display expenses,\n'a' to display annual report:");

      System.out.println("============================\n");
  }

  private void printInsertMenu(){
      System.out.println("\n============================");
      System.out.println("Enter\n't' to add tenant,\n'r' to record rent
payment,\n'e' to record expense:");
      System.out.println("============================\n");
  }
}
```

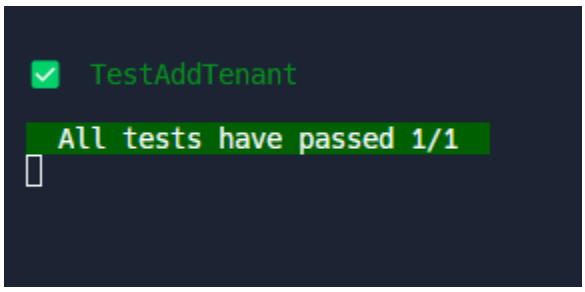
Unit Testing

For our unit testing, we are creating two tenantList objects, the first one we are using insertTenant() to add a tenant to the tenantList, and the second object is our expected result. And we use two boolean variables to test if the following insertions are successful or not. We expected a true value on assertFalse().

```
@Test
public void TestAddTenant() {
  TenantList tl = new TenantList();

  tl.insertTenant(new Tenant("Ben Benjamin", 101));
  tl.insertTenant(new Tenant("Sam Sammual", 102));
  tl.insertTenant(new Tenant("Gary Gray", 103));

  // Not expected to be added
  boolean b1 = tl.insertTenant(new Tenant("Gary Gray", 104));
  boolean b2 = tl.insertTenant(new Tenant("Tom Hanks", 101));
  assertFalse(b1);
  assertFalse(b2);
  //assertEquals(tl, expected);

}
```

Sample Run-time output:

Display Tenant



Summary:

The Capstone program was designed to organize all the data and keep track of all the tenant information. The designs show the flow of how the program will execute and how it might fit into actual hardware. Although there is no real tenant name or actual database, this program represents part of how the actual program will work.

The landlord will start the program with a prompt display input, display, or quit the program. The input option will contain add tenants, rent payment, and record expenses. All the input datas will be saved into the database for each use and will be deleted for a new use.

The display option will show another display prompt of display of tenants, rents, expenses, annual report. Those displays will show the amount for a year based on the option of display the landlord chooses.

The last option is to quit the program, the program will quit if this option is chosen or else it will continue to display the display prompt after each option was performed.