CSIS 3275 – Software Engineering

Term Project

Project Title: **Customer Self-Regularization System**

Student Name: **Ronaldo Aparecido de Oliveira**

Student ID: **300305192**

Submitted to: **Dr. Mahmood Al-Humaimidi**

Douglas College – CSIS department

New Westminster, BC, Summer 2020

# **1. Introduction**

Unfortunately, due to this recent outbreak, many people lost your jobs and, consequently, they are not able to pay your bills. In this reality, the *RAO Systems Solutions* proposed the development of a system to allow the customer to reorganize their debits, seeking to minimize default with the company.

The main justification for the development of this system was to allows each customer in default to define the best payment option among some conditions defined by company, within a specific and exclusive web area. In other words, the company wants to recover credits from defaulting consumers with an easy payment, accepting a reduction in interest and fees, and using a self-regularization procedure.

In the preliminary investigation, this information was identified:

* *Limitations*: some kind of debts cannot be included in the process because requires a personal negotiation – this rule will be defined by the Debt Manager and the customer will see the debts but he will not be able to install them.
* *Constraints*: customer must have an account and this profile should allow him to access the self-regularization module; and some debts will not be included in the procedure.
* *Costs*: besides the system development cost, the company will have an extra “cost” because it will not receive a part of the interest and fees. The team that will support the system is another cost (permanent).
* *Benefits*: recover credits that in normal situation would be difficult due to customer situation and strengthening the relationship with the costumer.
* *Feasibility*: the study showed that both technical and schedule are well-determined, however the operational and economic should be more detailed in the next steps.
* *Risks*: the integration with data warehouse due to access to private information; and the new definitions of accountability procedures to avoid any tax problem.

# **2. Project Planning**

After the Preliminary Investigation and meetings with stakeholders, it was possible to define the scope of the system:

* The system should be integrated with corporate systems, mainly with the data warehouse.
* Only customers with a specific profile will be included in this program – the profile is defined by Debt Manager.
* All complains, suggestions, and compliments must be saved, classified, and organized to allow the manager to create reports and indicators.
* The system should be web to allow the customers access from any place – it is not included any mobile application in this project.
* The system must be able to manager all communication with customers (electronic and paper-based) with traces and identification.
* The system must allow the manager to monitor all actions in each self-regularization program using a Dashboard – the indicators will be defined throughout the project.

It is important to highlight that the system will not develop payment system, using the solution already developed in the company, as well as the system to communicate with the customers, using a commercial version already purchased.

Regarding to the planning, it was forecasted these deliverables throughout the project:

* Final Report of Project Planning
* Models from Modeling Process
* Code of System Developed
* Code of Dashboard Developed
* System installed and tested
* Manuals, User’s Guide and Training

In the Gantt Chart below is possible to identify each task of the project and the milestones for each phase.

A screenshot of text

Description automatically generated

**Figure 1 – Project Schedule – Gantt Chart**

Considering the schedule (figure 1), it is estimated 182 business days (255 days) to complete all tasks and delivery all products.

# **3. Requirement Analysis Models**

After the definition of schedule, deliverables and scope, a meeting with all stakeholders was taken to present these products and to define the official team that will be in charge to develop the system. Moreover, all questions were answered by the Project Manager including, costs, time, technology, migration, benefits, and result projections. It is interesting to note that technology and its “complexity” were the main factor of concern.

It was determined by each department manager the name of the employee responsible to provide support to the development team, and to evaluate and validate the product when applicable to his department.

In the sequence is presented a set of details of the system, including the requirements (functional and non-functional) and diagrams to support the development of product.

## **3.1. Systems Requirements**

The requirements were separate in two groups: functional and non-functional.

*Functional requirements*

* Allow the customer to check the status of their debts, requesting assistance when necessary.
* Allow the client to make full payment of the debt or installments.
* Allow the debt manager to view each customer's debt, payment, and installment history.
* Allow the financial manager to generate a report containing an overview by sex, age range, debt range and geographic region.

*Non-Functional Requirements*

* Security: ensure that the customer accesses his information securely, avoiding improper access or misuse of the information.
* Traceability: it must be possible to monitor the entire process, identifying who, when, where and what was done.
* Ease of use: allow the use of the system with minimal training - it must be intuitive.

## **3.2. Scenarios and Use-Cases**

For each functional requirement was created a scenario and its use-case. From these results was design the Scenario-Based Model using UML, and the description of use-cases where is possible to understand the preconditions, postconditions, successful flow and alternative flow.

### 3.2.1. Scenarios for Each Functional Requirement

As a use case represents the functionality or actions, it can have multiple “paths”. A scenario is a specific path that the user selected to follow.

Below it is presented 4 (four) scenarios that was created to understand some “paths” of the system.

*Scenario # 1:*

Customer wants to pay their debts, accesses his personal area (with personal user and password), sees his debts, and indicates how he want to pay (in full, or in installments), or whether he would like to contest something. If in doubt or contestation, he requests clarification using virtual support or sending a message through the system itself.

*Scenario # 2:*

The customer decides to pay in full via credit card or bank transfer, inserting the information in the fields and accepts the conditions. Or the customer decides to install the debt in installments, using one of the system's options, as well as the means of payment of the installments. A contract is sent to his e-mail with all conditions and information.

*Scenario # 3:*

The debt manager would like to know how the process is and use the system monitoring module to monitor all operations. It is possible to analyze sets of information, such as, the debt stock, the amount of full and installment payments, the number of user accesses, and projections. With this information he can decide what to do in real time, depends on the necessity.

*Scenario # 4:*

In regular periods (days, weeks, months, or quarters), the financial manager would like to have a wider view of the whole process. To this, he decides to generate some management reports and analyze the general dashboard. In this dashboard, he can identify some important performance indicators, allowing him to take broader decision.

### 3.2.2. Scenario-Based Model using UML

This type of model allows to organize the scenarios and obtain the system functionality using some level of abstraction. The figure 2 shows 9 user cases (the oval elements) and 4 users (customer, debt manager, ERP, financial manager). The lines represent the interaction among the elements. For example: when a customer wants to check his debts, the user case “Check the Debts” is executed, using information from ERP and it can call the user case “Pay the Debts”, in case that the customer decides to pay a debt.

A close up of a map

Description automatically generated

**Figure 2 – Scenario-Based Model using UML**

### 3.2.3. Use-Cases Details

Each use-case must be “exploded” to become more understandable. The boxes below present the detailing of 4 (four) use-cases. The name of each one is on the top and the information is self-explanatory.

|  |  |
| --- | --- |
| **CHECK THE DEBTS** | |
| Name | Check the Debts |
| Actor | Customer |
| Description | Customer wants to know much he has in debts to decide if he will pay or not. |
| Successful Completion: | 1. Customer accesses his private area with his user and password  2. Customer selects the option to see his debts  3. Visualize the report with all debts  4. Print if necessary |
| Alternative: | 1. Customer access your private area with his user and password  2. Customer is not able to see the debts  3. Customer sends a message to support |
| Precondition: | Customer must have a user and password previously  Customer must have accessed his private area |
| Postcondition: | None |
| Assumptions: | None |

|  |  |
| --- | --- |
| **PAY THE DEBTS** | |
| Name | Pay the Debts |
| Actor | Customer |
| Description | Customer wants to pay all debts or installments them |
| Successful Completion: | 1. Customer accesses your private area with his user and password  2. Customer selects the debts that he would like to pay  3. Customer decides if he wants to pay or installments  4. Customer decides to pay all debits  4.1. Customer selects the payment method  4.2. Customer inserts the information to pay  4.3. Customer pays the debts |
| Alternative: | 1. Customer decides to pay only some debts (installment)  1.1. Customer selects the desired installment method  1.2. Customer selects the payment method  1.3. Customer inserts the information to pay  1.4. Customer pays the debts |
| Precondition: | Customer must have a user and password previously |
| Postcondition: | All or some selected debts paid |
| Assumptions: | None |

|  |  |
| --- | --- |
| **PROCESS MONITORING** | |
| Name | Process Monitoring |
| Actor | Debt Manager |
| Description | Debt manager needs to know how the operations is running |
| Successful Completion: | 1. Debt manager accesses your private area with his user and password  2. Debt manager selects the ‘live’ dashboard with current information  3. Debt manager analyzes the information to take fast decision |
| Alternative: | None |
| Precondition: | Debt manager must have a user and password previously  Debt manager must have registered the payment methods  Debt manager must have registered the installment rules |
| Postcondition: | None |
| Assumptions: | Customer must have accessed and paid or install some debts |

|  |  |
| --- | --- |
| **MANAGEMENT MONITORING** | |
| Name | Management Monitoring |
| Actor | Financial Manager |
| Description | Manager wants to have a wider view of all operations to take management decisions |
| Successful Completion: | 1. Financial manager accesses your private area with his user and password  2. Financial manager selects the manager dashboard  3. Financial manager analyzes the information to take decision |
| Alternative: | None |
| Precondition: | Financial manager must have a user and password previously |
| Postcondition: | None |
| Assumptions: | Debt manager must have registered the payment methods  Debt manager must have registered the installment rules  Customer must have accessed and paid or install some debts |

After to planning each use-case and its internal process, now we can design the “live” of each one, using activity and swimlane diagrams.

## **3.3. Activity Diagrams for Use-Cases**

For each scenario described above was created an activity diagram. In this type of diagram is possible to understand the “steps” that the customer can take when execute the system according to the scenario. It is very simple to understand and self-explanatory too.

*Scenario # 1:*

A picture containing table, kitchen, cat, standing

Description automatically generated

*Scenario # 2:*

A picture containing camera, table, white, man

Description automatically generated

*Scenario # 3:*

A picture containing indoor, kitchen, table, white

Description automatically generated

*Scenario # 4:*

A picture containing indoor, kitchen, sign, white

Description automatically generated

## **3.4. Swimlane Diagrams for Use-Cases**

Swimlanes add an extra layer of information of each activity diagram because, besides the customer actions, it is also represented the execution of other elements. For example, in the first figure below it is possible to see that 6 actions are taken by the customer and one is executed by ERP (data warehouse). Although, there is new information in the diagram, it is still simple to understand.

*Scenario # 1:*

A screenshot of a cell phone

Description automatically generated

*Scenario # 2:*

A screenshot of a social media post

Description automatically generated

*Scenario # 3:*

A screenshot of a cell phone

Description automatically generated

*Scenario # 4:*

A screenshot of a cell phone

Description automatically generated

## **3.5. Class Diagram**

Using the scenario descriptions and the design of each use-cases, it was performed a process to try to identify the information behind each nouns, adjectives, and verbs. From this analysis, it was executed another process to define each feature necessary to be stored in the database. The whole process, from the identification of nouns, adjectives, and verbs, until to obtain the final class diagram with its properties and methods, is presented below.

### 3.5.1. Identification of keywords

This step is very simple and consists to highlight the nouns and verbs that can be an object or a functionality in the class or system. This identification was performed for each scenario and the results were consolidated in tables showed in the item 3.5.2.

Scenario # 1:

**Customer** *wants* to pay their **debts**, *accesses* his **personal area** (with personal user and password), *sees* his **debts**, and *indicates* how he *wants* to *pay* (in full, or in **installments**), or whether he *would like* to *contest* something. If in **doubt** or **contestation**, he *requests* **clarification** using **virtual support** or *sending* a **message** through the system itself.

Scenario # 2:

The **customer** *decides* to *pay* in full via **credit card** or **bank transfer**, *inserting* the information in the fields and *accepts* the **conditions**. Or the **customer** *decides* to *install* the **debt** in **installments**, using one of the **system's options**, as well as the **means of payment** of the **installments**. A **contract** is *sent* to his e-mail with all **conditions** and **information**.

Scenario # 3:

The **debt manager** *would* *like* to *know* how the **process** *is* and use the **system monitoring module** to *monitor* all **operations**. It is possible to *analyze* sets of information, such as, the **debt stock**, **the amount of full** and **installment payments**, the **number of user accesses**, and **projections**. With this information he *can* *decide* what to *do* in real time, depends on the necessity.

Scenario # 4:

In **regular periods** (**days**, **weeks**, **months**, or **quarters**), the **financial manager** *would* *like* to *have* a wider view of the whole process. To this, he *decides* to *generate* some **management reports** and *analyze* the general **dashboard**. In this **dashboard**, he *can* *identify* some important **performance indicators**, allowing him to take broader **decision**.

### 3.5.2. Determining the function of each word

The tables below contain the information collected during the identification of keywords phase. In this step, each selected word is classified and characterized. If the word “passes” for all criteria, then it will be used in the next step “Class Diagram”.

|  |  |
| --- | --- |
| Potential Class | General Classification |
| customer | external entity |
| debts | occurrence |
| personal area | organizational unit |
| installments | occurrence |
| contestation | occurrence |
| message | occurrence |
| credit card | occurrence |
| bank transfer | occurrence |
| conditions | thing |
| means of payment | generalization |
| contract | thing |
| debt manager | external entity |
| system monitoring module | organizational unit |
| operations | occurrence |
| debt stock | not object, customer’s attribute |
| projections | not object, strategic definition |
| regular periods | not object, strategic definition |
| financial manager | external entity |
| management report | not object, strategic definition |
| dashboard | organizational unit |
| performance indicator | not object, strategic definition |

In the next table, the characteristic number that applies refers to the following:

1. Retained information
2. Needed services
3. Multiple attributes
4. Common attributes
5. Common operations
6. Essential requirements

|  |  |
| --- | --- |
| Potential Class | Characteristic number that applies |
| customer | accepted: all apply |
| debts | accepted: all apply |
| personal area | rejected: 1, 2, 3 fails |
| installments | accepted: all apply |
| contestation | accepted: all apply |
| message | rejected: 1, 2, 3 fails |
| credit card | accepted: all apply |
| bank transfer | accepted: all apply |
| conditions | rejected: |
| means of payment | accepted: all apply |
| contract | rejected: 3 fails, attribute of installment |
| debt manager | accepted: all apply |
| system monitoring module | rejected: 1, 2 fails |
| operations | rejected |
| debt stock | rejected: 3 fails, attribute of customer |
| projections | rejected: 3 fails, attribute of dashboard |
| regular periods | rejected: 3 fails, attribute of dashboard |
| financial manager | accepted: all apply |
| management report | rejected: 3 fails, attribute of dashboard |
| dashboard | accepted: 1, 2, 3, 4, 5 apply |
| performance indicator | rejected: 3 fails, attribute of dashboard |

### 3.5.3. Drawing the Class Diagram

After all previous steps, a class diagram can be built to describe the structure of the system – is shown the classes, with their attributes and operations, and the relationships among the objects.

In the figure 3, it can be identified 13 classes and the hierarchy and relationship among the classes.



**Figure 3 – Final Class Diagram**

# **4. Design Modeling**

## **4.1. State Transition Diagram**

Once the requirements analysis phase was concluded, it was started the design modeling. First of all, to understand the whole functioning of the system, a state transition diagram was designed to show how the information changes its status throughout the system.

A close up of a map

Description automatically generated

**Figure 4 – State Transition Diagram**

## **4.2. Architectural Design Model**

After to know how the information “lives”, it was designed the “ecosystem” of the product, presenting the interaction with external world. The figures below depict these designs.

|  |  |
| --- | --- |
| A close up of a logo  Description automatically generated | A screenshot of a cell phone  Description automatically generated |
| **Architectural Context** | **Archetype – Banking Operation for Payments** |

**Figure 5 – Architectural Context and an example of Archetype**

## **4.3. Component Design Model**

The next model to be design was the Component Model. The main characteristic of this model is to show the separation of concerns considering all functionality. The idea behind the scenes is to try to reuse available components to minimize time, cost, and resources. Using this approach, we can develop composing loosely coupled, aggreging the benefits to use something that it was developed and tested – the gain can extend for long-term for the software itself, as well as for the company.

The figure 6 below shows the interactions among all components of the system.

A close up of a map

Description automatically generated

**Figure 6 – Component Design Model**

## **4.4. User Interface Design Model**

One of the most important aspect of any software (system) is its interaction with the users. This factor is fundamental for the success of any application and that is why it is necessary spends a lot of time planning, projecting, and drawing the interface.

The figure 7 presents some examples of the interface planned for the system and which will be improved to better meet the user’s requirements. Of course, this definition is quite iterative, needing in some cases, many beta versions until the final one.

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generatedA screenshot of a cell phone

Description automatically generated

**Figure 7 – User Interface Design Model**

## **4.4. Internal and External Interfaces Design Model**

Although the user interface is so far the most evaluate for users, there are two other interfaces that they are crucial for any system: internal and external. Internal is related to connection with other internal systems or internal databases, while external is related to connection with external systems or external databases.

As example the figures 8 and 9 shows two interfaces – internal connection with data warehouse and the other the connection with the banking system.

A screenshot of a cell phone

Description automatically generated

**Figure 8 – Example of Internal Interface Design Model**

A screenshot of a cell phone

Description automatically generated

**Figure 9 – Example of External Interface Design Model**

## **4.5. Database Design Model**

Using the class diagram with the design model, it was performed the database design model. Although the most important set of information comes for the data warehouse, it is necessary a specific database to support the whole self-regularization process.

In this model, figure 10, we find details about the stored information as well as its data type. Furthermore, it is possible to identify all relationships among the tables and its cardinality. These relations are fundamental to guarantee the integrity of the database.

A screenshot of a cell phone

Description automatically generated

**Figure 10 – Database Design Model**

# **5. Software Construction**

After concluding the steps of planning, analysis, and modeling, the next is the development, or in other words, coding. Due to universal use and knowledge already acquired by the development team, the selected language was Java.

In the sequence, it is listed a part of the code showing the main class, some entity classes, and an internal interface. The complete code is found in the GitHub (<https://github.com/deoliveira-r5192/mySEJavaProject>).

## **5.1. Main Class Coding**

Below it is presented the code of the main class where many objects are created and populated with information from the database. This database was built using the model showed in the figure 10, however, using simulated information.

*Sample Code – Main Class*

package ronaldo\_5192\_SEproject;

import java.sql.\*;

import java.util.ArrayList;

import java.time.format.DateTimeFormatter;

import java.time.LocalDateTime;

public class CSRSystem {

static ArrayList<BankTransfer> bankTransfer = new ArrayList<BankTransfer>();

static ArrayList<Contestation> contestations = new ArrayList<Contestation>();

static ArrayList<CreditCard> creditCards = new ArrayList<CreditCard>();

static ArrayList<Customer> customers = new ArrayList<Customer>();

static ArrayList<Debt> debts = new ArrayList<Debt>();

static ArrayList<Installment> installments = new ArrayList<Installment>();

public static void main(String[] args) {

try {

// getting the connection with database

Connection conn = ConnectionManager.getConnection();

Statement stmt = conn.createStatement();

ResultSet rs = null;

// CONTESTATIONS

String sqlContestation = "SELECT \* FROM tbContestation;";

rs = stmt.executeQuery(sqlContestation);

while (rs.next())

contestations.add(new Contestation(rs.getString("issue"), rs.getString("dateStart"), rs.getString("dateEnd"), rs.getString("content"), rs.getString("conclusion")));

// CREDIT CARDS

String sqlCreditCard = "SELECT \* FROM tbCreditCard AS cc INNER JOIN tbPayment AS pa ON cc.pkCreditCard = pa.fkCreditCard;";

rs = stmt.executeQuery(sqlCreditCard);

while (rs.next())

creditCards.add(new CreditCard(rs.getDouble("valuePayment"), rs.getString("typePayment"), rs.getString("datePayment"), rs.getBoolean("recurrent"), rs.getString("number"),

rs.getString("name"), rs.getString("expireDate"), rs.getString("cvc"), rs.getString("brand")));

// BANK TRANSFER

String sqlBankTransfer = "SELECT \* FROM tbBankTransfer AS bt INNER JOIN tbPayment AS pa ON bt.pkBankTransfer = pa.fkBankTransfer";

rs = stmt.executeQuery(sqlBankTransfer);

while (rs.next())

bankTransfer.add(new BankTransfer(rs.getDouble("valuePayment"), rs.getString("typePayment"), rs.getString("datePayment"), rs.getBoolean("recurrent"), rs.getString("name"),

rs.getString("branch"), rs.getString("bank"), rs.getString("accountNumber"), rs.getString("branchNumber")));

// CUSTOMERS

String sqlCustomer = "SELECT \* FROM tbCustomer;";

rs = stmt.executeQuery(sqlCustomer);

while (rs.next()) {

int pkCustomer = rs.getInt("pkCustomer");

Customer customer = new Customer(rs.getString("firstName"), rs.getString("lastName"), rs.getString("birthDate"), rs.getString("firstPurchaseDate"),

rs.getString("lastPurchaseDate"), rs.getInt("numPurchases"), rs.getString("city"), rs.getString("state"));

// including DEBT into CUSTOMER

String sqlCustomerDebt = "SELECT \* FROM tbDebt WHERE fkCustomer = '" + pkCustomer + "';";

Statement stmtDebt = conn.createStatement();

ResultSet rsDebt = stmtDebt.executeQuery(sqlCustomerDebt);

while (rsDebt.next())

{

int pkDebt = rsDebt.getInt("pkDebt");

Debt debtTemp = new Debt(rsDebt.getString("typeDebt"), rsDebt.getString("dateDebt"), rsDebt.getFloat("valueDebt"));

// including INSTALLMENT into DEBT

String sqlDebtInstallment = "SELECT \* FROM tbInstallments WHERE fkDebt = '" + pkDebt + "';";

Statement stmtInstallment = conn.createStatement();

ResultSet rsInstallment = stmtInstallment.executeQuery(sqlDebtInstallment);

while (rsInstallment.next())

debtTemp.addInstallment(new Installment(rsInstallment.getString("dateInstallment"), rsInstallment.getFloat("valueInstallment"),

rsInstallment.getString("firstPayment"), rsInstallment.getString("frequency")));

// including CONSTESTATION into DEBT

String sqlDebtContestation = "SELECT \* FROM tbContestation WHERE fkDebt = '" + pkDebt + "';";

Statement stmtContestation = conn.createStatement();

ResultSet rsContestation = stmtContestation.executeQuery(sqlDebtContestation);

while (rsContestation.next())

debtTemp.addContestation(new Contestation(rsContestation.getString("issue"), rsContestation.getString("dateStart"),

rsContestation.getString("dateEnd"), rsContestation.getString("content"), rsContestation.getString("conclusion")));

// including PAYMENT - CREDIT CARD into DEBT

String sqlDebtCreditCard = "SELECT \* "

+ "FROM tbCreditCard AS cd "

+ "INNER JOIN tbPayment AS pa ON cd.pkCreditCard = pa.fkCreditCard "

+ "INNER JOIN tbDebt AS de ON pa.pkPayment = de.fkPayment "

+ "WHERE pkDebt = '" + pkDebt + "';";

Statement stmtDebtCreditCard = conn.createStatement();

ResultSet rsDebtCreditCard = stmtDebtCreditCard.executeQuery(sqlDebtCreditCard);

while (rsDebtCreditCard.next())

debtTemp.setPaymentMethod(new CreditCard(rsDebtCreditCard.getDouble("valuePayment"), rsDebtCreditCard.getString("typePayment"),

rsDebtCreditCard.getString("datePayment"), rsDebtCreditCard.getBoolean("recurrent"), rsDebtCreditCard.getString("number"),

rsDebtCreditCard.getString("name"), rsDebtCreditCard.getString("expireDate"), rsDebtCreditCard.getString("cvc"),

rsDebtCreditCard.getString("brand")));

// including PAYMENT - BANK TRANSFER into DEBT

String sqlDebtBankTransfer = "SELECT \* "

+ "FROM tbBankTransfer AS bf "

+ "INNER JOIN tbPayment AS pa ON bf.pkBankTransfer = pa.fkBankTransfer "

+ "INNER JOIN tbDebt AS de ON pa.pkPayment = de.fkPayment "

+ "WHERE pkDebt = '" + pkDebt + "';";

Statement stmtDebtBankTransfer = conn.createStatement();

ResultSet rsDebtBankTransfer = stmtDebtBankTransfer.executeQuery(sqlDebtBankTransfer);

while (rsDebtBankTransfer.next())

debtTemp.setPaymentMethod(new BankTransfer(rsDebtBankTransfer.getDouble("valuePayment"), rsDebtBankTransfer.getString("typePayment"),

rsDebtBankTransfer.getString("datePayment"), rsDebtBankTransfer.getBoolean("recurrent"), rsDebtBankTransfer.getString("name"),

rsDebtBankTransfer.getString("branch"), rsDebtBankTransfer.getString("bank"), rsDebtBankTransfer.getString("accountNumber"),

rsDebtBankTransfer.getString("branchNumber")));

// Add Debt into Customer

customer.addDebt(debtTemp);

}

customers.add(customer);

}

// DEBTS

String sqlDebt = "SELECT \* FROM tbDebt;";

rs = stmt.executeQuery(sqlDebt);

while (rs.next()) {

int pkDebt = rs.getInt("pkDebt");

Debt debt = new Debt(rs.getString("typeDebt"), rs.getString("dateDebt"), rs.getFloat("valueDebt"));

// including INSTALLMENT into DEBT

String sqlDebtInstallment = "SELECT \* FROM tbInstallments WHERE fkDebt = '" + pkDebt + "';";

Statement stmtInstallment = conn.createStatement();

ResultSet rsInstallment = stmtInstallment.executeQuery(sqlDebtInstallment);

while (rsInstallment.next())

debt.addInstallment(new Installment(rsInstallment.getString("dateInstallment"), rsInstallment.getFloat("valueInstallment"),

rsInstallment.getString("firstPayment"), rsInstallment.getString("frequency")));

// including CONSTESTATION into DEBT

String sqlDebtContestation = "SELECT \* FROM tbContestation WHERE fkDebt = '" + pkDebt + "';";

Statement stmtContestation = conn.createStatement();

ResultSet rsContestation = stmtContestation.executeQuery(sqlDebtContestation);

while (rsContestation.next())

debt.addContestation(new Contestation(rsContestation.getString("issue"), rsContestation.getString("dateStart"),

rsContestation.getString("dateEnd"), rsContestation.getString("content"), rsContestation.getString("conclusion")));

// including PAYMENT - CREDIT CARD into DEBT

String sqlDebtCreditCard = "SELECT \* "

+ "FROM tbCreditCard AS cd "

+ "INNER JOIN tbPayment AS pa ON cd.pkCreditCard = pa.fkCreditCard "

+ "INNER JOIN tbDebt AS de ON pa.pkPayment = de.fkPayment "

+ "WHERE pkDebt = '" + pkDebt + "';";

Statement stmtDebtCreditCard = conn.createStatement();

ResultSet rsDebtCreditCard = stmtDebtCreditCard.executeQuery(sqlDebtCreditCard);

while (rsDebtCreditCard.next())

debt.setPaymentMethod(new CreditCard(rsDebtCreditCard.getDouble("valuePayment"), rsDebtCreditCard.getString("typePayment"),

rsDebtCreditCard.getString("datePayment"), rsDebtCreditCard.getBoolean("recurrent"), rsDebtCreditCard.getString("number"),

rsDebtCreditCard.getString("name"), rsDebtCreditCard.getString("expireDate"), rsDebtCreditCard.getString("cvc"),

rsDebtCreditCard.getString("brand")));

// including PAYMENT - BANK TRANSFER into DEBT

String sqlDebtBankTransfer = "SELECT \* "

+ "FROM tbBankTransfer AS bf "

+ "INNER JOIN tbPayment AS pa ON bf.pkBankTransfer = pa.fkBankTransfer "

+ "INNER JOIN tbDebt AS de ON pa.pkPayment = de.fkPayment "

+ "WHERE pkDebt = '" + pkDebt + "';";

Statement stmtDebtBankTransfer = conn.createStatement();

ResultSet rsDebtBankTransfer = stmtDebtBankTransfer.executeQuery(sqlDebtBankTransfer);

while (rsDebtBankTransfer.next())

debt.setPaymentMethod(new BankTransfer(rsDebtBankTransfer.getDouble("valuePayment"), rsDebtBankTransfer.getString("typePayment"),

rsDebtBankTransfer.getString("datePayment"), rsDebtBankTransfer.getBoolean("recurrent"), rsDebtBankTransfer.getString("name"),

rsDebtBankTransfer.getString("branch"), rsDebtBankTransfer.getString("bank"), rsDebtBankTransfer.getString("accountNumber"),

rsDebtBankTransfer.getString("branchNumber")));

debts.add(debt);

}

// INSTALLMENTS

String sqlInstallment = "SELECT \* FROM tbInstallments;";

rs = stmt.executeQuery(sqlInstallment);

while (rs.next()) {

int pkInstallment = rs.getInt("pkInstallments");

Installment installment = new Installment(rs.getString("dateInstallment"), rs.getFloat("valueInstallment"), rs.getString("firstPayment"), rs.getString("frequency"));

// including CRITERIA into INSTALLMENT

String sqlInstallmentCriteria = "SELECT \* "

+ "FROM tbInstallments AS it "

+ "INNER JOIN tbInstallments\_tbCriteria AS ic ON it.pkInstallments = ic.fkInstallments "

+ "INNER JOIN tbCriteria AS cr ON ic.fkCriteria = cr.pkCriteria "

+ "WHERE pkInstallments = '" + pkInstallment + "';";

Statement stmtInstallmentCriteria = conn.createStatement();

ResultSet rsInstallmentCriteria = stmtInstallmentCriteria.executeQuery(sqlInstallmentCriteria);

while (rsInstallmentCriteria.next())

installment.addCriteria(new Criteria (rsInstallmentCriteria.getString("typeProgram"), rsInstallmentCriteria.getString("startProgram"),

rsInstallmentCriteria.getString("endProgram"), rsInstallmentCriteria.getString("frequency")));

installments.add(installment);

}

// DASHBOARD

DateTimeFormatter dtf = DateTimeFormatter.ofPattern("yyyy/MM/dd HH:mm:ss");

LocalDateTime now = LocalDateTime.now();

Dashboard dashboard = new Dashboard("SE Project", "1 hour", dtf.format(now));

dashboard.setCustomers(customers);

dashboard.setDebts(debts);

dashboard.setInstallments(installments);

dashboard.generateDashboard();

}

catch (SQLException e) {

System.out.println(e.getMessage());

}

}

}

## **5.2. Some Entity Class Coding**

Below it is presented the code of implementation of classes “Person”, “Dashboard”, and “Debt”. “Person” class is defined to work with information about each system user, either customer or manager (each one of this character has its correspondent class). “Dashboard” class has the main function to create and show information from the database. Finally, “Debt” class is used to organize and store information about each customer’s debts.

*Person Class*

package ronaldo\_5192\_SEproject;

import java.util.ArrayList;

public class Customer extends Person {

private String firstPurchase;

private String lastPurchase;

private int numPurchases;

private String city;

private String state;

private ArrayList<Debt> debts;

public Customer(String firstName, String lastName, String birthDate, String firstPurchase, String lastPurchase, int numPurchases, String city, String state) {

super(firstName, lastName, birthDate);

this.firstPurchase = firstPurchase;

this.lastPurchase = lastPurchase;

this.numPurchases = numPurchases;

this.city = city;

this.state = state;

this.debts = new ArrayList<Debt>();

}

public String getFirstPurchase() {

return firstPurchase;

}

public void setFirstPurchase(String firstPurchase) {

this.firstPurchase = firstPurchase;

}

public String getLastPurchase() {

return lastPurchase;

}

public void setLastPurchase(String lastPurchase) {

this.lastPurchase = lastPurchase;

}

public int getNumPurchases() {

return numPurchases;

}

public void setNumPurchases(int numPurchases) {

this.numPurchases = numPurchases;

}

public String getCity() {

return city;

}

public void setCity(String city) {

this.city = city;

}

public String getState() {

return state;

}

public void setState(String state) {

this.state = state;

}

public ArrayList<Debt> getDebts() {

return debts;

}

public void setDebts(ArrayList<Debt> debts) {

this.debts = debts;

}

public void addDebt(Debt debt) {

this.debts.add(debt);

}

public void printCustomer() {

System.out.println("Customer: " + super.getFirstName() + " " + super.getLastName() + " | BirthDate: " + super.getBirthDate() + " | City: " + this.city);

System.out.println("Number of purchases: " + this.numPurchases + " | First Purchase: " + this.firstPurchase + " | Last Purchase: " + this.lastPurchase);

if (debts.size() == 0)

System.out.println("Customer has NO debts");

else {

System.out.println("#Debts: " + debts.size());

for (int i=0; i<debts.size(); i++)

debts.get(i).printDebt();

}

}

}

*Dashboard Class*

package ronaldo\_5192\_SEproject;

import java.util.ArrayList;

public class Dashboard {

private String owner;

private String updateInterval;

private String lastUpdate;

private ArrayList<Customer> customers;

private ArrayList<Debt> debts;

private ArrayList<Installment> installments;

public Dashboard(String owner, String updateInterval, String lastUpdate) {

this.owner = owner;

this.updateInterval = updateInterval;

this.lastUpdate = lastUpdate;

this.customers = new ArrayList<Customer>();

this.debts = new ArrayList<Debt>();

this.installments = new ArrayList<Installment>();

}

public String getOwner() {

return owner;

}

public void setOwner(String owner) {

this.owner = owner;

}

public String getUpdateInterval() {

return updateInterval;

}

public void setUpdateInterval(String updateInterval) {

this.updateInterval = updateInterval;

}

public String getLastUpdate() {

return lastUpdate;

}

public void setLastUpdate(String lastUpdate) {

this.lastUpdate = lastUpdate;

}

public ArrayList<Customer> getCustomers() {

return customers;

}

public void setCustomers(ArrayList<Customer> customers) {

this.customers = customers;

}

public ArrayList<Debt> getDebts() {

return debts;

}

public void setDebts(ArrayList<Debt> debts) {

this.debts = debts;

}

public ArrayList<Installment> getInstallments() {

return installments;

}

public void setInstallments(ArrayList<Installment> installments) {

this.installments = installments;

}

public void generateDashboard() {

System.out.println("------------------------------------------------------------------------------------------------------");

System.out.println("--------------------------------------------- DASHBOARD ---------------------------------------------");

System.out.println("------------------------------------------------------------------------------------------------------");

System.out.println("Owner: " + this.owner + "\nInterval Update: " + this.updateInterval + "\nLast Update: " + this.lastUpdate + "\n");

System.out.println("------------------------------------------------------------------------------------------------------");

if (customers.size() == 0)

System.out.println("No Customer in the Dashboard");

else {

System.out.println("Total Customers: " + customers.size() + "\n");

for (int i=0; i<customers.size(); i++) {

System.out.println("\*\* Customer #" + (i+1));

customers.get(i).printCustomer();

System.out.println();

}

}

System.out.println("------------------------------------------------------------------------------------------------------");

if (debts.size() == 0)

System.out.println("No Debts in the Dashboard");

else {

System.out.println("Total Debts: " + debts.size() + "\n");

for (int i=0; i<debts.size(); i++){

System.out.println("\*\* Debt #" + (i+1));

debts.get(i).printDebt();

System.out.println();

}

}

System.out.println("------------------------------------------------------------------------------------------------------");

if (installments.size() == 0)

System.out.println("No Installments in the Dashboard");

else {

System.out.println("Total Installments: " + installments.size() + "\n");

for (int i=0; i<installments.size(); i++){

System.out.println("\*\* Installment #" + (i+1));

installments.get(i).printInstallment();

System.out.println();

}

}

System.out.println("------------------------------------------------------------------------------------------------------");

}

public void generateReport() {

System.out.println("Report successfully created - \*\* method not implemented \*\*");

}

public void generateProjection() {

System.out.println("Projection successfully created - \*\* method not implemented \*\*");

}

}

*Debt Class*

package ronaldo\_5192\_SEproject;

import java.util.ArrayList;

public class Debt {

private String typeDebt;

private String dateDebt;

private float valueDebt;

private PaymentMethod paymentMethod;

private ArrayList<Contestation> contestations;

private ArrayList<Installment> installments;

public Debt(String typeDebt, String dateDebt, float valueDebt) {

this.typeDebt = typeDebt;

this.dateDebt = dateDebt;

this.valueDebt = valueDebt;

this.paymentMethod = null;

this.contestations = new ArrayList<Contestation>();

this.installments = new ArrayList<Installment>();

}

public String getTypeDebt() {

return typeDebt;

}

public void setTypeDebt(String typeDebt) {

this.typeDebt = typeDebt;

}

public String getDateDebt() {

return dateDebt;

}

public void setDateDebt(String dateDebt) {

this.dateDebt = dateDebt;

}

public float getValueDebt() {

return valueDebt;

}

public void setValueDebt(float valueDebt) {

this.valueDebt = valueDebt;

}

public PaymentMethod getPaymentMethod() {

return paymentMethod;

}

public void setPaymentMethod(PaymentMethod paymentMethod) {

this.paymentMethod = paymentMethod;

}

public ArrayList<Contestation> getContestations() {

return contestations;

}

public void setContestations(ArrayList<Contestation> contestations) {

this.contestations = contestations;

}

public ArrayList<Installment> getInstallments() {

return installments;

}

public void setInstallments(ArrayList<Installment> installments) {

this.installments = installments;

}

public void addInstallment(Installment installment) {

this.installments.add(installment);

}

public void addContestation(Contestation contestation) {

this.contestations.add(contestation);

}

public void printDebt() {

System.out.println("Debt Date: " + this.dateDebt + " | Debt Value: " + this.valueDebt + " | Debt Type " + this.typeDebt);

if (paymentMethod == null)

System.out.println("Debt has NO payment");

else

paymentMethod.printPayment();

if (installments.size() == 0)

System.out.println("Debt has NO installment");

else {

System.out.println("#Installments: " + installments.size());

for (int i=0; i<installments.size(); i++)

installments.get(i).printInstallment();

}

if (contestations.size() == 0)

System.out.println("Debt has NO contestation");

else {

System.out.println("#Contestation: " + contestations.size());

for (int i=0; i<contestations.size(); i++)

contestations.get(i).printContestation();

}

}

}

## **5.3. Internal Interface Class Coding**

The implementation below shows how the code to connect to database is. The database was built using SQLite and the project folder.

package ronaldo\_5192\_SEproject;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.SQLException;

public class ConnectionManager {

private static String url = "jdbc:sqlite:D:/MyWorkSpace/ronaldo\_5192\_SEproject/SEproject.db";

private static Connection conn;

public static Connection getConnection() {

try {

// create a connection to the database

conn = DriverManager.getConnection(url);

}

catch (SQLException e) {

System.out.println(e.getMessage());

}

return conn;

}

}

# **6. Conclusion**

The purpose of this project was to create a system to help a company to recover the credits from defaulting customers, using the techniques of Software Engineering. The final product was a sample of the system including all classes and structures. In addition, we were able to design a set of diagrams and models to support the whole process.

Regarding to specific objectives of the company, now it is able to create self-regularization programs, manager each program, include and exclude customers, set rules, and track the results. It is expected that the company can recover the investment in the software sooner and, besides that, recover the credits from customer with a lower effort and costs. Moreover, the company’s directors believe that company’s image will be valorized because it is worried about the financial situation of its customers.

Considering the challenges faced in the project, I can list 2 (two) situations:

1. Determining the use-cases and creating the architecture and component diagrams, because I needed to create every situation and imagine a IT department with experience in other projects. The solution was very simple: use imagination and experience in my previous jobs, mainly when I worked as a project manager. I used different situations and tried to produce only one, so that it was realistic and feasible for the project.
2. Designing each diagram or model, because I could not find a single software that would execute all the necessary drawings. The solution was the most natural and obvious, that is, using different software and learning each one "on the fly", because I did not have time to study each one beforehand and needed to create the diagram as soon as possible due to the deadline.

Finally, it is important to highlight some aspects that could be improved in the system in the future. The main aspect is to develop a mobile version because this new feature would allow the customer to access and self-regularize much more quickly, including payment execution. Other improvements could make this tool more general and add it to the business flow, that is, allow the customer to install in installments at the time of purchase. Obviously, this type of decision must be made by senior executives. The last aspect would be to try to incorporate the database into the data warehouse to allow the rest of the company to use the information generated by the self-regulatory system, and not just the departments involved in the process.