Assignment 1

Analysis and Design Document

Student: Zahan Raul Dumitru

**Group:30433**

Table of Contents

1. Requirements Analysis 3

1.1 Assignment Specification 3

1.2 Functional Requirements 3

1.3 Non-functional Requirements 3

2. Use-Case Model 3

3. System Architectural Design 3

4. UML Sequence Diagrams 3

5. Class Design 3

6. Data Model 3

7. System Testing 3

8. Bibliography 3

1. Requirements Analysis

# Assignment Specification

Use Swing/C# API to design and implement an application for the front desk employees of a bank. The application should have two types of users (a regular user represented by the front desk employee and an administrator user) which have to provide a username and a password in order to use the application.

# Functional Requirements

The regular user can perform the following operations:

* Add/update/view client information (name, identity card number, personal numerical code, address, etc.).
* Create/update/delete/view client account (account information: identification number, type, amount of money, date of creation).
* Transfer money between accounts.
* Process utilities bills.

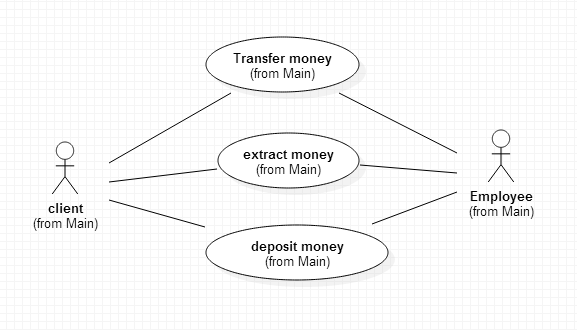
The administrator user can perform the following operations:

* CRUD on employees’ information.
* Generate reports for a particular period containing the activities performed by an employee.

# Non-functional Requirements

You need to have constraints when adding to db, you can’t have users with the same name.

2. Use-Case Model



*Use-Case description format:*

*Use case: The Employees are serving the bank clients, creating accounts and others; clients extract and deposit money*

*Primary actor: Client and Employee*

*Main success scenario: operations are done with success*

*Extensions: Users can’t pay their taxes*

3. System Architectural Design

**3.1 Architectural Pattern Description**

# Layered Architectural Style

Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other. Functionality within each layer is related by a common role or responsibility. Communication between layers is explicit and loosely coupled. Layering your application appropriately helps to support a strong separation of concerns that, in turn, supports flexibility and maintainability.

The layered architectural style has been described as an inverted pyramid of reuse where each layer aggregates the responsibilities and abstractions of the layer directly beneath it. With strict layering, components in one layer can interact only with components in the same layer or with components from the layer directly below it. More relaxed layering allows components in a layer to interact with components in the same layer or with components in any lower layer.

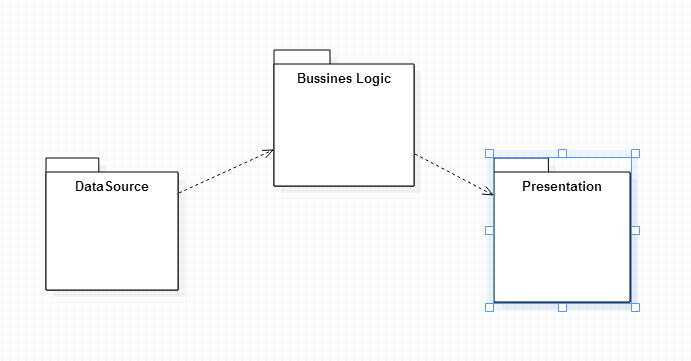
The layers of an application may reside on the same physical computer (the same tier) or may be distributed over separate computers (n-tier), and the components in each layer communicate with components in other layers through well-defined interfaces. For example, a typical Web application design consists of a presentation layer (functionality related to the UI), a business layer (business rules processing), and a data layer (functionality related to data access, often almost entirely implemented using high-level data access frameworks).

Common principles for designs that use the layered architectural style include:

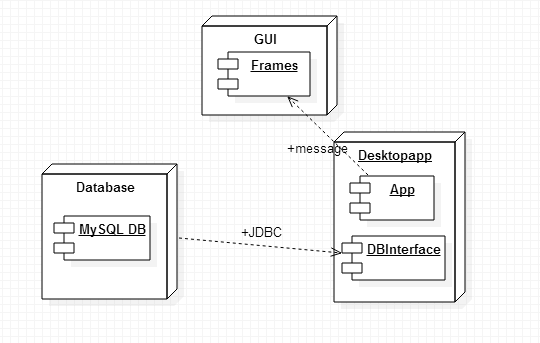
* **Abstraction**. Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.
* **Encapsulation**. No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.
* **Clearly defined functional layers**. The separation between functionality in each layer is clear. Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.
* **High cohesion**. Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.
* **Reusable**. Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.
* **Loose coupling**. Communication between layers is based on abstraction and events to provide loose coupling between layers.

**3.2 Diagrams**

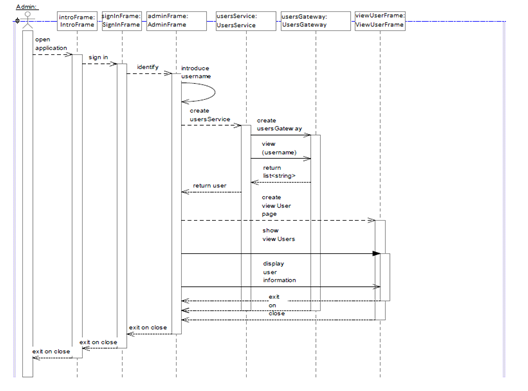
**Package diagram**

**

**Deployment and Component diagram**

****

4. UML Sequence Diagrams



5. Class Design

**5.1 Design Patterns Description**

**The active record pattern**

In [software engineering](http://en.wikipedia.org/wiki/Software_engineering), the **active record pattern** is an [architectural pattern](http://en.wikipedia.org/wiki/Architectural_pattern_(computer_science)) found in software that stores in-memory object data in [relational databases](http://en.wikipedia.org/wiki/Relational_database). It was named by [Martin Fowler](http://en.wikipedia.org/wiki/Martin_Fowler) in his 2003 book *Patterns of Enterprise Application Architecture*. The interface of an object conforming to this pattern would include functions such as Insert, Update, and Delete, plus properties that correspond more or less directly to the columns in the underlying database table.

**The active record pattern** is an approach to accessing data in a [database](http://en.wikipedia.org/wiki/Database). A [database table](http://en.wikipedia.org/wiki/Database_table) or [view](http://en.wikipedia.org/wiki/View_(database)) is wrapped into a [class](http://en.wikipedia.org/wiki/Class_(computer_science)). Thus, an [object](http://en.wikipedia.org/wiki/Object_(computer_science)) instance is tied to a single row in the table. After creation of an object, a new row is added to the table upon save. Any object loaded gets its information from the database. When an object is updated the corresponding row in the table is also updated. The wrapper class implements [accessor](http://en.wikipedia.org/wiki/Accessor" \o "Accessor) [methods](http://en.wikipedia.org/wiki/Method_(computer_programming)) or properties for each column in the table or view.

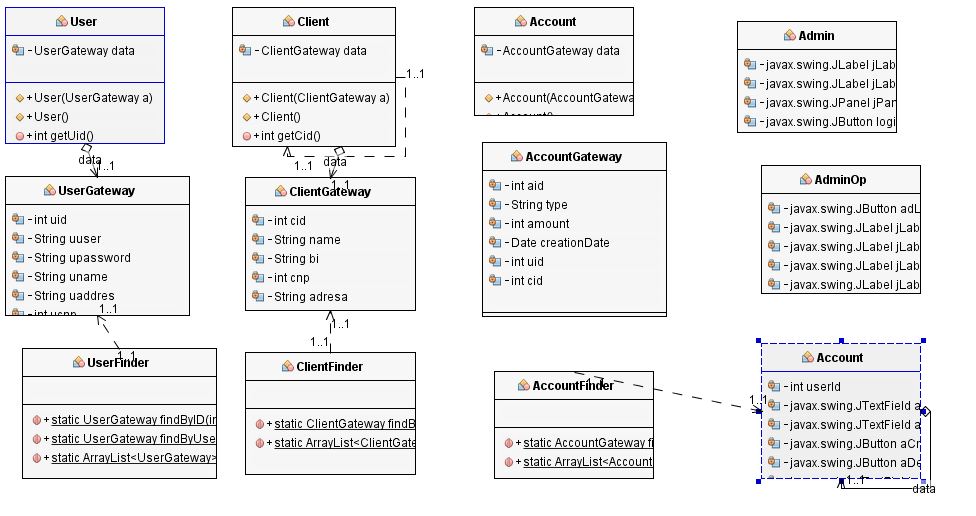
**Row Data Gateway pattern**

An object that acts as a Gateway to a single record in a data source. There is one instance per row.

Embedding database access code in in-memory objects can leave you with a few disadvantages. For a start, if your in-memory objects have business logic of their own, adding the database manipulation code increases complexity. Testing is awkward too since, if your in-memory objects are tied to a database, tests are slower to run because of all the database access. You may have to access multiple databases with all those annoying little variations on their SQL.

A **Row Data Gateway** gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language. All details of data source access are hidden behind this interface.

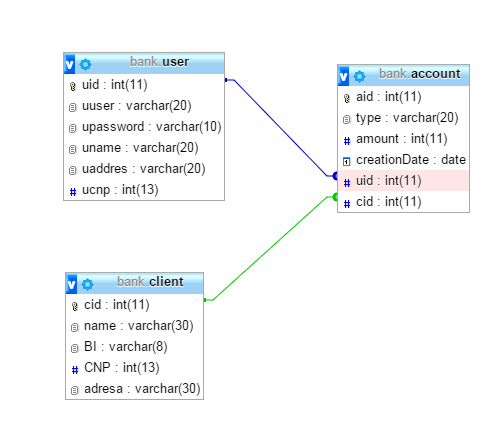
**5.2 UML Class Diagram**

**

The active record pattern and Row Data Gateway pattern are represented in the Gateway and Finder Classes

Are Structured in 3 Layers : Business Logic, Presentation and Data Source

6. Data Model

**

7. System Testing

In [computer programming](http://en.wikipedia.org/wiki/Computer_programming), unit testing is a [software testing](http://en.wikipedia.org/wiki/Software_testing) method by which individual units of [source code](http://en.wikipedia.org/wiki/Source_code), sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine whether they are fit for use.[[1]](http://en.wikipedia.org/wiki/Unit_testing#cite_note-kolawa-1) Intuitively, one can view a unit as the smallest testable part of an application. In [procedural programming](http://en.wikipedia.org/wiki/Procedural_programming), a unit could be an entire module, but it is more commonly an individual function or procedure. In [object-oriented programming](http://en.wikipedia.org/wiki/Object-oriented_programming), a unit is often an entire interface, such as a class, but could be an individual method. Unit tests are short code fragments created by programmers or occasionally by [white box testers](http://en.wikipedia.org/wiki/White-box_testing) during the development process. It forms the basis for component testing.

Ideally, each [test case](http://en.wikipedia.org/wiki/Test_case) is independent from the others. Substitutes such as [method stubs](http://en.wikipedia.org/wiki/Method_stub), [mock objects](http://en.wikipedia.org/wiki/Mock_object),  [fakes](http://en.wikipedia.org/wiki/Mock_object#Mocks.2C_fakes_and_stubs), and [test harnesses](http://en.wikipedia.org/wiki/Test_harness) can be used to assist testing a module in isolation. Unit tests are typically written and run by [software developers](http://en.wikipedia.org/wiki/Software_developer) to ensure that code meets its design and behaves as intended.

8. Bibliography

<http://en.wikipedia.org/wiki/Unit_testing>

<http://docs.oracle.com/javase/7/docs/api/java/sql/PreparedStatement.html>

<https://docs.jboss.org/hibernate/orm/4.0/devguide/en-US/html/chap-hql.html>

Patterns\_of\_Enterprise\_Application\_Architecture.pdf