**StarBank Application**

**Modelo de Análisis y Diseño del Sistema**

**Equipo de Trabajo**

**Cliente**

StarBank

**Responsables**

1017251689-David Andrés Torres betancour

**Historia de revisiones**

| Fecha | Versión | Descripción | Autor |
|---|---|---|---|
| 21/4/20 | 1 | Punto 6 | David Torres |
| 26/4/20 | 1.1 | Punto 1,2,3,4 | David Torres |
| 25/5/20 | 1.2 | Punto 5,7,8 | David Torres |

**StarBank**

**Modelo de Análisis y Diseño**

## 1. Introducción

En la entidad bancaria StarBank se encargan de la administración de cuentas de depósito, donde se realizan operaciones financieras como apertura, activación y desactivación de cuentas, suscripción de clientes, consignaciones, retiros, entre otras. Para lograr mayor desempeño en su servicio, requieren de un sistema electrónico, que supla dichas actividades.

Por lograr lo anterior,se tiene como objetivo principal realizar una aplicación de escritorio que respete y vaya acorde a las normas establecidas por el modelo de negocio de la entidad. Además que cuente con una interfaz gráfica, clara y amigable para el usuario, donde al mismo tiempo se apliquen buenas prácticas de programación y fundamentos de la programación orientada a objetos.

### 1.1 Propósito del documento

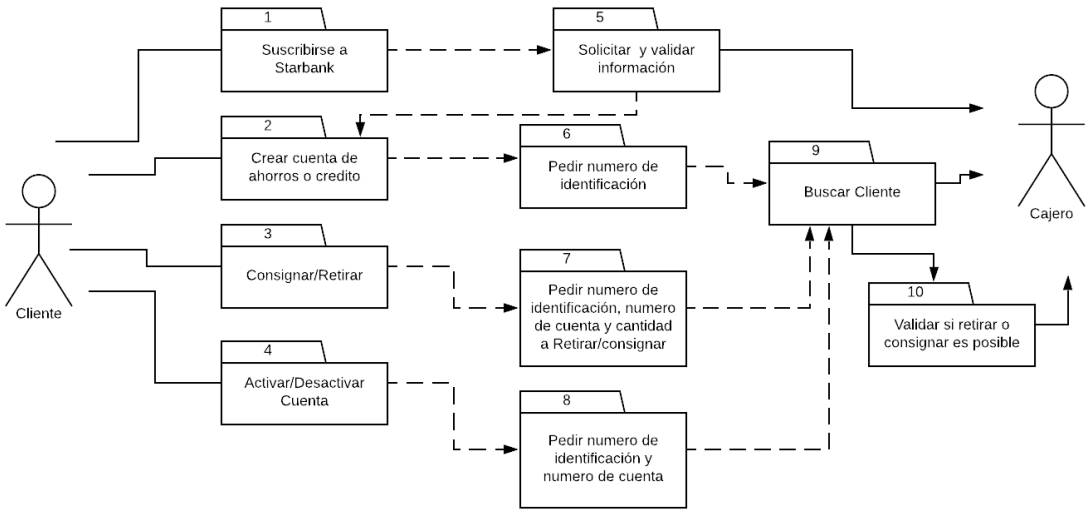
La intención de este documento es mostrar de una manera clara y estructurada, el análisis realizado a la solicitud del cliente, presentado a detalle las especificaciones funcionales, las líneas de vida que cada uno de los procesos y objetos que coexisten simultáneamente. De esta manera llegar a la etapa de desarrollo con planes concretos respecto al cómo se va a proceder en la construcción de la aplicación.

Este escrito está dirigido tanto a los clientes, como a los mismos desarrolladores, para que pueda circular una comunicación bidireccional, sin retrasos y el proyecto tengo mayor fluidez, logrando de esta manera los objetivos anteriormente planteados.

## ANÁLISIS DEL SISTEMA

### 2. Especificación funcional

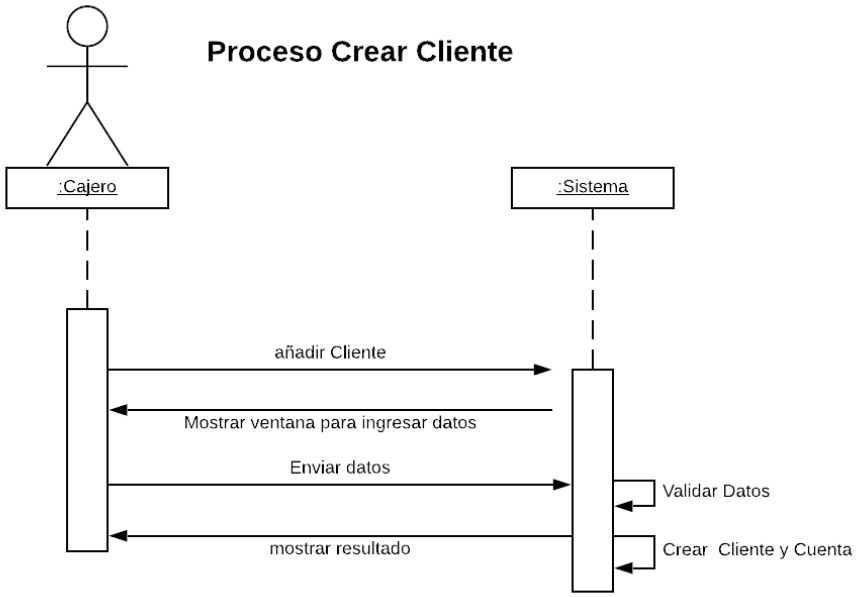
2.1 Funcionalidad.

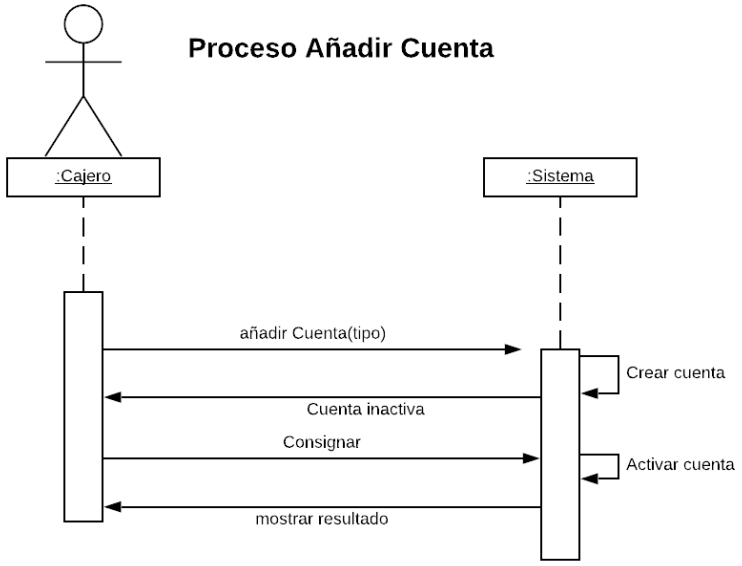


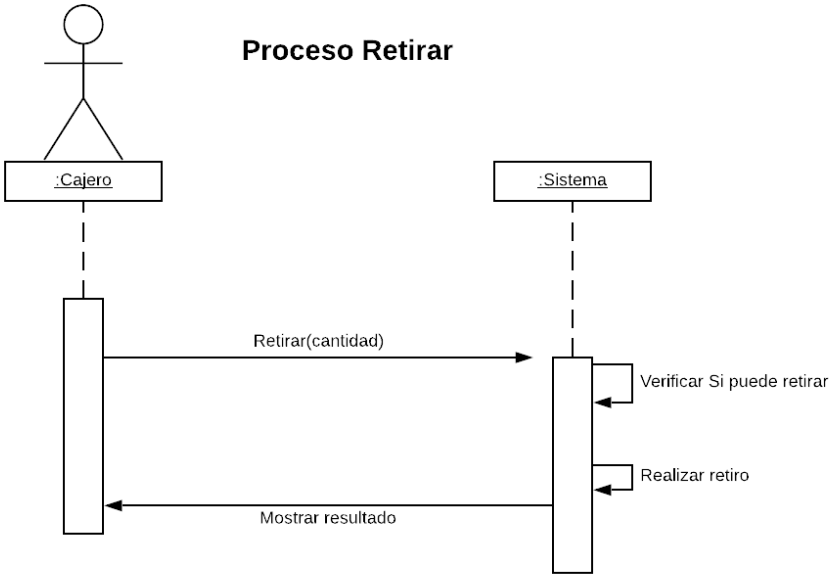

1. Es el momento en que el cliente se dirige al starbank para suscribirse al banco.
2. Si el cliente apenas se va a suscribir al banco se le solicita crear una cuenta de ahorros o créditos, pero si el cliente ya existe procede a crear la cuenta sin problemas.
3. El cliente llega a consignar el monto que el requiera y retirar el dinero que desee siempre y cuando sea menor o igual al monto total de la cuenta
4. Cuando el cliente lo requiera puede activar o desactivar una cuenta.
5. Antes de proceder a agregar al cliente se deben solicitar sus datos y validación de la información, en caso de que sea un cliente nuevo se procede a crear la cuenta de ahorros o crédito
6. Se solicita la identificación del cliente lo cual es solo unico que se necesita para crear una cuenta.
7. Para retirar o consignar se requiere el número de identificación, número de cuenta y cantidad de de dinero a retirar o consignar
8. Para activar o desactivar una cuenta se requiere el número de identificación y número de cuenta
9. Con la información ya validada, se procede a buscarla

**10.** En respuesta de la información encontrada se responde si es posible o no realizar el retiro o consignación.

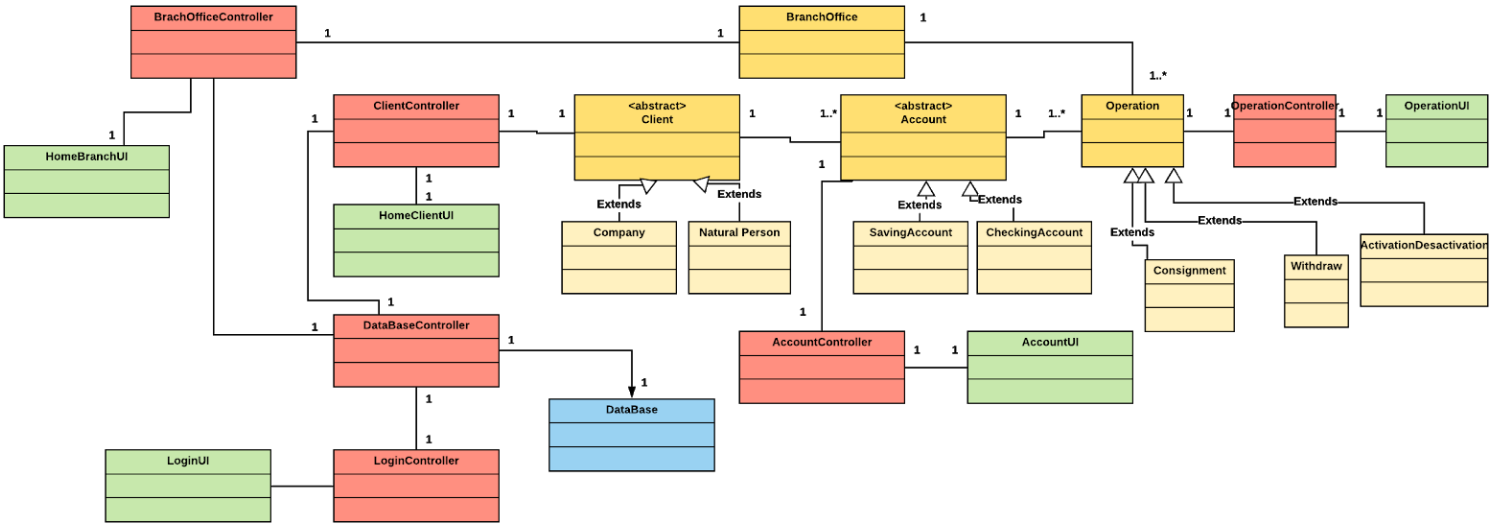
2.2 Diagrama de secuencias del sistema DSS.






## 3. Modelo conceptual refinado (modelo del dominio)



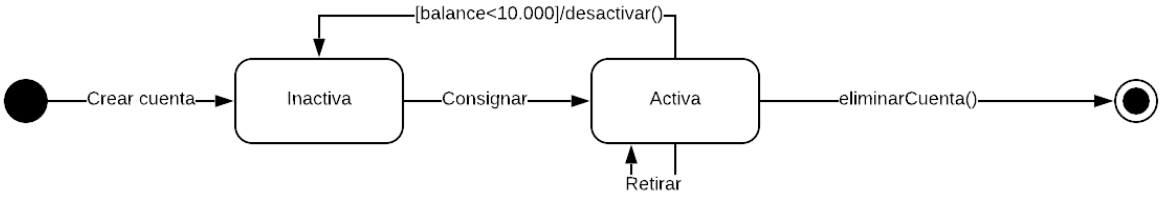

- **BranchOffice :** Esta clase contiene la información de la sucursal y el listado de operaciones
- **Client :**Esta clase es abstracta y tiene las principales parametros del cliente

- **Account:** Esta clase es abstracta y tiene los principales parámetros de la cuenta.
- **Operation:** Esta clase contiene la información de una operación
- **Company:** Esta clase es un tipo de cliente, por lo que hereda de la clase Client.
- **Natural Person:** Esta clase es un tipo de cliente, por lo que hereda de la clase Client.
- **Saving Account:** Esta clase es un tipo de cuenta , por lo que hereda de la clase Account
- **CheckingAccount :** Esta clase es un tipo de cuenta, por lo que hereda de la clase Account..
- **Consignment :** Esta clase es un tipo de operación , por lo que hereda de la clase Operación..
- **Withdraw :** Esta clase es un tipo de operación , por lo que hereda de la clase Operación.
- **ActivationDesactivation :** Esta clase es un tipo de operación , por lo que hereda de la clase Operación.
- **BranchOfficeController :** Esta clase se encarga para comunicar la información que tiene la sucursal con las demás clases, además de encargarse de crear la sucursal cuando se logea en la App.
- **ClientController :** Esta clase se encarga de comunicar la información del cliente con las demás clases, además de crear y eliminar el cliente:
- **DataBaseController :** Esta clase es la que inicializa la base de datos y se comunica con ella en caso que otras clases necesiten algún tipo de persistencia.
- **LoginController:** Se encarga de comunicarse con la base de datos para para responderle a la Interfaz de usuario si la información es válida
- **AccountController :** Esta clase se encarga de comunicar la información que tiene la cuenta con las demás clases, Además de crear y eliminar cuentas.
- **OperationController :** Esta clase gestiona las operaciones que se realicen en la sucursal.
- **HomeBranchUI :** Esta clase muestra la información de la sucursal y activa las acciones que sean requeridas.
- **LoginUI :** Esta clase se encarga de recibir los datos que van a ser verificados.
- **HomeClientUI :** Esta clase se encarga de mostrar la información del cliente y gestionar demás acciones que sean disparadas desde la interfaz
- **AccountUI :** Esta clase se encarga de mostrar la información de la cuenta y gestionar demás acciones que sean disparadas desde la interfaz,
- **OperationUI :** Muestra el total de operaciones realizadas
- **DataBase :** Es donde se configura el gestor de base de datos y están los métodos para realizar el CRUD.

## 4. Clases Reactivas

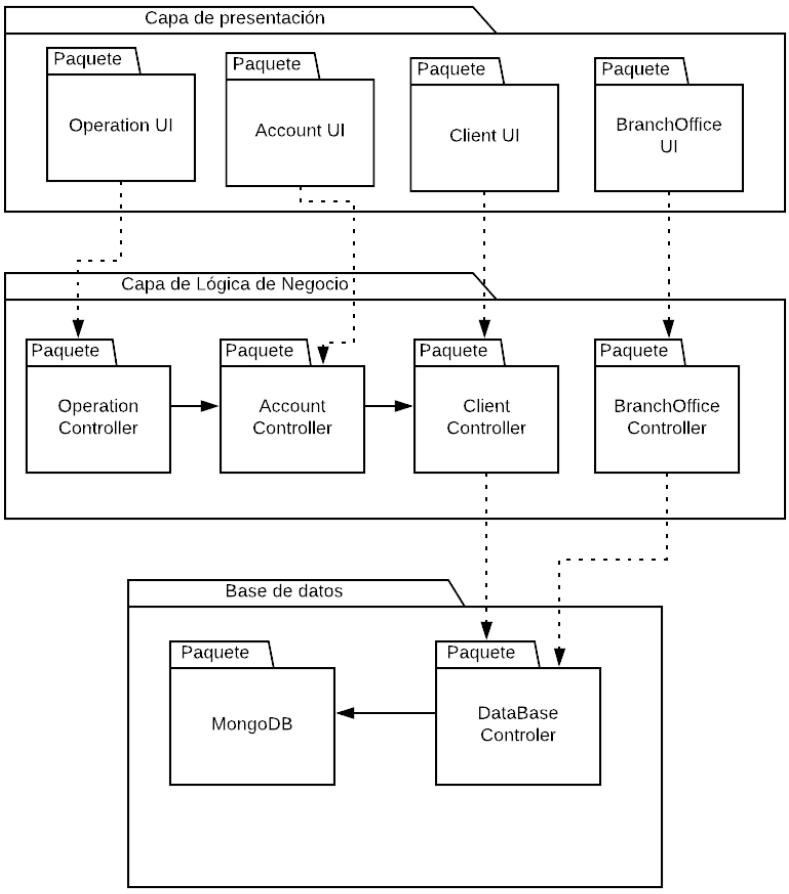
Este diagrama se muestra los estados por los que pasa una cuenta desde su momento de creación, iniciando en un estado inactivo, hasta que el cliente consigne el

valor correspondiente. En caso de realizar un retiro si el balance total es menor a 10.000 vuelve a su estado inactivo. Finalmente en caso de eliminar la cuenta, termina su ciclo




## DISEÑO DEL SISTEMA

### 5. Arquitectura lógica



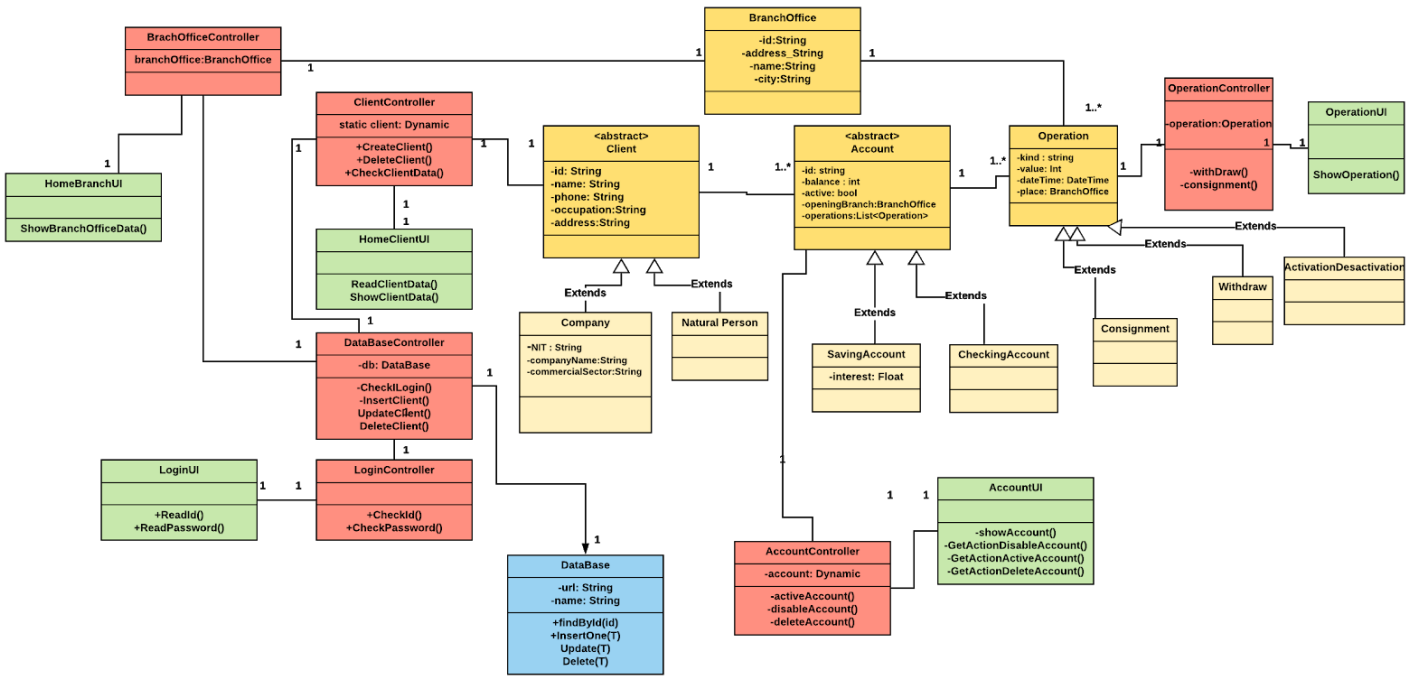

- **Capa de presentación :** Esta capa se encarga de la lectura y muestra en la UI de los datos

- **Operation UI:** Muestra las operaciones en la cuenta del cliente
- **Account UI :** Muestra las cuentas y activa las acciones de retirar y consignar
- **Client UI :** Muestra los datos del cliente, lee los datos cuando se va añadir
- **BranchOffice UI :** Muestra los datos de la sucursal y activa las acciones de búsqueda de clientes

- **Capa de lógica del negocio**
  - **Operation Controller :** Gestiona las operaciones realizada en las cuentas
  - **Account Controller :** Gestionar las cuentas existentes
  - **Client Controller :** Gestiona lo que se requiera hacer con el cliente
  - **BranchOffice Controller :** Gestiona las funcionalidades de la sucursal
- **Base de datos**
  - **MongoDB :** Es el gestor de base de datos
  - **DataBase Controller :** Es el que se comunica con el gestor de base de datos para solicitar o añadir información

## 6. Diseño de la estructura estática del sistema

6.1 Diagrama de Clases del paquete




A nivel estructural la idea fue definir unas clases que sean como **" los modelos"** que van a almacenar los datos, esas clases son las que están de color **Amarillo**, de esta

manera su única tarea es definir la estructura que va a tener los objetos que se van a instanciar.

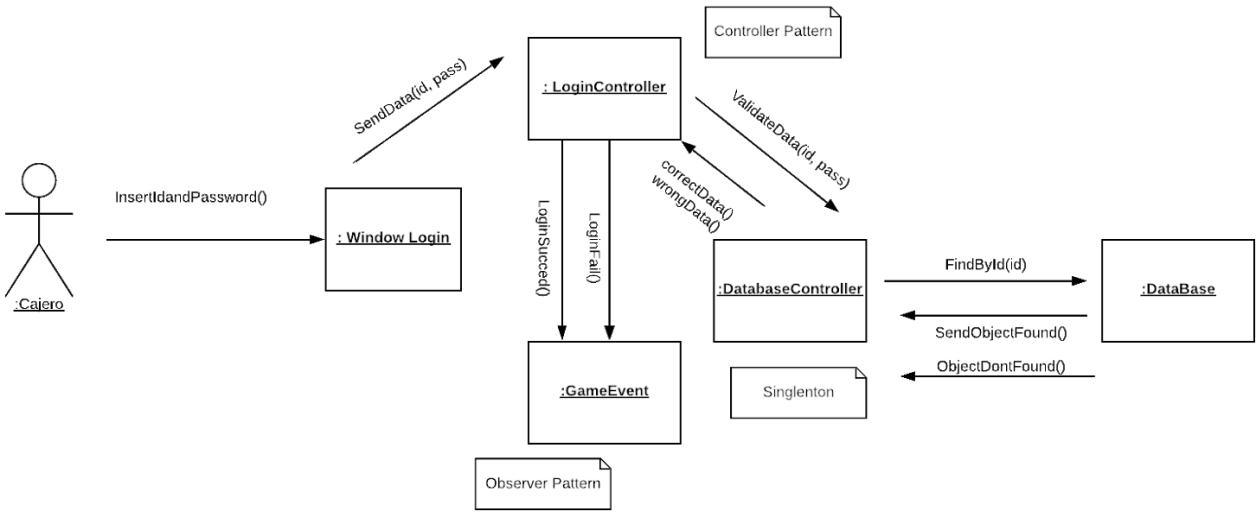
Hay varias clases **"Controladoras"** que son las de color **Rojo**, para así a cada una repartirle tareas específicas y no sobrecargar un solo controlador

Finalmente se definieron unas clases para mostrar los datos y manejar la interacción con el usuario, que son las de color **Verde**, así estas clases sólo se enfocan en mostrar y leer la información

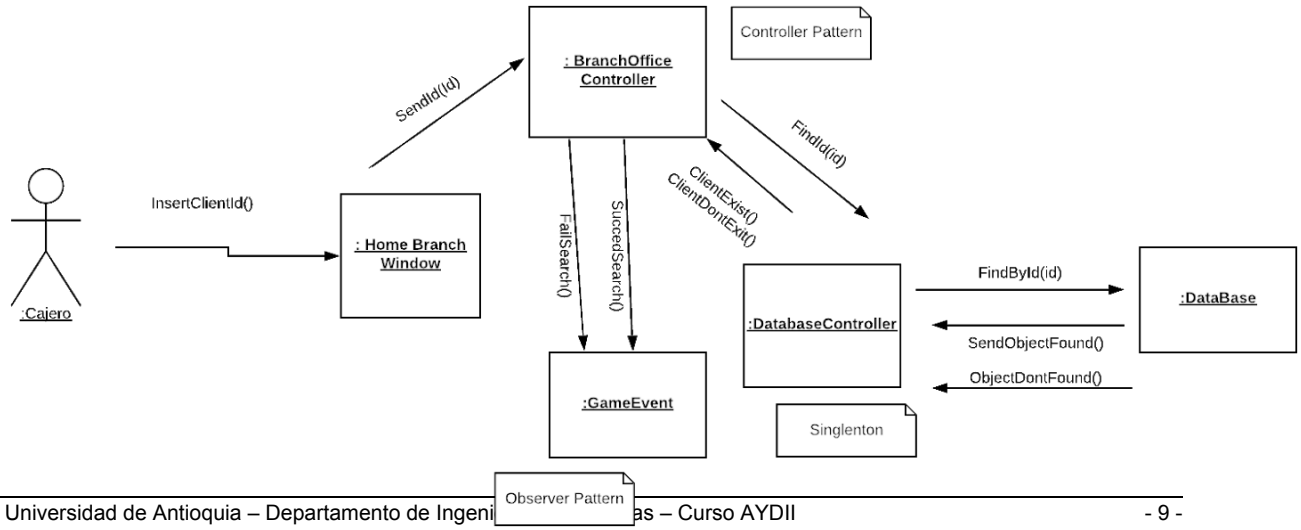
## 7. Asignación de responsabilidades y diseño de interacciones

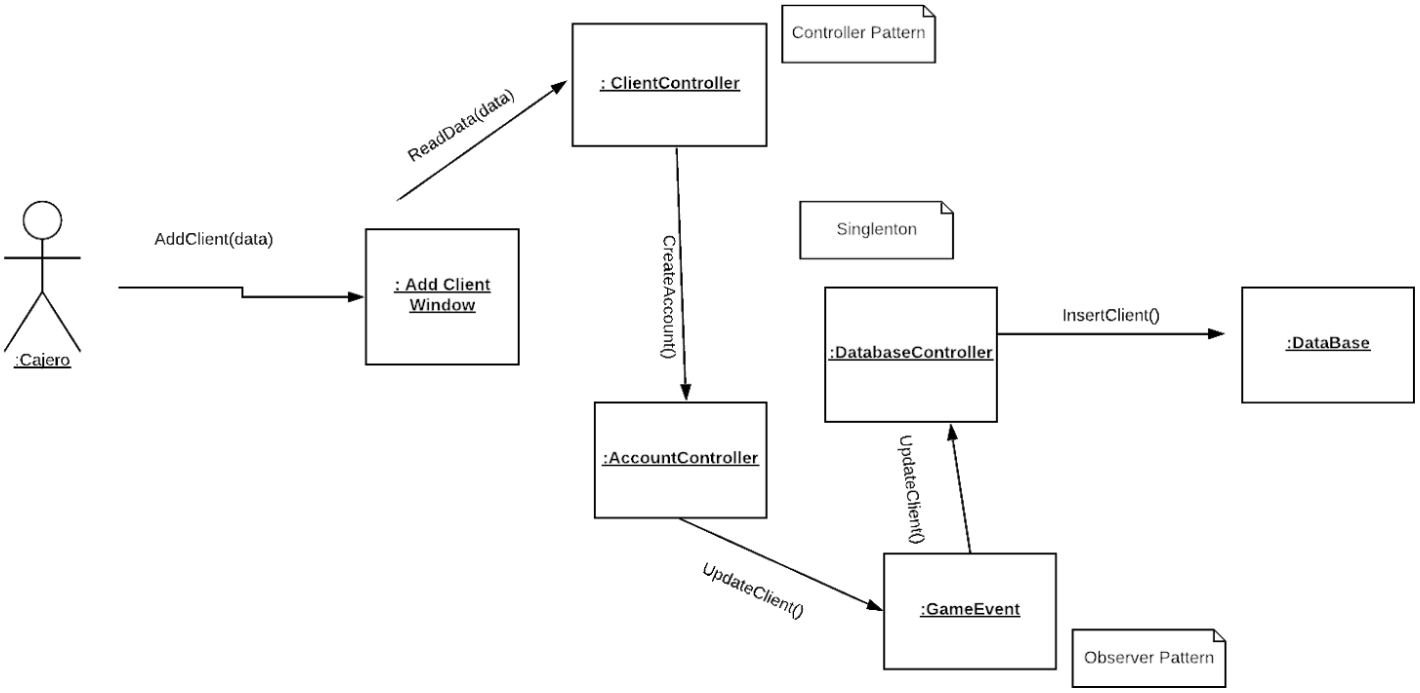
### 7.1 Colaboraciones **[2]**

#### 7.1.1 Colaboración para la operación:Iniciar Sesión en la sucursal




#### 7.1.2 Colaboración para la operación Buscar Cliente



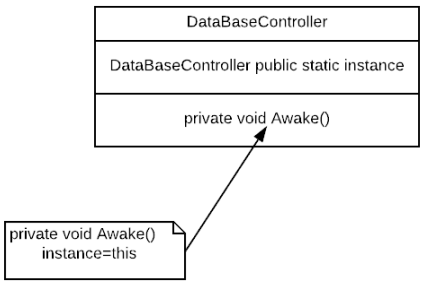



7.1.3 Colaboración para la operación: Añadir Cliente

7.2 Patrones de diseño usados

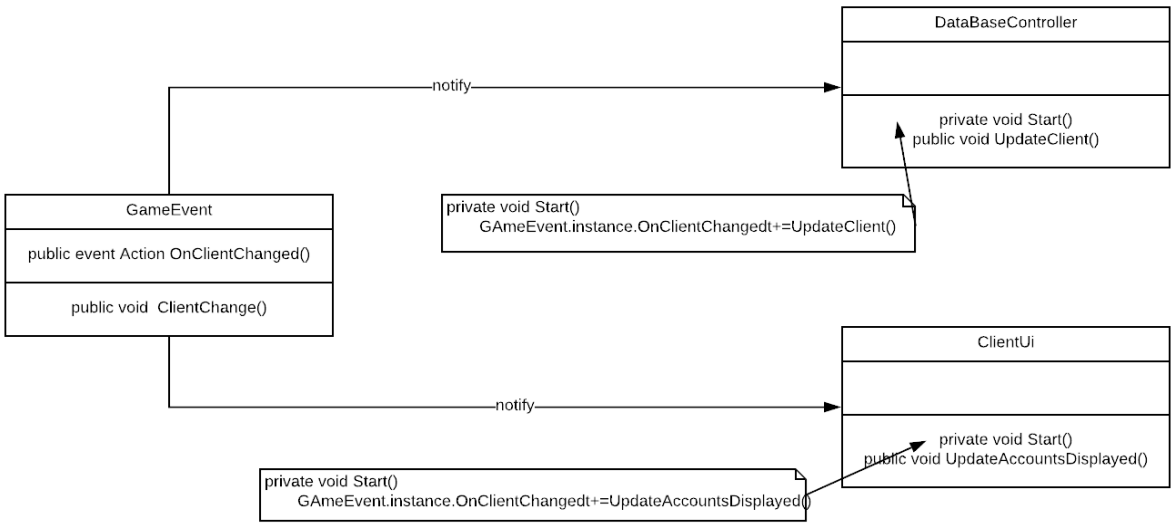
- **Patrones GoF:**
  - **Singleton:** Como hay algunos objetos que durante toda la aplicación van a tener una única instancia, Singleton me ayuda a asegurarse que solo haya una, provisionando a la aplicación un punto global de acceso a esa instancia.

    **Ejemplo** : Cuando se requiere enviar información a la base de datos, aplicando el patrón Singleton a la clase DataBase Controller, desde cualquier punto de la aplicación podría enviar información a la base de datos y estaré seguro que solo hay una instancia



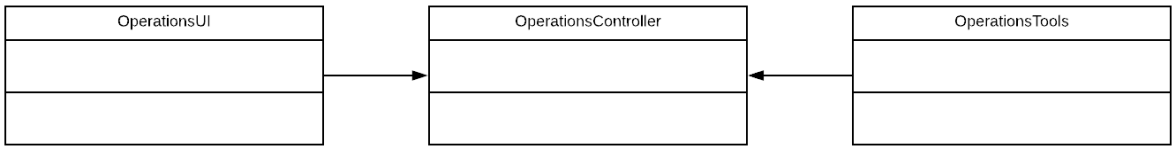

- **Observador: [1]**En el transcurso de la aplicación hay varias acciones que al ejecutarse, deben llamar a métodos de otras clases. Asi que para no crear dependencia entre objetos, aplicar este patrón ayuda a que cuando haya un cambio de estado , varios observadores sean notificados inmediatamente. Además se aprovecha que el lenguaje C# ya tiene el patrón observador implementado

  **Ejemplo:** Cuando un cliente añade una nueva cuenta, esta nueva cuenta del cliente debe ser actualizada en la base de datos por parte de la clase DataBaseController y la interfaz de usuario debe mostrar la cuenta agregada. Son dos observadores que serán notificados por parte del sujeto, en este caso GameEvent.



- **Patrones GRASP**
  - **Controlador:** Para no conectar directamente los objetos de la interfaz con los objetos del negocio creando alto acoplamiento y baja cohesión, el patrón controlador ayuda que al crear una clase Controlador, la lógica y la interfaz ya no se mezclen y soluciona el problema anteriormente dicho.

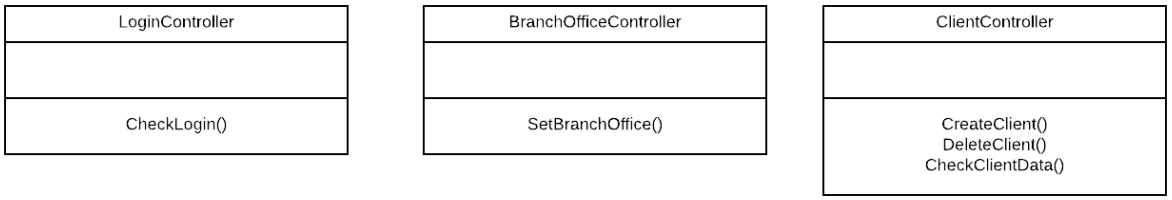
    **Ejemplo:** Para mostrar, gestionar y realizar las operaciones, se usaron tres clases. OperationsUI se encarga de mostrar las operaciones, OperationsController se encarga de leer los datos de la interfaz para mandarle los datos a la parte lógica, es decir a OperationsTools. De esta manera si se requieren hacer cambios o añadir condiciones al momento de consignar o retirar, solo habria



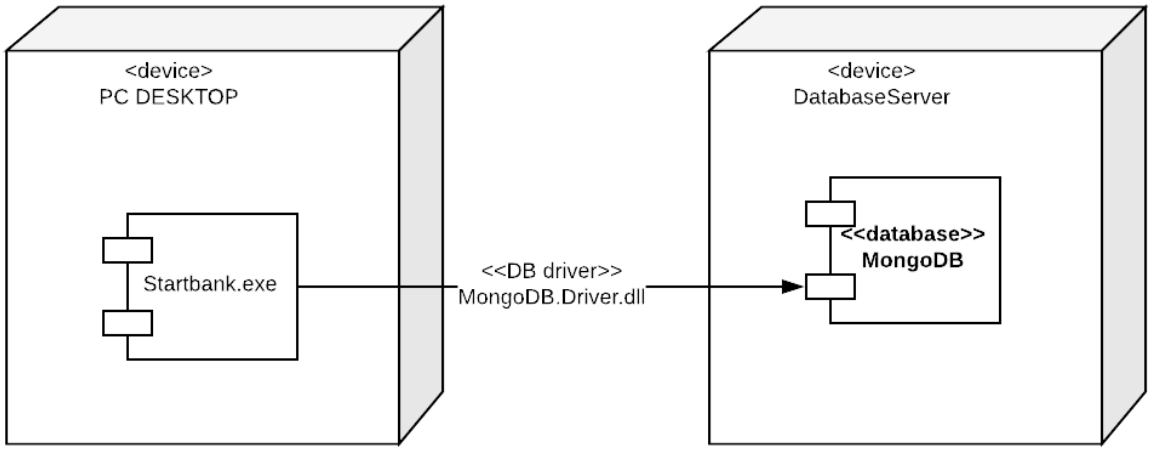

que cambiar la clase OperationsTools, así la clase OperationsUi, permanece intacta

  - **Bajo Acoplamiento : [3]** Como se puede notar en el diagrama de clases del punto 6, hay un bajo acoplamiento a nivel de dependencias, ya que las relaciones son de asociación y esta asociación se hace máximo con 1 o 2 clases, así que varias de las clases en el transcurso de la aplicación para realizar sus tareas, tienen poco conocimiento de las demás.
  - **Alta cohesión:** Hay demasiadas clases por esta razón, para que los comportamientos asignados a cada clase sean concretos y relacionados.

    Para evidenciar los dos patrones anteriormente descritos, se muestra que las tres clases no necesitan de la otra para realizar sus tareas, logrando bajo acoplamiento. Y cada una tiene una función en concreto y relacionada, por ejemplo LoginController verifica los datos del login y BranchOfficeController, se encarga solo de crear el objeto BranchOffice, así se logra alta cohesión




## 8. Despliegue




## 9. Referencias

**[1]** Refactoring Guru (2017). Observer. Tomado de https://refactoring.guru/design-patterns/observer

**[2]**LucidChart. Diagrama de colaboraciones UML. Tomado de https://www.lucidchart.com/pages/es/diagrama-de-secuencia

**[3]** HTML Rules(2017) Principios Grasp. Tomado de https://www.youtube.com/watch?v=Z8uxZrbZofQ&t=321s