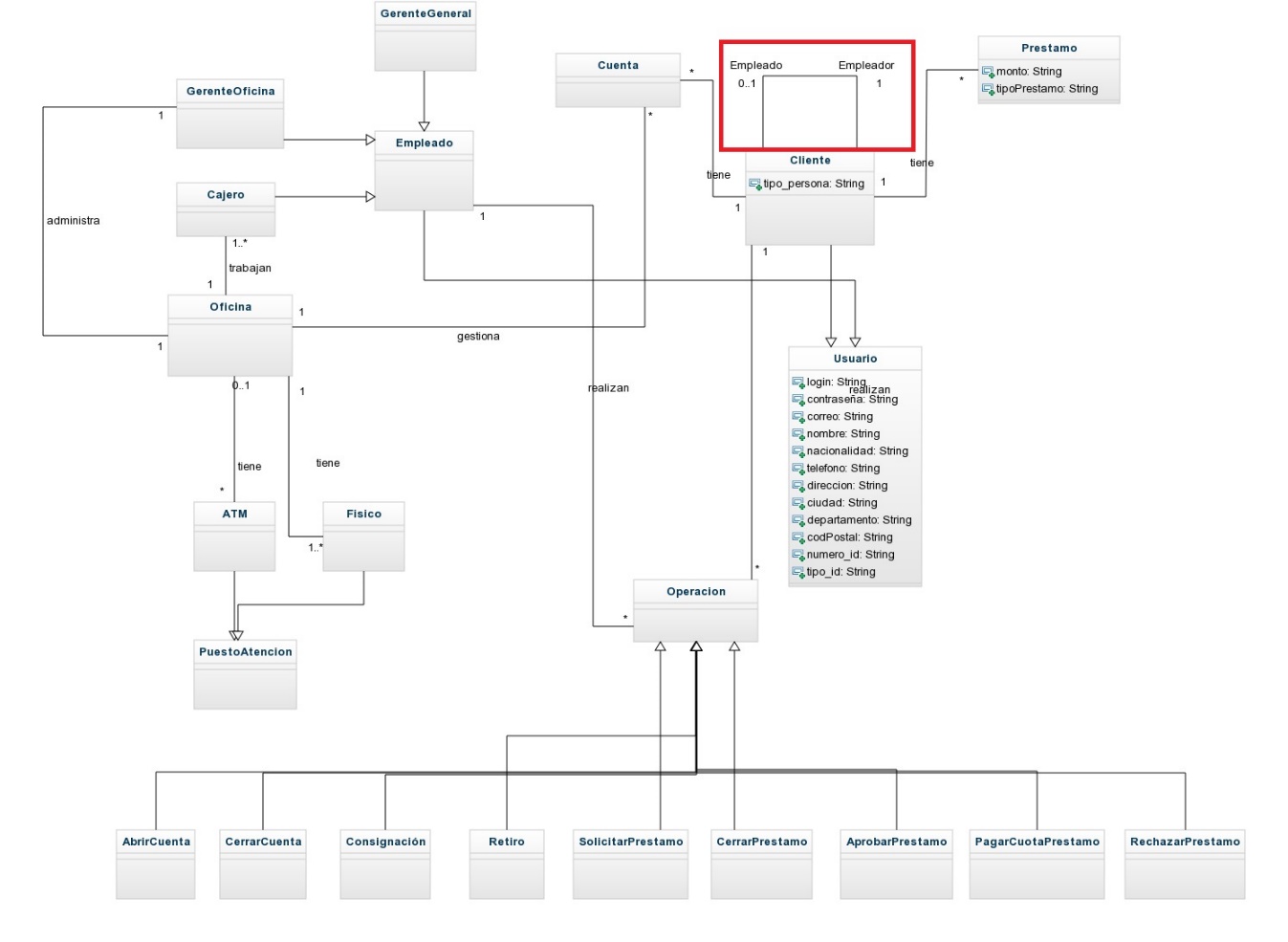
Andrés Camilo Zuleta Calderón – 201413405

Sergio Miguel Madera Rangel – 201417957

**ANALISIS Y DISEÑO ITERACIÓN 3**

1. **CAMBIOS UML ITER 2 A ITER 3**

El único cambio que se realizó al modelo del mundo entre la iteración 2 y la 3 fue la inclusión de una relación de clientes a clientes con el fin de poder modelar la relación Empleador – Empleado necesaria para poder realizar correctamente los requerimientos RF12, RF13 Y RF14 de esta iteración. Todo lo demás permaneció estable.



**Lógica de los nuevos requerimientos de la aplicación (Iteración 3)**

RF10: Es igual al RF6, la única diferencia es que ahora debe poder entrar la cuenta de origen. Es decir solo es añadir un nuevo valor dentro del INSERT INTO de la clausula SQL utilizada para realizar el requerimiento. Lo anterior es lo único debido a que nuestro modelo de tablas y html siempre soportó el requerimiento RF10.

RF11: Al igual que RF10 lo único que lo diferencia con su versión anterior es la posibilidad de agregar la cuenta de origen, y al igual que el requerimiento anterior tanto el modelo relacional del proyecto como los html soportaron siempre esta operación, el único cambio fue el INSERT INTO dentro del DAO que realizaba el registro.

RF12: Se necesita hacer una inserción dentro de una nueva tabla dentro de la cual se contenga una relación 1:N entre clientes dentro de los cuales se contiene toda la información de un empleador y un empleado donde el correo del empleador y el empleado es una FK a clientes y las id de sus cuentas son FKs a la tabla de cuentas.

RF13: En este requerimiento simplemente es llamar al RF10 debido a que pagar la nómina es una operación, al mismo tiempo se deben manejar savePoints con el fin de poder manejar la transaccionalidad solicitada por este punto.

RF14 : Esta operación debe ser igual que su versión a la anterior con la diferencia de que nos encontramos con una nueva entrada la cual debe ser manejada por medio de savePoints con el fin de garantizar el manejo transaccional solicitado por el requerimiento.

RFC5: Consultar prestamos es simplemente llamar a una de las subtransacciones manejae das por el RFC2 donde consultábamos los prestamos de una cuenta, solo que ahora lo hacemos tanto para general y oficinas específicas.

RFC6: Consultar prestamos es simplemente llamar a una de las subtransacciones manejae das por el RFC2 donde consultábamos las operaciones de una cuenta, solo que ahora lo hacemos tanto para general y oficinas específicas.

Se maneja la transaccionalidad de cada una de los requerimientos anteriores simplemente definiendo su nivel de aislamiento como serializable.

**JUSTIFICACION TOMA DE DECISIONES Y DISEÑO APLICACIÓN TRANSACCIONAL**

**PUNTO 1: ASEGURAMIENTO DE LA TRANSACCIONALIDAD DE LOS REQUERIMIENTOS**

**DEFICINIONES:**

**Concurrencia**

**PROBLEMAS:**

Lecturas sucias:

* Suponga que T2 escribe un nuevo valor sobre x y que a T1 se le permite leer ese valor antes que T2 termine. La lectura de T1 es sucia pues T2 no ha hecho commit ni abortado. Si T2 llegara a abortar o a modificar x de nuevo, cualquier cosa que hiciera T1 con x no sería consistente con el estado del modelo.
* Lecturas que se hacen de estados de la DB donde otra transacción activa ha hecho cambios parciales.

Lecturas consecutivas diferentes:

* Dos diferentes lecturas de los mismos recursos estuvieron separados por una transacción que actualizó esos recursos e hizo commit.

Fantasmas:

* Cuando el nivel de granularidad de los candados es de tuplas.
* Cuando una transacción T1 tiene candados sobre todas las tuplas que cumplen un predicado, y una transacción concurrente T2 inserta otra tupla que cumpliría con ese predicado. T1 no tendrá acceso a la información de T2 y podrá dar información no consistente.
* Tuplas que cumplen con un predicado pero no están presentes en una tabla.
* Una solución fácil pero ineficiente es poner un candado sobre toda la tabla.

**Candados sobre predicados**

* Si D es el conjunto de TODAS las posibles tuplas para una tabla (no solo las que existen) entonces se pueden tomar las tuplas que cumplan con el predicado P.
  + El predicado P para un SELECT es trivial. Para un INSERT y DELETE describe exactamente la tupla a insertar o borrar y para un UPDATE hay dos si se piensa en ésta última operación como la eiminación de unas tuplas y la inserción de otras.
* Es entonces posible que, al ejecutar una sentencia SQL, se coloque un candado sobre TODAS las tuplas de D que satisfagan P (o tanto P como P’ en el caso del UPDATE), incluso sobre las que no existan en la tabla.
* SE DEFINE ENTONCES UN CONFLICTO ENTRE DOS OPERACIONES COMO: Al menos una de ellas escribe y la intersección entre los conjuntos que cumplen sus predicados es no nula.
* Cuando una operación entra en conflicto con una concurrente, a la segunda no se le conceden los candados que ella requiere y queda con hambre :D.
* Resuelve el problema de los fantasmas.
* Checkear la intersección de predicados es costoso!

**Niveles de aislamiento:**

Cada transacción ejecuta a un nivel de aislamiento sin importar los niveles de las demás. Las anomalías que se garantizan prevenidas no dependen de cómo operen las otras transacciones.

Las transacciones pueden escoger entre cuatro niveles de aislamiento:

* Serializable: No permite fantasmas. Garantiza lo correcto de todas las aplicaciones. Los demás niveles aumentan la eficiencia pero dan riesgo a errores de consistencia. Todos los candados son de larga duración y de predicado.
* Repeatable Read: Lecturas sucesivas arrojan siempre el mismo resultado, pero se permiten fantasmas. Se logra pues cada SELECT coloca un candado de lectura de larga duración sobre sus recursos, por lo que no es posible que una transacción cambie los datos entre dos lecturas sucesivas. Como no hay candado sobre los predicados, sin embargo, se permiten fantasmas.
* Read Committed: No permite lecturas sucias, pero lecturas sucesivas pueden arrojar diferentes resultados. Un select coloca un candado de corta duración sobre sus recursos (no hay lecturas sucias pues el candado entra en conflicto con candados de escritura que hayan por otras operaciones). Hay lecturas variables pues los candados de lectura son de corta duración y se liberan al acabar con el SELECT.
* Read Uncommitted: Permite lecturas sucias. Un select no coloca un candado de lectura, por lo que no entrará en conflicto con otra operación que ya tenga candados sobre esos recursos.

**Tipos de candado:**

* De larga duración: aguantan hasta el commit. Candados de escritura de larga duración se asocian a UPDATE, DELETE e INSERT en todos los niveles de aislamiento.
* De corta duración: liberan el recurso cuando la sentencia que accedió a él se completó. No son suficientes para asegurar seriabilidad pero, al obligar a una transacción a pedir este tipo de candados, se puede verificar si hay otros candados en conflicto.

**NIVEL DE AISLAMIENTO ELEGIDO Y JUSTIFICACIÓN**

Se comprueba que el comportamiento transaccional de los requerimientos que implican actualización, inserción o borrado de información no presentan ningún tipo de anomalías.

Lo anterior debido a que todos los requerimientos presentados por la aplicación ( o al menos aquellos en los que estrictamente se específica el comportamiento transaccional de estos) presentan el mayor nivel de aislamiento transaccional, es decir, aislamiento serializable, lo cuál en primer lugar garantiza el cumplimiento de las propiedades **ACID** para cada transacción realizada por estos requerimientos y en segundo lugar al proteger la transacción de actualizaciones intermedias y reservar los derechos de escritura y lectura para el primer usuario que acceda a la información se evitando así problemas de concurrencia como pueden ser el problema del hambre y los abrazos mortales. Lo anterior debido a que el aislamiento de tipo serializable bloquea la tabla mientras un usuario interactúa con ella y deja en stand by al otro usuario que está intentando acceder a la tabla. Por las razones anteriores podemos justificar y demostrar que todos los requerimientos realizados para esta iteración presentan a la perfección un comportamiento transaccional.

**PUNTO 2: SISTEMA DE BACK-UP PARA LA BASE DE DATOS**

El sistema de respaldo propuesto para soportar la base de datos es el siguiente:

Se utilizan una serie de archivos JSon, cada uno representando una tabla respectiva del modelo relacional propuesto para este proyecto. Dichos archivos JSon contienen toda la información de las tablas que pertenecen a la base de datos. La infraestructura necesaria para lograr dicho sistema de back-up es simplemente la de la existencia y manejo de dichos archivos de texto.

**ES CONFIABLE PORQUE:**

Los archivos de texto JSon requieren poco espacio en memoria, pueden ser almacenados en memoria secundaria y terceria sin mayores costos lo cual asegura su persistencia en caso de que se presenten fallos dentro del sistema principal. La recuperación de estos es muy fácil de realizar y el manejo de estos no requiere conocimientos avanzados, además de que al ser uno de los archivos más manejados por la comunidad informática se puede encontrar gran soporte para realizar gran variedad de operaciones con estos sin complicación alguna.

**ES CONVENIENTE PORQUE:**

Adicionalmente en este al ser la aplicación propuesta una aplicación web el uso de archivos JSon nos permite convertir nuestra aplicación en una aplicación de tipo REST debido a que el envío de servicios REST está basada en JSON. El parseo de JSon a objetos java o a inserciones de tablas es fácil de manejas y además el hecho de que se pueda manipular de distintas formas la información contenida por los JSon resulta en un sistema que le brinda versatibilidad al programador para realizar distintas operaciones necesarias para el desarrollo de la aplicación.

**¿CÓMO FUNCIONA?**

1. Se guarda 1 copia de la carpeta que contiene los archivos JSon (cada Json con un nombre representativo que permita saber la tabla que representa) en memoria secundaria (disco local del computador) y otra copia en memoria terciaria (servicios cloud de almacenamiento)
2. Cada vez que se realice una modificación a las tablas (ya sea de inserción o de eliminación) se agregará la respectiva información de modificación en una nueva línea al archivo JSon que represeta dicha tabla con el fin de que haya consistencia de la información entre las tablas y los archivos Json.
3. En caso de presentarse una falla en el sistema BancAndes que cause la pérdida de los datos contenidos por las tabla de la base de datos se procederá a cargar con métodos ya implementados dentro de la lógica de la aplicación cada uno de los JSon con el fin de que el sistema sea reiniciado y las tablas posean la misma información que contenían antes de presentarse el fallo.
4. Los JSon serán parseados utilizando la librería de google de json, la implementación de dicho parseo se encuentra en la clase jsonParser de la aplicación que está siendo desarrollada.
5. La información será parseada en forma de objetos java de tipo VOS.
6. Finalmente usando los métodos get de dichas clases se podrá extraer la información de dichos VOS y se podrán hacer las respectivas inserciones en la base de datos.

**BONO: DIFERENCIAS MANEJO TRANSACCIONAL JBOSS VS PROGRAMADOR**

La principal diferencia entre delegarle el manejo transaccional al servidor de aplicaciones a dejárselo al programador radica en que el servidor de aplicaciones utiliza las configuraciones designadas por defecto para realizar cada una de las transacciones, es decir, autocommit en true, no manejo de savePoints según sea necesario, rollback hasta el inicio de la transacción sin guardar avances en la transacción y nivel de aislamiento en lectura confirmada. Mientras que cuando toda la responsabilidad de eso se le deja al programador debe ser este quien determine cada uno de las configuraciones que él quiere o necesita para realizar la transacción según lo necesite.

**VENTAJAS:**

**Servidor de aplicaciones:** Brinda un nivel de abstracción atractivo para el programador, liberándole de las tareas del manejo transaccional y permitiéndole así simplemente concentrarse en las tareas de desarrollo a la vez que el servidor de aplicaciones le brinda un nivel decente de manejo transaccional que permitirá realizar de forma coherente la gran mayoría de transacciones usuales.

**Manual:** Permite que el programador especifique todas y cada una de las características de la transacción de forma que esta salga tal cual el programador lo quiere y así de paso realizar transacciones de una forma más precisa que se ajusten a las reglas de negocio y a las necesidades particulares de un problema y de el programador.

**DESVENTAJAS:**

**Servidor de aplicaciones:** Las transacciones que requieran de configuraciones especificas o manejen problemas transaccionales particulares no podrán ser realizas de la forma esperada si se le deja el manejo transaccional al servidor de aplicaciones y en el peor de los casos se perderá la consistencia de la información y la coherencia entre estas y las reglas de negocio o puede que las transacciones nisiquiera funcionen.

**Manual:** Necesita de cierto nivel de conocimiento por parte del programador para que no sea este mismo quien vulnere la consistencia y coherencia de los datos.

Las arquitecturas de contenedores de aplicaciones normalmente proporcionan dos clases de transacciones: aquellas gestionadas por el contenedor (las transacciones se gestionan automáticamente usando algún API como JTS) y aquellas gestionadas explícitamente por el programador (el desarrollador DEBE incluir las transacciones en el código de su aplicación o cliente). Cada clase tiene entonces sus ventajas y desventajas:

|  |  |  |
| --- | --- | --- |
|  | En el Contenedor | Por el Programador |
| Ventajas | - Las interfaces ofrecidas por el contenedor (APIS) hacen que las transacciones sean fáciles de implementar y de utilizar. La escalabilidad, además, se hace más fácil.  - Como el contenedor es desarrollado alrededor de estándares pre-establecidos, sus modules se acoplarán más fácilmente a ambientes extraños de despliegue.  - Evita todas las labores de “carpintería” que pudiera traer el comportamiento transaccional.  - El manejo de la información es limpio. En particular, proveen un amplio soporte de seguridad. | - Las transacciones pueden adaptarse a la lógica y restricciones del negocio. Las ventajas se resumen en la LIBERTAD y el CONTROL que tiene el programador para interactuar con la base de datos al nivel de aislamiento que ésta le permita, con la lógica transaccional que él desee y utilizando los comandos e interfaces que él encuentre cómodos. |
| Desventajas | - El comportamiento transaccional es independiente de la lógica del negocio. Las operaciones son establecidas y están dadas por defecto.  - El contenedor maneja toda la interacción con la base de datos, incluyendo los niveles de aislamiento. Cambiar este comportamiento no es recomendable. | - El desarrollador debe darse a la tarea de comenzar explícitamente la transacción, de terminarla o anularla. Esto es más dispendioso (solo “carpintería”).  - La implementación de la transacción es más difícil, los módulos deben adaptarse a ambientes extraños uno por uno (y es tarea del desarrollador hacerlo). |

El cuadro, sin embargo, no es camisa de fuerza ni absoluto en ninguna medida. Algunos programadores encuentran indeseable que se les imponga un API (por la misma perdida de control) por lo que no sería una ventaja para ellos que los contenedores ofrezcan uno. Otros pensarán que la independencia del comportamiento transaccional a la lógica del negocio es deseable para poder ampliar las funciones que se le ofrecen al usuario. Es así como cada quién deberá construir su propia matriz y tomar su propia decisión.

Bastente, P., Almenares, F. *Transacciones y Seguridad en J2EE.* [En línea] Universidad Carlos III de Madrid. [Citado el: 20 de Octubre de 2013.] http://ocw.uc3m.es/ingenieria-telematica/software-de-comunicaciones-1/UDs\_JEE/jee-unidad-4.