Valentina Calderón: 202020771.

Daniela Espinosa: 202022615.

Juan Camilo Neira: 201922746.

**Etapa de análisis.**

**Requerimientos funcionales que la aplicación debe soportar.**

Un usuario puede crear un proyecto y agregar a otros usuarios como participantes del proyecto. El participante puede realizar actividades como parte del proyecto. Asimismo, el participante debe indicar el tipo de actividad que realiza y cuando inicia y cuando finaliza de trabajar en esta actividad. Además, el participante puede registrar una actividad que él o ella recién realizó o que realizó en días o horas pasadas. Por último, el usuario puede acceder a un reporte de las actividades realizadas por este o por otros participantes en el proyecto.

**Restricciones del proyecto.**

Tenemos las siguientes restricciones sobre la construcción del software. Debemos utilizar Git, que es un software de control de versiones. En este sentido, todas las extensiones de los archivos que utilicemos deben ser compatibles con este software. Por ejemplo, debemos utilizar archivos planos (csv, txt, etc.), ya que estos son en gran medida compatibles con Git al momento de almacenarlos y sincronizarlos en este repositorio. Asimismo, existe la restricción de que la interfaz de la aplicación debe estar basada en consola. Por último, contamos con tres entregas: análisis, diseño e implementación. Estas tres entregas deben ser completadas en un rango de un mes.

**Etapa de diseño.**

**Primera iteración.**

**Responsability- Driven Design:**

1. Roles:

* Proyecto.
* Participante.
* Actividad.

1. Responsabilidades.

* Proyecto:
  + Consultar el nombre del proyecto.
  + Consultar la descripción del proyecto.
  + Consultar el reporte de un participante.
  + Consultar la fecha estimada de inicio.
  + Consultar la fecha estimada de finalización.
  + Almacenar los participantes del proyecto.
  + Almacenar las actividades.
* Participante:
  + Consultar el nombre del participante que creó el proyecto.
  + Consultar el correo del participante que creó el proyecto.
  + Consultar el nombre de un participante.
  + Consultar el correo de un participante.
  + Agregar un participante al proyecto.
* Actividad:
  + Consultar el nombre.
  + Consultar la descripción.
  + Consultar la fecha.
  + Consultar el tipo de actividad.
  + Almacenar los reportes de cada participante.
  + Conocer el tiempo total invertido.
  + Conocer el tiempo promedio por tipo de actividad.
  + Conocer el tiempo por cada día.

1. Colaboraciones.

* Agregar participantes a un proyecto.
* Consultar el reporte de un participante.
* Consultar el participante que creó el proyecto.

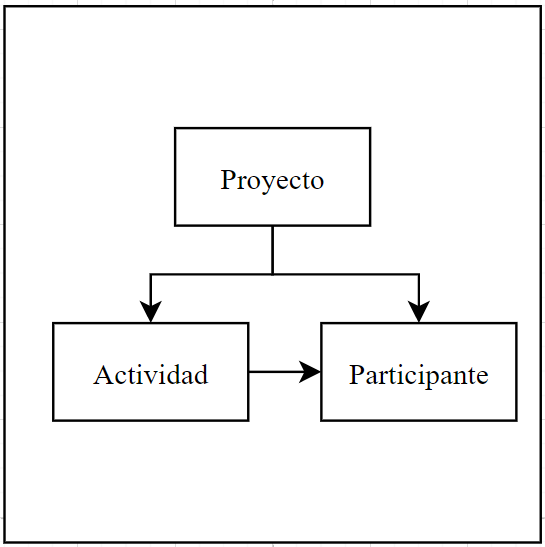


Figura. Esquema general.

**Segunda iteración.**

Realizamos cambios en el esquema general del diseño, debido a que los tres roles (Proyecto, Actividad y Participante) cumplen muchas responsabilidades. Así que, continuando con el diseño iterativo, subdividimos el esquema general de la siguiente forma.

**Responsability- Driven Design:**

1. Roles:

* Aplicación.
* Proyecto.
* Participante.
* Actividad.
* Estadística.

1. Responsabilidades.

* Aplicación:
  + Consultar la descripción del proyecto.
  + Consultar la fecha estimada de finalización del proyecto.
  + Consultar el nombre del participante que creó el proyecto.
  + Consultar el correo del participante que creó el proyecto.
  + Consultar el correo de un participante.
  + Consultar la cantidad de participantes del proyecto.
  + Consultar la descripción de una actividad
  + Consultar la fecha de una actividad.
  + Consultar el tipo de una actividad.
  + Consultar el reporte de un participante.
* Registro:
  + Crear un proyecto.
  + Crear una actividad.
  + Crear un participante.
* Proyecto:
  + Conocer el nombre del proyecto.
  + Conocer la descripción del proyecto.
  + Conocer la fecha estimada de finalización.
* Participante:
  + Conocer el nombre del participante que creó el proyecto.
  + Conocer el correo del participante que creó el proyecto.
  + Conocer el nombre de un participante.
  + Conocer el correo de un participante.
* Actividad:
  + Conocer el participante.
  + Conocer el nombre de la actividad.
  + Conocer la descripción.
  + Conocer la fecha.
  + Conocer el tipo de actividad.
  + Conocer el tiempo invertido en una sesión por participante.
* Reporte:
  + Calcular el tiempo total invertido.
  + Calcular el tiempo promedio por tipo de actividad.
  + Calcular el tiempo por cada día.

1. Colaboraciones.

* Calcular las estadísticas del reporte.
* Actualizar el reporte del participante.

**Tercera iteración.**

Esta es una actualización del documento de diseño después de terminar la implementación de la lógica del proyecto. Como era de esperarse, hubo cambios en el diagrama de clases, por ejemplo, se añadió nuevos métodos y atributos, se crearon nuevas relaciones entre clases e incluso se decidió no implementar algunas clases.

**Decisiones relevantes durante la implementación del modelo.**

En primer lugar, en la clase App, que es la clase que interactúa con el usuario, decidimos que el participante se identificaría principalmente por su correo. Esto se debe a que los nombres se pueden repetir, pero los correos no. Por esta razón, la App solicita, en algunas ocasiones, solo el correo y no el nombre.

En segundo lugar, la funcionalidad de cronómetro se llevó a cabo con la clase java.time y con métodos propios para obtener el tiempo en minutos entre un tiempo inicial y un tiempo final. Por ejemplo, se utilizó LocalDate para obtener el tiempo actual en formato (Hora: Minuto: Segundo). Asimismo, Duration se utilizó para encontrar la diferencia en minutos entre dos tiempos en el formato antes mencionado. Ahora bien, el cronómetro consistió en almacenar estas diferencias de tiempo en un ArrayList y posteriormente recorrer esta estructura de datos e ir sumando estas diferencias para calcular el tiempo total invertido.

En tercer lugar, se decidió, por un lado, registrar actividades pasadas y, por otro lado, realizar una actividad a partir del momento en que se ejecuta esta opción. En este último caso es cuando se utilizó el cronometro, a diferencia del primer caso en el cual solo es necesario determinar la diferencia entre la hora de inicio y la hora de fin ingresada por parámetro.

En cuarto lugar, no se implementó la clase Sesión que se encuentra en la segunda iteración de este documento. Esto se debe a que la clase java.time y los métodos creados en la clase Actividad fueron suficientes para calcular y almacenar una hora de inicio y una hora de fin, y por tanto no estuvimos en la necesidad de crear otra clase que tuviera esta responsabilidad.

En quinto lugar, no se implementó la clase Almacenamiento que se encuentra en la segunda iteración de este documento. Esto se debe a que lo más lógico, en la programación orientada a objetos, el que objeto principal, que en este caso es la clase Proyecto, almacene los objetos que le pertenecen, como la clase Actividad o Participante.

En sexto lugar, en la clase Registro y la clase Reporte se implementó solo atributos y métodos de tipo static. Esto se debe a que no es lógico que exista una instancia de alguna de estas dos clases.

En séptimo lugar, decidimos que el reporte de las actividades de un participante no se almacena en una estructura de datos, sino que este reporte se genera cada vez que se solicita. Esto se debe a que el reporte cambia cada ocasión que el participante realiza una nueva actividad y, por tanto, este almacenamiento sería obsoleto con cada nueva actividad registrada.

**Colaboraciones y Diagramas de secuencia.**

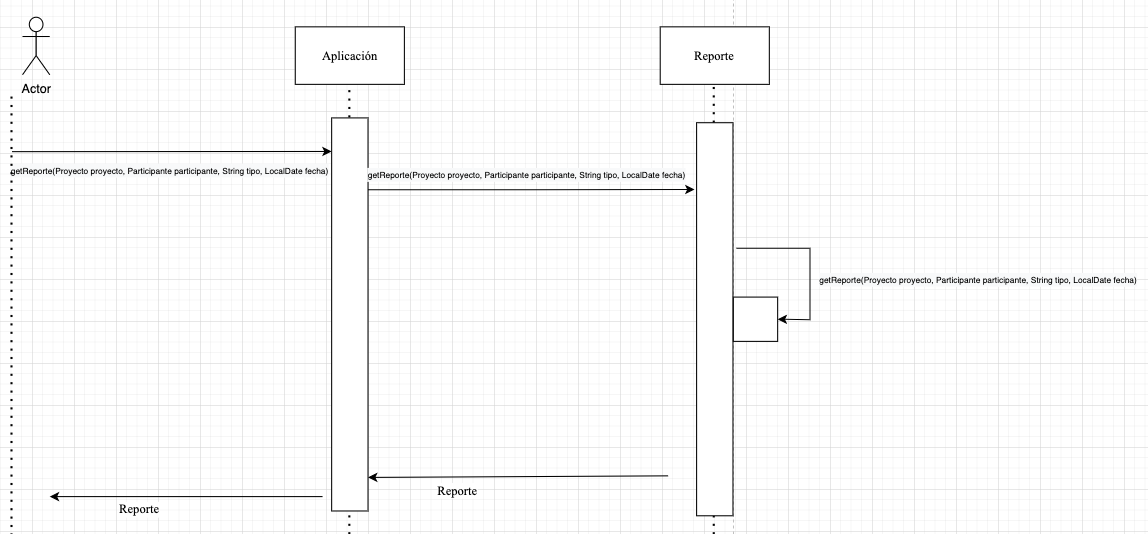


Figura. Diagrama de secuencia de buscar un reporte.

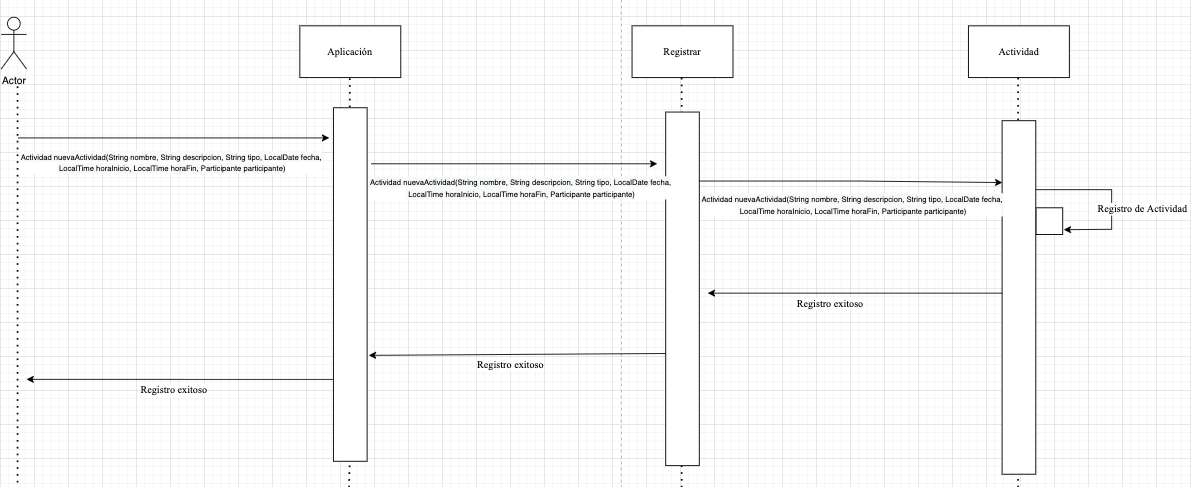


Figura. Diagrama de secuencia de crear una actividad.

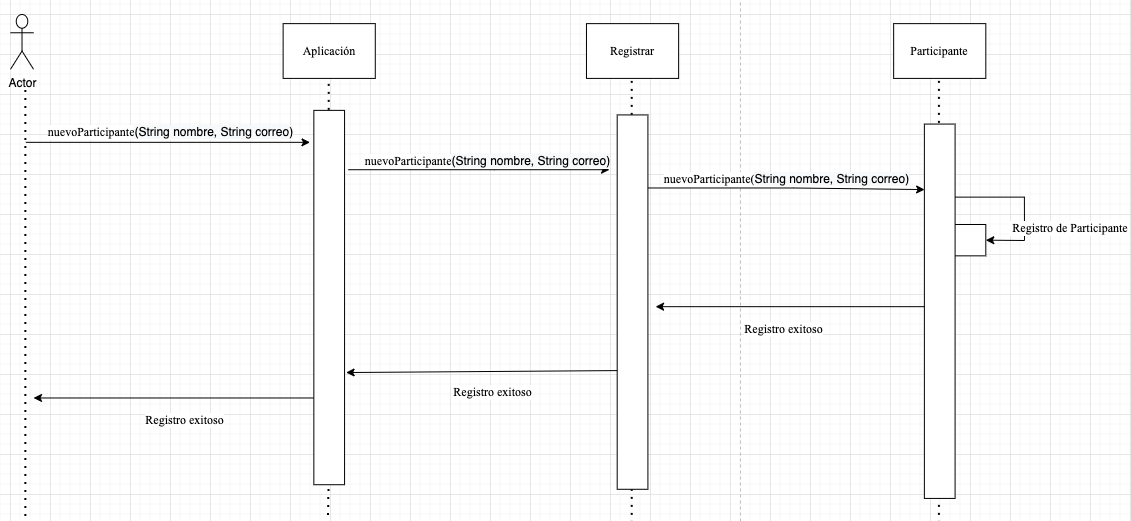


Figura. Diagrama de secuencia de crear un participante.

**Cuarta iteración.**

Esta iteración es una actualización del documento de diseño después de implementar la interfaz gráfica de usuario. No hubo cambios en el modelo del programa, es decir en los métodos e interacciones de las clases Proyecto, Actividad y Participante. En este caso, se añadió nuevas clases con el propósito de crear una interfaz gráfica con el usuario.

**Decisiones relevantes durante la implementación de la interfaz gráfica de usuario.**

En primer lugar, se tomó la decisión de cada clase de la interfaz gráfica de usuario fuera un JFrame, en cambio de reutilizar un mismo JFrame e ir agregando y eliminado distintos JPanel. Esto tuvo la implicación de que en cada cambio de pestaña se puede evidenciar que se cierra y se abre un nuevo JFrame.

En segundo lugar, se tomó la decisión de utilizar librerías externas, las cuales se almacenaron en la carpeta lib del proyecto. En este caso, utilizamos un calendario desplegable para que el usuario seleccione la fecha y un menú desplegable para que el usuario seleccione una hora. Únicamente para estos dos casos decidimos utilizar estos componentes ya implementados y probados.

En tercer lugar, tomamos la decisión de implementar el patrón de diseño de fachada con la clase Administrador. Consideramos que este es un patrón de diseño adecuado en nuestro proyecto porque es una forma de unificar la comunicación entre la interfaz gráfica de usuario y el modelo del programa.

**Quinta iteración.**

Esta iteración es una actualización del documento de diseño en la que se incluye nuevos requerimientos y correcciones del diseño hasta ahora implementado. Además, se incluye las estrategias para el manejo de los errores que surjan durante la ejecución del programa.

**Correcciones del diseño.**

Como se mencionó en la iteración anterior, estamos utilizando el patrón de diseño fachada mediante la clase Administrador. Ahora bien, vamos a implementar también el patrón de diseño singleton, con el fin de restringir la creación de objetos de la clase Administrador. Hasta ahora habíamos declarado a todos sus atributos y métodos como estáticos, con el fin de que no existiera una instancia de esta clase cuyo propósito es comunicar a la interfaz gráfica de usuario con la lógica del programa. El patrón singleton nos permite asegurar que la clase Administrador tenga una instancia única, y por tanto lo implementamos en nuestro proyecto.

En segundo lugar, en la clase Reporte existe el método getReporte cuya responsabilidad es generar las estadísticas del reporte y crear la interfaz gráfica correspondiente para que el usuario visualice el reporte. Esto lo identificamos como un problema, ya que esto no respeta el principio de diseño Single Responsibility, debido a que existen dos razones para modificar esta clase. Por tanto, nuestra solución es crear una instancia de la clase Reporte para así transferir las estadísticas en un paquete de datos a la clase Ventana\_Mostrar\_Reporte.

En tercer lugar, identificamos que estamos implementando el patrón de diseño Inyección de dependencias. Esto se debe a que los constructores de las clases Proyecto, Tarea, Actividad y Reporte reciben, dentro de sus parámetros, variables de clases. En este caso, el constructor de la clase Proyecto, por ejemplo, no tiene la responsabilidad de crear una variable de tipo Participante, sino que la responsabilidad de crear esta variable es del método de la clase Registro que llama al constructor de Proyecto.

**Diagrama de clases.**

En el siguiente enlace se encuentra un diagrama de clases simplificado. Este tiene el propósito de explicar la colaboraciones y dependencias entre las clases del modelo y entre estas clases y la interfaz gráfica de usuario, la cual se simplifica en una sola clase.

Enlace: [UML\_Simplificado.svg](https://uniandes-my.sharepoint.com/:u:/g/personal/j_neira_uniandes_edu_co/Eejt-YmNdGhDsXEH2ak-3x8Bc71RsY0AXmgjBD1m4or8vQ?e=Edew1X)

En el siguiente enlace se encuentra el diagrama de clases completo de la interfaz gráfica de usuario junto a la lógica del programa.

Enlace: [UML.svg](https://uniandes-my.sharepoint.com/:u:/g/personal/j_neira_uniandes_edu_co/ES2z37iMfiBEvPHeTec0D80BGysvGKUbzUA8TT2vQ4hzag?e=3zNYep)

**Estrategias para el manejo del error.**

En primer lugar, en la interfaz gráfica verificamos que los JTextField no estén vacíos y que, en general, el usuario haya diligenciado todos los datos que se le solicita. En caso de un error, la misma clase es la encargada de informar al usuario, mediante el despliegue de una ventana, cuál dato no fue diligenciado. El usuario no podrá avanzar hasta solucionar este error.

En segundo lugar, verificamos que la fecha de fin de una actividad sea anterior a la fecha de inicio. En caso de error, la misma clase se encarga de desplegar una ventana e informarle al usuario sobre el error. Este no puede avanzar hasta que la fecha de fin sea posterior a la fecha de inicio. Lo mismo sucede en caso de que la hora de fin sea anterior a la hora de inicio al momento de registrar una actividad pasada.

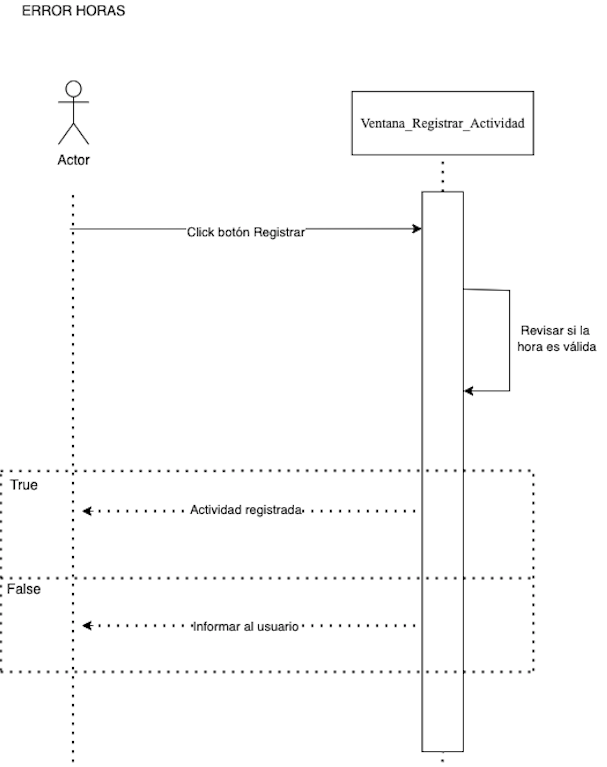
En tercer lugar, en caso de que el usuario diligencie los datos, pero estos estén incorrectos o cambien en el tiempo, existen las funcionalidades para que posteriormente este actualice estos datos. Por ejemplo, si el usuario después desea cambiar la fecha aproximada de fin del proyecto, existe una funcionalidad para tal propósito.

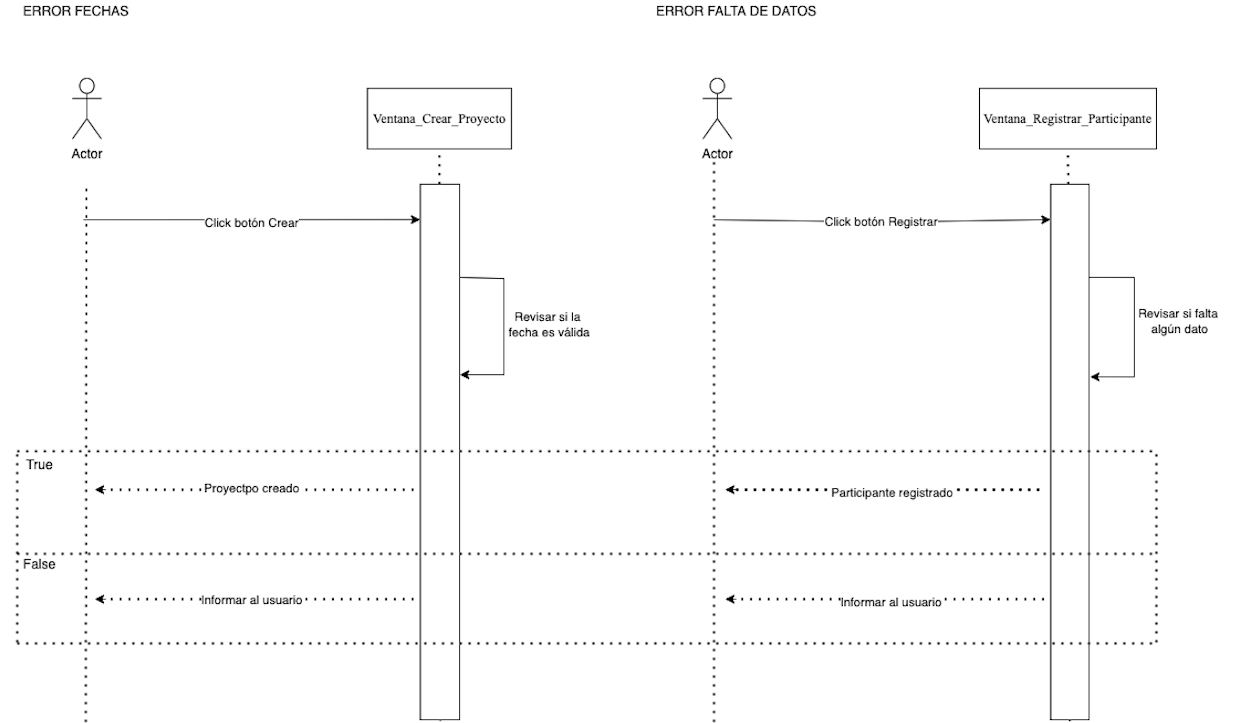
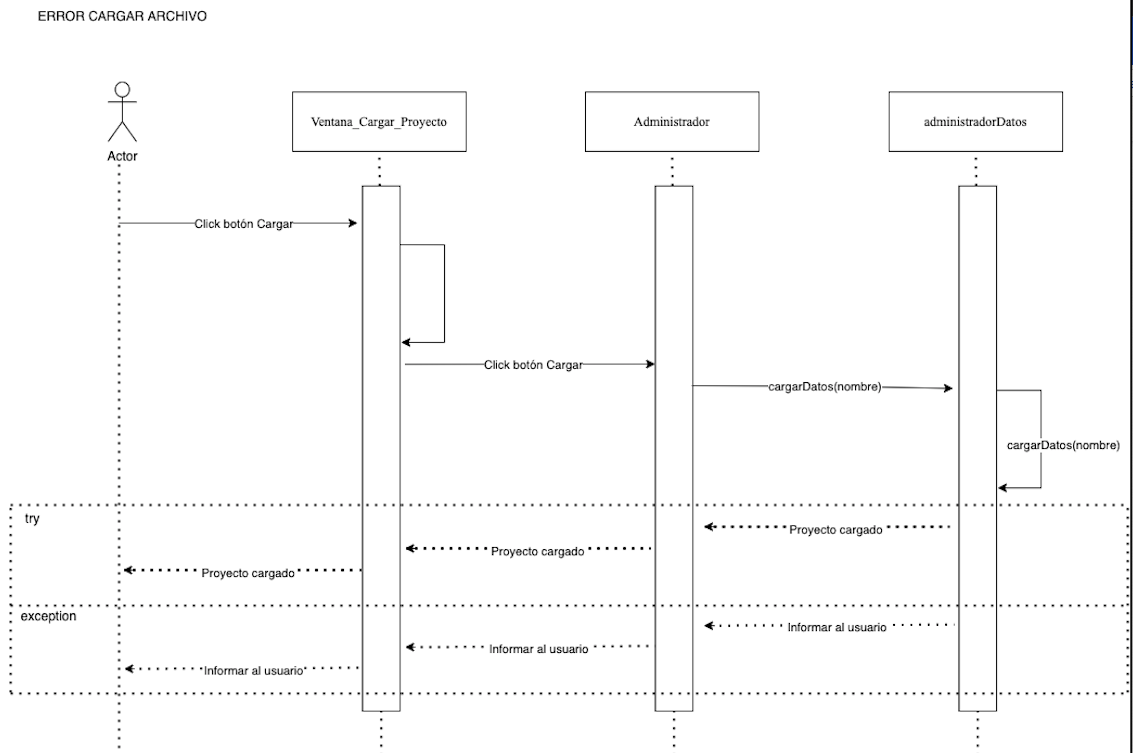
En cuarto lugar, en caso de que no exista el correo del participante cuando se registra una actividad pasada o se crea el reporte, la clase que detectó este error tiene la responsabilidad de informarle al usuario. Detectar este error se hace mediante el llamado de un método que retorna un valor booleano después de consultar si existe el correo. Esto mismo puede suceder si el usuario consulta un paquete, tarea o actividad que no existe.

En quinto lugar, cuando la clase administradorDatos no encuentra el archivo del proyecto en el disco local, este error es atrapado por la clase Ventana\_Cargar\_Proyecto, la cual despliega una ventana en la que se le informa al usuario que no se encontró el archivo.

En sexto lugar, en nuestro proyecto es posible descargar y cargar archivos planos del disco local. Estos archivos almacenan un proyecto, es decir, sus atributos, sus participantes y los paquetes creados. Sin embargo, al momento de cargar estos datos es posible que estos se hayan corrompido, en otras palabras, que los datos que se estén cargando a la aplicación no sean precisamente los que espera el usuario. Lo anterior requiere de una detección del error activa, la cual consiste en desplegar en una ventana los atributos, participantes y paquetes del proyecto, con el fin de que el usuario verifique que estos sean correctos antes de cargarlos en la aplicación.

**Diagrama de secuencia.**





**PROYECTO 3 – PARTE 2**

**Reflexión:**

Después de realizar las 3 fases del proyecto, consideramos que es fundamental hacer la siguiente reflexión sobre el proceso de diseño. Presentaremos una posición crítica acerca de nuestro trabajo, identificando con ejemplos puntuales las partes problemáticas del proceso, así como los aspectos positivos.

En primer lugar,

Proyecto 1.

* + ¿Qué cosas salieron bien? ¿Qué decisiones resultaron acertadas?
    - En el documento de diseño seguimos la metodología Responsibility Driven Design que nos facilitó la planificación del proyecto. Esto nos permitió descomponer un problema complejo en los distintos roles y responsabilidades que después implementaríamos.
    - En general, el proceso de análisis salió bien porque identificamos las entidades que aparecen, las características (atributos) de esas entidades y establecimos relaciones entre estas, incluyendo asociaciones y relaciones de herencia.
    - Asimismo, hicimos una descripción de los requerimientos funcionales y de las restricciones del proyecto. Esto nos ayudó a identificar los objetivos que buscábamos cumplir con la realización de la aplicación.
    - Con respecto al diseño, justificamos las decisiones clave e incluimos un diagrama de clases, incluyendo sus relaciones, atributos y métodos. Asimismo, incluimos varios diagramas de secuencia para las funcionalidades de: Buscar un reporte, crear una actividad y crear un participante.
    - Por otro lado, una decisión acertada fue haber utilizado estereotipos para clasificar las clases y sus responsabilidades. Por ejemplo, la clase Reporte la identificamos como un Service Provider ya que su función era generar las estadísticas de un participante. Sin embargo, asignar estos estereotipos no fue fácil pues la mayoría de las clases podían ser clasificadas en más de uno.
    - Utilizar estereotipos facilitó la planeación del proyecto y nos dio claridad sobre la responsabilidad de cada clase y si cumple sólo una; y en caso de que no, esto no mostraba que debíamos seguir descomponiendo el proyecto. Esta decisión fue correcta pues como se vio más adelante en el curso, es importante respetar el principio de Single Responsibility.
  + ¿Qué cosas salieron mal? ¿Qué decisiones fueron problemáticas?
    - En esta primera entrega, la estructura y formulación del diagrama pudo haber sido mejor. Por un lado, las convenciones que escogimos, por ejemplo, las divisiones entre cuadrantes dificultaban la comprensión de este. Asimismo, estaba incompleto y las relaciones entre clases no eran lo suficientemente claras. Además, los nombres que escogimos eran poco significativos, por ejemplo, el nombre “Registrar” no permitía inferir que este era un Controller o en general, su rol dentro del proyecto.
  + ¿Qué tipo de problemas tuvieron durante el desarrollo de los proyectos y a qué se debieron?
    - Por un lado, el hecho de que hasta ahora estuviéramos aprendiendo Java, dificultó la implementación del diseño.
    - Asimismo, no estábamos familiarizados con los patrones de diseño y las estrategias que estos proporcionan a problemas recurrentes, por tal motivo, nuestras soluciones no fueron óptimas. Po ejemplo, decidimos que los métodos y atributos de la clase Administrar Datos fueran static, para que no existiera una instancia. No obstante, desconocíamos del patrón de diseño Singleton y la estrategia que este nos brinda para tener acceso a una única instancia de una clase.

En segundo lugar,

* Proyecto 2:
  + ¿Qué cosas salieron bien? ¿Qué decisiones resultaron acertadas
    - Una decisión acertada es que no reutilizamos un JFrame para la interfaz gráfica, sino que cada pestaña era un JFrame. Esto tiene la desventaja de que se puede ver que se cierra y se abre en cada cambio de pestaña. Sin embargo, tomamos esta decisión porque nos parece que es lo más conveniente para realizar los requerimientos de cada una.
    - Una decisión acertada fue utilizar el framework Swing que nos proporcionó todas las herramientas necesarias para implementar la interfaz gráfica, de tal forma que podamos reutilizarlo.
    - Otra decisión acertada fue haber utilizado librerías ya que nos permitieron reutilizar elementos gráficos tal como, el calendario que indicaba la fecha de fin del proyecto o de una actividad; y un menú desplegable para seleccionar una hora.
    - En general, la implementación fue coherente y consistente tanto con el modelo de clases, como con los diagramas de secuencia.
  + ¿qué cosas salieron mal? ¿qué decisiones fueron problemáticas?
    - Una decisión problemática que tomamos fue unir generar reporte y visualizar reporte. Nos dimos cuenta de que era mejor, por un lado, generar la estadística y, por otro lado, mostrarla. Durante la exposición en clase de nuestro proyecto, el profesor nos recomendó transmitir un paquete de datos, así que la clase Reporte (que antes generaba y mostraba el reporte) ahora es una instancia de la que se puede consultar distintos aspectos sobre el reporte de un participante.
    - El diseño de la interfaz gráfica en la primera entrega pudo haber sido mejor. A pesar de que nos guio en el proceso de implementación y nos sirvió para planear el trabajo, este pudo haber sido más completo y claro en las relaciones entre los componentes de la interfaz gráfica.
    - Una decisión que puede ser problemática a futuro es que utilizamos el patrón de diseño Facade con el propósito de proveer una interfaz unificada para comunicar la interfaz gráfica con el modelo del programa. No obstante, en un futuro puede traer problemas porque esta interfaz unificada se vuelve muy grande conforme aumenta el modelo del programa lo cual no respeta el principio de segregación de la interfaz porque habría muchos métodos que el cliente (la interfaz gráfica) no utiliza.
    - Algo que no supimos hacer fue la visualización de alto nivel que muestre cuándo ha habido actividades realizadas en un proyecto.
  + ¿Qué tipo de problemas tuvieron durante el desarrollo de los proyectos y a qué se debieron?
    - Un aspecto tecnológico que se nos dificulto fue subir y guardar un archivo al disco local porque teníamos que utilizar un Buffer Reader con el cual no estábamos familiarizados.
    - Otra dificultad fue que era la primera vez que hacíamos interfaces gráficas de usuario y además, tuvimos que aprender sobre SWING y todas las herramientas que este proporciona.

En tercer lugar.

* Proyecto 3:
  + ¿Qué cosas salieron bien? ¿Qué decisiones resultaron acertadas?
    - Cambiamos algunos nombres de las clases para que estos fueran más significativos.
    - En esta ocasión, teniendo en cuenta la retroalimentación que dimos sobre el proyecto 2, implementamos el patrón de diseño Singleton.
    - Una decisión que tomamos fue que los paquetes, tareas y actividades iban a ser almacenadas en su respectivo HashMap. Creemos que es una decisión acertada, teniendo en cuenta la regla de dominio de que un paquete puede contener paquetes, ya que otras estructuras de datos no son apropiadas; por ejemplo, un árbol no es apropiado porque este es binario y, como se mencionó, un paquete puede almacenar más de 2 paquetes, tareas o actividades.
  + ¿qué cosas salieron mal? ¿Qué decisiones fueron problemáticas?
    - Una dificultad en este proyecto fue que, debido a que un paquete puede almacenar paquetes, ahora necesitamos una ruta para llegar a la tarea o actividad que el cliente busque. Lo anterior fue problemático porque no teníamos certeza de cómo esto se iba a visualizar en la interfaz gráfica.
    - El hecho de que ahora hay paquetes almacenados en paquetes, dificulta generar el archivo que se guarda en el disco local.
    - Una decisión problemática en la entrega 1fue que planeamos crear una interfaz que fuera un Jframe para que todas las demás clases de la interfaz gráfica heredaran de este. Sin embargo, al implementarlo, nos encontramos con el problema de que los métodos son Static Final y esto impedía implementar la interfaz tal como lo estábamos haciendo. Así que para la entrega 2 decidimos implementarlo de tal modo de que cada clase fuera un JFrame.
  + ¿Qué tipo de problemas tuvieron durante el desarrollo de los proyectos y a qué se debieron?
    - Un problema que tuvimos durante el desarrollo del proyecto, derivado al desconocimiento de la tecnología o a la dificultad de diseñar en un entorno incierto fue que no supimos visualizar el árbol en su totalidad, sólo el primer nivel.
    - Otro problema fue que no pudimos generar reportes que muestren el avance del proyecto y el desempeño del equipo de trabajo. Si bien sí se logró el reporte de un participante en una actividad determinada.

**Instrucciones de uso:**

* En la primera ventana se ponen los datos del participante principal.
* Para crear el proyecto, se oprime el primer botón del menú y se llenan los datos requeridos.
* Para crear los paquetes y las tareas se debe oprimir el segundo botón que se llama “realizar la planeación del proyecto”. Ahí se podrán registrar paquetes tareas, y en el botón que se llama “ver carpetas” permite ver la visualización de los mismos.
* Existen todas las funcionalidades de las entregas pasadas como: Modificar la fecha final de una actividad o de un proyecto, agregar un participante, etc. Cada botón indica su propia funcionalidad.
* El reporte es el reporte de una actividad de un participante.
* Para consultar el reporte se pone el correo del participante, la fecha de la realización de la actividad y seleccionar la ubicación de la tarea (la actividad que hace parte de la tarea).