




Informe de Laboratorio FIS

Sistema de gestión de monitorías

Daniel Esteban Ladino Torres 20142020043
Neider Alejandro Fajardo Cardona 20142020025
Cristian Camilo Cuervo Castillo 20142020010



Índice general

I	Proyecto	
1	Problema	7
1.1	Introducción	7
1.2	Definición	8
1.3	Solución	9
1.4	Objetivos	10
1.4.1	Objetivo general	10
1.4.2	Objetivos específicos	10
1.5	Justificación	11
1.5.1	Justificación social	11
1.5.2	Justificación tecnológica	11
1.5.3	Justificación ambiental	11
2	Metodología	13
2.1	Introducción	13
2.2	Scrum	14
2.3	Ventajas y Desventajas	15
2.3.1	Ventajas	15
2.3.2	Desventajas	15
2.4	Cronograma	16

II**UML**

3	Análisis	19
3.1	Introducción	19
3.2	Diagrama de Casos de Uso	20
3.3	Interacciones	22
3.3.1	Diagrama de Secuencia	23
3.3.2	Diagrama de Comunicación	27
3.3.3	Diagrama de Temporización	33
3.4	Diagramas de Actividades	34
3.5	Diagramas de Workflow	35
3.6	Diagramas de Descripción de la Interacción	36
4	Diseño	37
4.1	Introducción	37
4.2	Diagrama de Clases de Análisis	38
4.3	Diagrama de Clases	41
4.4	Diagrama de Objetos	45
4.5	Diagrama de Estructura Compuesta	46
4.6	Patrones	47
4.6.1	Patrones creacionales	47
4.6.2	Patrones Estructurales	49
4.6.3	Patrones de Comportamiento	52
5	Despliegue	55
5.1	Introducción	55
5.2	Diagrama de Sistemas	56
5.3	Diagrama de Componentes	57
5.4	Diagrama de Artefactos	58
5.5	Diagrama de Nodos	59

III**Conclusiones**

6	Conlusiones	63
7	Trabajos Futuros	65
8	Anexos	67
8.1	Código	67
8.1.1	Patrón Singleton	67
8.1.2	Patrón Fachada	68
8.1.3	Patrón Componente	70
8.1.4	Patrón Proxy	72
8.1.5	Patrón Comando	75



Proyecto

1	Problema	7
1.1	Introducción	
1.2	Definición	
1.3	Solución	
1.4	Objetivos	
1.5	Justificación	
2	Metodología	13
2.1	Introducción	
2.2	Scrum	
2.3	Ventajas y Desventajas	
2.4	Cronograma	



1. Problema



Figura 1.1: Monitores

1.1 Introducción

Las monitorías en la Universidad Distrital, son un incentivo que ofrece la institución a estudiantes de todas las carreras a cambio de una serie de beneficios. Actualmente la forma en la que se controla no es la adecuada, lo cual genera una serie de trabas en la entrega de incentivos. A continuación se especificará en detalle el problema y la solución que se propone.

1.2 Definición

La universidad Distrital Francisco José de Caldas al igual que algunas otras universidades ofrece estímulos a sus estudiantes por medio de convocatorias tales como la de los asistentes académicos e investigativos, a los cuales se les denomina comúnmente como monitores. Las labores de los monitores son diversas, pero las más comunes son el apoyo al docente y el estudiante. Son los encargados de apoyar diferentes tareas dentro de la academia y deben trabajar 12 horas semanales durante el periodo académico en el que se postulan. Los beneficios obtenidos a la hora de ser monitor van desde 2 salarios mínimos legales vigentes hasta un descuento de la mitad del valor de una especialización.

El estudiante espera que dichos incentivos sean entregados lo más pronto posible apenas acabada su labor, sin embargo el proceso de entrega para estos tienen una serie de trabas por parte de la universidad, hasta tal punto que el estudiante es recompensado hasta 4 meses después de culminada su tarea. El problema generalmente radica en el incumplimiento por parte de algunos monitores en el desempeño de sus labores, ello retrasa por un buen tiempo el pago de las nóminas.

Pero no todo es culpa del estudiante, algunas veces es simplemente culpa de la planeación, el control y los docentes. Todo monitor cuenta con un horario fijo en el cual deben cumplir sus labores, el estudiante supone que cumplirá sus 12 horas semanales durante las 16 semanas de estudio, pero esto no sucede así. Durante algunas semanas dadas las circunstancias el monitor se atrasa en el cumplimiento de sus tareas, como por ejemplo cuando el docente falla sin previa notificación, esto genera un coste de oportunidad para el estudiante pues no solo pierde su valioso tiempo, sino que también deberá recuperar una inasistencia de la cual no es responsable.

Al final de la última semana de estudios es cuando se esperaría que todos los monitores hicieran entrega de sus planillas de asistencia, sin embargo es en esa semana donde el estudiante tiene que rebuscarse dentro de la academia para lograr completar sus horas, aquellos monitores responsables lo consiguen en el último día, mientras que los otros sencillamente no le prestan importancia y afectan el proceso para todos los demás. Dado estos hechos aquí mencionados se hace necesario la implementación de un control más riguroso y automatizado que dé vía libre a un procedimiento más rápido en la entrega de incentivos para los monitores.

1.3 Solución

Para lograr que el proceso de las monitorías sea más eficiente y dar solución a los problemas más comunes que se encuentran a la hora de asignar monitores, se quiere sistematizar la información de manera que sea más organizado este proceso, de igual forma se busca facilitar el acercamiento entre profesores y estudiantes, con los monitores que estén disponibles, lo cual ayuda a que estos últimos puedan completar sus horas laborales y garantiza que cumplan con su deber, para de esta forma recibir los incentivos de forma oportuna. Se quiere reducir de forma significativa la pérdida de horas por parte de los monitores debido a algún imprevisto que se presente con el profesor, como por ejemplo, que este no pueda asistir a clase y que un monitor encuentre diferentes alternativas para recuperar estas horas.

Con el fin de lograr lo anteriormente planteado, se propone desarrollar una aplicación híbrida, con la que se pueda gestionar las actividades realizadas por los monitores y tanto las horas cumplidas, como las que estén aún pendientes, así como la certificación de estas por parte de los profesores. Con la aplicación se busca dar un control a las horas de trabajo realizadas, ya que al terminar una sesión el profesor podrá certificarla por medio de la aplicación, con lo que se controla que los monitores cumplan con sus labores establecidas. En caso de que el profesor no asista a la clase o que no necesite del monitor, este podrá buscar un profesor que si requiera uno, esto se hará a través de un módulo que contiene a los monitores libres en ese momento, los cuales podrán ser vistos por los profesores quienes tendrán la opción de contactarlos y con esto evitar que el monitor pierda tiempo, igualmente ayudar para que complete las horas en las que estaba atrasado o que adelante tiempo de trabajo, siempre que un profesor certifique su labor.

Uno de los objetivos es organizar la información, de forma que se pueda eliminar el papeleo que se emplea en este proceso y que sea más fácil acceder a ella, por medio de la aplicación se podrá manejar toda la información referente a los profesores, los horarios, los monitores y las horas laboradas por estos. Toda la información será registrada en una base de datos y las personas autorizadas podrán consultarla para comprobar que los procesos se estén realizando con normalidad, los estudiantes tendrán la opción de llevar un control de su tiempo de forma más sencilla y práctica.

1.4 Objetivos

1.4.1 Objetivo general

Desarrollar un software de plataforma web, en el cual se realizara la automatización del sistema de monitorias con el fin de apoyar la organización y control en la Universidad Distrital Francisco José de Caldas.

1.4.2 Objetivos específicos

- Realizar un diseño inicial de la aplicación mediante técnicas y modelos lógicos de tal manera que se pueda realizar un esbozo de la aplicación de manera coherente y eficiente.
- Diseñar una interfaz accesible para administradores mediante la cual se pueda llevar un permanente control de los estudiantes que pertenecen al programa de monitorias de la universidad.
- Mejorar la oportunidad de completar y verificar las horas que los estudiantes tienen que cumplir durante el semestre y con ello constatar su desempeño durante el semestre.

1.5 Justificación

El presente proyecto de software aquí expuesto buscará dar solución a los retrasos de las nóminas de los monitores de la Universidad Distrital, por medio de una automatización parcial del proceso logrando así beneficiar a cientos de estudiantes que ejecutan sus labores como asistentes académicos e investigativos.

1.5.1 Justificación social

Llevar a cabo una aceleración del proceso de las monitorías beneficiaría más que todo a la comunidad académica, pues permitiría que los pagos se efectuaran casi inmediatamente luego de finalizada las correspondientes labores de tal forma que los estudiantes puedan invertir su dinero en nuevos proyectos sin tener que esperar largas trabas del proceso.

1.5.2 Justificación tecnológica

Un proyecto de esta índole no se había llevado a cabo dentro de la universidad, lo cual hace que el producto sea innovador dentro de este aspecto. Por otro lado la automatización del proceso y la digitalización de la documentación requerida para la certificación de las labores, permitirá ahorrar tiempo y recursos.

1.5.3 Justificación ambiental

El hecho de digitalizar algunos componentes del proceso como las planillas de asistencia, documentos del estudiante, horarios y entre otros, no significan un gran impacto en beneficio al medio ambiente. Sin embargo a largo plazo, al usar menos hojas de papel y tinta se contribuye al cuidado del planeta y la generación de menos residuos.

2. Metodología

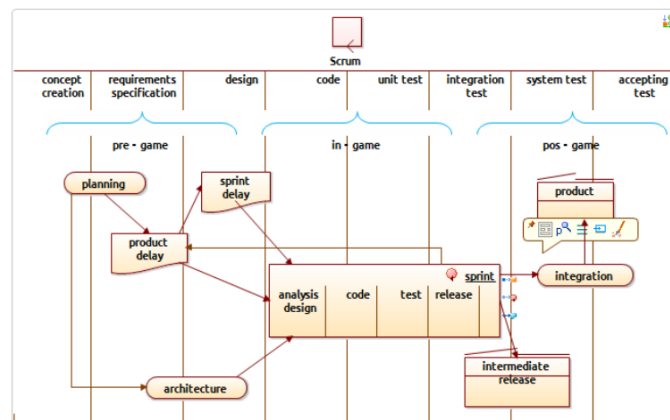


Figura 2.1: Metodología Scrum

2.1 Introducción

Durante mucho tiempo se buscaron procesos que se pudieran aplicar en casi cualquier proyecto de desarrollo, los cuales garantizaran la calidad del software y optimizara los tiempos de trabajo, para de esta forma garantizar un producto final de calidad entregado en el plazo de tiempo establecido. El no contar con un proceso de desarrollo puede verse reflejado en el tiempo de producción, ya que este puede ser mucho mayor que el esperado, esto implica igualmente un aumento en los costos. Las metodologías de desarrollo ágiles surgen debido a la necesidad de generar entregas en el menor tiempo posible, teniendo como prioridad el contacto con el cliente, y la satisfacción de este, para adaptarse de forma rápida al cambio de los requerimientos, del mercado y a las tecnologías nuevas que se desarrollan, sin dejar de lado la calidad del software en cada entrega.

2.2 Scrum

El scrum es un marco de referencia para el desarrollo de software que va de la mano con los principios del manifiesto ágil, el cual busca servir como guía para los procesos de establecer los requerimientos, análisis, diseño, avances y entregas. Este método fue desarrollado por Jeff Sutherland y su equipo de desarrollo en el año 1990. En años más recientes Schwaber y Beedle han desarrollado más el método Scrum.[6] En esta metodología el trabajo es dividido en sprints, que por lo general duran dos semanas, pero puede variar dependiendo del tiempo disponible, durante cada sprint se realizan reuniones diarias de por lo general 15 minutos de duración, los cuales sirven para actualizar los avances, informar de los obstáculos que se han presentado y planear lo que se hará hasta la próxima reunión. Esto funciona como estrategia para garantizar que los procesos planteados para el sprint se cumplan.

En cada Sprint se incluyen los requerimientos más importantes, los que se van a desarrollar en esa iteración, los cuales están ordenados en una pila de trabajo, en donde las prioridades se encuentran en la parte superior de la pila. Se discute un plan para la realización de los requerimientos, listando las tareas necesarias para estos. Al final de cada sprint se espera que las funcionalidades que entraron estén terminadas y listas para ser presentadas al cliente, se hace también una retrospectiva para evaluar la forma de trabajar del equipo, todos los aspectos a mejorar para la próxima iteración.

En la metodología Scrum se cuenta con distintos actores, los cuales tienen un papel importante en el desarrollo del producto:

- Dueño del producto: Esta persona define los requerimientos del producto y es el único que dado el caso puede cambiarlos.
- Equipo de trabajo: Son grupos multidisciplinarios y auto-organizados, que trabajan en el desarrollo del producto.
- Scrum master o Facilitador: Es una persona externa, preferiblemente, a los grupos de trabajo, la cual dirige los encuentros diarios, ayuda a dar solución a los diferentes problemas que se puedan presentar y evalúa el progreso que tiene cada persona con el fin de conseguir los objetivos de cada iteración.

2.3 Ventajas y Desventajas

2.3.1 Ventajas

- A diferencia de las metodologías tradicionales, el scrum nos permite un desarrollo del proyecto dinámico y ágil, es decir podemos estar enviando entregables al cliente y al mismo tiempo, si es necesario, modificarlo de acuerdo a nuevas especificaciones o necesidades.
- Permite una mayor eficiencia por parte de cada persona perteneciente al proyecto, esto debido a que cada quien sabe cual es su tarea, es decir requiere de un gran orden para ser aplicado.
- Da un rol mas incluyente al cliente y a los desarrolladores del software, esto debido a que es más frecuente las pruebas y entregables del software para el perfeccionamiento del mismo.
- Permite una planeación detallada de los objetivos a alcanzar, por lo que los imprevistos son menos probables a la hora de la codificación.

2.3.2 Desventajas

- El hecho de que se le de un rol tan importante a cada miembro del proyecto, puede traer el inconveniente de que alguno de ellos por distintas causas, no pueda cumplir con su tarea, esto implicaría el retraso en el desarrollo o la sobrecarga de trabajo de otros miembros.
- Muchas veces se puede presentar que no se cumpla con las fechas estipuladas para los entregables, o simplemente que a la hora de presentarlos no se haya logrado un avance significativo.
- Es necesario excesivas reuniones entre equipo desarrollador y cliente, todo para la verificación y seguimiento del proyecto, ante esto se pueden presentar problemas de disponibilidad.
- Un error en Pila de Trabajo significaría la carencia de alguna parte esencial del producto, o por el contrario un desperdicio de tiempo y recursos.

2.4 Cronograma

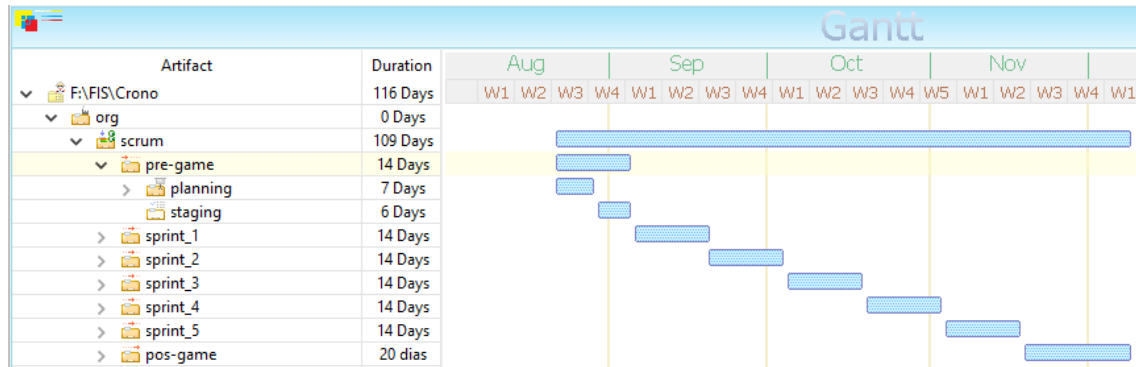
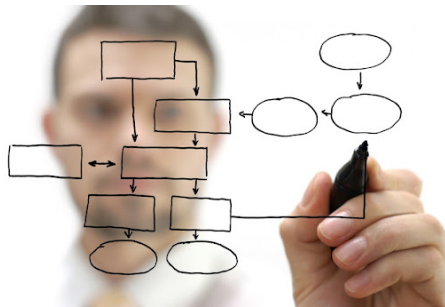


Figura 2.2: Cronograma de la metodología de scrum

3	Análisis	19
3.1	Introducción	
3.2	Diagrama de Casos de Uso	
3.3	Interacciones	
3.4	Diagramas de Actividades	
3.5	Diagramas de Workflow	
3.6	Diagramas de Descripción de la Interacción	
4	Diseño	37
4.1	Introducción	
4.2	Diagrama de Clases de Análisis	
4.3	Diagrama de Clases	
4.4	Diagrama de Objetos	
4.5	Diagrama de Estructura Compuesta	
4.6	Patrones	
5	Despliegue	55
5.1	Introducción	
5.2	Diagrama de Sistemas	
5.3	Diagrama de Componentes	
5.4	Diagrama de Artefactos	
5.5	Diagrama de Nodos	



3. Análisis



3.1 Introducción

El análisis de los requerimientos puede ser uno de los procesos más complejos que se presentan a la hora de desarrollar un programa de software, esto se debe a que muchas veces no basta con entender el problema y la posible solución para este, sino que igualmente es necesario comprender las necesidades del cliente, saber qué es lo que quiere, pero muchas veces es el mismo cliente el que no sabe esto último, lo cual complica bastante el proceso.

Realizar un buen análisis del proyecto nos garantiza una base sólida para la elaboración de nuestro producto, de ahí que sea una de las partes más importante, quizá la más importante, a la hora de realizar un proyecto, ya que es nuestro punto de partida, la forma que tenemos para guiarnos durante el proceso de desarrollo, los planos de nuestro proyecto, por lo que sin el análisis inicial lo más probable es que se este destinado al fracaso.

3.2 Diagrama de Casos de Uso

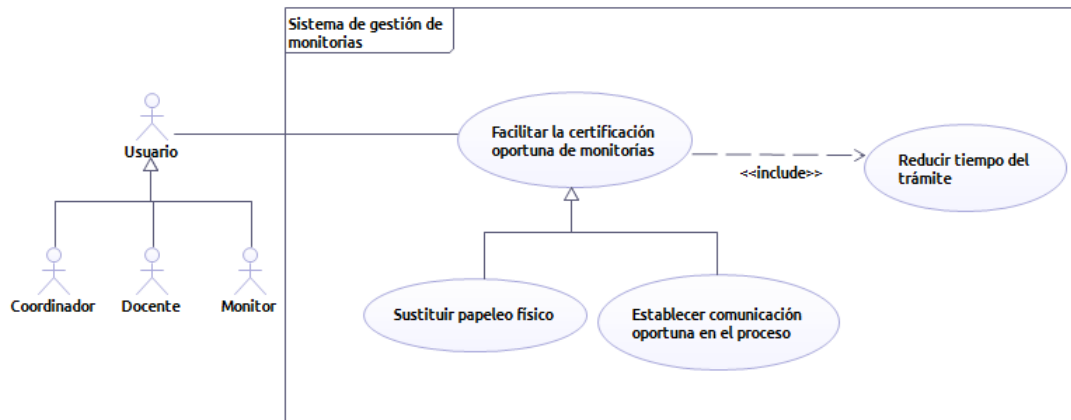


Figura 3.1: Caso de uso del sistema de gestión de monitorías

Nombre:	Sustituir papeleo físico	Id:	CU01
Actores:	Monitor, Profesor		
Objetivo:	Agilizar la certificación de la monitoria		
Escenarios			
Primario:	Digitalizar toda la documentación requerida en el proceso		
Secundario:	-Documentación incompleta -Horas incompletas -Objetivos incompletos		
Excepcionales:	-Tamaño virtual de los soportes muy grande -Formato de los soportes incompatible -Falta de comunicación -Inconsistencia en los datos		

Cuadro 3.1: Especificaciones de caso de uso 01

Nombre:	Establecer una comunicación oportuna en el proceso.	Id:	CU02
Actores:	Monitor, Profesor, Coordinador		
Objetivo:	Contribuir a la interacción entre los actores del sistema		
Escenarios			
Primario:	Visualizar la información de contacto de los actores		
Secundario:	-Inexistencia de datos registrados -Información de contacto incompleta -Inconsistencia de los datos de contacto		
Excepcionales:	-Fallo en la persistencia de los datos -Falla de comunicación		

Cuadro 3.2: Especificaciones de caso de uso 02

Nombre:	Reducir tiempo de tramite	Id:	CU03
Actores:	Monitor, Profesor, Coordinador		
Objetivo:	Contribuir a la rápida certificación de monitores		
Escenarios			
Primario:	Generar informe de certificación dentro de fechas preestablecidas		
Secundario:	-Vencimiento de plazo para la certificación		
Excepcionales:	-Falta de oportunidades para completar las horas/ objetivos		
	-Retraso en el cumplimiento de las tareas.		
	-Fallas en el calendario del servidor del sistema		

Cuadro 3.3: Especificaciones de caso de uso 03

3.3 Interacciones

Todos los sistemas incluyen interacciones de algún tipo. Éstas son interacciones del usuario, que implican entradas y salidas del usuario; interacciones entre el sistema a desarrollar y otros sistemas; o interacciones entre los componentes del sistema. El modelado de interacción del usuario es importante, pues ayuda a identificar los requerimientos del usuario. El modelado de la interacción sistema a sistema destaca los problemas de comunicación que se lleguen a presentar. El modelado de interacción de componentes ayuda a entender si es probable que una estructura de un sistema propuesto obtenga el rendimiento y la confiabilidad requeridos por el sistema.[4]

Los diagramas de interacción de más frecuente uso son los siguientes:

- Diagrama de Secuencia

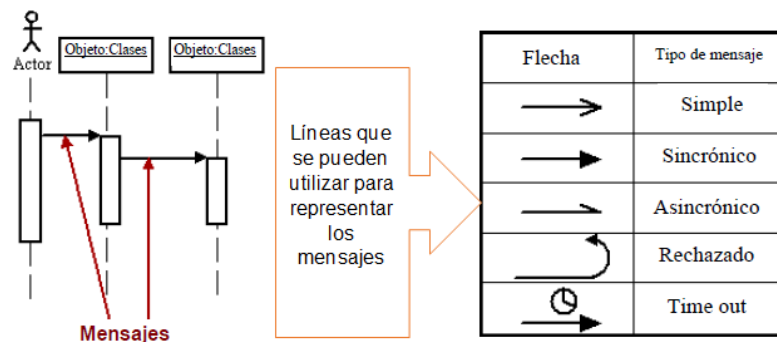


Figura 3.2: Ejemplo de diagrama de secuencia

- Diagrama de Comunicación

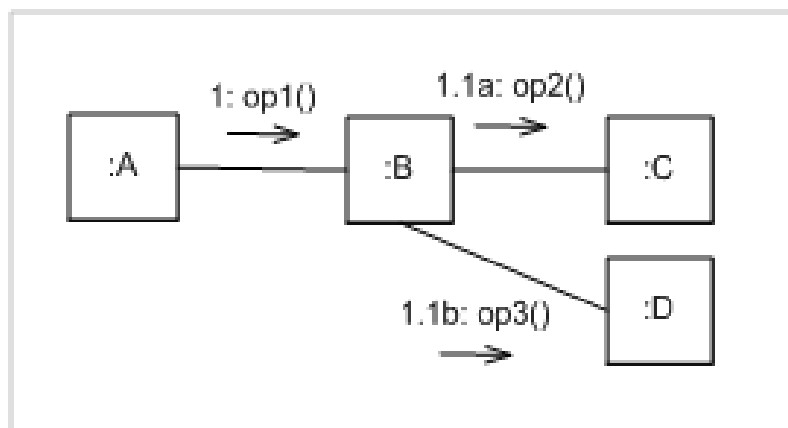


Figura 3.3: Ejemplo de diagrama de comunicaciones

3.3.1 Diagrama de Secuencia

El diagrama de secuencia es un tipo de diagrama usado para modelar interacción entre objetos en un sistema. Este tipo de diagrama muestra la interacción de un conjunto de objetos en una aplicación a través del tiempo y se modela para cada caso de uso. Mientras que el diagrama de casos de uso permite el modelado de una vista del negocio para el escenario, el diagrama de secuencia contiene detalles de implementación del escenario, incluyendo los objetos y clases que se usan para implementar el escenario y mensajes intercambiados entre los objetos.

A continuación se muestran algunos de los escenarios más importantes de las interacciones del sistema:

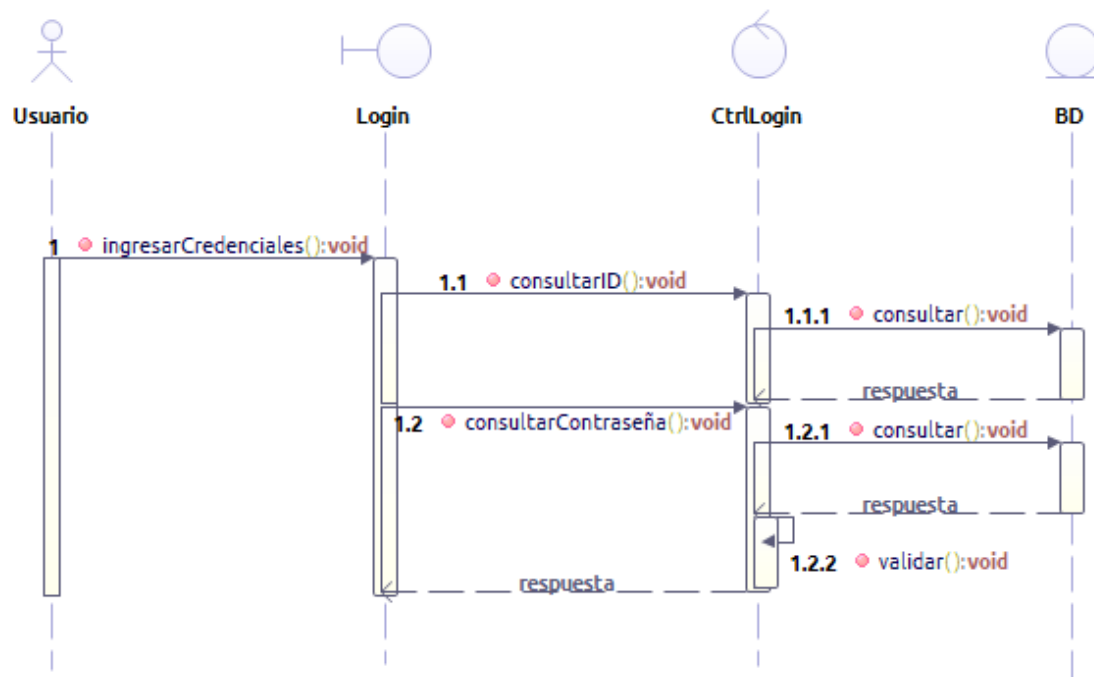


Figura 3.4: Diagrama de secuencia para la autenticación de un usuario en el sistema

En el anterior diagrama de secuencia sucede lo primero que debe hacer un usuario antes de hacer uso de las funcionalidades del sistema: autenticarse como un usuario, este usuario es único y tiene distintas funcionalidades dependiendo de su rol.

Por ejemplo en el siguiente diagrama de secuencia se muestra la funcionalidad del escenario primario del caso de uso 03 apreciado en el cuadro 3.3, donde el coordinador se encarga de generar los certificados de cada monitor.

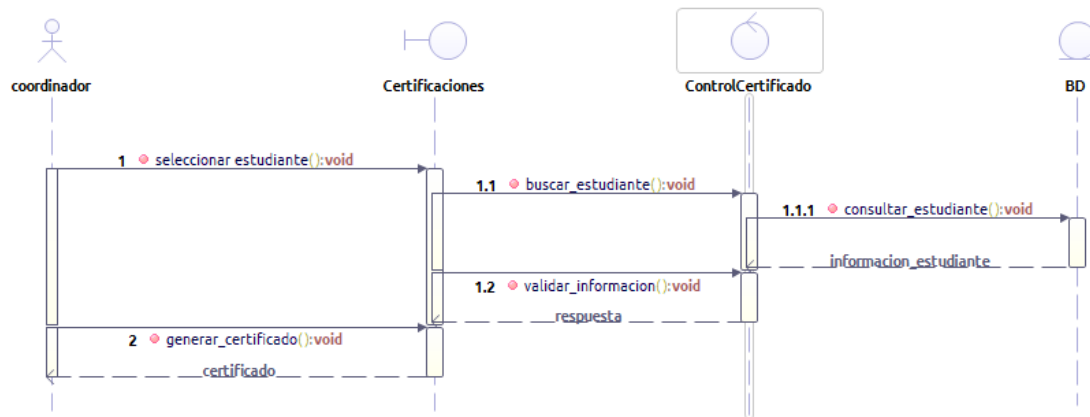


Figura 3.5: Diagrama de secuencia para la certificación de las monitorías

Por otro lado una funcionalidad clave del sistema es permitir el contacto oportuno entre los usuarios del sistema, observado en el caso de uso 02 del cuadro 3.2. En este diagrama de secuencia se generaliza primordialmente para todos los usuarios, pues todos tienen la posibilidad de visualizar bien sea el correo o el teléfono de los involucrados en el proceso.

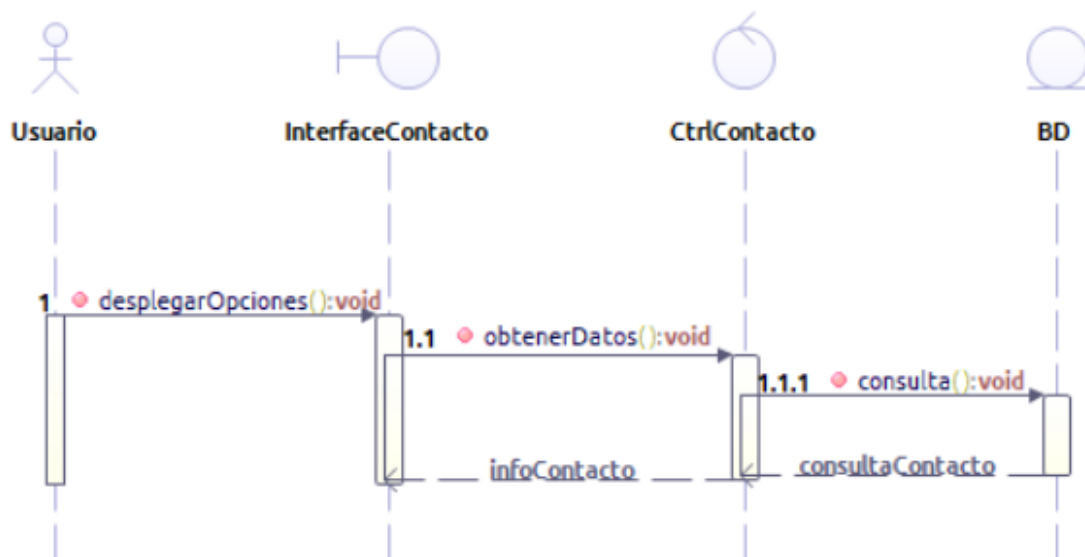


Figura 3.6: Diagrama de secuencia para el contacto entre usuarios del sistema

Ahora, existirán situaciones en las que un docente necesita de ayuda extra dentro de la academia, y un estudiante necesita horas para poder certificarse, es en este escenario donde el docente puede publicar un clasificado en el sistema a la espera de que algún estudiante lo contacte, a continuación se visualiza esta situación en un diagrama de secuencia:

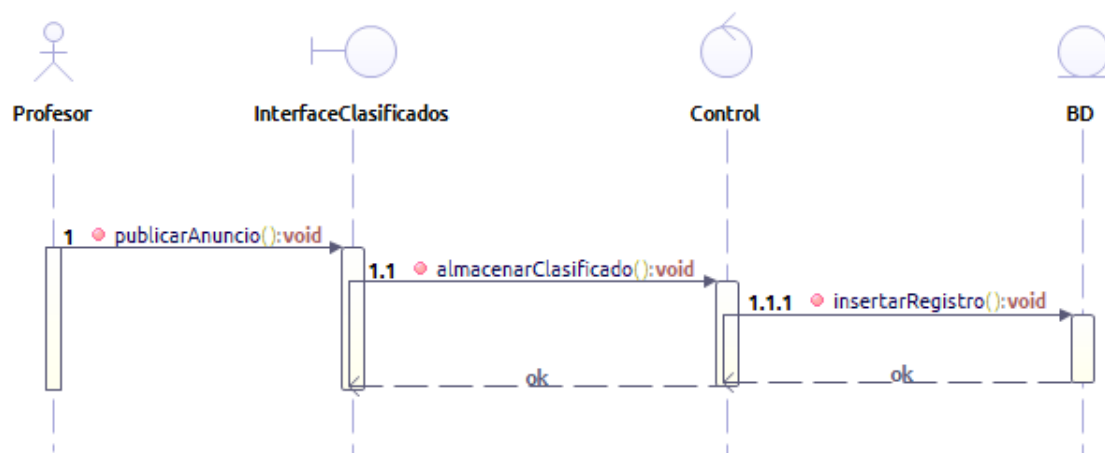


Figura 3.7: Diagrama de secuencia para la publicación de clasificados por parte del profesor

En el siguiente diagrama de secuencia puede que describamos uno de los procesos más fundamentales de nuestro aplicativo, el hecho de permitirle a el docente poder llevar un registro de las horas de los monitores a su cargo es algo esencial, por ello nuestro actor principal en esta abstracción sera el docente, aquel que tiene acceso a una interfaz gráfica que le luego le permitirá seleccionar a un estudiante, por medio de una petición la interfaz accede al control para que este posteriormente, pueda acceder a la base de datos de los estudiantes y pueda regresar información sobre las horas de monitoria de un estudiante, al obtener esta información en un proceso inmediatamente continuo, se desplegara en la interfaz esta información, para que el docente pueda modificarla, es decir para que el docente tenga la capacidad de llenar las horas que le hacen falta al estudiante. De este modo nos aseguramos de una total transparencia en el proceso, ademas de el hecho de ahorrarnos un montón de papeleo y firmas innecesarias:

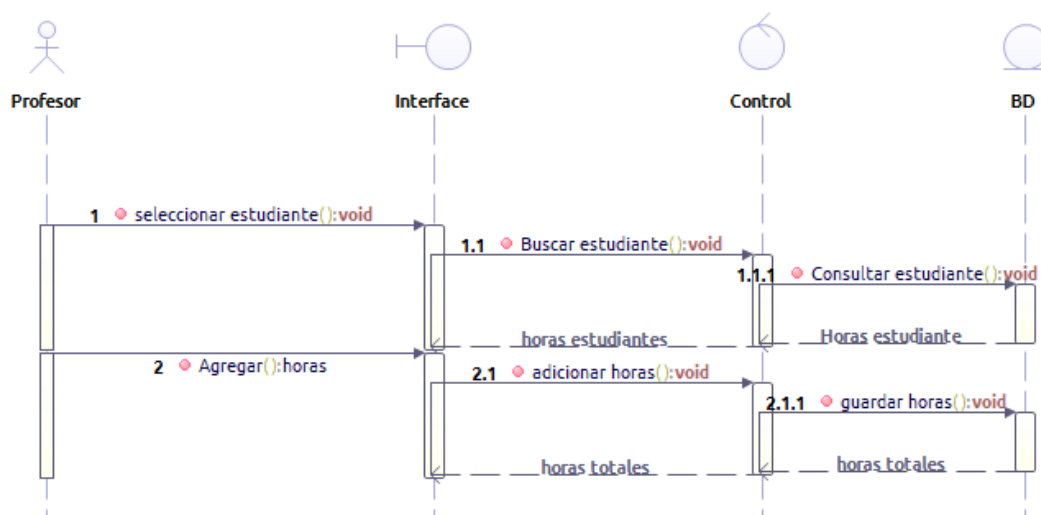


Figura 3.8: Diagrama de secuencia para la certificación de horas de monitoría

El diagrama de digitalización de los documentos nos describe una automatización y visualización virtual, completa de los documentos que se tienen que presentar en los procesos de monitorias, para ello haremos uso de una base de datos que lleve registro de cada uno de los estudiantes implicados en las monitorias, como es obvio nuestro actor principal sera el estudiante, para ser exactos

los monitores, ellos podrán acceder a la plataforma mediante sus cuentas, en donde encontraran una interfaz que les permitirá llenar su información principal y donde podrán subir todo documento requerido, para ello deberán subir una imagen de dichos requisitos, la cual sera llevada a un control que procederá a guardar la en la base de datos, si no hay errores externos, la plataforma le verificara al estudiante el estado de su información:

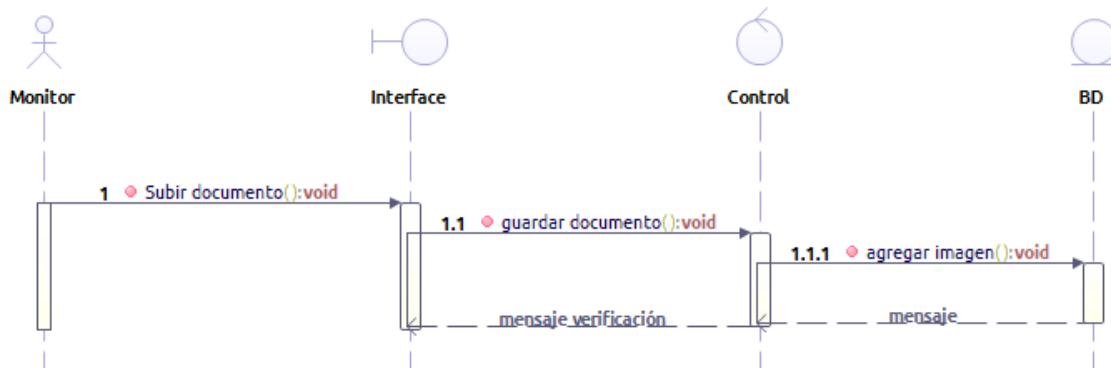


Figura 3.9: Diagrama de secuencia para la digitalización de documentos

3.3.2 Diagrama de Comunicación

Un diagrama de comunicación nos permite modelar las distintas interacciones y acciones entre objetos de nuestro proyecto, normalmente son combinaciones de información de otro tipo de diagramas tales como el de clases o el de secuencia que previamente hemos realizado, este nos ayuda a describir el comportamiento de nuestro sistema por medio de mensajes, por ello es normal que el nombre de los objetos del diagrama de comunicación sean los mismo que los del diagrama de secuencia, estos mensajes al igual que en el anterior diagrama están organizados por importancia a la hora de ser realizados, no podemos acceder a una base de datos si previamente no nos identificamos con el sistema.

En el siguiente diagrama de comunicación podemos ver el proceso del sistema a la hora de ingresar al sistema, pero podemos observar un cambio en el control al acceso de la base de datos, como podemos ver en este objeto tenemos un mensaje que esta redireccionando a si mismo, es decir que este tiene la capacidad de validar antes de devolver la información a la interfaz, esto es esencial en aplicativo debido a que si no pensáramos en este tipo de control, en un nivel alto de nuestra aplicación cualquiera podría acceder a los datos y dañar nuestra información.

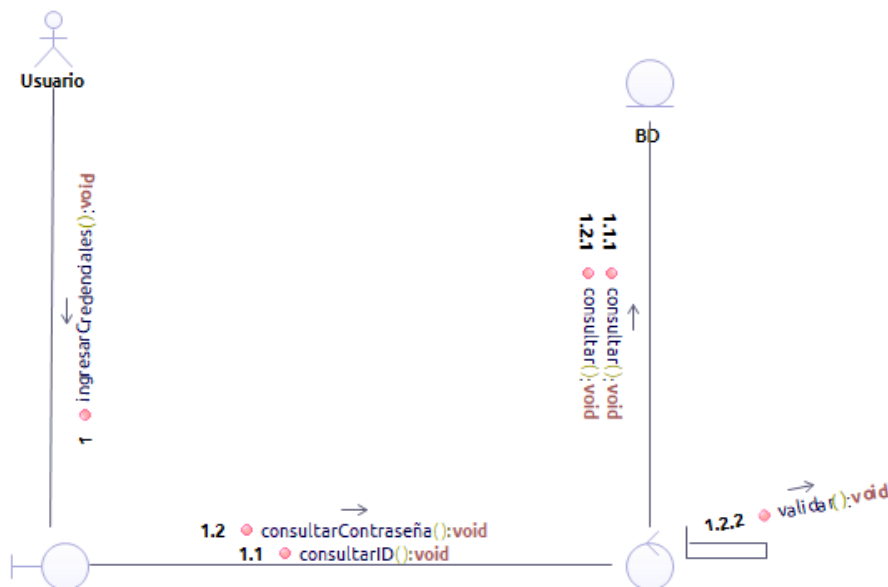


Figura 3.10: Diagrama de comunicación para el diagrama de secuencia de la figura 3.2

Cada doble mensaje que observamos nos permite visualizar una acción que toma nuestro sistema, aun y sin embargo este diagrama no nos permite visualizar los mensajes de retorno, esto se debe a que estos diagramas de comunicación son un complemento de nuestro proceso de abstracción, en este caso nos retornaría la información del estudiante.

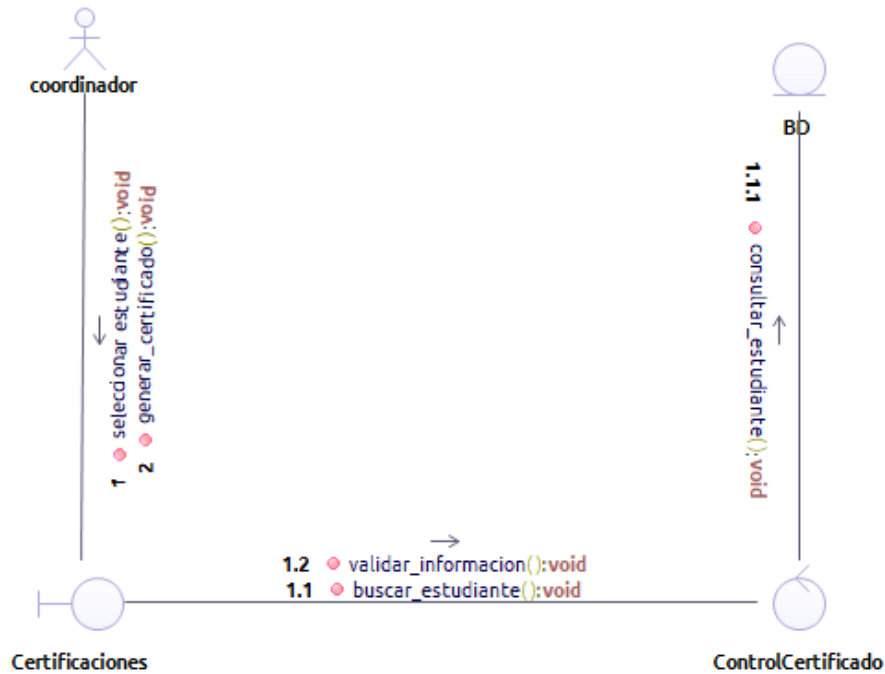


Figura 3.11: Diagrama de comunicación para el diagrama de secuencia de la figura 3.3

El siguiente diagrama de comunicación refleja la secuencia de mensajes realizada por el programa para permitir al usuario obtener la información necesaria, en este caso la información de contacto de otro usuario, que puede ser profesor o monitor y con lo que se garantizaría la oportuna comunicación entre los diferentes usuarios de la aplicación.

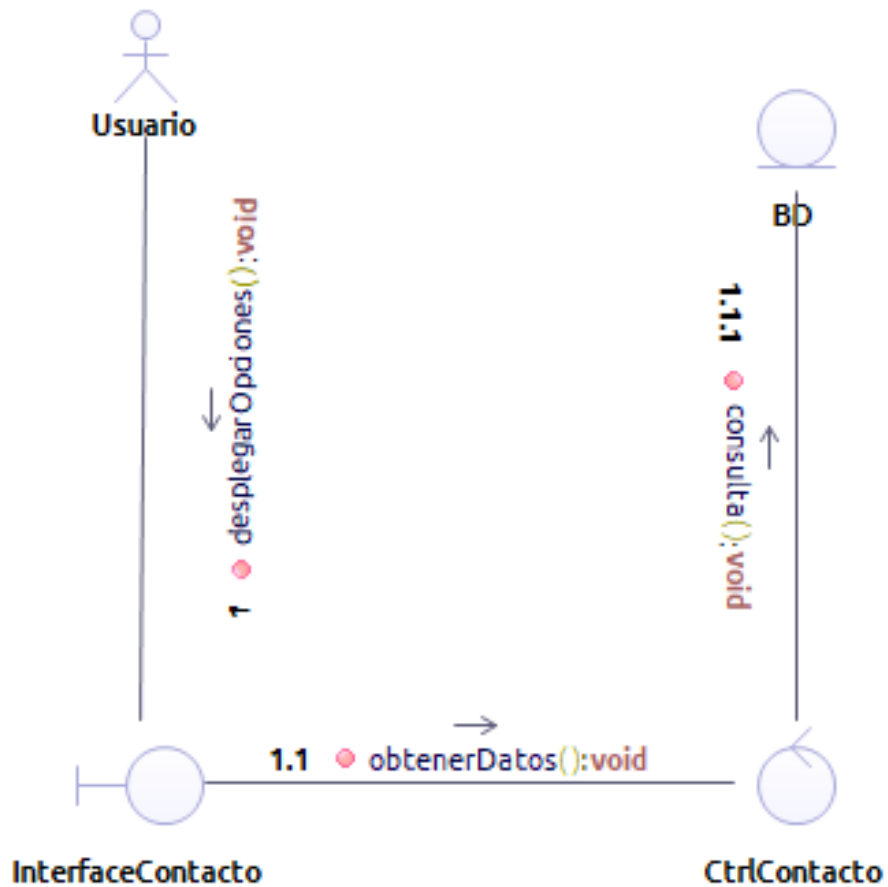


Figura 3.12: Diagrama de comunicación para el diagrama de secuencia de la figura 3.6

Del mismo modo la siguiente figura ilustrar la forma en la que el programa permite a un docente buscar un monitor disponible, por medio de un clasificado que será guardado y posteriormente mostrado a los demás usuarios, específicamente a los monitores que estén disponibles para completar sus horas de monitoria.

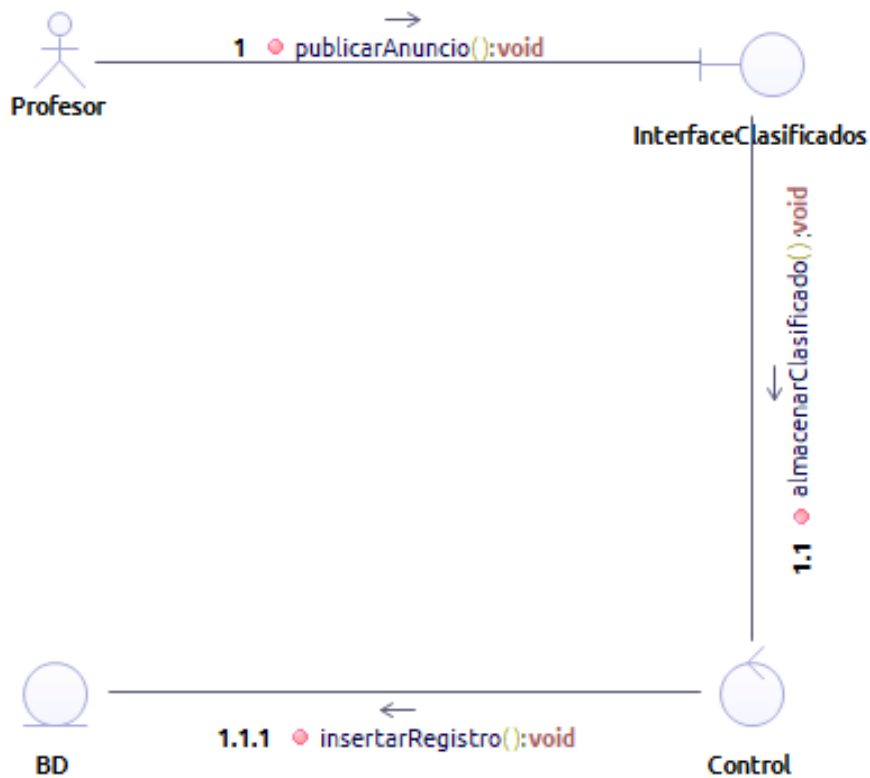


Figura 3.13: Diagrama de comunicación para el diagrama de secuencia de la figura 3.7

El siguiente diagrama de comunicación nos da a conocer las direcciones de los mensajes entre los distintos objetos, desde el profesor y seleccionar a un estudiante y agregarle las horas correspondientes por su labor de monitorías, lo cual lo realiza mediante un objeto interface, este nos permite representar la parte grafica de nuestra app, a su vez conectado por un objeto control, este se encarga de transmitir la información capturada en el objeto que interactúa con el actor principal, haciendo las veces de puente entre una interface y una persistencia, este referencia la información a nuestro objeto base de datos (BD) este realiza la función de consulta y modificación de la base, mediante los mensajes de consultar datos del estudiante y guardar las horas.

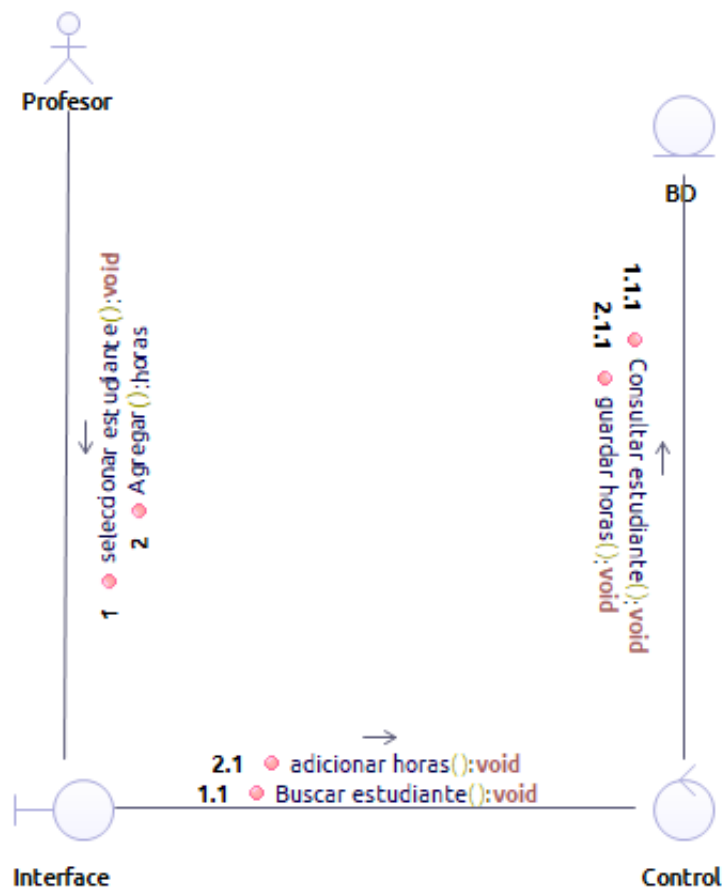


Figura 3.14: Diagrama de comunicación para el diagrama de secuencia de la figura 3.6

Este diagrama de comunicación es el que nos permite la virtualización en gran parte del papeleo, para ello el actor principal en este caso un estudiante, al llenar su información personal, junto con algunos archivos específicos e imágenes, interactúa con un objeto Interface, aquel le permite al usuario digitar y capturar sus datos para evitar todo trámite presencial y molesto, mediante el mensaje subir documento, se planea capturar toda información de un estudiante para luego ser llevada a un control que se encargue mediante otro enlace de comunicación de decirle al objeto BD, que es el que interactúa directamente con nuestra base de datos, que actualice cierta información, dado cierto punto del proceso, el actor solo podrá actualizar su información, ya que la modificación de sus datos personales solo será supervisada por un actor externo de mayor jerarquía, esto con el fin de evitar información corrupta

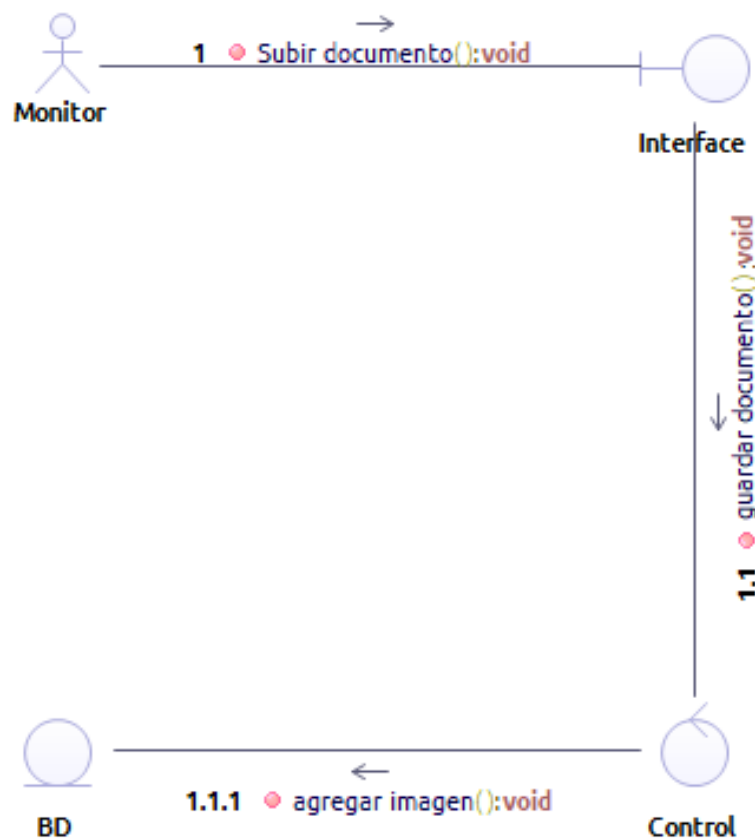


Figura 3.15: Diagrama de comunicación para el diagrama de secuencia de la figura 3.7

3.3.3 Diagrama de Temporización

3.4 Diagramas de Actividades

3.5 Diagramas de Workflow

3.6 Diagramas de Descripción de la Interacción



4. Diseño



4.1 Introducción

El diseño del software, constituye una de las bases más importantes a la hora de crear nuevos sistemas. Hoy en día la complejidad creciente del software hace que el riesgo de construir sistemas que no alcancen los objetivos sea muy alto. Para evitar ese riesgo conviene construirlo a partir de una buena fase de planificación mediante diagramas UML que permitan cumplir las expectativas tanto del desarrollador como las del cliente. A continuación se presentarán algunos de los diagramas más importantes relacionados con el diseño del sistema de gestión de monitorías.

4.2 Diagrama de Clases de Análisis

Los diagramas de clases de análisis facilitan la realización de los casos de uso del sistema y sirve como una simplificación del modelo de diseño. Entre los 3 elementos que se visualizan en estos diagramas se encuentran la Interfaz, qué es una clase estereotipada la cual describe la realización de los casos de uso del sistema. El otro elemento es el controlador, donde se plasma la lógica del negocio y el último elemento es la entidad, donde se encuentra la persistencia del sistema.

A continuación se muestra el diagrama de clases de análisis para la parte de la autenticación del usuario.

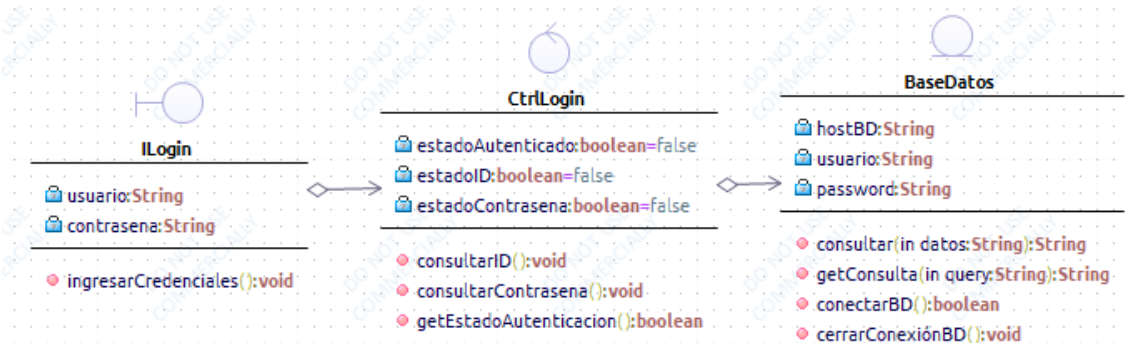


Figura 4.1: Diagrama de Clases de análisis para el Login de la plataforma.

Por otro lado se tiene para la certificación de monitorías el siguiente diagrama:

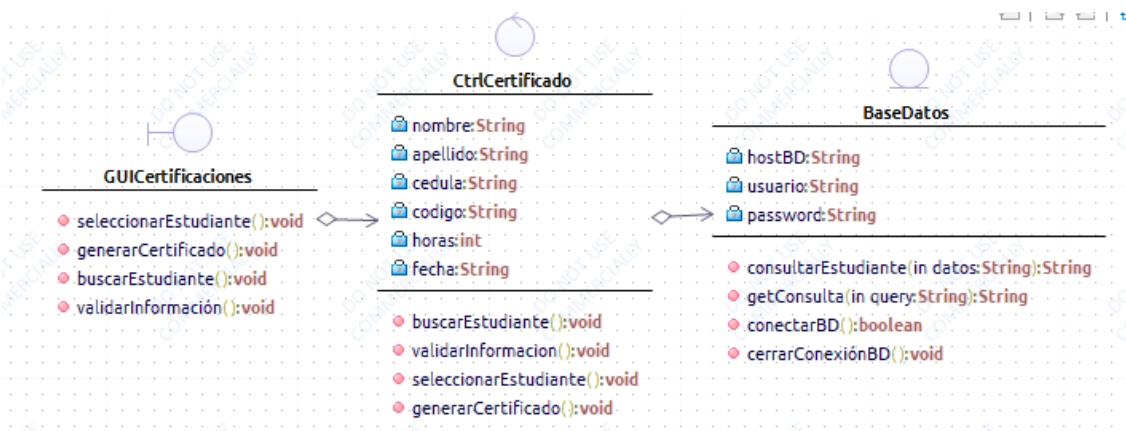


Figura 4.2: Diagrama de Clases de análisis para la certificación de monitoría.

La siguiente figura representa el diagrama de clases de análisis de el contacto entre monitor y profesor

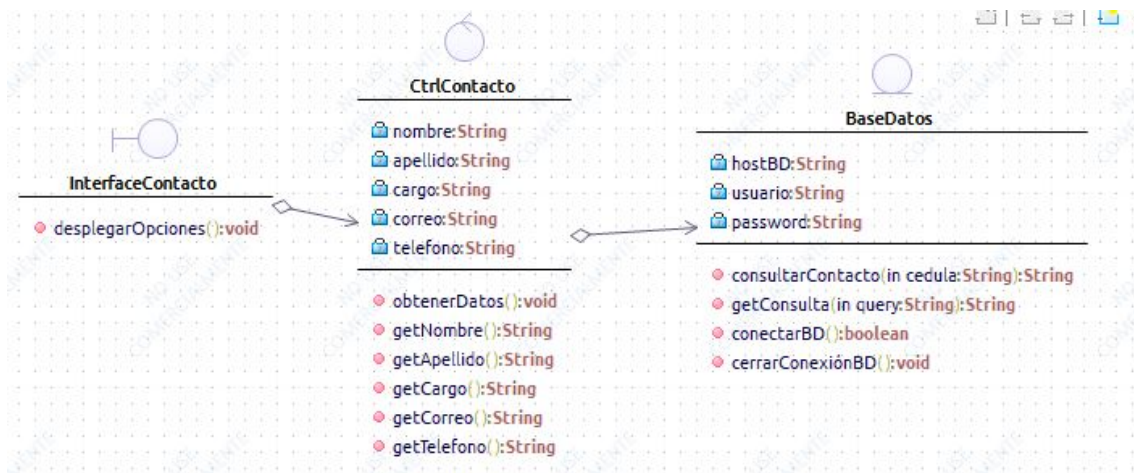


Figura 4.3: Diagrama de Clases de análisis para el contacto Docente-Monitor.

En el próximo diagrama de clases de análisis se puede observar las clases que interfieren en el proceso de publicar clasificados.

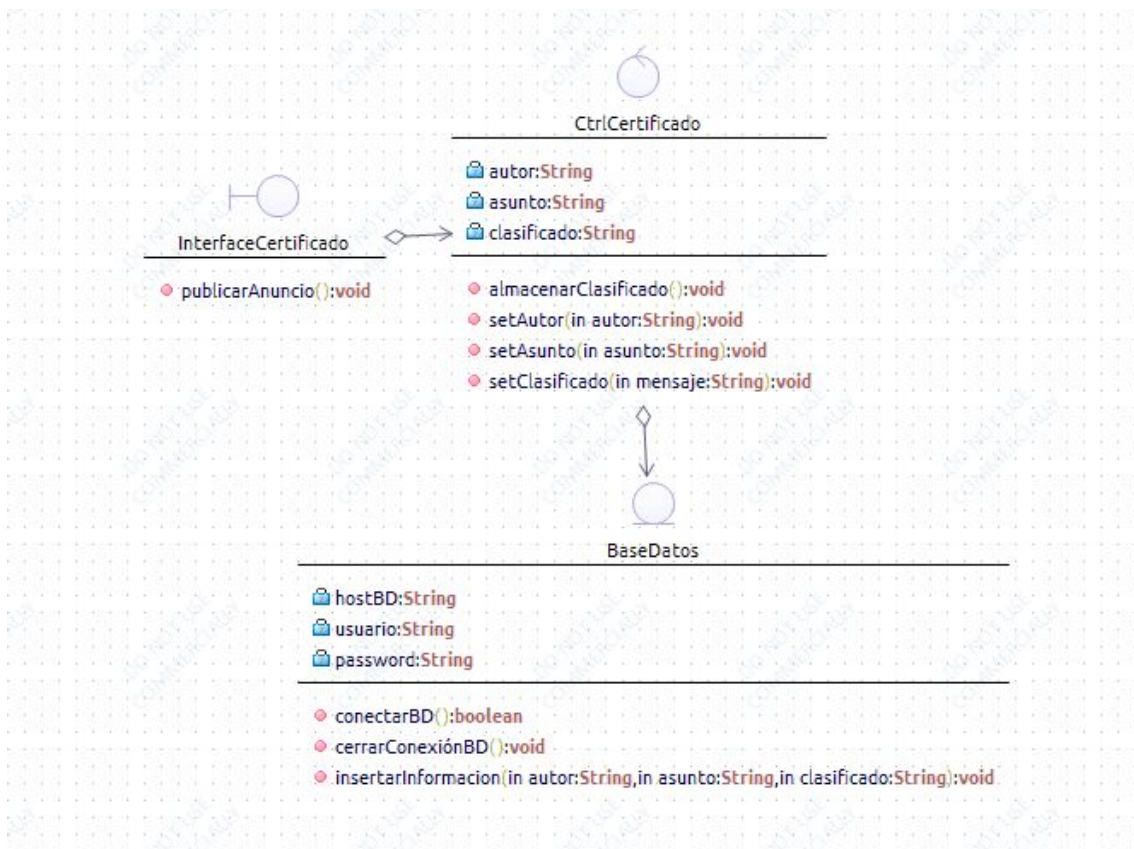


Figura 4.4: Diagrama de Clases de análisis para el proceso de publicar clasificados .

Por último se tiene el diagrama de clases de análisis de la certificación de las horas de las monitorías.

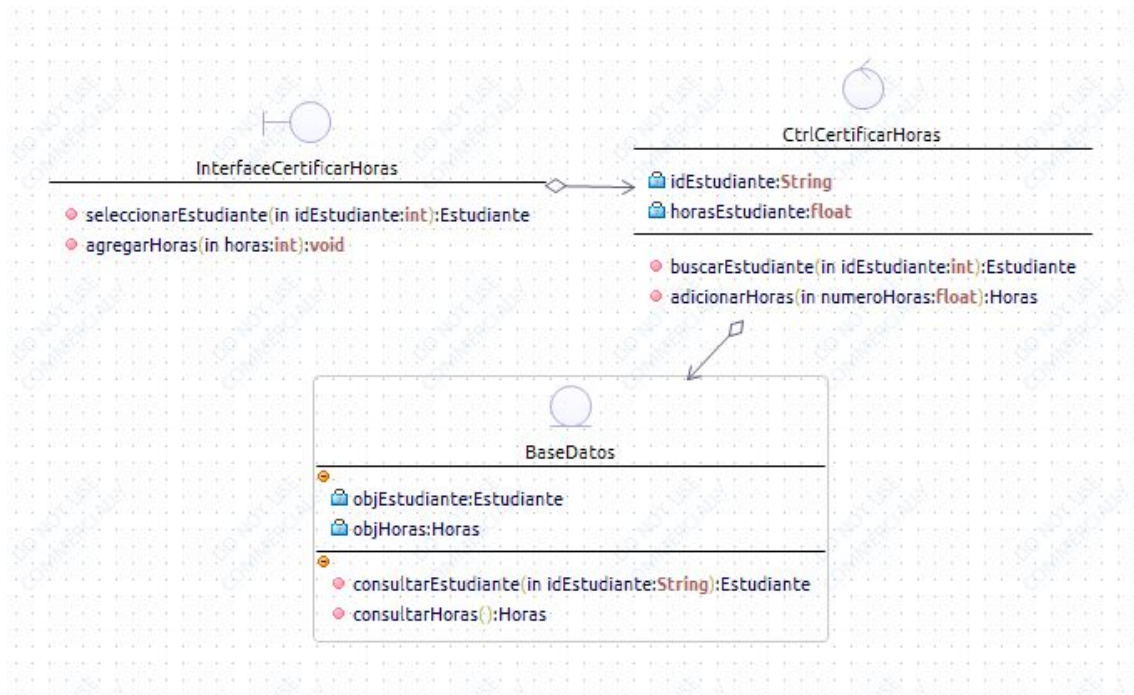


Figura 4.5: Diagrama de Clases de análisis para el proceso de certificación de horas.

4.3 Diagrama de Clases

Un diagrama de clases describe los tipos de objetos que hay en el sistema y las diversas clases de relaciones estáticas que existen entre ellos. Es importante identificar las relaciones entre las clases. El nombre de la asociación debe dejar claro que una entidad utiliza a otra clase como parte de sus atributos o características. Se debe tener en cuenta que entre dos clases puede existir más de una relación.[3]

De acuerdo a los diagramas de secuencia y objetos, se realizó una construcción de los diagramas de clase de acuerdo con la utilidad que tiene coloso, a partir de las clases generadas se procedió a editar cada una de estas clases, agregando una serie de atributos y métodos que en un principio se consideran necesario.

El primer diagrama consiste en las clases que administran la forma de ingresar a la plataforma:

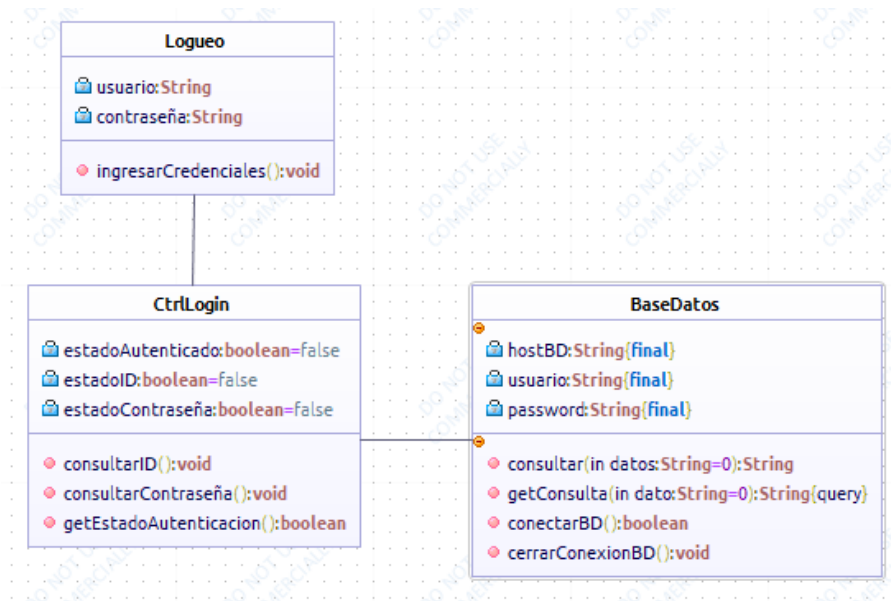


Figura 4.6: Diagrama de Clases para el Login de la plataforma.

Ahora, en la siguiente imagen se muestra el diagrama de clases para la generación de certificados, en este diagrama la clase control depende de la clase de base de datos para ejecutar su lógica y ésta a su vez usa de una interfaz gráfica donde el usuario interactúa con el aplicativo y este visualiza los datos entregados.

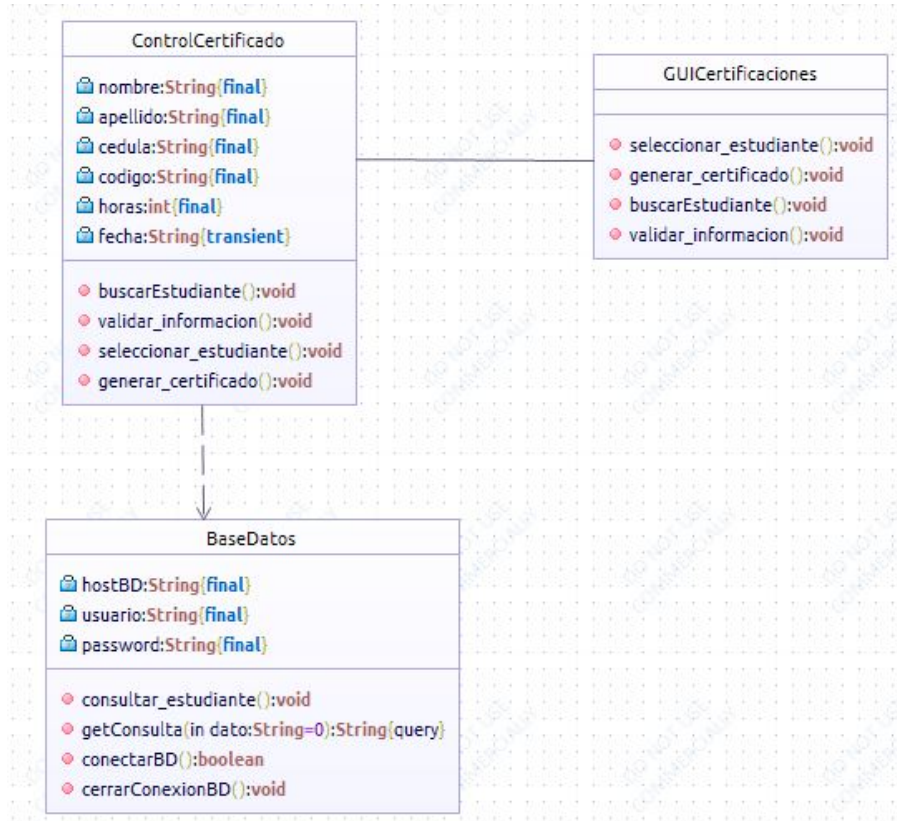


Figura 4.7: Diagrama de Clases para la generación de certificados monitoria.

Manera similar sucede con el diagrama de clases para la visualización de contactos, cuya finalidad es obtener unicamente los datos de contacto de los usuarios.

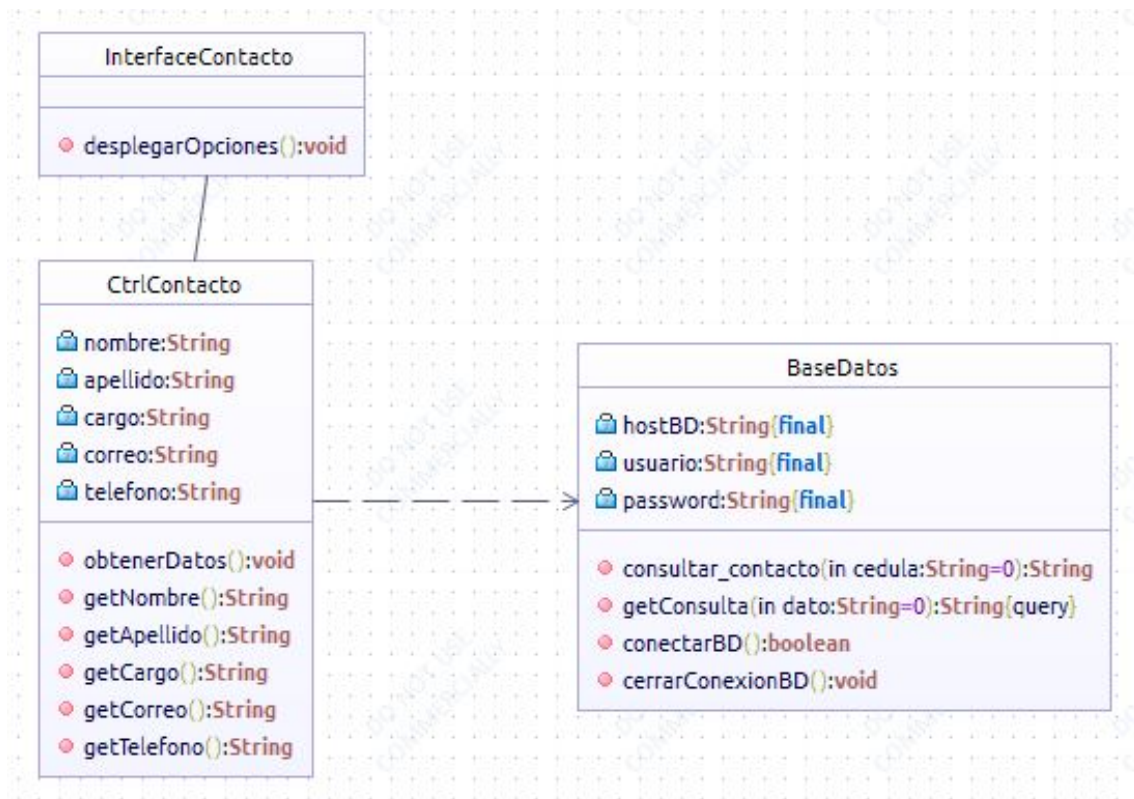


Figura 4.8: Diagrama de clases para la visualización de contactos.

Uno de los beneficios que ofrece la plataforma de gestión de monitorías es el uso de clasificados por parte de los profesores para completar las horas requeridas por el monitor, en base al diagrama de secuencia de la figura 3.7 se propone el siguiente diagrama de clases:

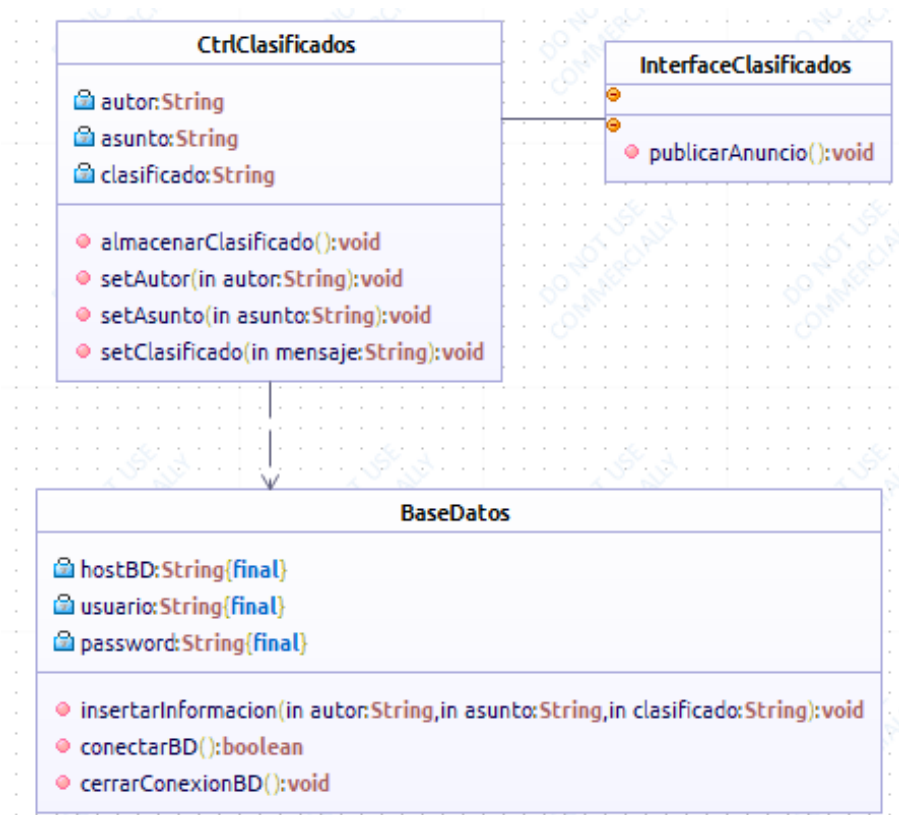


Figura 4.9: Diagrama de clases para la publicación de clasificados

La certificación de las horas de monitoria es uno de los procesos imprscindibles en el desarrollo del proyecto, puesto que con este se busca aportar al cumplimiento de varios requerimientos, como los de eliminar el papeleo y facilitar el proceso de certificación, por lo que es importante hacer una correcta espificación de este.

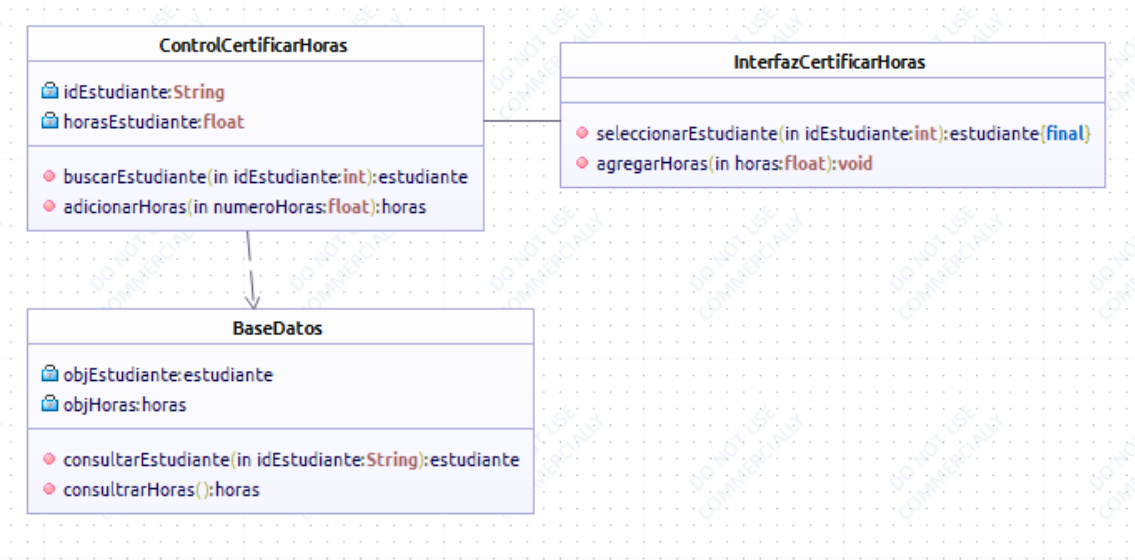


Figura 4.10: Diagrama de clases para la certificación de horas de monitoría

4.4 Diagrama de Objetos

4.5 Diagrama de Estructura Compuesta

4.6 Patrones

Según Gamma en su libro Patrones de Diseño, define a un patrón de diseño como una descripción de clases y objetos comunicándose entre sí adaptada para resolver un problema de diseño general en un contexto particular. Con los patrones de diseño es posible saber cual fue la intencionalidad con la que se desarrolló una aplicación y cual será a futuro la intención con la que se desarrolla en el presente. El patrón de diseño no es sólo una plantilla para reusar; es todo un idioma de diseño que estandariza la forma de diseñar aplicaciones, garantizando su fácil evolución, prueba y mantenimiento[2].

Existen 3 tipos de patrones:

- Creacionales:
- Estructurales
- De comportamiento

4.6.1 Patrones creacionales

Los patrones creacionales proporcionan ayuda a la hora de crear instancias de objetos. El objetivo de este tipo de patrones es el de abstraer el proceso de instanciación y ocultar los detalles de cómo los objetos son creados o inicializados [2]. Dentro de los patrones creacionales se pueden encontrar los siguientes:

1. Método Fábrica
2. Fábrica Abstracta
3. Prototype
4. Constructor
5. Singleton

Para el caso concreto de nuestro proyecto, se hará uso del patrón creacional singletón. Por lo que sólo se hará énfasis profundo en éste patrón.

Singleton

El patrón Singleton, también conocido como instancia única, nos permite tener un control en número de instancias de un objeto, en este caso, como su nombre lo indica, define la creación de una única instancia.

Estructura

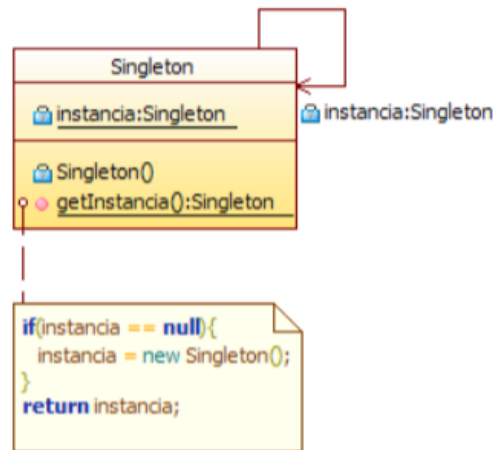


Figura 4.11: Estructura del patrón Singleton.

Caso de estudio

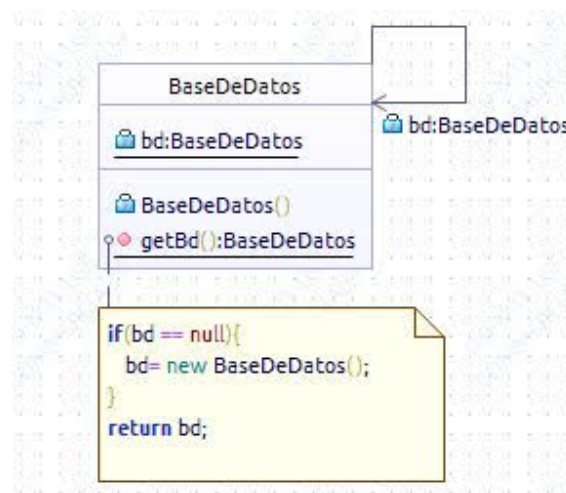


Figura 4.12: Singleton del sistema de gestión tutoríaas.

Dentro del diseño del aplicativo es necesario garantizar que la base de datos de la que se hace uso sea la misma en todo el aplicativo y que no se creen multiples instancias de esta, por lo que se hará uso del patrón de diseño Singleton para garantizar una única instancia de la base de datos y que esta sea utilizada en todo el aplicativo. Podrá observar un ejemplo de su implementación en el anexo 8.1.1.

4.6.2 Patrones Estructurales

Los patrones estructurales se enfocan en como las clases y objetos se componen para formar estructuras mayores, los patrones estructurales describen como las estructuras compuestas por clases crecen para crear nuevas funcionalidades de manera de agregar a la estructura flexibilidad y que la misma pueda cambiar en tiempo de ejecución lo cual es imposible con una composición de clases estáticas[1]. Dentro de los patrones estructurales se pueden encontrar los siguientes:

1. Puente
2. Adaptador
3. Componente
4. Decorador
5. Fachada
6. Peso Ligero
7. Proxy

Para el caso concreto de nuestro proyecto, se hará uso de los patrones estructurales Componente, Fachada y Proxy. Por lo que sólo se hará énfasis profundo en estos patrones.

Patrón Fachada

El patrón fachada provee una interface unificada para un conjunto de interfaces en un subsistema, la fachada es una interface de alto nivel que facilita el uso de un subsistema. Su uso principal consiste en la posibilidad de crear un sistema de niveles[2].

Estructura



Figura 4.13: Estructura del patrón fachada.

Caso de estudio

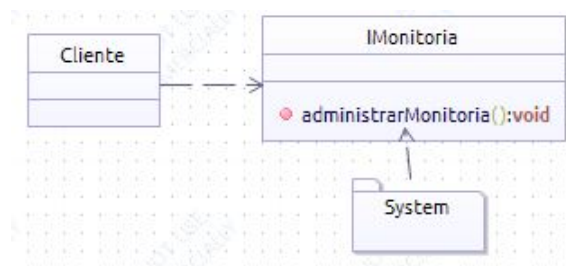


Figura 4.14: Fachada del sistema de gestión de tutorías.

Para el diseño del aplicativo de gestión de monitorías se utilizará una fachada que reúna las implementaciones más fundamentales con las que el usuario interactuará. Desacoplando la implementación del cliente con todo el subsistema y simplificando su uso. Puede observar un ejemplo de su implementación en el Anexo 8.1.2.

Patrón Componente

El patrón componente simplifica el uso de varios objetos similares organizándolos en una estructura tipo árbol que representa el todo-parte[2], lo cual simplifica la creación de algoritmos y objetos complejos haciendo uso de la recursión al tratar al todo igual que a una parte.

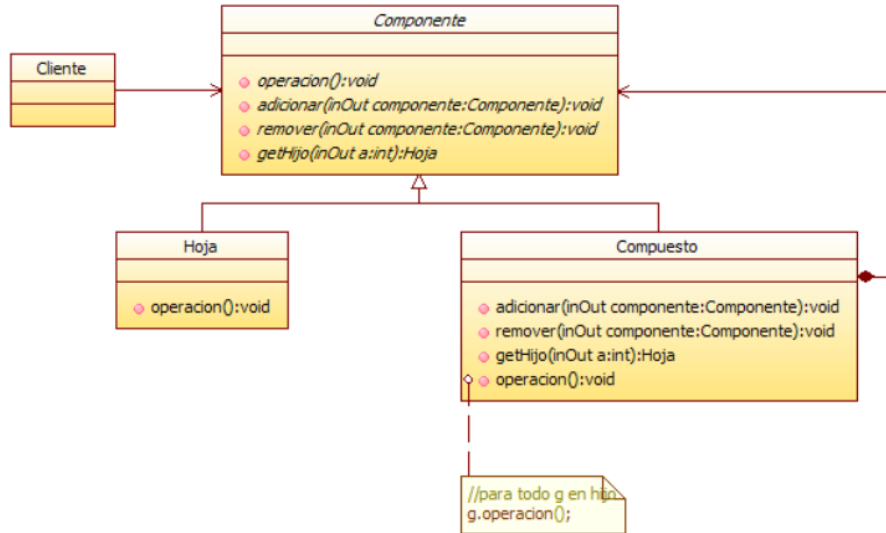


Figura 4.15: Estructura del patrón componente.

Caso de estudio

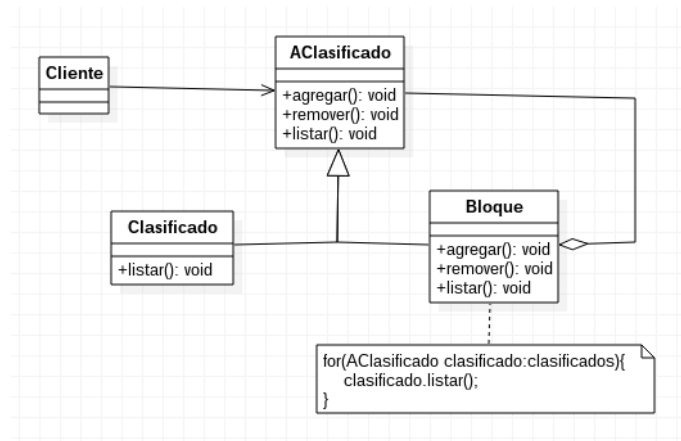


Figura 4.16: Patrón componente del sistema de gestión de tutorías..

Dentro de la aplicación de monitorías se creará una zona de clasificados en donde los monitores podrán observar las publicaciones hechas por los profesores que requieran de sus servicios. Para el manejo de estos clasificados se ha decidido hacer uso del patrón componente, el cual permite agrupar los clasificados en una estructura en forma de árbol, lo que permite hacer uso de la recursión para facilitar la forma en la que se van a visualizar los clasificados. Podrá observar un ejemplo de su implementación en el anexo 8.1.3.

Patrón Proxy

Provee un sustituto o marcador para otro objeto que controla el acceso a él. Controla el acceso a un objeto o lo representa localmente o por su alta demanda.[2].

Estructura

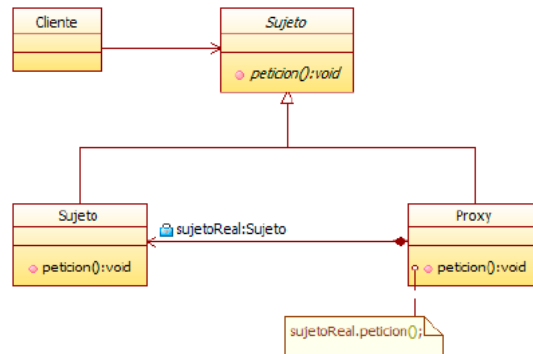


Figura 4.17: Estructura del patrón proxy.

Caso de estudio

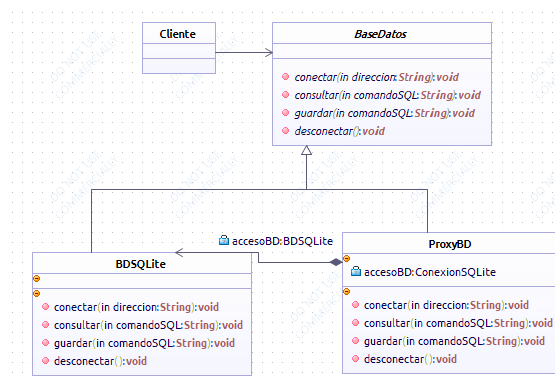


Figura 4.18: Implementación del patrón proxy.

El patrón Proxy actúa en el diseño del sistema de gestión de monitorías como intermediario para poder realizar operaciones en la base de datos SQLite, el proxy recibe la solicitud de alguna operación en la base de datos, y si es posible hacerla se delegan las funciones a la clase BDSQLite, caso contrario no se ejecuta ninguna acción lo cual reduce el uso de recursos. Podrá observar un ejemplo de su implementación en el anexo 8.1.4.

4.6.3 Patrones de Comportamiento

Los patrones de comportamiento explican la forma en la que los objetos interactúan entre sí. Describe como los diferentes objetos y clases intercambian mensajes entre ellos para hacer que las cosas funcionen y describir como los pasos de una tarea en específico es dividida a lo largo de distintos objetos. A diferencia de los patrones creacionales que solo describen el momento en que se crean los objetos y los patrones estructurales que describen una estructura más o menos estática, los patrones de comportamiento describen un proceso o flujo de información[5]. Dentro de los patrones de comportamiento se pueden encontrar los siguientes:

1. Cadena de Responsabilidad
2. Comando
3. Interprete
4. Iterador
5. Mediador
6. Momento
7. Observador
8. Estado
9. Estrategia
10. Método Plantilla
11. Visitador

Para el caso concreto de nuestro proyecto, se hará uso del patrón de comportamiento comando. Por lo que sólo se hará énfasis profundo en éste patrón.

Patrón Comando

El patrón comando nos permite encapsular una petición en un objeto, es decir que nos permite parametrizar a los clientes con diferentes peticiones, esto significa que podemos hacer peticiones a objetos de nuestra aplicación sin ser especificados, este es un método importante, debido a que como desarrolladores muchas veces no tenemos modo de conocer al receptor de la petición, ni de saber que operaciones se efectuaran, en términos conceptuales este patrón nos permite asegurar el encapsulamiento y por ende la seguridad de nuestra información

Estructura

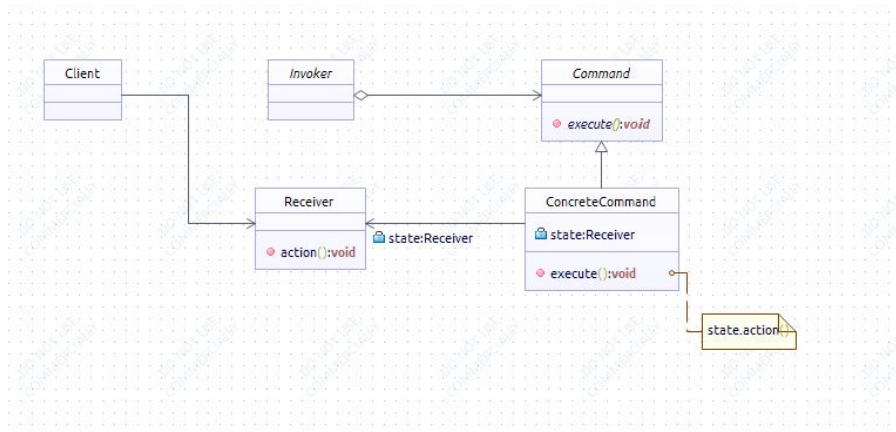


Figura 4.19: Estructura general del patrón comando.

Caso de estudio

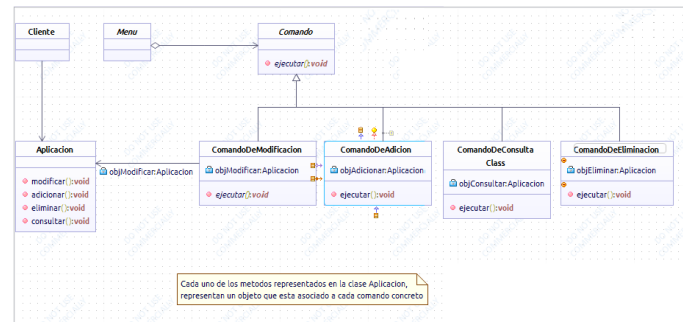


Figura 4.20: Comando del sistema de gestión de tutorías.

La clave de este patrón es la clase abstracta Comando, esta declara una interfaz para ejecutar las líneas de operaciones, en este caso, esta clase tiene consigo una operación, también abstracta la cual se le suele llamar Ejecutar, con ello las subclases de orden, especifican un receptor-acción (en nuestro caso el receptor sería nuestra aplicación, por el otro lado la acción sería el comando) guardando el receptor como una variable de instancia e implementado, mientras que Ejecutar se usa para invocar la petición. Por ende, el receptor debe poseer el conocimiento necesario para llevar a cabo la petición. Una de las características y principales, y quizá la que nos da una conveniente ventaja es la de tener la posibilidad de deshacer las operaciones realizadas, también y por obvias razones buscamos independizar el momento de la petición del momento de la ejecución.

Como ya habíamos mencionado, al tener nuestra clase comando, debemos tener la especificación, es decir la clase concreta de nuestro comando, con ello nos permitimos separar tareas, y es esta clase la que conocerá la procedencia de la petición. La clase menu será nuestra interfaz gráfica, es decir un menú con opciones desplegadas sobre ciertas tareas a cumplir. Y por otro lado la clase app será la que implemente la funcionalidad real, es decir nuestra aplicación en sí, esta será nuestra clase receptor. Podrá observar un ejemplo de su implementación en el anexo 8.1.5.

A close-up photograph of an architectural floor plan. A yellow pencil with a pink eraser and a wooden ruler are placed diagonally across the drawing. The drawing shows various rooms and corridors, with labels such as 'CONFERENCE', 'CORRIDOR', 'MEN', 'WOMEN', and 'EXIST. WOMEN'. Dimensions and room numbers like '131-711', '126', '123', '128', and '181-711' are visible. The text 'MADE IN U.S.A.' is printed on the ruler.

5. Despliegue

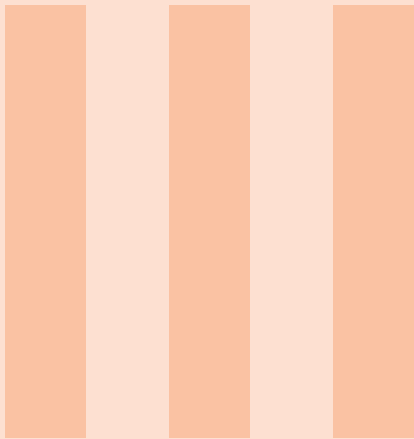
5.1 Introducción

5.2 Diagrama de Sistemas

5.3 Diagrama de Componentes

5.4 Diagrama de Artefactos

5.5 Diagrama de Nodos



Conclusiones

6	Conlusiones	63
7	Trabajos Futuros	65
8	Anexos	67
8.1	Código	



6. Conlusiones



7. Trabajos Futuros



8. Anexos

8.1 Código

8.1.1 Patrón Singleton

```
1 package PatronProxy;
2
3 public abstract class BaseDatos {
4     public abstract void conectar(String direccion);
5     public abstract void consultar(String comandoSQL);
6     public abstract void guardar(String comandoSQL);
7     public abstract void desconectar();
8
9     public static BaseDatos getBaseDatos();
10
11
12
```

Figura 8.1: Clase abstracta BaseDatos, parte del patrón proxy, con singleton implementado

```
@Override
public static BaseDatos getBaseDatos() {
    if(bd == null) {
        bd = new BFSQLite();
    }
    return bd;
}
```

Figura 8.2: Clase BDSQLite, parte del patrón proxy, con singleton implementado

```
@Override
public static BaseDatos getBaseDatos() {
    if(proxy == null) {
        proxy = new ProxyBD();
    }
    return proxy;
}
```

Figura 8.3: Clase ProxyBD, parte del patrón proxy, con singleton implementado

8.1.2 Patrón Fachada

```
1 package patronFachada;
2
3 public interface IMonitoria {
4
5     void administrarMonitoria();
6
7 }
8
```

Figura 8.4: Interfaz monitoría (fachada)


```

public class HorasMonitoria implements IMonitoria{
    int opcionElegida;
    BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
    @Override
    public void administrarMonitoria() {

        System.out.println("Horas de Monitoria"
            + "¿Que desea hacer?\n"
            + "1)Consultar horas\n"
            + "2)Registrar horas\n");

        try {
            opcionElegida=Integer.parseInt(in.readLine());
        } catch (NumberFormatException e) {

            e.printStackTrace();
        } catch (IOException e) {

            e.printStackTrace();
        }

        switch(opcionElegida) {
        case 1:
            consultarHoras();
            break;
        case 2:
            registrarHoras();
            break;
        default:
            System.out.println("Ingrese una opcion valida");
        }

    }
    public void consultarHoras() {
        System.out.println("Usted esta siendo redireccionado"
            + " al módulo de consulta de horas");

    }
    public void registrarHoras() {
        System.out.println("Usted esta siendo redireccionado"
            + " al módulo de registro de horas");

    }

}

```

Figura 8.5: Ejemplo de implementación para la fachada

```

public class Cliente {

    public static void main(String[] args) throws NumberFormatException, IOException {
        int opcionElegida;
        Logueo login =new Logueo();

        BufferedReader in = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Bienvenido al sistema de gestión de monitorías\n"
            + "¿Que desea hacer?\n"
            + "1)Administrar horas\n"
            + "2)Ir a clasificados\n"
            + "3)Ver perfil\n"
            + "4)Consultar horario\n"
            + "5)Cerrar sesion");
        opcionElegida=Integer.parseInt(in.readLine());
        switch(opcionElegida) {
            case 1:
                System.out.println("Redirigiendo al subsistema HorasMonitoria");
                HorasMonitoria horas =new HorasMonitoria();
                horas.administrarMonitoria();
                break;
            case 2:
                System.out.println("Redirigiendo al subsistema Clasificados");
                Clasificados clasificado =new Clasificados();
                clasificado.administrarMonitoria();
                break;
            case 3:
                System.out.println("Redirigiendo al subsistema usuario");
                Usuario user =new Usuario();
                user.administrarMonitoria();
                break;
            case 4:
                System.out.println("Redirigiendo al subsistema horarios");
                Horarios horario =new Horarios();
                horario.administrarMonitoria();
            case 5:
                System.out.println("Saliendo del sistema...");
                login.iniciarSesion();
        }
    }
}

```

Figura 8.6: Cliente ejemplo fachada

8.1.3 Patrón Componente

```

1
2 public interface AClasificado{
3
4     public String listar();
5
6 }
7

```

Figura 8.7: Clase AClasificado, contiene interfaz que implementan los clasificados

```
import java.util.ArrayList;

public class Bloque implements AClasificado{

    private ArrayList<AClasificado> clasificados = new ArrayList<AClasificado>();

    @Override
    public String listar() {
        String texto="";
        for(int i=0; i<clasificados.size();i++) {
            texto = clasificados.get(i).listar();
        }
        return texto;
    }
    public void agregar(AClasificado clasificado) {
        clasificados.add(clasificado);
    }
    public void remover(AClasificado clasificado) {
        clasificados.remove(clasificado);
    }
}
```

Figura 8.8: Clase Bloque, contiene los objetos tipo clasificados

```
public class Clasificado implements AClasificado{

    private String autor;
    private String info;
    private String mensaje;

    public Clasificado(String a, String i, String m) {
        this.autor = a;
        this.info = i;
        this.mensaje = m;
    }

    public String visualizar() {
        String texto="";
        texto += "Autor: "+autor;
        texto += "Información: "+info;
        texto += mensaje;
        return texto;
    }
    @Override
    public String listar() {
        return visualizar();
    }
}
```

Figura 8.9: Clase Clasificado

8.1.4 Patrón Proxy

```
package PatronProxy;

public class Cliente {
    public static void main(String[] arguments){
        BaseDatos bd= new ProxyBD();
        bd.conectar("F:\\FIS\\Proyecto FIS\\BaseDatos.sqlite");
        bd.guardar("INSERT INTO Clasificados\r\n" +
            "VALUES (cla01, HorasMonitoria, Solicito estudiante..., 21/10/2017);");
        bd.conectar("SELECT * FROM Clasificados;");
    }
}
```

Figura 8.10: Cliente con un ejemplo de implementación

```
package PatronProxy;

public abstract class BaseDatos {
    public abstract void conectar(String direccion);
    public abstract void consultar(String comandoSQL);
    public abstract void guardar(String comandoSQL);
    public abstract void desconectar();
}
```

Figura 8.11: Clase abstracta de las bases de datos

```
package PatronProxy;
public class ProxyBD extends BaseDatos{

    private BDSQLite accesoBD =null;

    @Override
    public void conectar(String direccion) {
        if(accesoBD!=null){
            accesoBD.conectar(direccion);
        }
        else{
            accesoBD= new BDSQLite();
            accesoBD.conectar(direccion);
        }
    }
    @Override
    public void consultar(String comandoSQL) {
        if(accesoBD!=null){
            accesoBD.consultar(comandoSQL);
        }
        else{
            accesoBD= new BDSQLite();
            accesoBD.consultar(comandoSQL);
        }
    }
    @Override
    public void guardar(String comandoSQL) {
        if(accesoBD!=null){
            accesoBD.guardar(comandoSQL);
        }
        else{
            accesoBD= new BDSQLite();
            accesoBD.guardar(comandoSQL);
        }
    }
    @Override
    public void desconectar() {
        if(accesoBD!=null){
            accesoBD.desconectar();
        }
        else{
            accesoBD= new BDSQLite();
            accesoBD.desconectar();
        }
    }
}
```

Figura 8.12: Clase proxy para el manejo de la conexión en SQLite

```

package PatronProxy;
import java.sql.Connection;

public class BDSQLite extends BaseDatos{
    private Connection conexion;
    @Override
    public void conectar(String direccion) {
        String driver = "org.sqlite.JDBC";
        String connectString = "jdbc:sqlite:"+direccion;
        try {
            Class.forName(driver);
            conexion = DriverManager.getConnection(connectString);
        } catch (Exception e) {
            System.out.println(e.getMessage());
        }
    }

    @Override
    public void consultar(String comandoSQL) {
        try {
            Statement stmt = conexion.createStatement();
            ResultSet rs = stmt.executeQuery(comandoSQL);
            ResultSetMetaData rsmd = rs.getMetaData();
            int numeroColumna = rsmd.getColumnCount();
            while (rs.next()) {
                for(int i = 1 ; i <= numeroColumna; i++){
                    System.out.print(rs.getString(i) + " ");
                }
                System.out.println();
            }
            stmt.close();
            desconectar();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage());
        }
    }

    @Override
    public void guardar(String comandoSQL) {
        try {
            Statement stmt = conexion.createStatement();
            stmt.executeUpdate (comandoSQL);
            stmt.close();
            desconectar();
        } catch (Exception e) {
            JOptionPane.showMessageDialog(null, e.getMessage());
        }
    }
}

```

```

@Override
public void desconectar() {
    try {
        conexion.close();
    } catch (SQLException e) {
        System.out.println(e.getMessage());
    }
}
}

```

Figura 8.13: Clase con la lógica de la base de datos para SQLite

8.1.5 Patrón Comando

```

package launcher;

import vista.Aplicacion;
import vista.Menu;
import control.ComandoDeAdicion;
import control.ComandoDeConsulta;
import control.IComando;

public class Cliente {

    public static void main(String[] args) {
        Aplicacion interfaz = new Aplicacion();
        IComando miComando = new ComandoDeConsulta(interfaz);
        Menu ventanaGUI = new Menu(miComando);
        ventanaGUI.correrMenu();
        miComando = new ComandoDeAdicion(interfaz);
        ventanaGUI.correrMenu();
    }
}

```

Figura 8.14: Clase Cliente que ejecuta el comando

```

package control;

public interface IComando {

    public void ejecutar();

}

```

Figura 8.15: Interface comando

```

package vista;

public class Aplicacion {

    public void modificar(){
        System.out.println("Accediendo a la Base de Datos del estudiante...");
        System.out.println("Verificando existencia del estudiante...");
        System.out.println("En espera de modificacion...");
        System.out.println("Informacion modificada...");
        System.out.println("Cerrando Base de Datos");
    }

    public void adicionar(){
        System.out.println("Accediendo a la Base de Datos del estudiante...");
        System.out.println("Verificando existencia del estudiante...");
        System.out.println("Informacion adicionada");
        System.out.println("Cerrando Base de Datos");
    }

    public void eliminar(){
        System.out.println("Accediendo a la Base de Datos del estudiante...");
        System.out.println("Verificando existencia del estudiante...");
        System.out.println("Informacion eliminada");
        System.out.println("Cerrando Base de Datos");
    }

    public void consultar(){
        System.out.println("Accediendo a la Base de Datos del estudiante...");
        System.out.println("Verificando existencia del estudiante...");
        System.out.println("Mostrando informacion del Estudiante");
        System.out.println("Cerrando Base de Datos");
    }
}

```

Figura 8.16: Clase Aplicacion donde se manejan las consultas a la BD

Figura 8.17: Implementación de la interfaz comando por cada metodo de Aplicacion

```

package control;

import vista.Aplicacion;

public class ComandoDeModificacion implements IComando{

    private Aplicacion app;

    public ComandoDeModificacion(Aplicacion aplicacion) {
        this.app = aplicacion;
    }

    @Override
    public void ejecutar() {
        app.modificar();
    }

}

```

Figura 8.18: Comando de Modificación


```
package control;

import vista.Aplicacion;

public class ComandoDeAdicion implements IComando{

    private Aplicacion app;

    public ComandoDeAdicion(Aplicacion aplicacion){
        this.app = aplicacion;
    }

    @Override
    public void ejecutar() {
        app.adicionar();
    }

}
```

Figura 8.19: Comando de Adición

```
package control;

import vista.Aplicacion;

public class ComandoDeEliminacion implements IComando{

    private Aplicacion app;

    public ComandoDeEliminacion(Aplicacion aplicacion){
        this.app = aplicacion;
    }

    @Override
    public void ejecutar() {
        app.eliminar();
    }

}
```

Figura 8.20: Comando de Eliminación

```
package control;

import vista.Aplicacion;

public class ComandoDeConsulta implements IComando{

    private Aplicacion app;

    public ComandoDeConsulta(Aplicacion aplicacion){
        this.app = aplicacion;
    }

    @Override
    public void ejecutar() {
        app.consultar();
    }

}
```

Figura 8.21: Comando de Consulta

```
package vista;

import control.IComando;

public class Menu {

    private IComando objComando;

    public Menu(IComando comando){
        this.objComando = comando;
    }

    public void correrMenu() {
        System.out.println("\t **** INTERFAZ DE LA APLICACION **** ");
        System.out.println("Pidiendo Datos del estudiante: ");
        objComando.ejecutar();
        System.out.println("Saliendo de la interfaz");
    }

}
```

Figura 8.22: Clase Menu con la que interactua el cliente en la GUI



Bibliografía

- [1] Díaz A. Patrones estructurales.
- [2] S Bolaños. *Presentación patrones de diseño GoF*.
- [3] Villegas M. Cardona S., Jaramillo S. *Introducción a la programación en Java*.
- [4] Modelado del sistema. Modelos de interacción.
- [5] Robert K. Behavioral design patterns.
- [6] R Pressman. *Ingeniería de Software. Un Enfoque Práctico*.