Universidad de los Andes
ISIS 1126 – Sección 3
Diseño y Programación Orientada a Objetos
Proyecto 3 – Entrega 2

REFLEXIÓN SOBRE EL PROCESO DE DISEÑO

1. INTRODUCCIÓN

Normalmente, cuando se habla de desarrollo de software, aspectos como la lógica algorítmica, la sintaxis del lenguaje y los requisitos funcionales ocupan la mayor parte de la atención de los programadores Juniors o con poca experiencia. Sin lugar a dudas, la etapa más importante dentro del desarrollo de software es la del diseño.

Durante el diseño se toman decisiones que repercutirán en las fases posteriores del proceso. Dependiendo de que tan bueno sea este, la implementación, será más sencilla de llevar a cabo, así como también las posibles modificaciones que se deseen hacer al programa en un futuro. Un buen diseño hace que un programa funcional genérico pase a ser una aplicación optima que se moldea a las necesidades de los usuarios.

El presente documento tiene como objetivo analizar las decisiones que fueron tomadas en la etapa de diseño en los proyectos de "Diseño y programación orientada a objetos". De tal manera que resulte ser un recurso valioso para todas aquellos estudiantes que cursen la materia.

2. DECISIONES EN EL ANALISIS DEL DOMINIO

Persistencia en los datos: Durante el desarrollo de proyecto 1 se optó por que el <<information holder>> denominado ArchivoUsuarios fuera el encargado de cargar y guardar la información. La anterior decisión en si no fue un error como tal, pero al momento de evaluar el dominio del problema no se tuvo en cuenta un aspecto fundamental.

Aunque no se expresara de manera explícita en el enunciado, por el tipo de aplicación que se pedía, era obvio que los datos debían ser persistentes. Si bien existía una función de guardado, esta solo funcionaba durante la misma ejecución del programa, por lo que en diseño propuesto se perdía la información con cada nueva ejecución.

El no entender a cabalidad el contexto del problema supuso que la implementación fuera funcional, pero inutilizable en un entorno real. Desde un principio se debió diseñar pensando en este requisito funcional implícito.

Errores menores en la autentificación de los usuarios: En el proyecto 1 para autentificar al usuario se decidió implementar el método ingresarLogin() en la clase Aplicación. Al momento de hacerlo no se empleó ningún sistema de contraseñas, solo bastaba colocar el login para acceder al sistema. Tampoco existía ninguna opción para intentar iniciar sesión en caso de

equivocarse, al hacerlo se creaba automáticamente un nuevo usuario, por lo que era necesario salir de la aplicación.

El no contar con opciones específicas para registrarse e iniciar sesión dificulta la interacción con el sistema. Por otra parte, el que no se requiera una contraseña para ingresar, a pesar de que disminuye la complejidad de la implementación, causa que la aplicación se vuelva demasiado insegura.

3. DECISIONES EN LA ESTRUCTURA DEL DISEÑO

Guardado en los datos: Inicialmente, en la etapa de diseño del proyecto 1, se propuso un diagrama en el cual existía una única clase encargada de cargar y guardar toda la información. Tras un extenso análisis se descartó tal idea, debido a que el sistema dependía demasiado de esta clase y que esta última sería muy extensa y compleja.

Al final se optó por implementar dos clases: ArchivadorUsuarios y ArchivadorProyectos. Aunque tal decisión disminuyó considerablemente el acoplamiento entre clases, también dificultó levemente establecer las relaciones entre los proyectos y los usuarios guardados.

Distribución de los menús: En el desarrollo del proyecto 2, se ideo una interfaz con 3 menus o ventanas principales: un menú para el login, otro para elegir el proyecto y un ultima ventana para realizar acciones con base al proyecto seleccionado. Como diseño se propuso que existieran tres clases principales para cada uno de los menus mencionados. Para evitar que estas clases fueran demasiado elaboradas y complejas, se optó por crear dos subclases por cada clase principal que se encargaran de actualizar la ventana en tiempo real.

Por ejemplo, la clase padre MenuLogin solo se encarga de la parte lógica para el inicio de sesión y registro de nuevos usuarios. Para actualizar la ventana, dicha clase invoca algunos métodos de sus clases hijas PanelLogin1 y PanelLogin2. En concreto, la subclase PanelLogin1 cumple la función de actualizar el panel correspondiente al inicio de sesión, mientras que PanelLogin2 hace lo propio con el panel con la información del registro.

Tal diseño permite que cada clase tenga solo un rol principal a seguir, lo que facilita la detección de errores, en el caso que se presenten contratiempos durante la ejecución del programa. Otras ventajas son que, al seguir el principio de “single responsability”, las responsabilidades son más fáciles de detectar y cambiar, el acoplamiento es menor y los cambios a una parte del sistema afectaran menos las otras.

Manejo de ventanas emergentes: Además de las subclases descritas en el numeral anterior, para el caso de las superclases MenuProyecto y MenuEleccionProyecto existen algunas subclases adicionales de tipo dialog. Estas fueron preconcebidas para crear ventanas emergentes que obligaran a los usuarios a diligenciar cierta información antes de interactuar con otras ventanas. En otras palabras, provocan la aparición de ventanas emergentes.

Por ejemplo, cuando se crea un nuevo proyecto, la clase DialoCrearProyectos, la cual es clase hija de MenuEleccionProyecto, construye una ventana emergente solicitando algunos datos para crear el proyecto. Para hacer efectiva la acción es necesario llenar todos los campos y dar en aceptar. Mientras que la ventana este abierta, no será posible ejecutar una nueva acción hasta que esta se haga exitosamente o se cancele cerrando la ventana.

El haber optado por tal alternativa, disminuyo la complejidad de las clases de los paneles y ayudó a seguir el principio de "single responsability". Esta decisión también hizo que fuera menos común la ocurrencia de errores, debido a que restringe al usuario a interactuar solo con la ventana emergente en cuestión antes de usar otras partes del programa. Por otra parte, la estructura propuesta, al tener un estilo de control de carácter delegado, ocasiona que hallan algunas colaboraciones poco útiles.

Interacción entre la interfaz y la lógica: Para que las clases de la interfaz pudieran acceder a la información de los proyectos y los usuarios, en el diagrama de diseño se estableció una clase denominada VentanaAplicacion para mediar la comunicación entre ambas partes, ideándose, para que fuera clase padre de todas las clases de los menús (MenuProyecto, MenuLogin, MenuEleccionProyecto)

Una de las principales desventajas del modelo propuesto es que, al haber tantas colaboraciones con la clase mencionada, aplicar un cambio en esta implicara probablemente modificar otras partes de la aplicación. Además, cuando el programa arroje un error, existirá una especie de efecto embudo, lo que ocasiona que la mayoría de los errores se produzcan en VentanaAplicacion, haciendo más difícil la corrección del problema. Por otro lado, al seguir un estilo de control más centralizado, es más fácil identificar los puntos más interesantes del programa.

4. PROBLEMAS POR EL DESCONOCIMIENTO DE LA TECNOLOGIA

Problemas con Swing: Debido a que los integrantes del grupo prácticamente no tenían experiencia con interfaces gráficas, así como también, al desconocimiento de parte de la sintaxis y algunas funciones de swing, se dieron algunos contratiempos.

En primer lugar, se desconocía como conservar las proporciones de los widgets al aumentar el tamaño de la ventana. Esto implico que se tuviera que desistir de implementar algunas funciones para ajustar el tamaño de la ventana, para minimizarla o para ponerla a pantalla completa, a pesar de estar incluidas en el diseño inicial. Actualmente, ya sabiendo el concepto de layout en java, probablemente se hubiera podido implementar estas funciones y muchas más.

A la hora de personalizar la interfaz de usuario, otra limitante fue el conocimiento superficial que se tenía acerca de la librería Swing. De haber tenido un mayor bagaje acerca del gran número de posibilidades que esta ofrece, la interfaz habría sido más moderna y responsiva. Además, se hubiera evitado tener que diseñar algún que otro widget desde cero, simplificando el diseño y la implementación considerablemente.

5. DIFICULTADES EN DISEÑAR EN UN ENTORNO INCIERTO

El mayor reto a lo largo del curso fue implementar nuevas funcionalidades sobre el proyecto sin tener la certeza de cuál sería la índole de los cambios. Si bien el docente a cargo mencionó desde el inicio del curso que el proyecto 2 consistiría en diseñar e implementar una interfaz, más allá de ello no fue hasta el último momento que se supo que tocaba modificar.

La dificultad bajo tal escenario se debió mayormente a que era la primera vez que se reutilizaba un proyecto para nuevas entregas, por lo que las metodologías aplicadas anteriormente influyeron en todas las futuras entregas. Afortunadamente, como grupo supimos sobreponernos a las circunstancias y salir adelante.

El hecho de diseñar un programa sin saber cómo iba a extenderse o cambiar más adelante, fue extremadamente útil para afianzar una correcta metodología de diseño que se ajustara a la mayor parte de las circunstancias. Como muchas de las personas y empresas que mandan a desarrollar software no tienen claro exactamente lo que quieren, el ser capaz de diseñar adecuándose a cambios imprevistos es una habilidad fundamental para todo programador.