Proyecto 3 - DPOO
Reflexión y análisis proyectos 1, 2 y 3

Miguel Arturo Reina Rocabado

10 de diciembre de 2022

Aciertos y desaciertos sobre decisiones de diseño

Durante el desarrollo de los proyectos, pude observar varios aciertos e inconvinientes sobre todo entre lo que se pensaba que era lo ideal y sobre cómo se implementa. En el proyecto 1, se planteó un diagrama de dominio que no cumplía con las reglas UML y se veía mal en general (Figura 1). Por esto, para el proyecto 2 y 3 se cambió de herramienta a GenMyModel para que (1) fuera mucho más sencillo desplazar las clases y (2)que el programa se encargara de la nomenclatura y reglas UML, por tanto considero que este fue un acierto. Sin embargo, revisando nuevamente el diagrama, claramente puedo ver que hacen falta relaciones entre las clases y creo que fue una falla desde el inicio no haber entendido desde el principio el dominio.

Enseñanza

Asegurarse de entender el dominio antes de si quiera empezar a programar, el diagrama de dominio puede ser una herramienta para identificar huecos en el conocimiento del que lo analiza.

Por otro lado, creo que fue acertado haber separado las clases de registrador vs. los demás controladores así se desacopla el proceso de registarse/ingresar del relacionado al equipo de fantasía. Sin embargo, creo que se acumuló mucha de la lógica en las clases de los controladores, dejando las clases del dominio casi sin responsabilidades más allá de ser utilizados como *information holder*. Ya con más conceptos en la cabeza, puedo identificar que varios métodos pudieron estar en estas clases y como está en el momento, se pudo haber desacoplado mucho mejor las clases cumpliendo sobre todo el principio de responsabilidad única de SOLID. Me pude dar cuenta de esto sobretodo porque en la carga de datos tuve que pasar mucho de los datos entre dos clases (ControladorTemporada y Jugador) por lo que me quedaría con el consejo que vimos en clase.

Enseñanza

Al asignar responsabilidades, es útil asignar métodos a las clases que contienen los datos en la misma clase para que los datos no pasen entre clases y se incremente el encapsulamiento.

Adicionalmente, me gustaría mencionar sobre el cambio de diseño que se tuvo desde la entrega 1 a la entrega 2 donde se pasó de persistir la información a través de .csv a .ser gracias a implemenzar `Serializable` esto permitió una persistencia

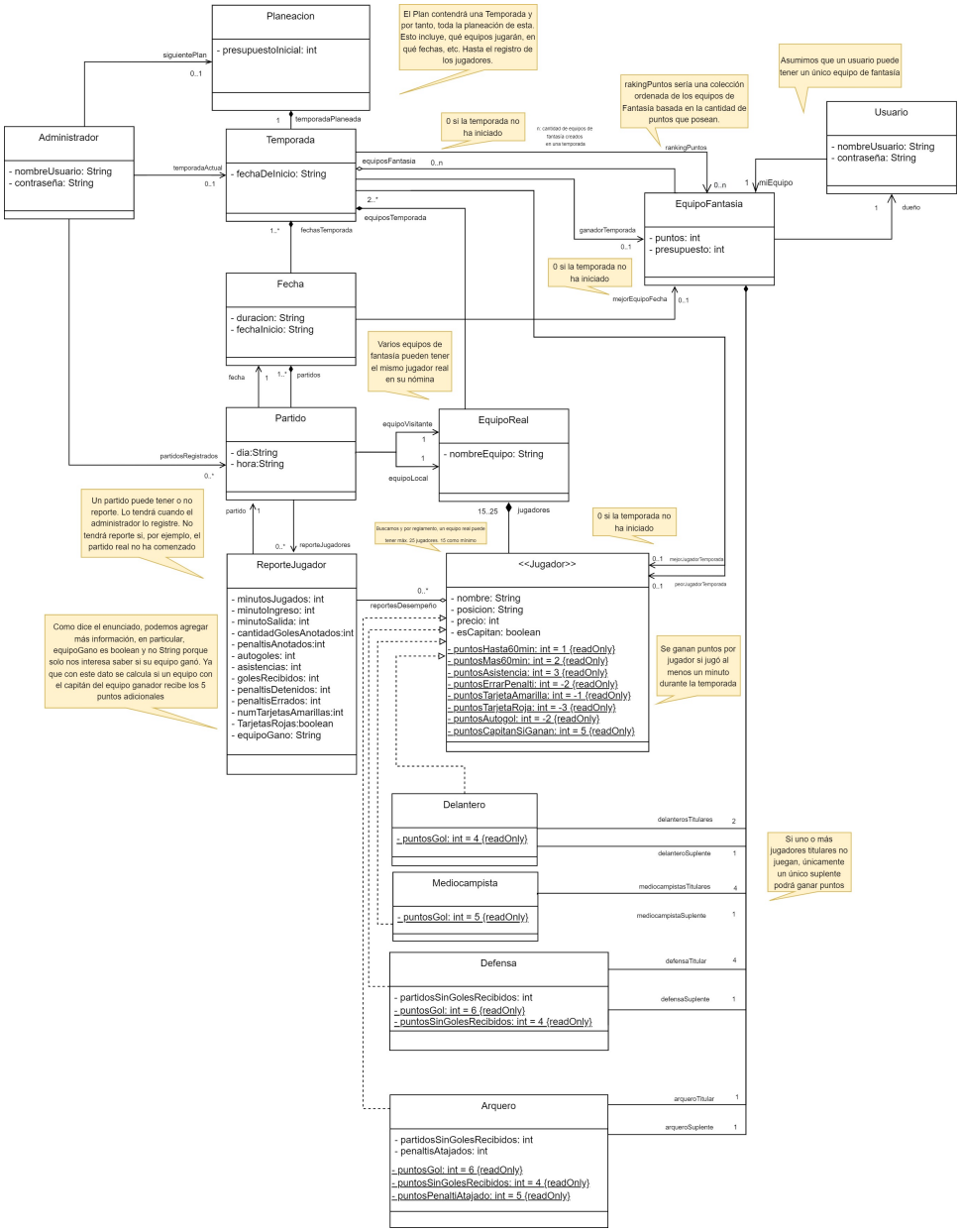

Figura 1: Diagrama de dominio - proyecto 1

mucho más sencilla que con los .csv sobretodo para objetos complejos como son los equipos de fantasía. Sin embargo, creo que una falla de diseño es hacer persistir a los controladores, aunque esto permite que las estructuras de datos que donde se persiste e.g. la lista de todos los equipos de fantasía de la temporada esto mismo hace que se generen errores cuando se hacen colaboraciones entre los diferentes controladores ya que uno se carga primero que otro y a veces salían errores de que un controlador no estaba inicializado. Ya leyendo, es mucho mejor crear una clase que tenga como única funcionalidad tener estos datos y que sea el controlador quien la cargue.

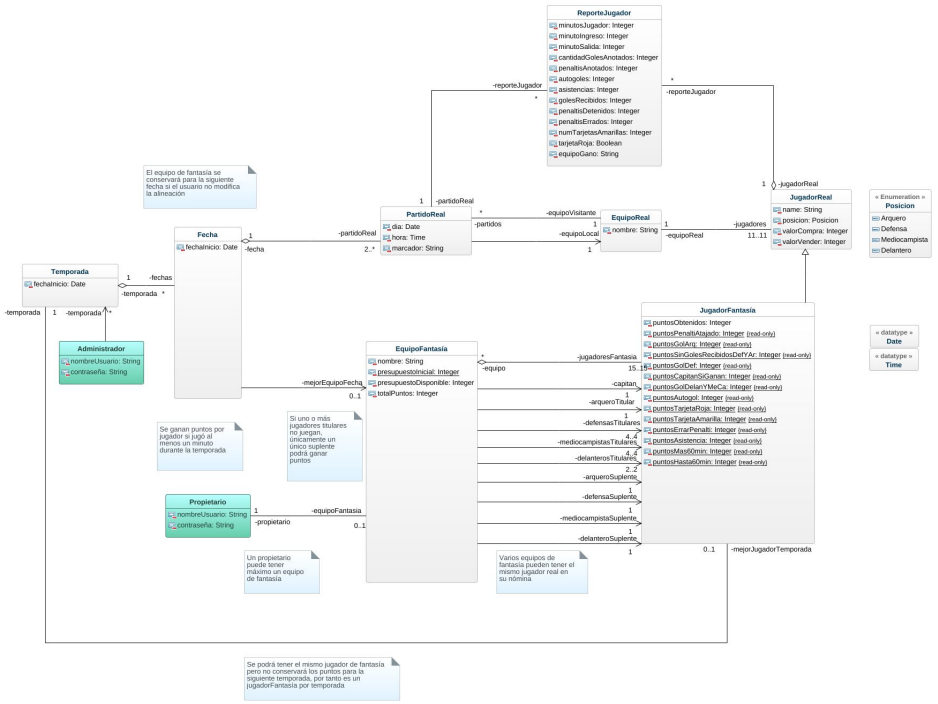

Figura 2: Diagrama de dominio - proyecto 2

Enseñanza

Crear una única clase que sea *information holder* y hacer que persista cargándola desde el controlador que la utiliza; no persistir el controlador.

Finalmente, me pareció una decisión de diseño acertada crear una plantilla de paneles de la cual se heredara para tener en común ciertas fuentes, layout y sobre todo, para que cada uno de los páneles tuviera acceso a la clase de FacadeGUI. Creo que esto facilitó enormemente a la creación del diagrama de clases y también a la programación de las interfaces ya que no tenía que llamar a

Tipos de problemas

Desconocimiento de la tecnología

Ya realizando este documento, realmente me he dado cuenta lo amplío y customizable lo que pueden ser las interfaces hechas con Swing y AWT. Sin embargo, la documentación de Oracle sobre estos framework me parece que no es eficiente explicando los conceptos y me fue costoso en tiempo encontrar soluciones a cosas particulares. Además, creo que no fue hasta los patrones de diseño y con lectura de textos académicos sobre los patrones que realmente pude tener algo concreto sobre lo cual evaluar cómo deberían interactuar y formular las clases. Por lo que creo que mi eficiencia para llevar a cabo los requerimientos en el programa muchas veces se vio limitada por mis conocimientos de la tecnología que requería (Swing, Java, etc.) y eso se vio reflejado en la completitud de las entregas. Por otro lado, un compañero mio no desconocía/no pudo/no se podía integrar git con Eclipse por lo que se tuvo muchos problemas en la integración de código, por lo que eso también

fue un problema.

Problemas sobre el análisis de dominio, diseño sobre este

Como se mencionó anteriormente, definitivamente considero que se tuvo problemas en el análisis, ya que al principio se estaba asignando puntos a los jugadores reales cuando es a los equipos de fantasía quienes se les asignan los puntos. Por esto y otros casos, creo que mi falta de conocimiento sobre el dominio de estudio hizo que algunos atributos estuvieran fuera de lugar y que se confundieran conceptos entre sí. Por otro lado, sobre el diseño creo que realmente hicieron falta interfaces para hacer que la aplicación fuera robusta ante cambios

Enseñanza

¡Incluir interfaces! No tiene sentido que un registrador dependa de una clase particular (ControladorTemporada) para redigir al usuario. Las interfaces establecen un contrato entre clases disminuyendo el acoplamiento.

Cierre

Soy plenamente consciente que a la aplicación le faltan funcionalidades, mas creo que es lo que pude hacer con mis conocimientos actuales, tiempo disponible y compañeros infiables para trabajar. En vacaciones, pienso hacer un mini proyecto aplicando patrones y principios para asegurarme de que sí entendí y si no repasar para poder estar seguro de formarme como un buen profesional.