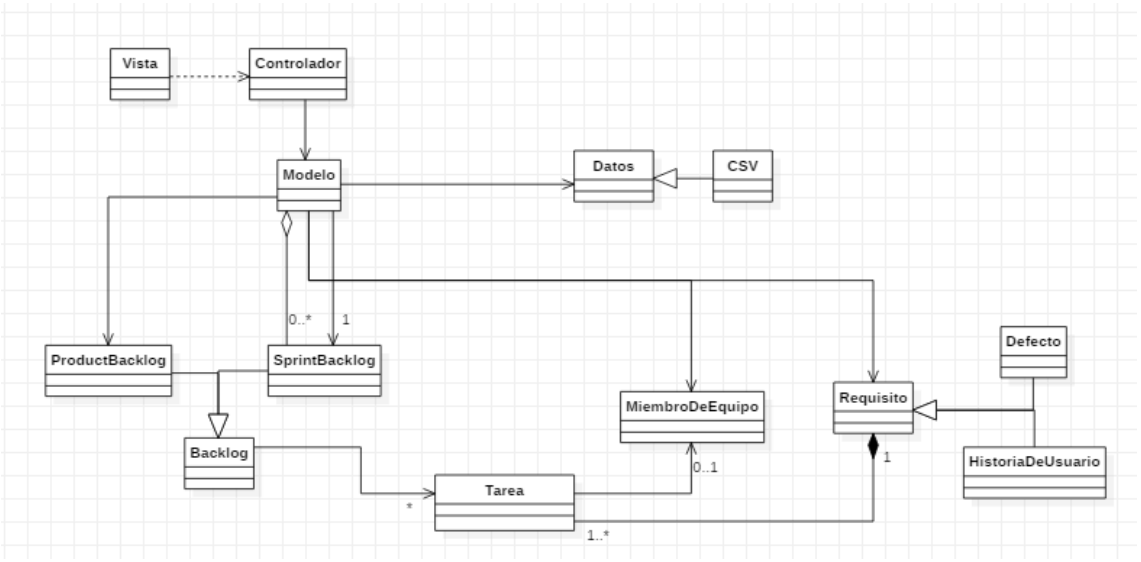
MEMORIA DE PRÁCTICAS

Diseño y Mantenimiento del Software

Luis Miguel Cabrejas Arce y Hector Cogollos Adrian

Contenido

Diagrama de Clases



Patrones de diseño empleados

Patrón Singleton

Utilizamos el patrón Singleton en clases como Modelo, Controlador, CSV y para ProductBacklog. El motivo para haber empleado el patrón en estas clases es que únicamente queremos una instancia de cada una de esas clases así que para evitar que se puedan crear más aplicamos el patrón.

Patrón MVC

Utilizamos el patrón MVC con las clases Modelo, Vista y Controlador para aislar la parte lógica de la Vista, esto nos es útil debido a que en la segunda parte de la práctica una de las opciones es implementar una nueva forma de interacción, de forma que si decidimos implementar la nueva forma de interacción usar este patrón nos facilitará el trabajo.

Justificaciones de diseño

Vista

No se utiliza el patrón Estados para hacer la interfaz de texto de la aplicación porque consideramos que las condiciones que definen el estado son demasiado simples.

Modelo

Lo mismo que pasaba con las vistas pasa con los sprintsBacklog los estados de las tareas son excesivamente simples dado que la única condición es cual es el estado previo. Seria interesante si dependiese de otros factores como que no haya terminado el sprint o que se pueda mover a estados previos. En su lugar hemos utilizado una función para cada movimiento, simplemente si no esta en la lista del estado previo no lo mueve.

No permitimos mover tareas que no tengan miembro asignado a Doing debido a que no tiene sentido que una tarea se esté haciendo si no hay nadie asignado a la realización de esa tarea. Tampoco permitimos modificar el miembro asignado a una tarea mientras no esté en el product backlog o en TO DO en el sprint backlog.

SprintBacklog

Como hemos comentado no utilizamos estados para las tareas debido a que la definición de los estados es demasiado simple y sus condiciones para el cambio de estados también, y consideramos que es más difícil definir a que sprint pertenece una tarea mediante estados que utilizando listas, porque no solo hay que identificar el estado, sino que también el sprint al que pertenece.

Por otra parte, cuando terminamos un Sprint guardamos el estado del sprint que hemos terminado y todas las tareas que no se han finalizado vuelven al productBacklog. Hemos considerado que cuando se termina un sprint hay que mantener el estado a modo de historial pero que todas las tareas que no han sido terminadas pasaran al producto backlog y no al nuevo sprint pensando en que este puede volver a ser replanteado.

Otro aspecto clave para justificar no haber utilizado el patrón estados es que solo se pueden mover tareas en una dirección. Solo dejamos volver a reubicar las tareas que no han sido finalizadas cuando termina el sprint. Consideramos que no tiene sentido volver atrás en su estado a no ser que no se puedan llevar a cabo por el motivo que sea.

Datos

Para los datos utilizamos una interfaz que es “Datos” esta interfaz lo que nos permite es aislar la implementación concreta con la que se van a guardar los datos del método que se va a usar para guardarlos. Esto nos permitiría que en caso de que implementemos otro método como una base de datos no sea necesario tocar la lógica del programa.

CSV

Es la implementación concreta para implementar la persistencia, esta clase tiene varios puntos de mejora que más tarde comentaremos, pero la idea es implementar esas funciones que hemos definido en “Datos” esta clase es un singleton para evitar que haya múltiples instancias.

Posibles mejoras

CSV

En este caso podríamos abstraer la lógica de convertir un modelo orientado a objetos a uno relacional del guardado de datos. Esto también serviría en el caso de que utilizásemos una base de datos.

También creemos que se puede separar la función de guardar datos de la estructura concreta que deba tener el fichero. Esto sería crear una función que reciba una dirección de fichero y una lista de listas de tipo String y la guarda en el fichero.

Conclusiones

Hemos podido comprobar que muchos de los patrones aprendidos no tienen sentido en implementaciones tan simples dado que su objetivo es simplificar modelos complejos de implementar. También que, aunque no se implementen los patrones por su simplicidad en el modelo de negocio actual puede ser interesante tenerlos en cuenta si cambian estas reglas.

En cuanto a el diseño es importante tenerlo claro antes de implementarlo. Esto no quiere decir que haya que tener todo el diseño claro, pero si aquel módulo que vamos a implementar. Por ejemplo, el módulo para la persistencia, aunque posteriormente pueda haber ciertas demandas de datos que actualmente no han sido contempladas, si has conseguido estructurar bien este módulo puede ahorrar tiempo. Un ejemplo concreto es guardar un objeto, si separas la función de guardar, de la estructura concreta del objeto, implementar el guardado de otro objeto solo supone tratar esa estructura concreta. En nuestro caso todo se guarda y carga como texto, lo que hemos hecho es una función que lee un documento y nos lo devuelve como una lista de listas luego es la función concreta la que deberá interpretar que es lo que hay en cada fila y columna.