DISEÑO Y MANTENIMIENTO DEL SOFTWARE

4º GRADO DE INGENIERÍA INFORMÁTICA

# PRÁCTICA OBLIGATORIA

# APLICACIÓN SCRUM

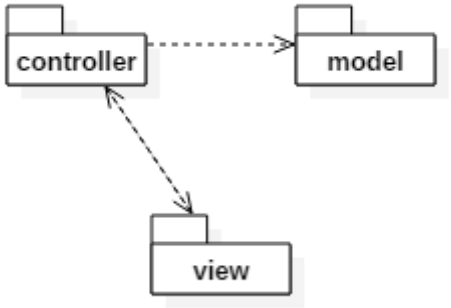
ALUMNOS:
Yi Peng Ji
Alicia Olivares Gil

# Contenido

# 1. Estructura principal del programa. Patrón MVC

El programa se divide en 3 paquetes principales correspondiendo con los participantes del patrón arquitectónico Modelo-Vista-Controlador:

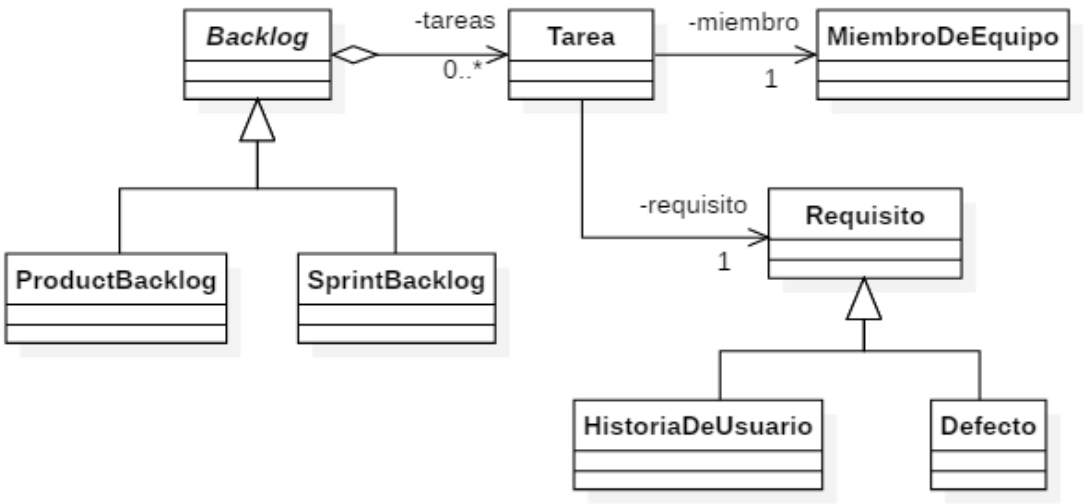
- **model**: contiene la estructura de los datos que maneja el programa. En nuestro caso añadimos otro paquete **persistence** que se encarga de la gestión de la persistencia y de instanciar y mantener la referencia a los objetos del modelo durante la ejecución.
- **view**: corresponde con la interfaz de la aplicación, la parte del programa con la que interactúa el usuario.
- **controller**: responde a los eventos generados por la interacción del usuario con la vista y realiza peticiones al modelo. Se puede ver como un intermediario entre la vista y el modelo.




# 2. Modelo

En el **paquete model** se incluyen las entidades necesarias para representar los datos que emplea el programa.

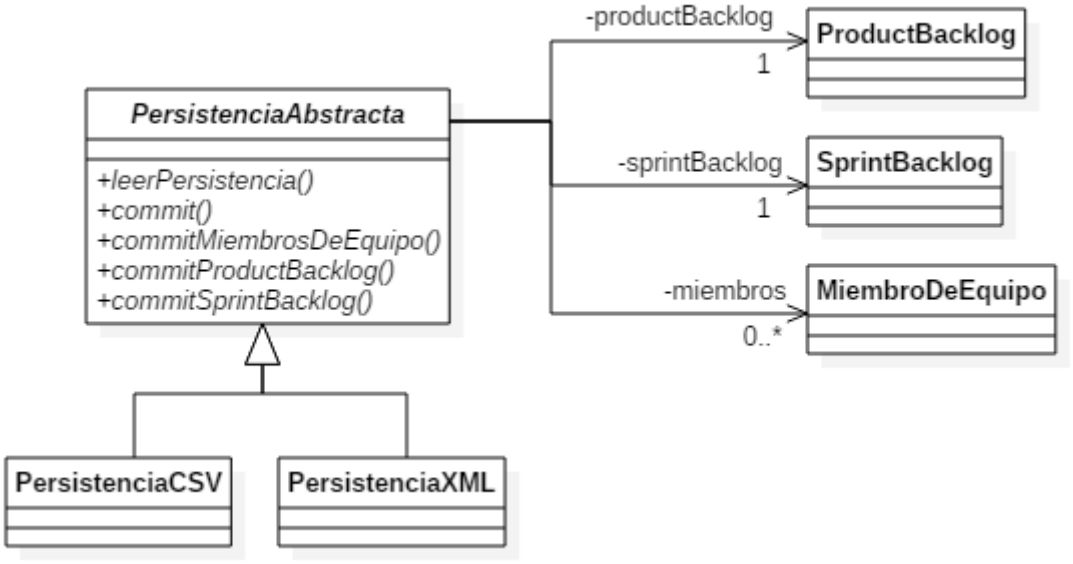
La estructura básica de los datos se define con las siguientes relaciones entre entidades:



- Una clase abstracta **Backlog** que contiene un Mapa de Tareas y que implementa los mecanismos para añadir tareas y acceder a ellas.
- Una clase concreta **ProductBacklog** que extiende la clase Backlog y que contiene las tareas pendientes que no han sido añadidas al Sprint actual. Esta clase implementa el **patrón Singleton** ya que solo se necesita una instancia a la que se debe tener un acceso global.
- Una clase concreta **SprintBacklog** que extiende la clase Backlog y que contiene las tareas que han sido añadidas al Sprint actual. Además, contiene dos atributos adicionales:
    - **descripcion**: Descripción del sprint.
    - **fechaInicio**: Fecha en la que se inicia el sprint.

  Esta clase también implementa el **patrón Singleton** ya que solo existirá un Sprint al mismo tiempo.
- Una clase **Tarea** con los siguientes atributos:
    - **idTarea**: Identificador entero de la tarea.
    - **coste**: coste entero de la tarea.
    - **beneficio**: Beneficio entero de la tarea.
    - **requisito**: Objeto de la clase Requisito asociado. Cada Tarea se asocia a un único requisito.
    - **miembro**: Objeto de la clase MiembroDeEquipo asociado, corresponde con el miembro al que se le asigna esa tarea. Cada tarea se asigna a un único miembro.
    - **estado**: Objeto de la clase EstadoTarea asociado, corresponde con el estado de la tarea dentro del ciclo de scrum (pendiente, en proceso, en validación, completada). Decidimos usar un objeto en lugar de un atributo entero o String para implementar el patrón Estado que gestione las transiciones de la tarea entre los distintos estados.
- Una clase **MiembroDeEquipo** con un atributo nombre que representa a un miembro del equipo de desarrollo.
- Una clase **Requisito** con los siguientes atributos:
    - **titulo**: título del requisito (o de la tarea, ya que la relación es 1 a 1).
    - **descripcion**: descripción del requisito (o de la tarea).
- Una clase **Defecto** que extiende la clase Requisito y añade un nuevo atributo:
    - **tarea**: identificador entero de la tarea anterior con la que se relaciona dicho defecto.
- Una clase **HistoriaDeUsuario** que extiende la clase Requisito y añade un nuevo atributo:
    - **actor**: Actor al que se asocia la historia de usuario (p.e. administrador, cliente, programador…).

## 3. Persistencia




Damos dos implementaciones alternativas de la persistencia, una mediante ficheros **csv**. y otra mediante ficheros **xml.** Para ello contamos con una clase **PersistenciaCSV** y una clase **PersistenciaXML** que extienden la clase abstracta **PersistenciaAbstracta**, de forma que si se quisiera dar una implementación diferente a la persistencia solo sería necesario crear una nueva clase que extendiera de PersitenciaAbstracta, implementando los métodos necesarios, y modificar la instanciación del objeto persistencia en la clase Main del paquete controller.

La lectura y escritura en la persistencia se realiza en la implementación de los métodos abstractos de PersistenciaAbstracta en PersistenciaCSV o PersistenciaXML. Para la interacción con los ficheros de persistencia csv. empleamos la librería **opencsv-4.3.2**.

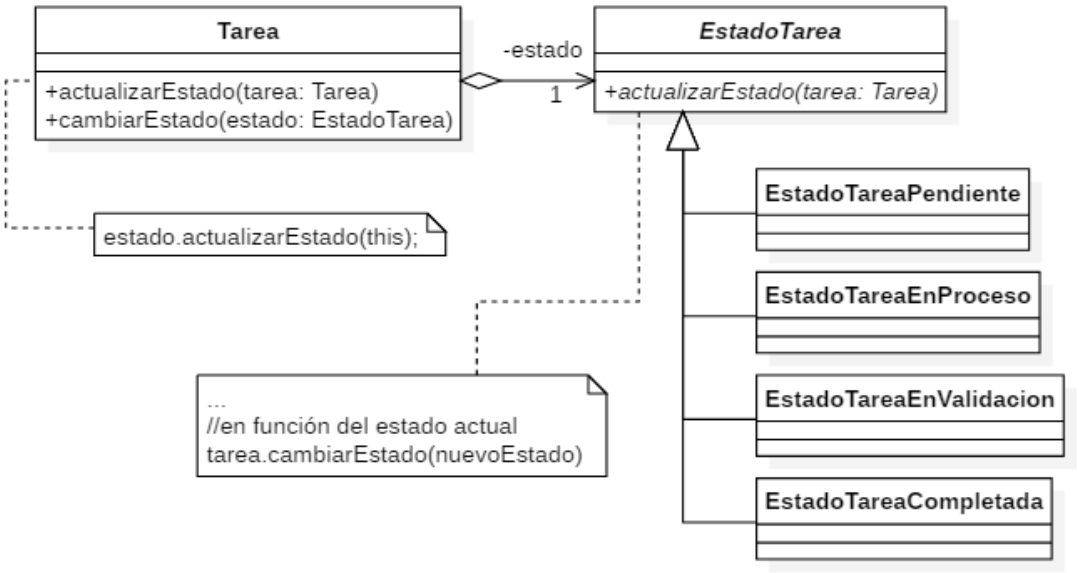
Las clases PersistenciaCSV y PersistenciaXML implementan el **patrón Singleton** ya que solo se necesita un acceso a la persistencia al que acceder de forma global.

La clase PersistenciaAbstracta contiene las relaciones hacia los datos dinámicos que maneja el programa:

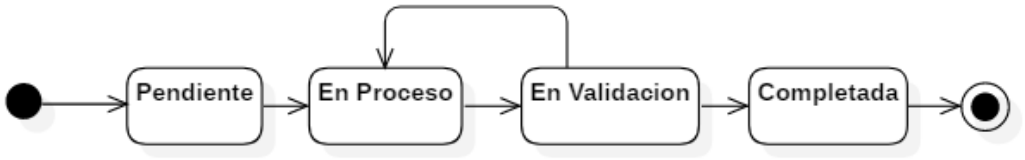
- **productBacklog**: contiene las tareas que no se han asignado a un sprint.
- **sprintBacklog**: contiene las tareas del sprint actual
- **miembros**: Mapa de todos los miembros del equipo, se les haya asignado o no una tarea.

## a) Patrón Estado para la clase Tarea

Para gestionar las transiciones entre los estados de una tarea decidimos emplear el patrón Estado. Cada objeto Tarea tiene un **EstadoTarea** asociado que además de indicar su estado actual, define el siguiente estado al que se moverá al actualizar su estado. Todos los EstadoTarea concretos implementan el **patrón Singleton**, ya que, dado que no tienen estado interno, cada instancia puede ser compartida por varios objetos Tarea sin que perjudique al correcto funcionamiento. Otra opción sería implementar el patrón Flyweight no visto en clase.



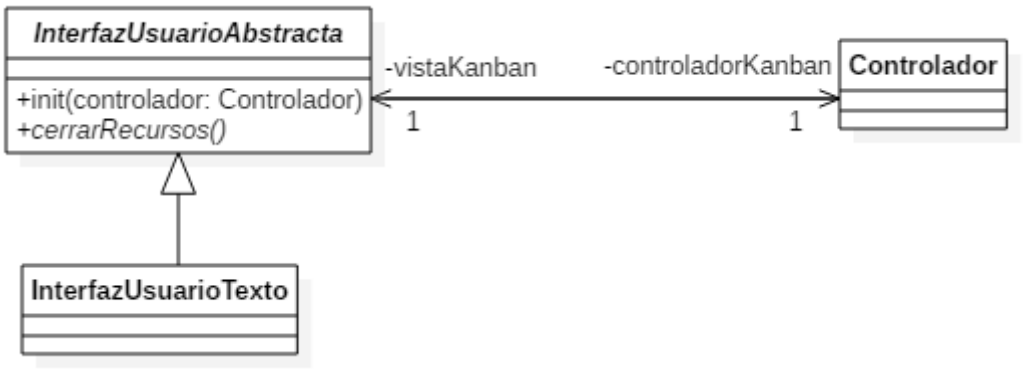

Permitimos las siguientes transiciones:




Si una tarea en validación pasa a estar en proceso o completada será decisión del usuario. El resto se moverán al siguiente estado definido.

## 4. Vista

El paquete view contiene la implementación de la interfaz de usuario. Damos una implementación de la interfaz en modo texto mediante la clase **InterfazUsuarioTexto**, que extiende la clase abstracta **InterfazUsuarioAbstracta**, que será la que se comunique con el controlador. De esta forma, para dar una implementación alternativa de la interfaz solo será necesario crear una nueva clase que extienda InterfazUsuarioAbstracta e implemente los métodos necesarios. Se da una implementación por defecto al método init donde crea la referencia al objeto Controlador, por lo que de ser necesario sobrescribir el ese método sería conveniente llamar al init del padre o iniciar la referencia a Controlador de otra forma.




La navegación es bidireccional ya que la vista hace peticiones a través de los métodos del controlador, y el controlador se encarga de iniciar la vista llamando a su método init.

Para gestionar las transiciones entre los distintos submenús de la interfaz de texto decidimos emplear el **patrón Estado**.

### a) Patrón Estado para la clase InterfazUsuarioTexto

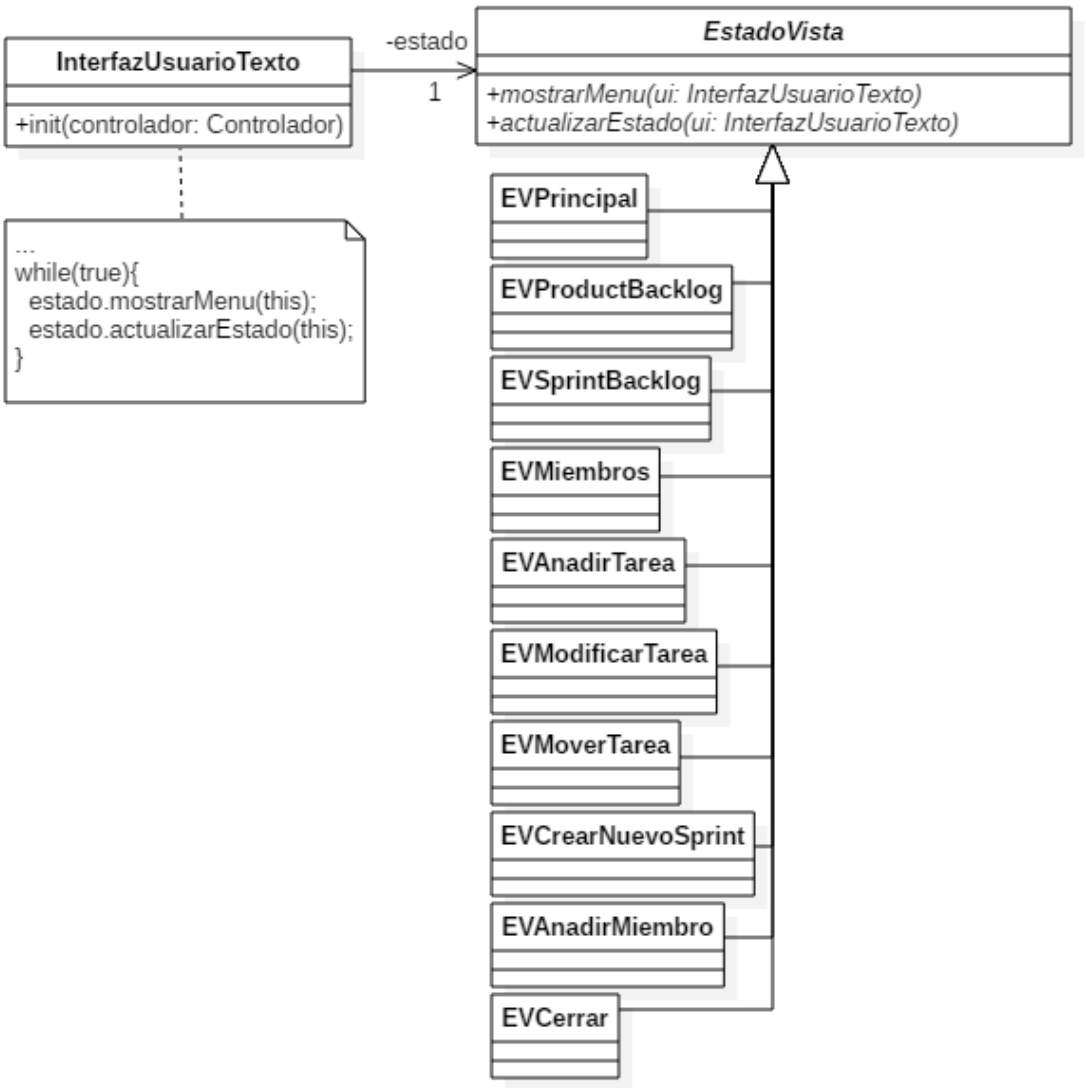
Definimos una clase **EstadoVista** por cada submenú en la cual se definirá la transición al siguiente EstadoVista, es decir, al siguiente submenú a mostrar.

Cada EstadoVista concreto da su propia implementación de dos métodos:

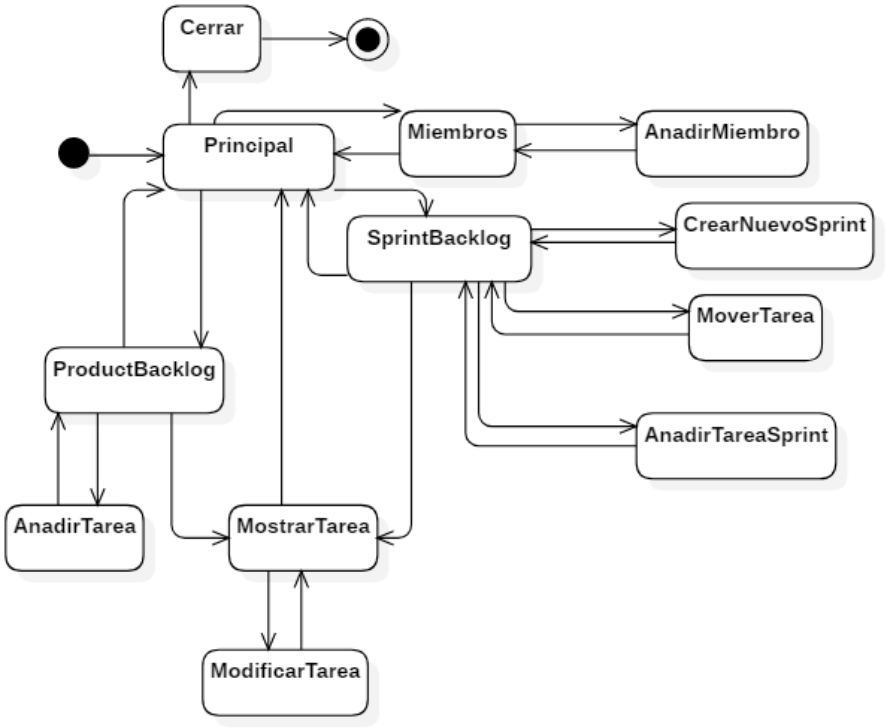
- **mostrarMenu**: muestra el submenú correspondiente a ese estado, el cual recibirá las acciones del usuario.
- **actualizarEstado**: modifica el estado del controlador al siguiente estado en función del estado actual y de las acciones del usuario.

La ejecución principal del programa consistirá en ejecutar estos dos métodos en ese orden hasta la finalización del programa.

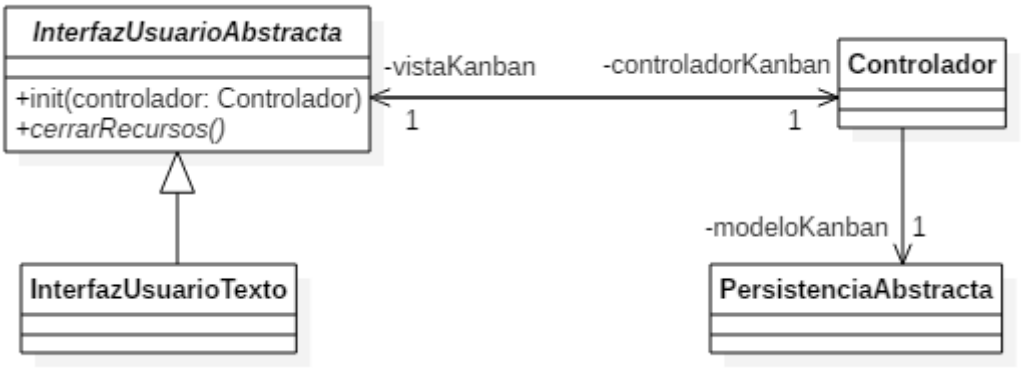
Además, cada EstadoVista concreto implementa el **patrón Singleton**. Esto no provoca problemas ya que durante la ejecución del programa solo existirá una instancia de InterfazUsuarioTexto (al ser también Singleton).




Definimos las siguientes transiciones entre submenús:



## 5. Controlador




El paquete controller contiene la clase **Controlador**, encargada de contener la lógica de negocio transmitiendo las peticiones de la vista al modelo y a la persistencia a través de sus métodos definidos.

Mantiene referencias a la Interfaz de usuario y a la persistencia, y contiene los métodos que implementan la lógica de negocio: modificarTarea, anadirTarea, anadirTareaSprint, moverTarea, anadirNuevoSprint y anadirNuevoMiembro.