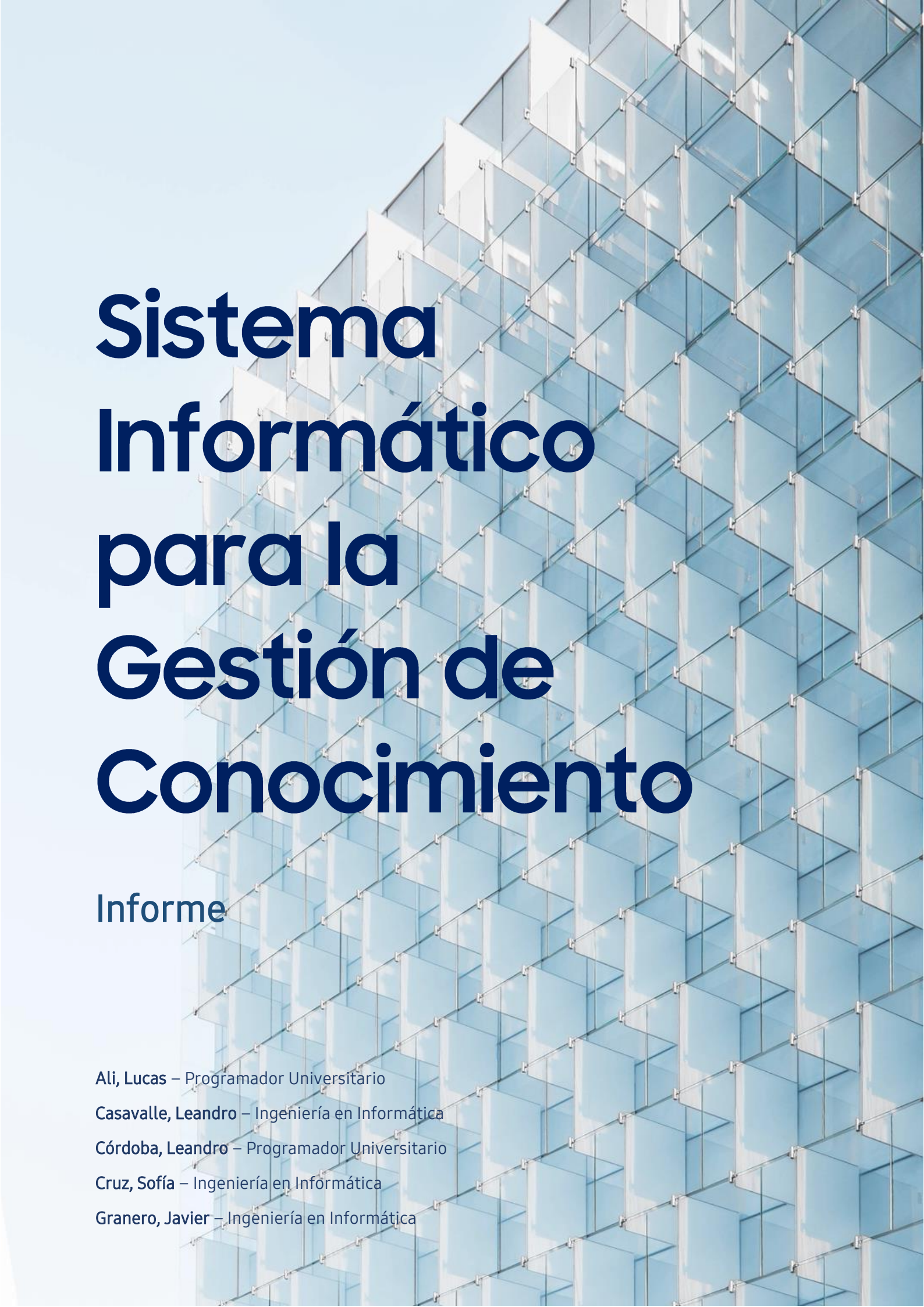


Sistema Informático para la Gestión de Conocimiento

Informe

Ali, Lucas – Programador Universitario

Casavalle, Leandro – Ingeniería en Informática

Córdoba, Leandro – Programador Universitario

Cruz, Sofía – Ingeniería en Informática

Granero, Javier – Ingeniería en Informática

1. Requerimientos del proyecto

Una comprensión clara y detallada de los requisitos de un proyecto es importante, puesto que guiarán su diseño, implementación y funcionalidad. En esta sección, se presentan los requerimientos identificados para el proyecto "Gestión de Conocimiento". Estos requerimientos han sido derivados tanto del enunciado original del proyecto como de las interacciones con el docente, asegurando así una comprensión integral de las expectativas y funcionalidades clave que el sistema debe abordar.

A lo largo de esta sección, se detallan los requisitos agrupados en áreas específicas, como la gestión de usuarios, la interacción colaborativa, el ranking y valoración, entre otros. Cada requisito se presenta de manera clara y concisa, respaldado por las decisiones de diseño correspondientes que han guiado el desarrollo del sistema.

❖ Gestión de Usuarios:

- Registro de usuarios con datos personales (**nombre**, **apellido**, **país de origen**), dirección de correo electrónico (**email**) y **contraseña**.
- Cada usuario puede realizar **preguntas** y/o **respuestas**.
- Puede expresar su aprobación mediante un "**me gusta**" a una respuesta.
- Si es un usuario **consultor** (creador de una pregunta), puede marcar una respuesta como **aceptada**, considerándola la **solución** a su consulta.

❖ Gestión de Preguntas:

- Creación de preguntas con **título**, **descripción** y la opción de adjuntar una **imagen**.
- Registro de la **fecha** de creación de cada pregunta.
- Asociación de **tags** (etiquetas) a cada pregunta, útiles para realizar búsquedas.
- La pregunta se considera **inactiva** si no recibe respuestas en más de 6 meses.
- Posibilidad de marcar una respuesta como **aceptada** y cambiar el estado de la pregunta a **solucionada**, impidiendo la recepción de más respuestas.
- Cada pregunta puede tener **múltiples respuestas**, realizadas por cualquier usuario, pudiendo hacerlas más de una vez.
- **Notificación** al usuario consultor cuando su pregunta recibe una **respuesta**.
- En caso de que una pregunta esté **suspendida**, **no** se emite **notificación** al usuario consultor al recibir una respuesta.

❖ Gestión de Respuestas:

- Creación de respuestas con su **contenido** (texto) y, opcionalmente, una **imagen**.

- Registro de la **fecha** de creación de cada respuesta.
- Una respuesta puede ser marcada como **aceptada**, convirtiéndola en la **solución** a una consulta (pregunta).
- Posibilidad de valorar las respuestas mediante "**me gusta**".
- Posibilidad de **listar** las respuestas según la **cantidad** de "**me gusta**".

❖ Estados de las Preguntas:

- Las preguntas pueden tener varios estados: **activa**, **inactiva**, **solucionada**, **suspendida**.
- Automatización del cambio de estado:
  - **Inactiva**, después de 6 meses sin respuestas.
  - **Activa**, si su estado actual es inactiva y recibe una respuesta.
  - **Suspendida**, si el usuario consultor elimina su cuenta.
  - **Solucionada**, si el usuario consultor marca alguna respuesta como aceptada.

❖ Interacción Colaborativa:

- Los usuarios pueden realizar **preguntas** y **responderlas**.
- **Notificación** automática al usuario que realizó la pregunta cuando alguien responde.
- Preguntas inactivas pueden ser respondidas y cambian al estado activo.

❖ Ranking y Valoración:

- Los usuarios pueden ser rankeados según la **cantidad** de **respuestas aceptadas**.
- Las preguntas pueden ser rankeadas según su **cantidad** de "**me gusta**".
- La respuesta **aceptada** ocupa el **primer lugar** en el ranking si la **pregunta** está **solucionada**.

❖ Búsqueda y Filtrado:

- Listar preguntas dado un **tag** específico.
- Listar usuarios rankeados según la **cantidad** de **respuestas aceptadas**.

2. Consideraciones particulares de diseño

El diseño de una solución es una etapa crucial, ya que sienta las bases para su funcionalidad, flexibilidad y mantenibilidad. En esta sección, se expondrán las consideraciones particulares de diseño tomadas por nuestro equipo para cumplir con los requisitos establecidos en la sección anterior. Cada decisión de diseño fue evaluada y seleccionada para optimizar la eficiencia del sistema y mejorar la experiencia del usuario.

Desde la conceptualización de la clase principal hasta la implementación de las clases especializadas, cada elección se encuentra respaldada por un análisis y una comprensión de los requisitos del proyecto. La estructura del sistema, las relaciones entre clases y el manejo de estados se abordarán en detalle, ofreciendo una visión clara de las decisiones que impulsan el diseño integral de esta solución.

A lo largo de esta sección, se presentan las justificaciones detrás de las decisiones clave, la aplicación de patrones de diseño para resolver desafíos específicos, y demás aspectos.

Generalizando, esta sección busca ofrecer una visión detallada y transparente del proceso de diseño, demostrando la abstracción que subyace en la construcción de nuestra solución.

Para el diseño de la solución se eligió la Programación Orientada a Objetos (POO) como paradigma de diseño. Este paradigma ofrece una base robusta y estructurada que se alinea de manera efectiva con la naturaleza del problema a resolver. La POO es una metodología que se centra en modelar el sistema a través de objetos, entidades que encapsulan tanto los datos (atributos) como las operaciones (métodos) relacionadas con una entidad específica. Esta elección se fundamenta en varios principios clave que aportan beneficios significativos al proceso de diseño y desarrollo de software.

Diseño de la clase Sistema

Nuestro equipo consideró diseñar una clase principal que actúe como el sistema central llamada Sistema, encargada de gestionar integralmente el conjunto de clases. Esta clase sirve como interfaz para otra clase, Usuario, que se mencionará a continuación, facilitando así las funciones que los usuarios podrán tener al utilizar nuestra solución.

La clase sistema, en su rol central, asume varias responsabilidades clave, entre las cuales se destacan la capacidad de añadir preguntas y respuestas, *rankear* a los usuarios, eliminar usuarios y otorgar "Me Gusta" a respuestas, delegando responsabilidades a otras clases más internas para su correcto funcionamiento.

Además, se implementa una relación de composición entre las clases Sistema y Usuario, ya que se considera que sin sistema no podríamos existir usuarios. Del mismo modo, existe una relación de composición entre Sistema y Publicación (una clase abstracta de la cual se hablará más adelante), ya que la existencia de la segunda depende intrínsecamente de la primera.

Diseño de la clase Usuario

Nuestro equipo planteó de manera definitiva la creación de una clase Usuario. Esta clase desempeñará un papel fundamental al contener la información de los usuarios del sistema, convirtiéndola en una entidad clave. Almacenará elementos vitales como el nombre de usuario, la

dirección de correo electrónico, la contraseña, las publicaciones que han recibido "Me Gusta" y la gestión de notificaciones.

Además, se diseñó la clase Usuario con funciones específicas que facilitarán la construcción de otros objetos, como Preguntas y Respuestas. Siguiendo este enfoque, se estableció una relación asociativa **bidireccional** entre la clase **Usuario** y la clase **Publicación**, que actúa como la clase base abstracta de Respuesta y Pregunta.

Diseño de la clase Publicación

Nuestro equipo tenía claro que la solución requería tanto de una clase **Pregunta** como de una clase **Respuesta**. En una primera instancia, las diseñamos como clases independientes entre sí. Sin embargo, a medida que avanzaba el diseño, se observó que estas clases compartían varias similitudes, como el ID (identificador numérico), una imagen (utilizando la dirección absoluta de almacenamiento de un archivo de imagen) y la fecha de creación. Además, ambas clases tenían la funcionalidad de listar información.

En vista de estas similitudes, se consideró que la manera más óptima de abordar este desafío era aplicar la **generalización** de estas clases en una superclase llamada **Publicación**. Publicación se convirtió en la clase base tanto de Pregunta como de Respuesta, heredando sus atributos y métodos, y especificando aquellos que se consideraron convenientes.

Diseño de la clase Pregunta

Sin dudas, la clase **Pregunta** fue complicada pero interesante a su vez de diseñar, en un principio se la planteó como una clase que tendría como atributos un título, una descripción y una serie de etiquetas (esto último se lo implementaría como un arreglo de cadenas de caracteres, *strings*), además de los atributos que heredaba de la superclase Publicación. Por otro lado, se pensó la funcionalidad de poder añadir respuestas dentro de esta misma clase, ya que esta tendría una relación de **agregación** con la clase Respuesta, que podría contener o no, valga la redundancia, respuestas una pregunta.

Lo más desafiante del diseño de esta clase fue plantear como se manejarían los estados que podría tener una Pregunta. En un principio se planteó que dentro de esta misma clase se manejarían todos los métodos de cambio de estado mediante la estructura de selección múltiple “según” (*switch*), pero se pudo ver que era muy poco práctico, es por ello que en consulta con profesores se decidió implementar una técnica de diseño conocida como **Patrón de Estado** (*State Pattern*), que simplificó el manejo del comportamiento de la adición de respuestas y notificación de la clase Pregunta.

Patrón de Estado en el diseño de la clase Pregunta

El **Patrón de Estado** desempeña un papel fundamental en el diseño de la clase Pregunta, afrontando el desafío de gestionar los diferentes estados que puede asumir una pregunta a lo largo de su ciclo de vida. Como se mencionó anteriormente, en un principio, se planteó manejar estos estados directamente dentro de la clase Pregunta mediante una estructura de selección múltiple, pero esta aproximación se reveló como poco práctica y difícil de mantener, pues si se deseaba

incorporar otros estados posibles, implicaría mayores modificaciones, interfiriendo con el diseño en general.

En lugar de ello, se optó por implementar el Patrón de Estado, una técnica de diseño que permite que un objeto altere su comportamiento cuando su estado interno cambia. En el contexto de la clase Pregunta, el Patrón de Estado simplifica la gestión de estados como "Suspendida", "Activa", "Solucionada" o "Inactiva", así como las transiciones entre ellos.

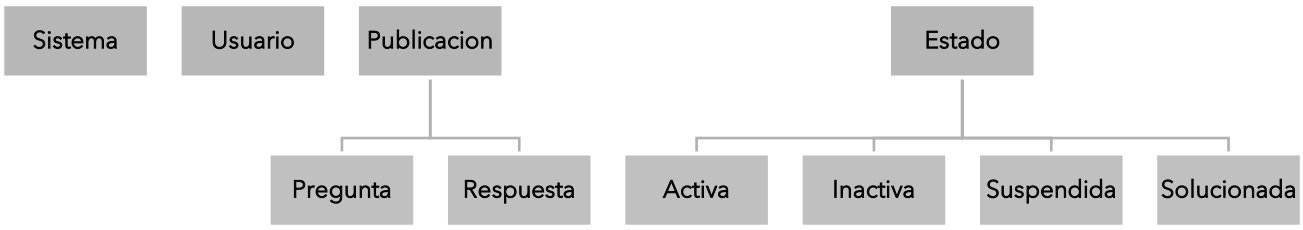
Este patrón implica la creación de clases de estado concretas que encapsulan el comportamiento asociado a un estado específico. Es decir, que cada estado podría definir cómo se manejan las respuestas y las notificaciones. Esto se logra creando una superclase (abstracta) llamada **Estado**, y sus clases derivadas correspondientes a cada estado (en este caso, **Activa**, **Inactiva**, **Suspendida** y **Solucionada**), donde se redefiniría un método según los requerimientos especificados.

La clase Pregunta, por lo tanto, mantiene una referencia a un objeto de estado, delegando las operaciones relacionadas con el estado a este objeto. Esta abstracción permite una mayor modularidad y extensibilidad en el código, ya que se pueden agregar nuevos estados o modificar el comportamiento existente sin alterar la clase Pregunta en sí.

Diseño de la clase Respuesta

La clase que menos complicaciones presentó fue **Respuesta**, una clase con algunos atributos importantes, como un contador de la cantidad de "Me Gusta" que recibió una respuesta, un booleano que determina si es aceptada o no. Estos atributos ayudarían a *rankear* las respuestas y a identificar las aquellas que son prioritarias para una pregunta.

Diagrama de clases identificadas



3. Desarrollo del trabajo

En esta sección, se detallan las etapas fundamentales en el diseño e implementación del sistema de gestión del conocimiento, para la obtención de una solución.

A lo largo de las tres etapas mencionadas a continuación, se presenta una visión minuciosa del trabajo realizado, desde la identificación de clases y sus responsabilidades hasta la concreción del código en la etapa de implementación. Cada etapa brinda aspectos cruciales del desarrollo, destacando decisiones de diseño, estructuras de código y la lógica detrás de cada componente.

En la Etapa 1, se presentarán las clases identificadas, junto con sus roles y responsabilidades específicos en el contexto del sistema. Esto se complementará con un detallado diagrama UML (*Unified Modeling Language*) que exhibirá la estructura de cada clase, incluyendo atributos, métodos y las relaciones clave.

La Etapa 2, se presentará la visualización de la solución a través de un Diagrama de Clases (UML). Este diagrama proporciona una visión global de la arquitectura del sistema, mostrando cómo las clases interactúan y se relacionan entre sí para lograr los objetivos del proyecto.

Finalmente, en la Etapa 3, se hará énfasis en los detalles de la implementación, donde el código materializa el diseño resultante de la etapa anterior. Se destacarán las decisiones de diseño, consideraciones de rendimiento y cualquier aspecto distintivo que haya guiado la implementación en C++.

A través de estas etapas, se busca proporcionar una visión detallada y coherente del desarrollo del sistema, desde la concepción de las clases hasta la realización de la solución a través del código implementado.

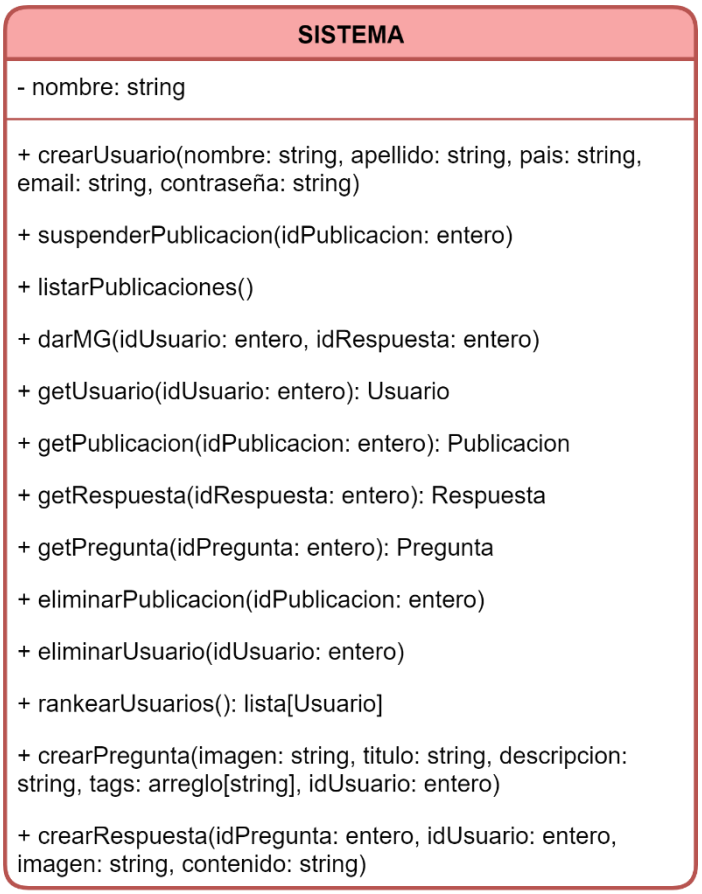
3.1.Clases identificadas y sus responsabilidades

Del proceso de abstracción de la situación planteada, se lograron identificar las siguientes clases, incluyendo sus atributos y métodos:

Sistema

Esta es la clase principal, encargada de manejar la aplicación. Posee una serie de métodos que permiten la interacción con las clases Usuario y Publicación (y sus derivadas). Permite manipular todos los datos dentro de la aplicación, como eliminar o crear un usuario, eliminar o crear una publicación (pregunta o respuesta), y mostrar las publicaciones realizadas en el gestor de conocimientos.

El diagrama de esta clase es el siguiente:



El único atributo (privado) de esta clase es 'nombre', un *string* que da identidad al sistema.

Entre los métodos (públicos) destacados, se encuentran:

- crearUsuario(nombre: string, apellido: string, pais: string, email: string, contraseña: string): permite crear un usuario en el sistema, recibiendo como parámetros los elementos necesarios para cargar los atributos de la clase correspondiente. Sistema tiene una relación de composición con la clase Usuario, y una lista con objetos de dicha clase; al crearse un nuevo usuario, éste se añade a la lista usuarios.

- suspenderPublicacion(idPublicacion: entero): permite cambiar el estado de una pregunta a "Suspendida" una vez que hayan transcurrido 6 meses desde su publicación. La pregunta se busca en el sistema usando el método getPregunta(idPublicacion: entero) indicando su identificador numérico (recibido como parámetro), y se procede al cambio de estado invocando a un método en la clase Pregunta. Aquí está el pseudocódigo:

```
pregunta <-- getPregunta(idPublicacion)
fechaActual <-- Fecha.getFecha()
fechaPublicacion <-- pregunta.getFecha()

Si (fechaActual.getMes() - fechaPublicacion.getMes() >= 6) Entonces
      pregunta.cambiarEstado(Suspendida)
```

- darMG(idUsuario: entero, idRespuesta: entero): permite incrementar el contador de "Me Gusta" de una respuesta. Para hacerlo, recibe como parámetro el identificador numérico del usuario que desea dar "Me Gusta" y el identificador de la respuesta en cuestión; con esos datos, el sistema buscará el usuario y la respuesta con los métodos getUsuario(idUsuario: entero) y getRespuesta(idRespuesta: entero), de forma correspondiente, y luego realiza un control para asegurar que el usuario no haya dado una valoración antes a esta respuesta (para evitar la repetición de "Me Gusta" por parte de un mismo usuario), en caso afirmativo, se accede a un método de la clase Respuesta para incrementar el contador de "Me Gusta" y un método de la clase Usuario para añadir la respuesta a un listado especial (una función agregada, no solicitada por el proyecto). Aquí el pseudocódigo de este método:

```
usuario <-- getUsuario(idUsuario)
respuesta <-- getRespuesta(idRespuesta)

Si (Not usuario.pertenece(idRespuesta)) Entonces
      respuesta.darMG()
      usuario.agregrarRespuestaLikeada(respuesta)
```

- crearPregunta(imagen: string, titulo: string, descripcion: string, tags: arreglo[string], idUsuario: entero): permite recibir como parámetros los elementos necesarios para poder crear una pregunta. Recibe, además, un identificador numérico para el usuario que desea realizar la pregunta; con este dato, el sistema identifica a dicho consultor con el método getUsuario(idUsuario: entero) y luego invoca al método para crear la pregunta dentro de la clase Usuario, usando la instancia del elemento identificado anteriormente; de esta forma, delega la tarea de la creación de una pregunta a otra clase. Aquí el pseudocódigo en cuestión:

```
usuario <-- getUsuario(idUsuario)

Si (usuario <> Null) Entonces
      pregunta <-- usuario.crearPregunta(imagen, titulo, descripcion, tags)
      publicaciones.Add(pregunta)
```

- crearRespuesta(idPregunta: entero, idUsuario: entero, imagen: string, contenido: string): el funcionamiento de este método es similar al anterior: recibe como parámetros los elementos necesarios para poder crear una respuesta, y un identificador numérico del usuario que desea crear la respuesta; con este dato, el sistema lo identifica con el método getUsuario(idUsuario: entero) en la lista usuarios, y luego invoca al método para crear la pregunta dentro de la clase Usuario, usando la instancia del elemento identificado anteriormente; de esta forma, delega la tarea de la creación de una respuesta a otra clase. Aquí el pseudocódigo de este método:

```
pregunta <-- getPregunta(idPregunta)
usuario <-- getUsuario(idUsuario)

Si (pregunta <> Null) Entonces
      respuesta <-- usuario.crearRespuesta(pregunta, imagen, contenido)
      publicaciones.Add(respuesta)
```

- eliminarUsuario(idUsuario: entero): permite eliminar un usuario del sistema, indicando como parámetro su identificador numérico. Este método busca el usuario identificado en la lista usuarios, luego invoca a un método de eliminación dentro de la instancia identificada de la clase Usuario, y finalmente remueve el elemento de la lista antes mencionada. Este es el pseudocódigo en cuestión:

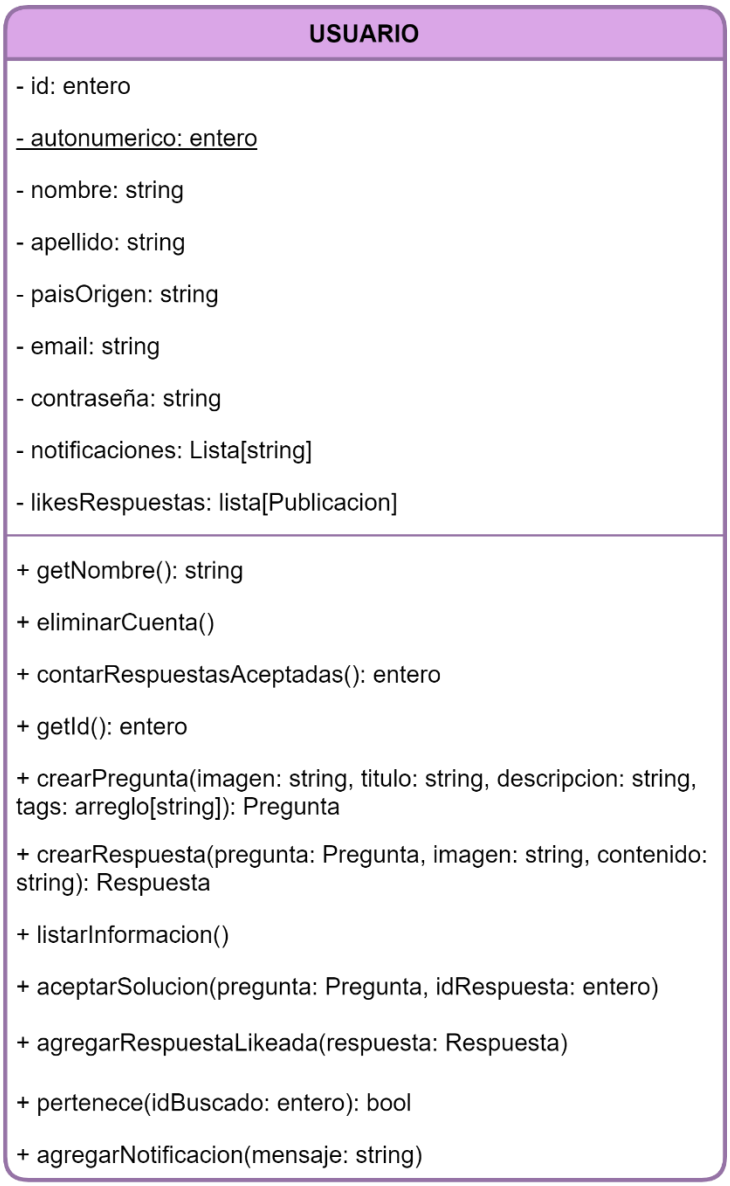
```
usuarioAEliminar <-- getUsuario(idUsuario)

Si (usuarioAEliminar <> Null) Entonces
      usuarioAEliminar.eliminarCuenta()
      usuarios.Remove(usuarioAEliminar)
```

Usuario

La clase Usuario tiene como propósito representar y gestionar las acciones que un usuario puede llevar a cabo dentro del sistema. Anteriormente, se había mencionado que un usuario puede realizar una pregunta y ésta puede ser respondida por cualquier otro usuario del sistema (pudiendo responder más de una vez una misma pregunta), además, distintos usuarios pueden dar "Me Gusta" a las respuestas, una única vez, y el usuario que realizó la pregunta es el encargado de aceptar la respuesta como solución si cree que responde a su pregunta de la mejor forma.

Este es el diagrama de esta clase:



La clase Usuario tiene los siguientes atributos: ID (identificador numérico), nombre, país de origen, dirección de correo electrónico y contraseña, que se utilizan para almacenar la información personal del usuario (cumpliendo así con los requisitos básicos de la identificación).

Adicionalmente, se han incorporado dos listas: notificaciones (almacena cadenas de caracteres que representan las notificaciones recibidas cuando alguien responde a la pregunta del usuario), y likesRespuestas (almacena las respuestas a las cuales el usuario ha dado "Me Gusta". Esta funcionalidad adicional, no solicitada por el proyecto, se consideró conveniente incorporar.

- crearPregunta(imagen: string, titulo: string, descripcion: string, tags: arreglo[string]): Pregunta: recibe como parámetros los elementos necesarios para crear una pregunta. Este método es

invocado desde la clase Sistema cuando se delega la tarea de crear una pregunta por un usuario identificado. Este método simplemente invoca al constructor de la clase **Pregunta** con los parámetros recibidos, y agrega la nueva instancia a la lista **publicaciones** del usuario. Este es el pseudocódigo:

```
pregunta <-- Pregunta(imagen, Fecha(), titulo, descripcion, tags)
publicaciones.Add(pregunta)

Retorna pregunta
```

- **crearRespuesta(pregunta: Pregunta, imagen: string, contenido: string): Respuesta:** de forma análoga con el método anterior, este recibe los parámetros necesarios para crear una respuesta. También es invocado desde la clase Sistema, cuando se delega la tarea de creación de una respuesta por un usuario identificado. Este método simplemente invoca al constructor de la clase **Respuesta** con los parámetros recibidos, y agrega la nueva instancia a la lista **publicaciones** del usuario. Aquí el pseudocódigo:

```
respuesta <-- Respuesta(imagen, Fecha(), this, contenido)
pregunta.addRespuesta(respuesta)

Retorna respuesta
```

- **eliminarCuenta():** uno de los requisitos del proyecto establece que las preguntas realizadas por un usuario deben permanecer en el sistema incluso si este decide eliminar su cuenta, cambiando su estado a "**Suspendida**". Este método se encarga de modificar el estado de cada pregunta asociada al usuario correspondiente, como se mencionó anteriormente. Cuando el Sistema desea eliminar una instancia de la clase Usuario, invoca este método, delegando parte de la tarea a esta clase. Aquí el pseudocódigo correspondiente:

```
Para cada P en publicaciones
      Si (P.getTipo = 1) Entonces
            P.cambiarEstado(Suspendida)
```

- **contarRespuestasAceptadas(): entero:** este método simplemente cuenta la cantidad de respuestas aceptadas que están asociadas a un usuario. Esto es útil a la hora de realizar un ordenamiento de usuarios en el sistema ponderando la cantidad de soluciones brindadas. Este es el pseudocódigo:

```
cont <-- 0
Para cada P en publicaciones
      Si (P.getTipo = 2) Entonces
            Si (P.esAceptada()) Entonces
                  cont <-- cont + 1
Retorna cont
```

- **aceptarSolucion(pregunta: Pregunta, idRespuesta: entero):** este método recibe como parámetros una referencia a un objeto **Pregunta** y un identificador numérico para la respuesta que se desea marcar como solución. Dentro de la instancia referenciada de Pregunta, se accede a la lista **respuestas** y se busca la publicación según el ID proporcionado, una vez encontrada, se invoca al método para marcarla como solución. Aquí el pseudocódigo:

```
Para cada R en pregunta.getRespuestas()
      Si (R.getId() == idRespuesta) Entonces
            respuesta.aceptarSolucion()
```

- **agregarRespuestaLikeada(respuesta: Respuesta):** este método simplemente agrega una respuesta (recibida como parámetro) a la que el usuario dio "Me Gusta" a la lista **likesRespuestas**. Este es el pseudocódigo:

```
this.likesRespuestas.Add(respuesta)
```

- **pertenece(idBuscado: entero): bool:** un método que verifica si una respuesta se encuentra o no en la lista **likesRespuestas**. La respuesta se busca en dicha lista según el identificador numérico recibido como parámetro. Este método es usado cuando un usuario da un "Me Gusta" a una respuesta, y evitar la reiteración de la valoración (en la clase Sistema, **darMG(...)**). A continuación, el pseudocódigo:

```
Para cada R en likesRespuestas
    Si (R.getId() == idBuscado) Entonces
        Escribir("Ya le diste me gusta a esta respuesta, no puedes darle más de uno")
      Retorna true
Retorna false
```

- **agregarNotificacion(mensaje: string)**: este método simplemente enlista una cadena de caracteres a la lista **notificaciones** del usuario. Es invocado por un método dentro de la clase **Pregunta**. Este es el pseudocódigo:

```
this.notificaciones.Add(mensaje)
```

Publicación

Esta clase sirve como la clase base tanto para la clase Pregunta como para la clase Respuesta. Aquí se declaran los métodos y atributos que serán compartidos por ambas clases mencionadas. Entre los atributos presentes en la clase Publicación, se incluyen un ID (identificador autonumérico), autonumérico (para autoincrementar el ID), tipo e imagen. En cuanto a las funciones, se han definido los correspondientes *getters* para cada atributo, además de la función para listar información. Esta última, como su nombre indica, tiene la tarea de presentar la información de los atributos de un objeto Publicación en un formato legible.
Este es el diagrama de clase diseñado:

Pregunta

La clase Pregunta hereda tanto atributos como métodos de su clase base, Publicación, como mencionamos anteriormente. Esta clase tiene la responsabilidad principal de almacenar información y llevar a cabo diversas funciones cuando los usuarios de nuestra aplicación realizan preguntas. Además de los atributos heredados, Pregunta incluye los siguientes: título, descripción y tags, siendo este último un array de strings en el cual se almacenan las etiquetas asociadas a la pregunta.

Este es el diagrama de clase diseñado:

Entre los métodos a destacar se encuentran los siguientes:

- **agregarRespuesta(respuesta: Respuesta):** este sencillo método se encarga, básicamente, de agregar a nuestra lista de Respuesta un objeto de tipo respuesta el cual es recibido como parámetro (una referencia).

```
this.respuestas.Add(respuesta)
```

- **addRespuesta(respuesta: Respuesta) :** este método de encarga de llamar a la clase Estado y delegarle la tarea de agregar una respuesta en función a que tipo de clase hija de estado sea la actual en nuestro objeto Respuesta, este funcionamiento fue implementado en base al patrón de diseño "Pattern State". Este es el pseudocódigo:

```
estado.agregarRespuesta(respuesta, this)
```

- **notificarUsuario(respuesta: Respuesta):** este método se encarga sencillamente de que al momento que una respuesta es agregada a nuestro objeto pregunta, se agrega un mensaje de tipo *string* a nuestra lista de notificaciones en el objeto usuario dueño de esta respectiva Pregunta. Este es el pseudocódigo:

```
mensaje <-- "El usuario " + this.respuesta.usuario.getNombre()
+ " respondió a tu pregunta " + this.titulo

this.usuario.notificaciones.Add(mensaje)
```

- cambiarEstado(nuevoEstado: Estado): este método se encarga que la referencia de estado referencie a un nuevo estado, el cual es recibido como parámetro. Este es el pseudocódigo:

```
this.estado <-- nuevoEstado
```

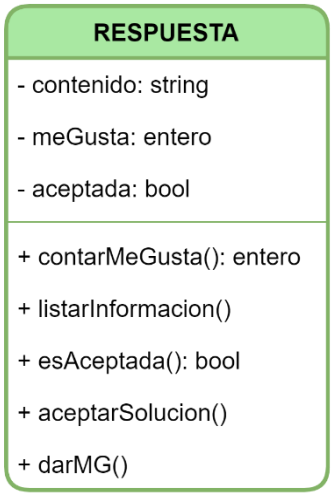
- rankearRespuesta(): este método como, su nombre indica, se encarga de rankear los objetos de tipo Respuesta que contiene la Pregunta. El ordenamiento se realiza en base a la cantidad de "Me Gusta" que contiene cada Respuesta, facilitando la implementación de ordenamiento utilizando la función *sort()*, de la librería STD (C++).

- listarRespuestas(): este método se encarga de recorrer todos los objetos de tipo Respuesta que contiene cada Pregunta y llamar a su método listarInformacion() para mostrar la información de cada objeto Respuesta.

Respuesta

La subclase Respuesta, hereda atributos de la superclase Publicación, se encarga de contener la información de la respuesta realizada por el usuario, registrar la cantidad de "Me Gusta", y ser aceptada o no por el usuario.

El diagrama de clase diseñado es el siguiente:



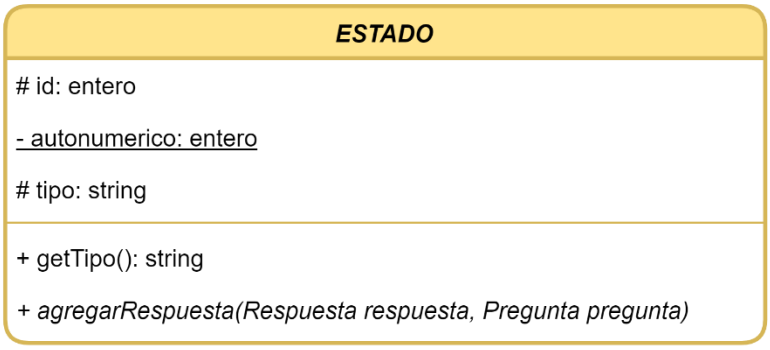
Entre los métodos a destacar están los siguientes:

- **contarMeGusta(): entero:** este método se encargar de devolver el atributo **meGusta** de Respuesta.
- **darMG():** este método de encarga principalmente de aumento en 1 el valor de nuestro atributo **meGusta**.
- **listarInformacion():** este método lista toda la información del objeto Respuesta.
- **esAceptada():** retorna nuestro atributo booleano **aceptada**.
- **aceptarSolucion():** este método se encarga de cambiar el valor de nuestro atributo **aceptada**, que es booleano.

Estado

Esta clase es necesaria para definir el estado de la pregunta, permite la transacción entre estados, y la posibilidad de agregar un nuevo estado para futuro.

Este es el diagrama de la clase:



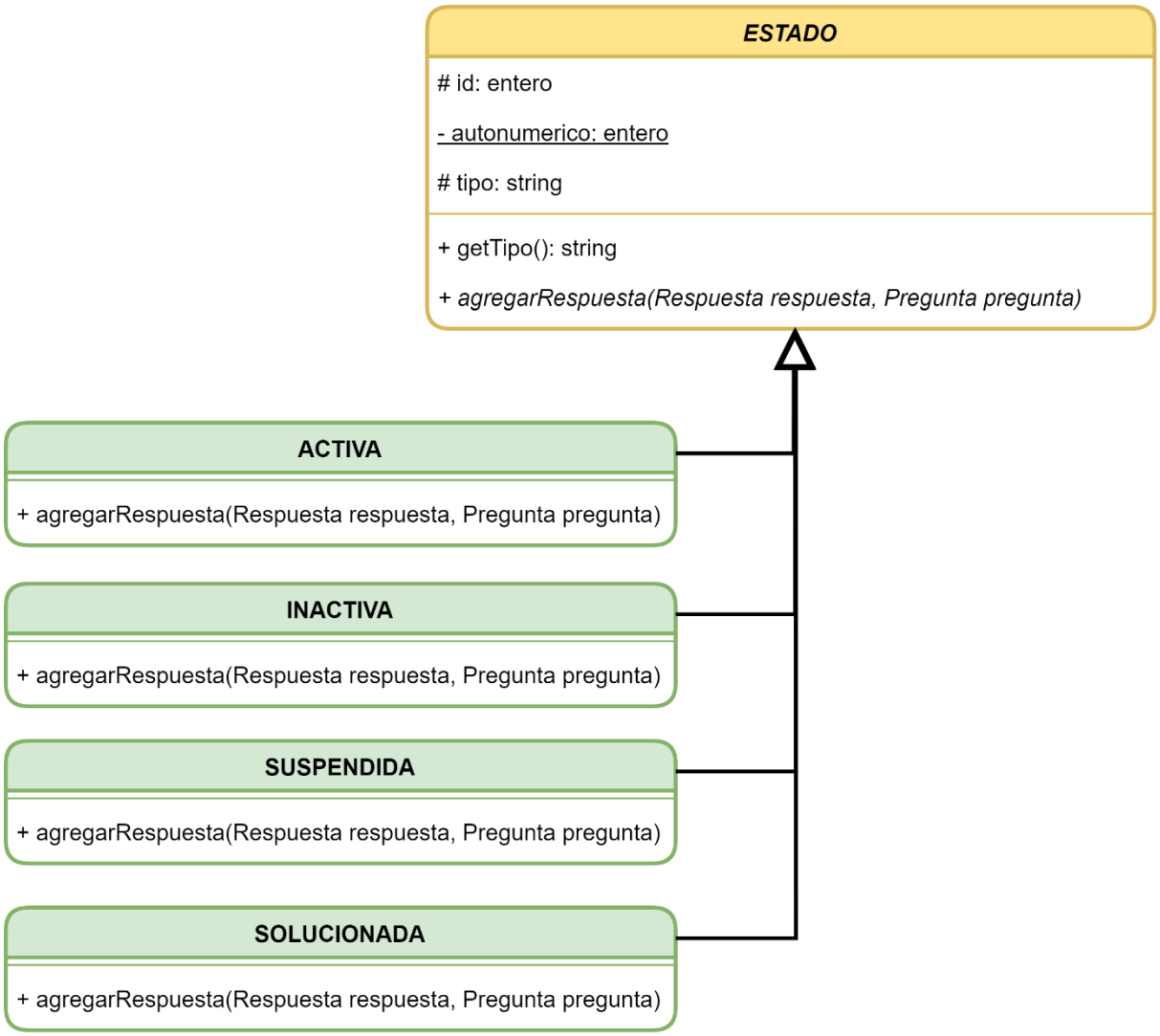
El método abstracto es el más importante en este caso:

- **agregarRespuesta(Respuesta respuesta, Pregunta pregunta):** este método de la clase abstracta es heredado por cada clase derivada (correspondiente a cada estado, según el Patrón de Estado), y es redefinido dependiendo de la subclase, lo cual se describe a continuación:
  - **Activa:** el método agrega una respuesta que recibe como parámetro a la lista **respuestas** de la Pregunta, que también recibe como parámetro.

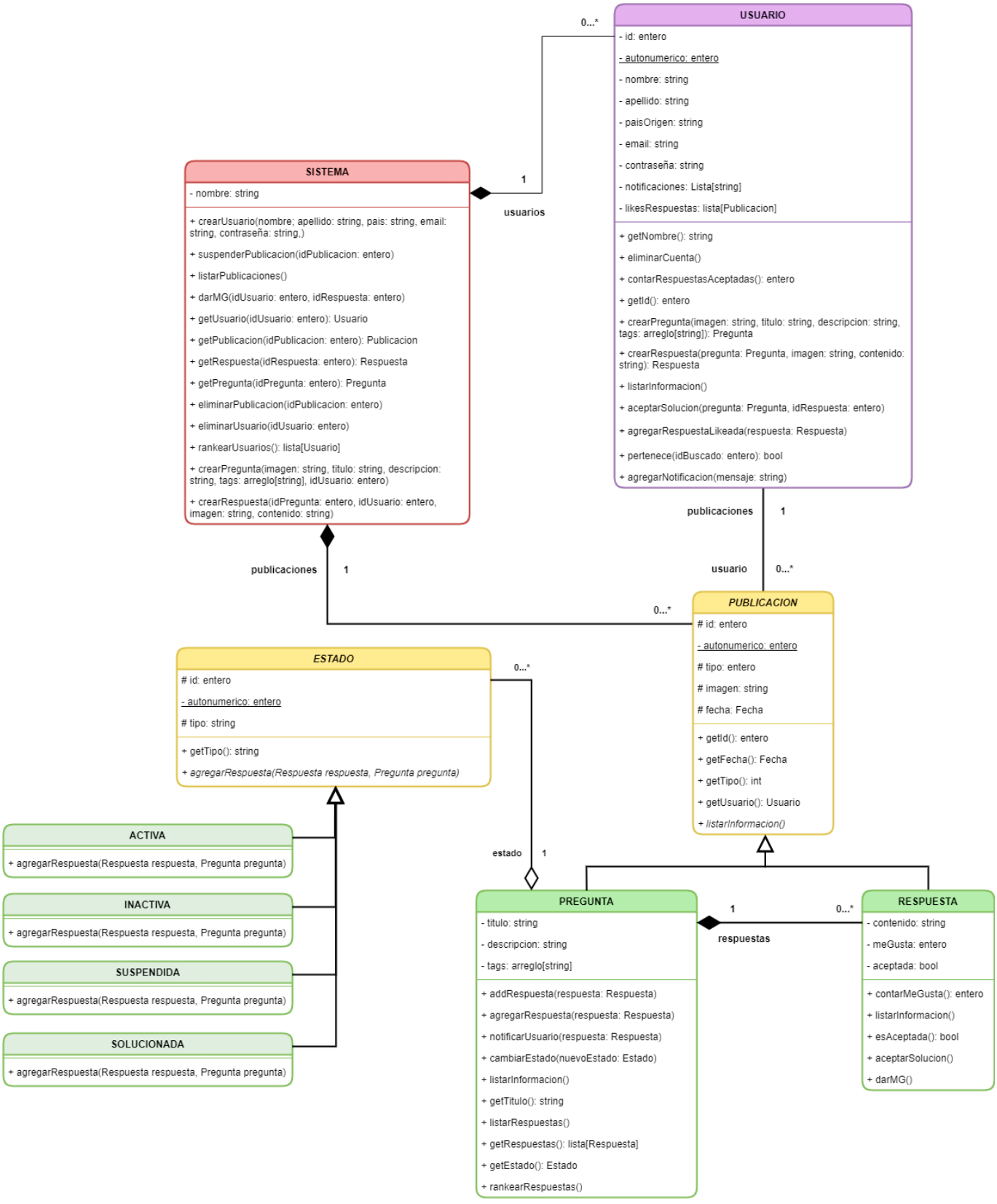
- Inactiva: el método, además de agregar la respuesta que recibe como parámetro la lista respuestas de la Pregunta (que recibe también como parámetro), cambia el estado de la Pregunta a Activa.

- Suspendida: en este caso, el método cambia el estado de la Pregunta a Activa, y vuelve a ejecutar el método en el nuevo estado.

- Solucionada: en este caso, el método indica que no se admite agregar respuesta a la pregunta, puesto que está solucionada.



3.2.Diagrama de clases (UML)



Para ver el diagrama de clases completo, ingresar al siguiente enlace.

3.3. Detalles de la implementación (C++)

Para llevar a cabo la implementación de nuestro proyecto, optamos por utilizar el lenguaje de programación **C++** en conjunto con el entorno de desarrollo **Eclipse**. C++ es un lenguaje de programación de alto rendimiento y ampliamente utilizado, que ofrece una amplia gama de características, incluyendo la programación orientada a objetos (**POO**) lo cual utilizamos bastante en nuestro proyecto. Por otro lado, Eclipse es un entorno de desarrollo integrado (IDE) que proporciona una plataforma robusta y herramientas eficientes para facilitar el proceso de desarrollo de software.

En nuestra metodología de organización del proyecto, adoptamos un enfoque que se basa en la utilización de archivos con extensión "**.h**" para la declaración de clases, junto con la especificación de sus atributos y métodos. Estos archivos ".h" actúan como interfaces, proporcionando una visión clara de la estructura de nuestras clases. Por otro lado, los archivos de extensión "**.cpp**" albergan la implementación concreta de las clases y sus métodos, permitiendo separar claramente la **interfaz** de la **implementación**.