PROGRAMACIÓN ORIENTADA A OBJETOS II

TRABAJO GRUPAL FINAL

SISTEMA DE ALQUILERES TEMPORALES

INTEGRANTES:

Cabral Joel (Comisión 2)

- joel.c.98@hotmail.com

Deo Matias Francisco (Comisión 1)

- matiasfd.deo@gmail.com

Troche Leonardo Martin (Comisión 1)

- leo.m.troche@gmail.com

SITIO WEB

- **Decisiones de diseño:**

Por un lado, decidimos separar al **sitio web** en 3 partes para evitar sobrecargarlo de responsabilidad, una que maneja las diversas opciones (**ManagerDeOpciones**) tales como categorías, servicios y tipos de inmueble, y cualquier otra que se pueda añadir en un futuro; por otro lado, un **RecolectorDeDatos** que permite acceder a la **información de gestión** del sitio web de manera directa, con la función que un administrador luego pueda utilizarla; y por último, el **SitioWeb** que representa al sitio web como tal, que maneja a los **usuarios** y permite realizar **búsquedas** con los alquileres de todo el sitio.

- **Detalles de implementación:**

El recolector de datos conoce a Sitio Web de manera que puede recolectar la información directamente de él, lo cual es escalable a futuro.

- **Patrones de diseño:**

Ninguno a destacar.

BÚSQUEDA

- **Decisiones de diseño:**

Para la **búsqueda**, decidimos **no** utilizar el patrón Composite para simplificar mucho su funcionamiento. De esta manera el objeto **ListaDeFiltrosPredeterminada** se encarga de **obligar (**por constructor) el uso de los filtros ciudad, fecha de inicio, y fecha de salida, y luego efectúa el **filtrado** con **todos** los filtros que posee (con **intersecciones**). También se pueden **añadir** más filtros luego, pero no vimos la necesidad que pueda añadir a objetos de su misma clase, ya que con tal de añadir más filtros comunes se llega al mismo resultado de una manera más simple (haciendo lo contrario, deberías volver a ingresar los mismos filtros obligatorios, porque si se ingresara un filtro con un valor distinto devolvería siempre vacío, y de cualquier manera ya se pueden añadir otros filtros simples manualmente).

El filtrado dentro de **ListaDeFiltrosPredeterminada** se realiza de forma que cada alquiler se comprueba con los filtros que haya en la lista de filtros, y de esta manera aquellos alquileres que se mantengan a través de todos ellos serán los resultados del filtrado.

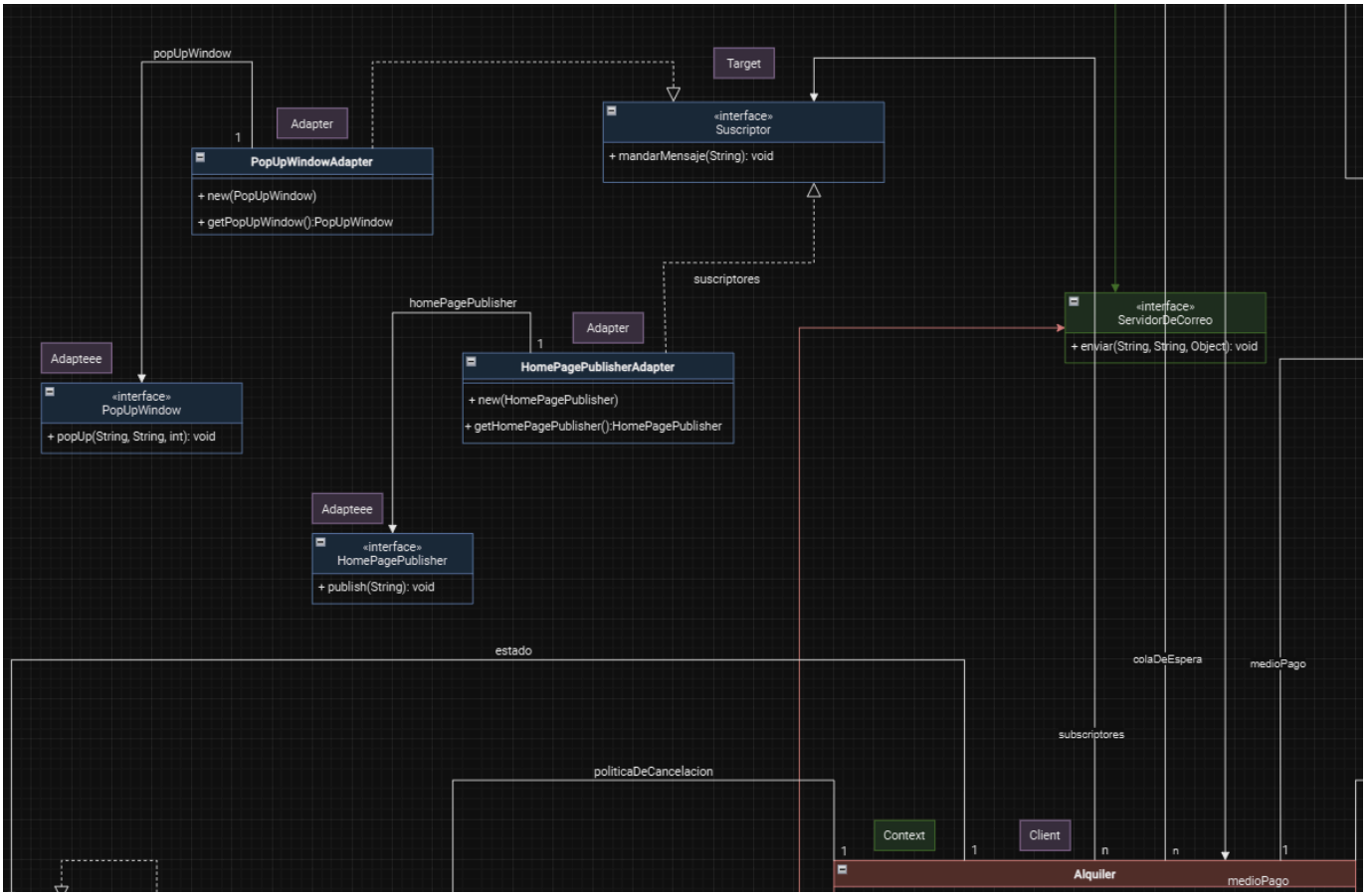
- **Detalles de implementación:**

La búsqueda se realiza con intersecciones (con filtros “**and**”), de forma tal que se puede agregar filtros para buscar alquileres que cumplen sólamente con lo pedido y **nada más**. Esto se realiza a través de una **iteración** sobre los alquileres, donde cada uno **debe pasar** el filtro especificado, que responde con un booleano. Esto supera a la anterior implementación, donde cada filtro recibía los alquileres del anterior, iterando muchas más veces en promedio.

NOTIFICACIONES

- **Patrones de diseño:**

Para las notificaciones, pensamos en utilizar un patrón **Object Adapter** para las **interfaces fuera** de nuestro dominio, pero requerían ser adaptadas a él. De esta manera, ambas **PopUpWindow** y **HomePagePublisher** pueden ser suscritas al sitio web (concretamente a las publicaciones de alquileres).



Se utilizaron **2 adapters**:

- **PopUpWindowAdapter** que se adapta a **PopUpWindow.**
- **HomePagePublisherAdapter** que se adapta a **HomePagePublisher.**

Dado que ambos son **Object Adapters**, esto implica que contienen a PopUpWindow y a HomePagePublisher como variable de instancia, respectivamente.

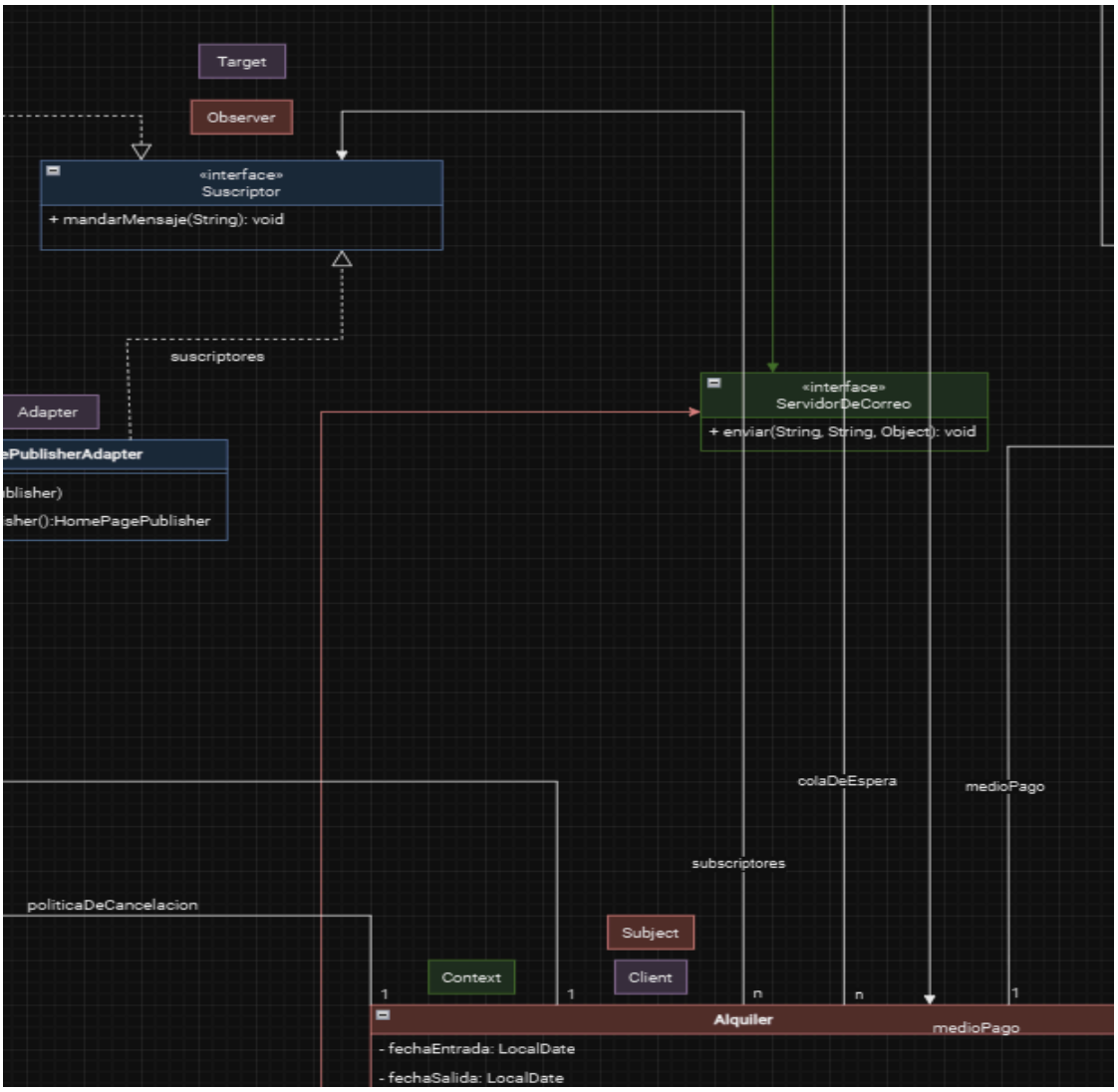
Tanto **PopUpWindow** como **HomePagePublisher** son **Adaptees** (respectivamente de su adapter)**.**

El **Target** para ambos es la interface **Suscriptor,** a la que se deben adaptar para poder suscribirse a **Alquiler.**

El **Client** es la clase **Alquiler**, que representa a las publicaciones de alquileres, a las que se suscriben las clases que implementan a la interface Target **Suscriptor**.

Por otro lado, se utiliza al patrón **Observer** para notificar a los suscriptores de **Alquiler (Subject)** cuando ocurre uno de los siguientes **eventos** (donde Alquiler es quien se encarga de avisar):

- Cancelación de un inmueble.
- Reserva de un inmueble.
- Baja de precio precio de un inmueble.



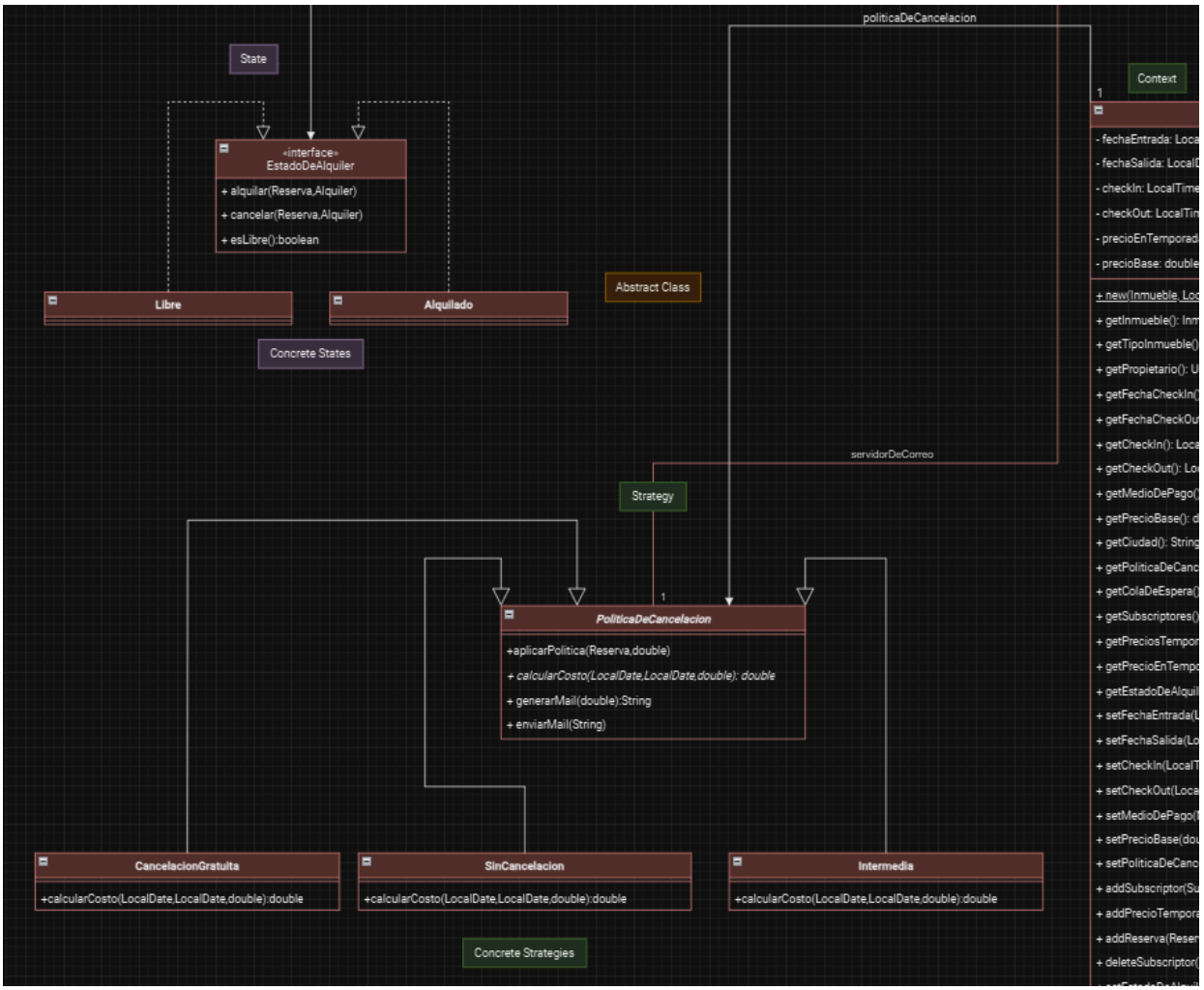
De esta manera, **todos los suscriptores (Observers)** se enteran de estos eventos, pero cada uno luego los utiliza como prefiera.

POLÍTICAS DE CANCELACIÓN

- **Patrones de diseño:**

Inicialmente, se adoptó el patrón de diseño **Strategy** para implementar las políticas de cancelación, utilizando una interfaz común para todas las políticas. Sin embargo, durante el proceso de desarrollo y diseño de las clases correspondientes a cada política, se observó que la mayoría de los métodos contenían **código equivalente**, con excepción de **uno** en particular donde se realizaban los cálculos específicos que diferenciaban a cada política.

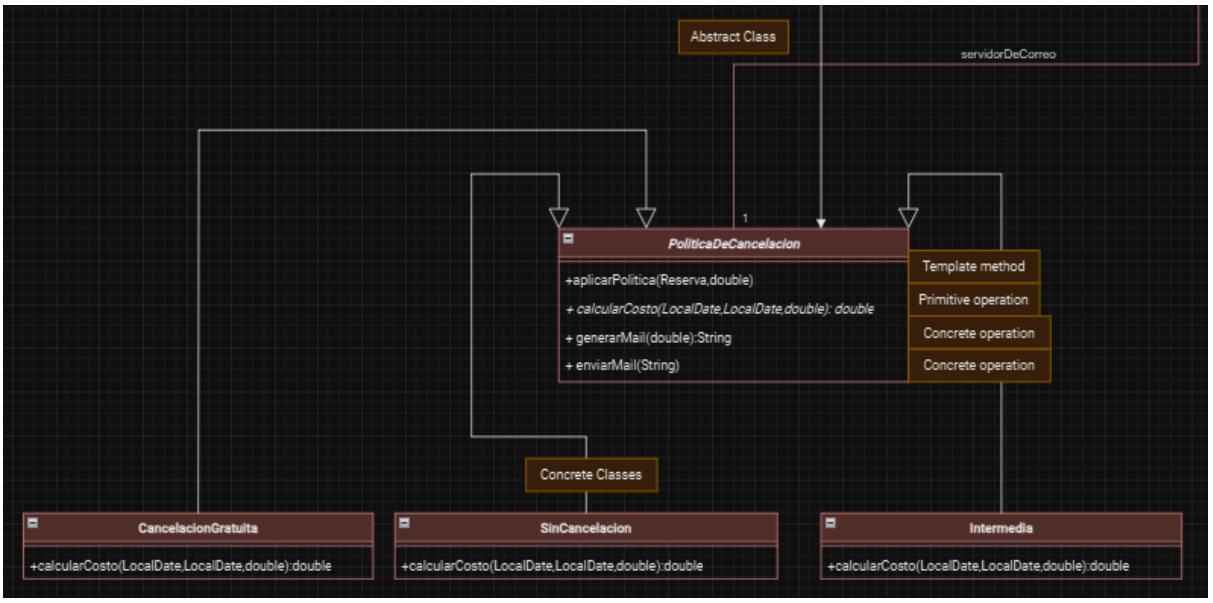
A raíz de esta repetición de código, se decidió refactorizar la solución sustituyendo la interfaz por una **superclase abstracta**, que centraliza la implementación del código común a todas las políticas. De esta manera, se combinó el patrón **Strategy** con el patrón **Template Method**, permitiendo que las políticas compartieran la implementación común, mientras que cada una de ellas podía definir el comportamiento específico de los cálculos en el método correspondiente.



Se utilizaron 3 **estrategias (ConcreteStrategy)**:

- **CancelaciónGratuita**: Hasta 10 días antes de la fecha de inicio de la ocupación y luego abona el equivalente a dos días de reserva.
- **SinCancelación**: en caso de cancelar el usuario de todas formas paga los días que había reservado.
- **Intermedia**: Hasta 20 días antes es gratuita, entre el día 19 anterior y el día 10 anterior paga el 50 %, después del décimo día paga la totalidad.

Donde **Alquiler** es el **Contexto** de este patrón, y la clase abstracta **PolíticaDeCancelación** es **Strategy**.



Se utiliza un **Template Method** combinado con el mencionado **Strategy** para las politicas de cancelacion:

Se crea una clase abstracta (**AbstractClass**) **PoliticaDeCancelacion** la cual implementa las **concrete operations** aplicarPolitica**,** generarMail y enviarMail ya que las distintas políticas que heredan de estas clase utilizan estos métodos de la misma manera. El método **calcularCosto** se utiliza como método abstracto para que sus subclases (**CancelaciónGratuita**, **SinCancelación** e **Intermedia**) puedan implementarlo de la manera que le corresponda a cada política, cada una correspondiendo al lugar de **ConcreteClass**.

ALQUILER e INMUEBLE

- **Decisiones de diseño:**

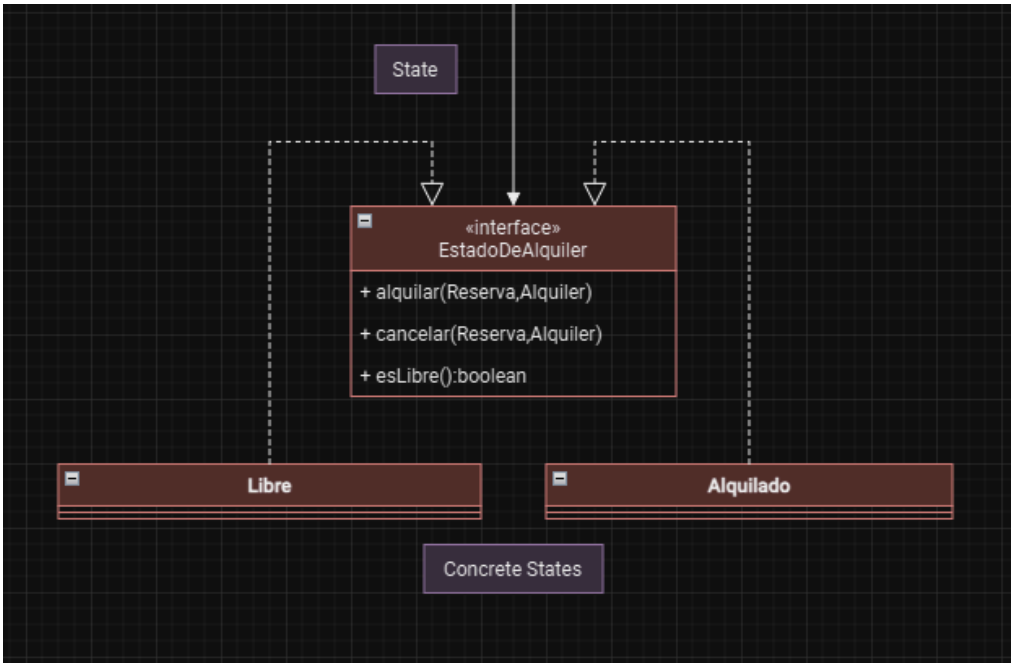
Decidimos que la clase **Alquiler** represente a las **publicaciones** realizadas por propietarios, a las que se pueden suscribir agentes externos, y también pueden ser alquilados por posibles inquilinos. De esta manera, depende de si este está libre o no, se encargará de permitir a un inquilino u otro, esperando la confirmación del dueño en cada caso.

Por su parte, la clase **Inmueble** representa al inmueble físico del propietario, que posee ciertos atributos ligados a describirlo. La publicación de este en el sistema proviene desde él. Cabe destacar que Inmueble conoce a la Interface **Foto**, que permite abstraerse de cómo esta se implemente luego.

- **Patrones de diseño:**

Para los **estados** de **alquiler** decidimos usar el patrón **State**, para poder manejar de forma escalable el código que dependía de ciertos momentos precisos en el que se encontraba el alquiler, evitando los condicionales excesivos.

Decidimos optar por dos estados**, Libre** y **Alquilado** creados como clases que aplican una interface **EstadoDeAlquiler**, las cuales permiten gestionar al alquiler a través de los métodos alquilar y cancelar de la interface.



Entonces, según **Gamma**:

- **Context** → Alquiler
- **State** → EstadoDeAlquiler (interface)
- **ConcreteState** → Libre, Alquilado

- **Detalles de implementación**

En un primer momento, se decidió que las clases encargadas estos fuesen responsables tanto del cambio de estado del alquiler como de la gestiones asociadas, como el **guardado**, la **confirmación** y la **cancelación** de las reservas. Sin embargo, a medida que se identificó una alta frecuencia de llamadas a los métodos relacionados con la gestión del alquiler, se optó por un **rediseño**.

En lugar de que las clases de estado manejen directamente estas operaciones, se decidió **delegar** la responsabilidad de ejecutar el código relacionado con la gestión del alquiler a la propia clase **Alquiler**. Los estados, en este nuevo enfoque, se encargan exclusivamente de **determinar** qué acción o código debe ejecutarse, manteniendo así la lógica de **control** separada de la gestión de datos. Este cambio permite una mejor **organización** del código y facilita la extensión o modificación de las operaciones sin afectar a los estados directamente.

USUARIOS

- **Decisiones de diseño**

En un principio, se planteó separar los usuarios en **Inquilino** y **Propietario**, y que éstos hereden la funcionalidad básica de una **clase abstracta de Usuario**.
Sin embargo, al analizar los requisitos del sistema, se observó que los roles de inquilinos y propietarios no son estáticos, ya que un propietario puede alquilar un inmueble y un inquilino puede ofrecer inmuebles para alquilar.
Por lo tanto, se optó simplificar el diseño y **crear una única clase concreta Usuario** que cubriera todas las funcionalidades necesarias para ambos roles.

- **Detalles de implementación**

Se optó por utilizar listas separadas para los inmuebles y los alquileres registrados, ya que, según los requisitos del sistema, un Usuario puede registrar un inmueble sin necesariamente alquilarlo. Es decir, el hecho de **registrar un Inmueble no implica que esté en alquiler**. En cambio, cuando un inmueble es **registrado como Alquiler**, significa que **está disponible para ser arrendado por otros usuarios**.

- **Patrones de diseño**

Para la representación de los usuarios en el sistema, no se empleó ningún patrón de diseño específico.

RESERVAS

- **Decisiones de diseño**

Se decidió utilizar un **ServidorDeCorreo** dentro de la clase **Reserva** para desacoplar la lógica de envío de correos, lo que facilita el cumplimiento del principio de Responsabilidad Única (**Single Responsibility Principle**) de los principios **SOLID**. Este enfoque permite que la clase Reserva se enfoque en su funcionalidad principal, mientras que el servidor de correo maneja la tarea específica de enviar los correos.

Por otro lado, la clase **Reserva** puede tener diferentes comportamientos según su estado (como **PendienteDeAprobacion**, **Vigente**, **Finalizado**, **Cancelado** o **EnCola**). Para gestionar este cambio de comportamiento de manera más eficiente, se optó por aplicar el **patrón de diseño State**, que permite modificar el comportamiento de un objeto en tiempo de ejecución según su estado.

- **Detalles de implementación**

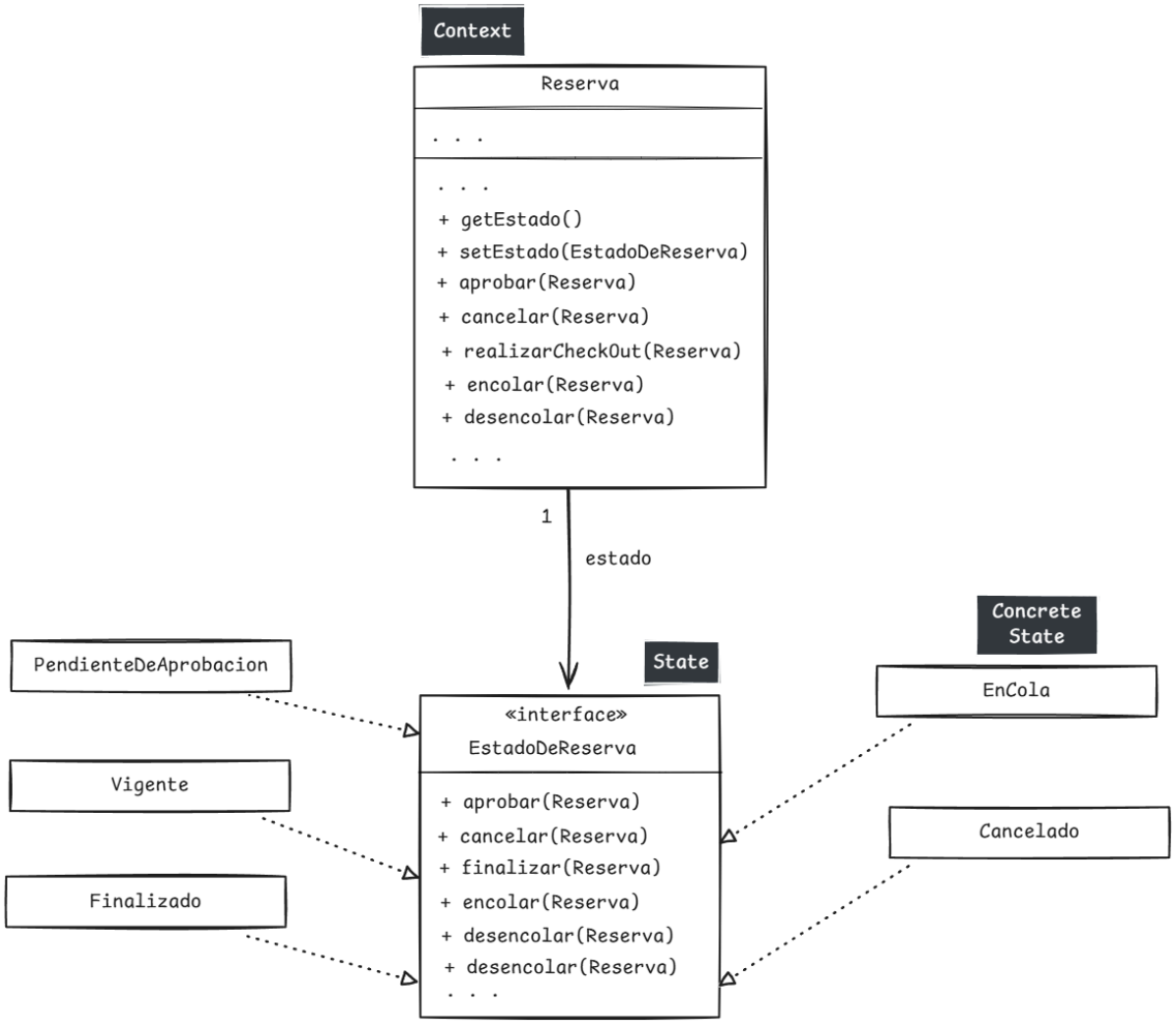
Los métodos **enviarCorreoDeAprobacion** y **enviarCorreoDeCancelacion** son responsables de tareas muy específicas, como enviar un correo con la información correspondiente. Para simplificar su implementación, se forzó el mismo asunto para los mensajes y se incluyó la información de la reserva directamente en el cuerpo del correo.

- **Patrones de diseño**

En el contexto de la clase **Reserva**, se decidió implementar el **Patrón de Diseño State** para manejar los distintos estados de una reserva, como **PendienteDeAprobacion**, **Vigente**, **Finalizado**, **Cancelado** y **EnCola**. Este patrón es útil cuando un objeto puede cambiar su comportamiento dependiendo de su estado interno, permitiendo que el mismo objeto responda de manera diferente a los mismos métodos según el estado en que se encuentre. En lugar de escribir condiciones if o switch en el código, el patrón State delega el comportamiento a objetos de estado especializados, lo que mejora la flexibilidad y mantenibilidad del sistema.

El objetivo principal de aplicar este patrón es simplificar la gestión del cambio de comportamiento de la clase Reserva según su estado, sin tener que modificar el código cada vez que se añade un nuevo estado o comportamiento. Al utilizar este patrón, el comportamiento de la reserva es más modular y cada estado puede ser tratado como una clase independiente que se ocupa de su lógica de manera encapsulada.

El **UML** que representa a la **Reserva y sus estados**, según los **roles** definidos en el libro de **Gamma et al**., es el siguiente:



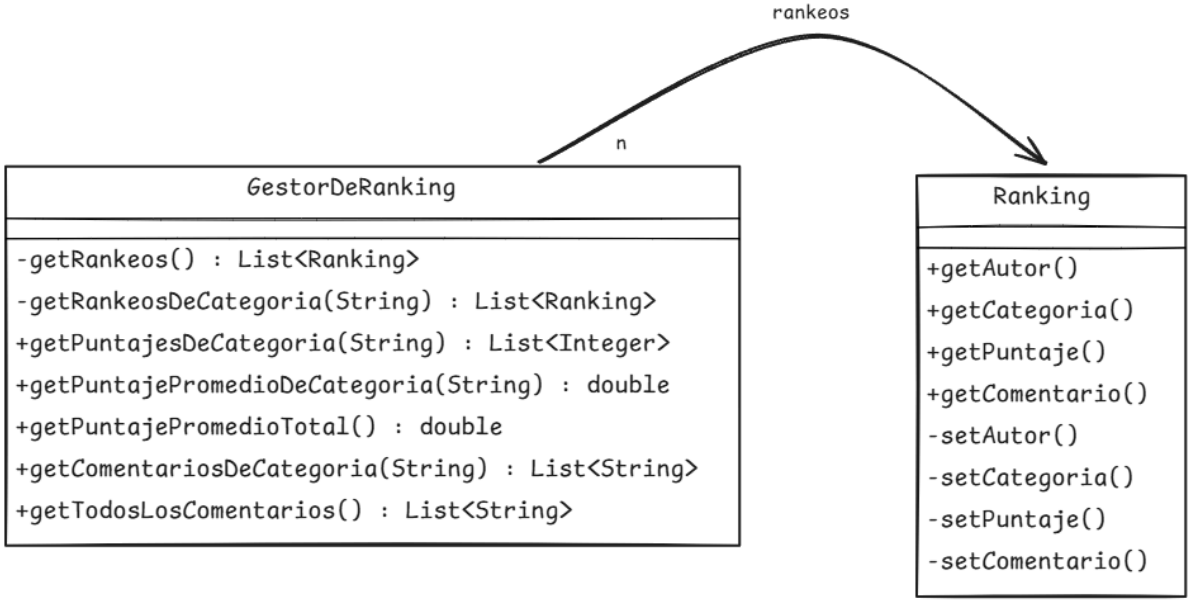
- **Context** → Reserva
- **State** → EstadoDeReserva (Interface)
- **Concrete State** → PendienteDeAprobacion, Vigente, Cancelado, Finalizado, EnCola

RANKING

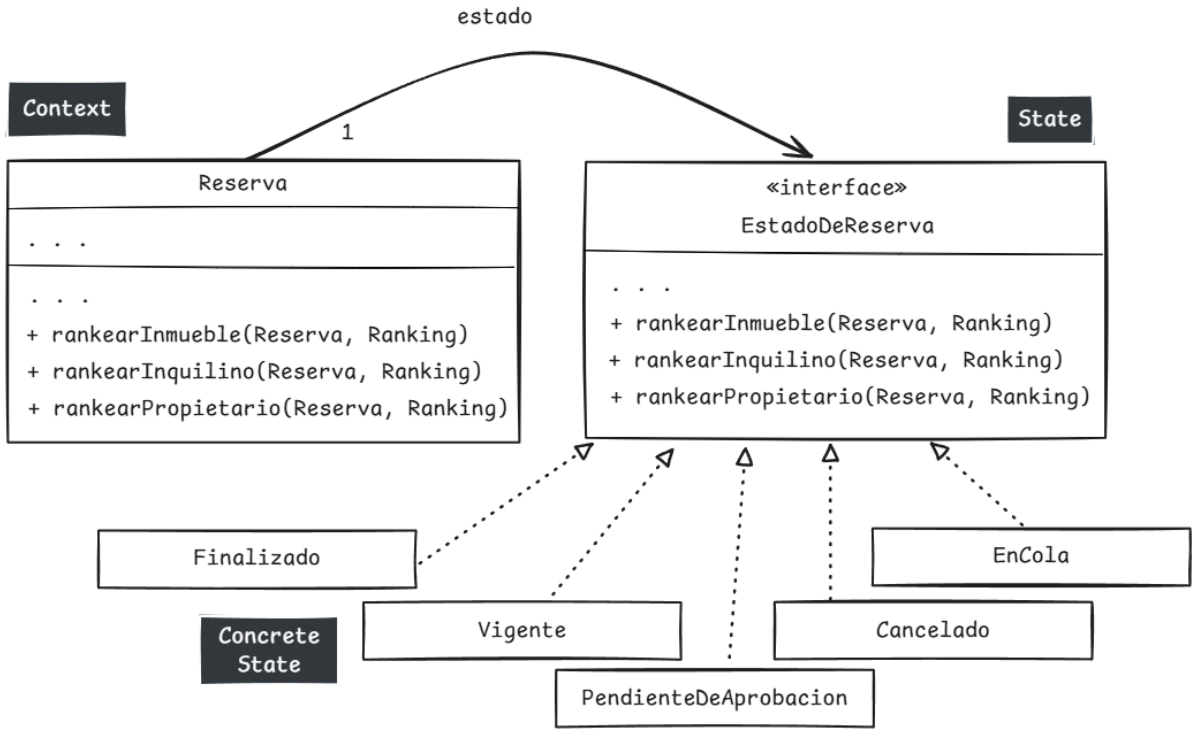
- **Decisiones de diseño**

En un principio, se consideró la creación de una interfaz **Rankeable** que sería implementada tanto por la clase **Usuario** como por la clase **Inmueble**. Sin embargo, al analizar la implementación, nos dimos cuenta de que la lógica sería prácticamente idéntica en ambas clases. **Esto nos llevó a cuestionarnos** si sería adecuado que ambas clases heredaran esos métodos comunes desde una clase abstracta.

Al no encontrar un concepto común en el dominio que justificara la herencia entre Usuario e Inmueble, decidimos replantear la solución. Optamos por crear un **GestorDeRanking**, una clase encargada de gestionar toda la lógica relacionada con el ranking de estas entidades. De esta manera, tanto **Usuario** como **Inmueble** reciben una instancia de **GestorDeRanking** a través de su constructor, delegando en esta clase la responsabilidad del manejo del ranking, lo que mantiene una separación clara de responsabilidades (**Single Responsibility Principle**).



Al diseñar los métodos **rankearInmueble**, **rankearInquilino** y **rankearPropietario**, que dependen del estado de la reserva (ya que solo se puede realizar un ranking después de que se haya hecho el checkout), se decidió incorporar estos métodos tanto en la clase **Reserva** como en la clase **EstadoDeReserva**. Esto permite que el ranking sólo pueda realizarse cuando la reserva se encuentra en el estado **Finalizado**, asegurando que la lógica de negocio esté estrictamente controlada según el estado de la reserva.



- **Detalles de implementación**

Inicialmente, se consideró que la clase **GestorDeRanking** debería contener un **Map<String, List<Ranking>>** para organizar los rankings por categorías. Sin embargo, esta solución añadiría una mayor complejidad a la implementación sin un beneficio claro, ya que el sistema no requiere devolver todos los rankings de una entidad, sino solo aquellos correspondientes a una categoría específica o un promedio general.

Por lo tanto, se optó por una implementación más sencilla utilizando una única **List<Ranking>,** que almacena todos los rankings realizados para una entidad. Luego, cada método de la clase **GestorDeRanking** se encarga de filtrar esta lista según los parámetros solicitados (por ejemplo, por categoría o promedio general), devolviendo solo la información relevante según el caso.

- **Patrones de diseño**

Para la representación del ranking de las entidades en el sistema, no se utilizó ningún patrón de diseño específico.