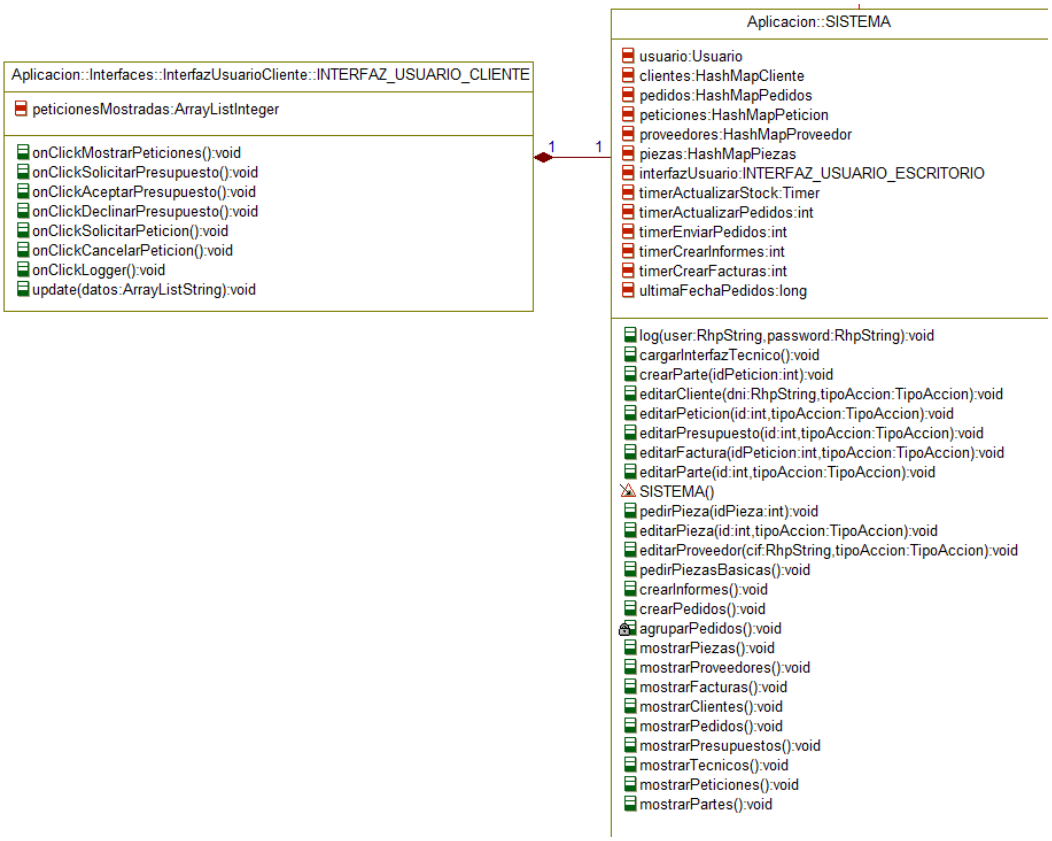
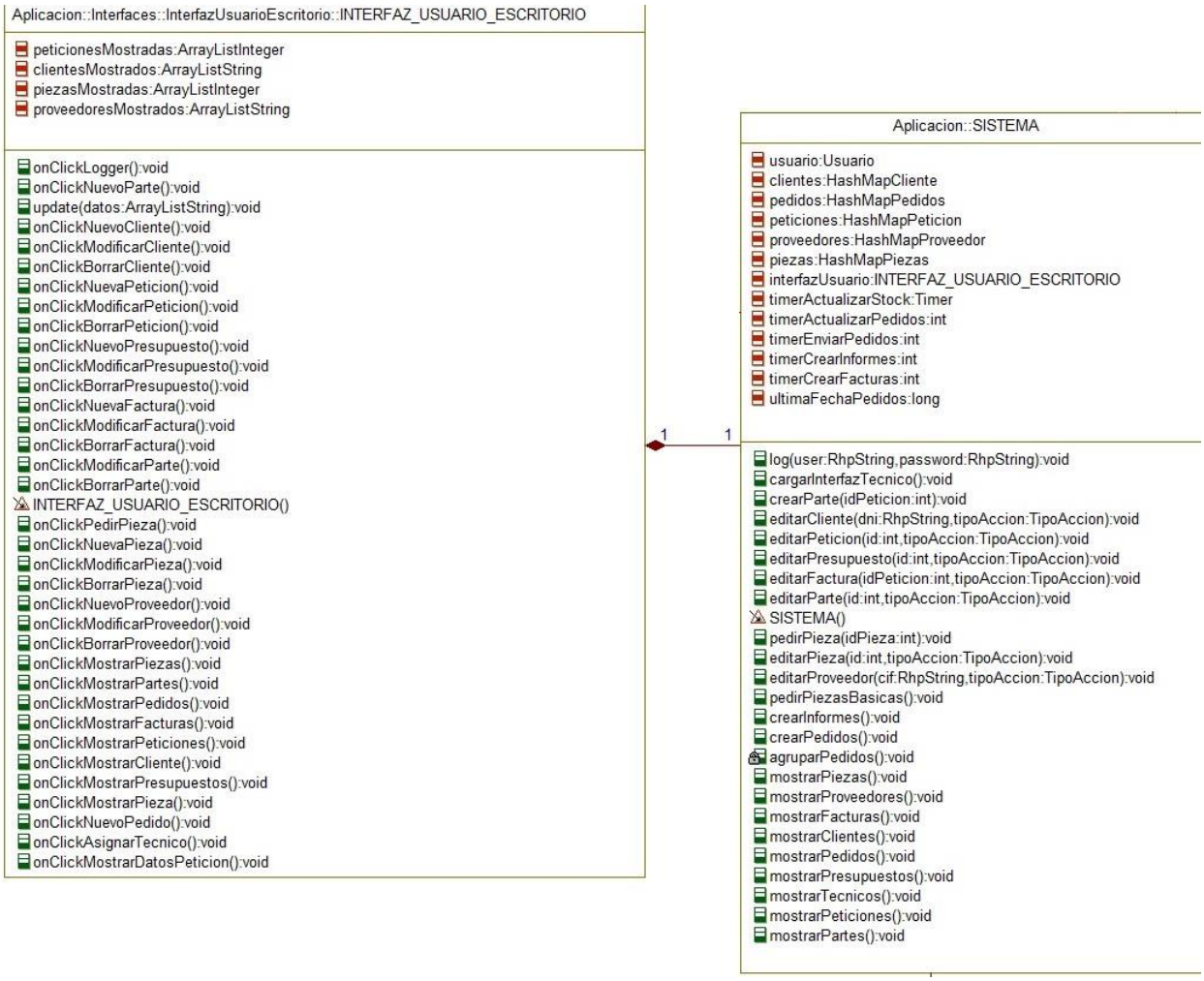
12 Aplicación de patrones

Hemos aplicado los patrones sw en aquellos casos en los que hemos encontrado alguno de los problemas de asignación de responsabilidades vistos en teoría. Hemos analizado de qué problema se trataba para poder saber qué patrón poder aplicar para resolverlo.
Como línea general hemos procurado en todo momento mantener bajo el acoplamiento entre clases y alta la cohesión dentro de las mismas. Para lograr esto hemos intentado aislar dentro de cada clase aquella funcionalidad que estuviera fuertemente relacionada. En el caso de las clases propias del dominio de trabajo esto se correspondía con las entidades del modelo entidad relación que actúan sobre sus propios datos. En el caso de las clases auxiliares hemos creado un controlador que es la clase Sistema que actúa como una interfaz común para todas las clases auxiliares.

12.1 Patrón vista-controlador.

Lo implementamos entre la clase sistema y las clases de la interfaz. De modo que aquellos cambios que sean realizados por el usuario en la interfaz lleguen a la clase sistema desde una única en vez de llegar desde cada una de las que estén formando parte de la interfaz. De este modo se reduce el acoplamiento entre clases ya que el dominio solo se comunica con un único controlador de la interfaz y no con cada clase que la esté formando.
Todas las clases de interfaz que tengan algo que comunicar al dominio deberán hacerlo a través del controlador.





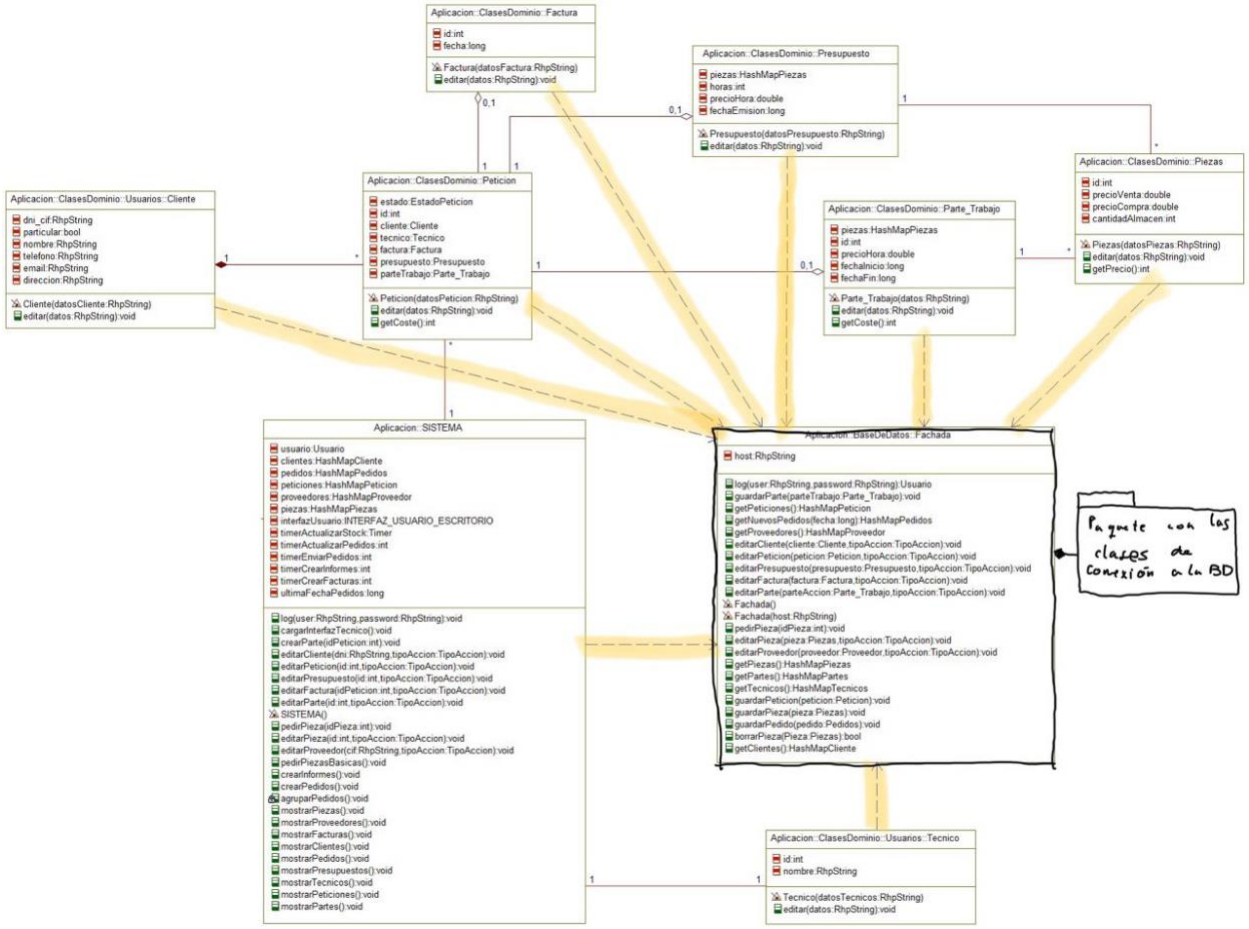
Aplicando este patrón conseguimos que el sistema sea independiente de las interfaces, recibe información de ellas a partir de un controlador. También podría no existir el paquete de interfaces.

12.2 Patrón fachada.

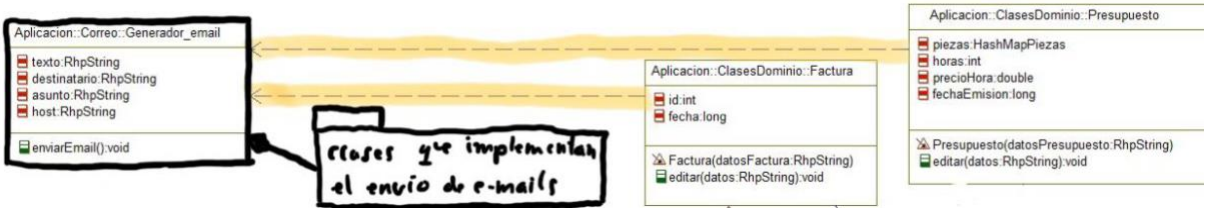
Este patrón da solución a la comunicación entre clases de distintos paquetes. Cuando las clases de dominio requieren de utilizar un grupo de clases que se encuentran implementando una funcionalidad común (por lo general estas clases estarán en un mismo paquete) deberán evitar utilizar dichas clases de forma directa. Para ello la clase fachada deberá resumir toda la funcionalidad que el paquete deba proporcionar al exterior de modo que para usarlo solo sea necesario acceder a la clase que haga de fachada con el exterior de modo que se reducirá el acoplamiento entre clases.

Lo utilizaremos en el caso de la comunicación con la base de datos. Las clases que implementen la comunicación con la base de datos pueden ser variadas y complejas, pero gracias a la fachada dicha comunicación quedará enormemente simplificada. Las clases de dominio solo necesitan comunicarse con la fachada para poder utilizar la base de datos y será la fachada la encargada de llamar a las clases del paquete en el que se implemente la comunicación con ella de modo que se realice aquello que se desea.

También utilizamos este patrón entre las clases Sistema y Gestor de archivos para ocultar la implementación de cómo los informes se almacenan y comparten en formato JSON a través del servidor.
Por último, este mismo patrón lo aplicamos en el gestor de correos para ocultar también las clases del interior de este paquete.



Al aplicar este patrón la única forma de acceder al paquete con las clases de conexión a la base de datos es a través de la fachada.



Del mismo modo, al aplicar este patrón la única forma de acceder a las clases que implementan el envío de emails es mediante la fachada.

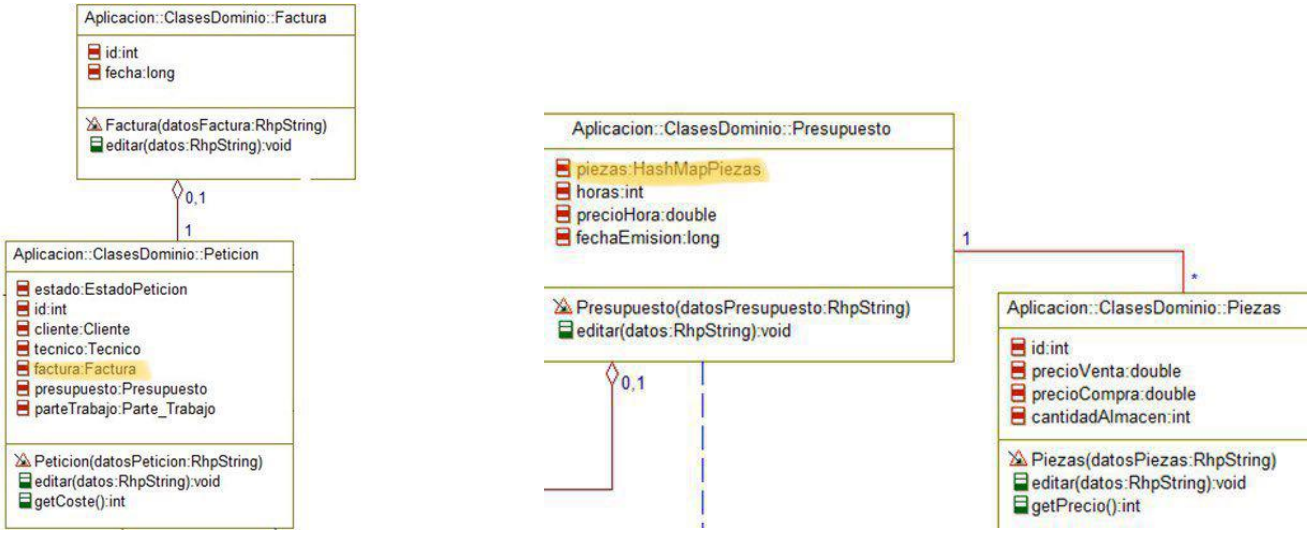
Así logramos reducir el acoplamiento entre clases.

12.3 Patrón listener - observer para comunicación indirecta.

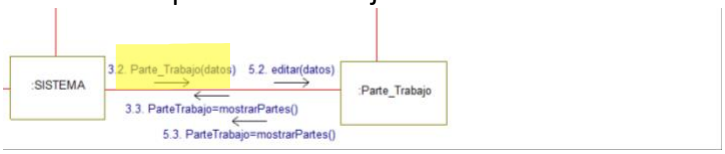
Se implementa un patrón similar a este entre la clase timer y la clase sistema. El Sistema creará un timer al que se subscribirá indicándole un tiempo y un método suyo en el que quiere recibir una llamada cuando dicho tiempo haya transcurrido. El timer tendrá una referencia del método que debe llamar y realizará dicha llamada cuando el tiempo finalice.
De este modo el Sistema podrá suscribirse o cancelar la suscripción de métodos suyos para recibir llamadas cuando lo deseé en tiempo de ejecución.
Al estar esta funcionalidad fuera de la clase sistema en el caso de ser necesario otras clases podrían aprovecharse de ella en un futuro también y realizar suscripciones del mismo modo.

12.4 Patrón creador.

Las clases que contengan como uno de sus atributos una colección de otra clase serán candidatas para asumir el rol de crearlas.

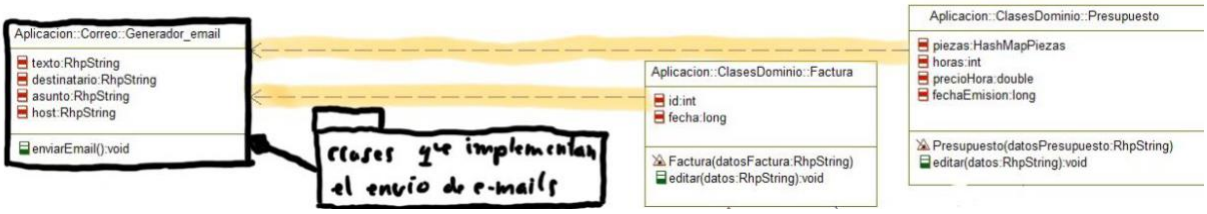


También puede verse esto mismo en los diagramas de colaboración. El sistema contiene una colección de partes de trabajo los cuales son creados desde él.



12.5 Patrón experto.

Este patrón se encarga de solucionar el problema de quién se encargará de asignar responsabilidades a los objetos. Es más, una idea que una nueva implementación, ya que para solucionar el problema hacemos uso del controlador y los creadores.
Este patrón es observable en todas las comunicaciones con las fachadas.



Las clases se guardan a si mismas en la base de datos, o en este caso envían sus datos por correo a quien corresponda. Otra clase no debería tener acceso a sus datos privados por la necesidad de tener que incorporarlos al correo o a la base de datos si esto puede evitarse.

Adicionalmente el patrón se observa con mayor claridad con los diagramas de colaboración, aquí se aprecia cómo la pieza es la que se guarda a si misma cuando es creada. Ocurriría lo mismo antes de ser borra o cuando esta es modificada ya que la pieza es la única que debería conocer todos sus datos. Esto aumenta la cohesión centro de las clases.