

**UNIVERSIDAD PRIVADA BOLIVIANA**

**FACULTAD DE INGENIERÍA Y ARQUITECTURA**

**CARRERA DE SISTEMAS COMPUTACIONALES**

**PRIVACIDAD, ARREBATADA U OLVIDADA**

**Estudiantes:** Carlos Andrés Coronado Arrazola

, Fabricio Fernández Jauregui

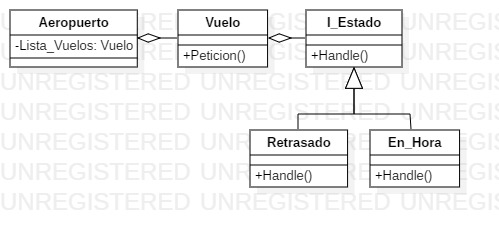
**Materia:** Patrones de Diseño

**Docente:** Raúl Antelo Jurado

**Cochabamba, marzo 2021**

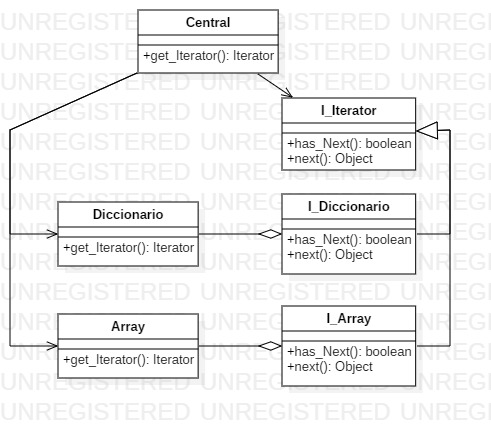
# Diagrama de clases genérico de cada patrón y la explicación del problema al que fue aplicado

## State



El patrón State fue aplicado para resolver el problema que plantea el cambio de estado de un vuelo. Como Central esta conectada a la entidad Aeropuerto, esta necesita saber la lista de vuelos, los cuales están almacenados en Aeropuerto, y a su vez necesita saber si estos están en hora o retrasados. Es de suponer que la clase que representa a contexto es el Vuelo, pues este es quien cambiara de estado si ciertas condiciones se cumplen, condiciones que serán calculadas en tiempo de ejecución.

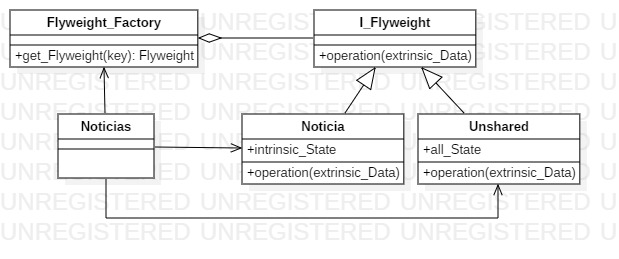
## Iterator



El patrón Iterator fue implementado para resolver el problema que había al momento de tener que recorrer las estructuras en las que se encontraban almacenadas las notificaciones provenientes del Gobierno Central y el Gobierno Departamental. Este patrón permite un acceso secuencial a los elementos de un objeto sin exponer su representación como tal, en otras palabras, independientemente de la estructura que sea y de la clase que provenga, si se tiene su iterator no habrá problemas al momento de obtener la información almacenada en la estructura, sin revelar los detalles de su implementación a quién desea recorrerla.

La clase llamada Diccionario representa a la estructura que usa el Gobierno Central y el array a la estructura utilizada por el Gobierno Departamental.

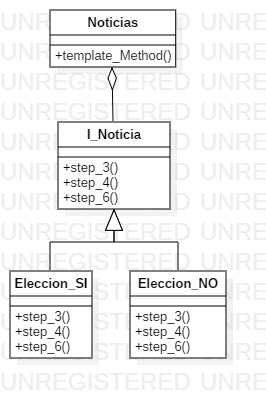
## Flyweight



El patrón Flyweight fue usado para resolver la clasificación de las noticias por categorías. La primera razón que se vio para implementar este patrón fue que este mismo permite organizar los elementos como si de una tabla de pertenencia se tratara (se marca las categorías a las que una noticia pertenece y se dejas las demás en blanco). La segunda razón fue el hecho de que si se implementara una instancia diferente de la misma noticia en cada categoría el programa consumiría muchos recursos y utilizaría demasiado especio de memoria con el tiempo, en cambio el patrón de Peso Ligero permite tener una sola instanciación a la que las listas a la que esta noticia es agregada apuntan, optimizando por mucho su uso de memoria y recursos.

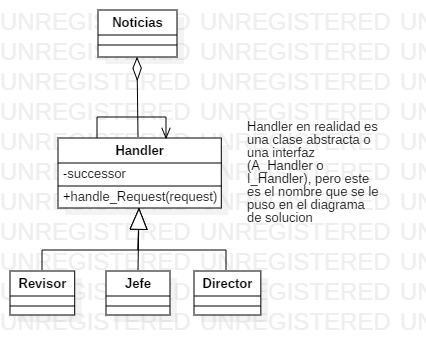
La clase Unshared solo esta para representar la forma genérica del diagrama de clases del patrón.

## Template



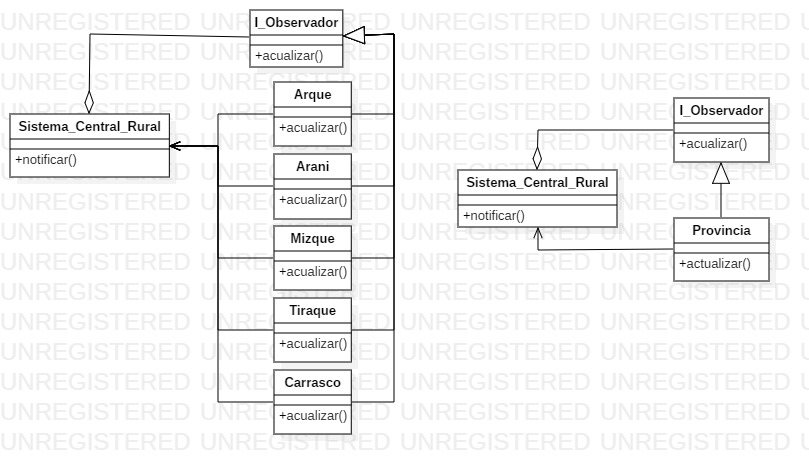
El patrón Template fue implementado para resolver las variaciones que había en la implementación del algoritmo de publicación de una noticia. En ciertos momentos durante la ejecución de este algoritmo el cliente tenía que escoger entre dos opciones, pero para el resto de los pasos no había esta opción, es debido a esto que se vio que el patrón Template encaja perfectamente ya que lo que este hace es ir difiriendo algunos pasos del algoritmo a las subclases para que estas las redefinan de la forma que se desea. En este caso, el problema planteaba una elección de si o no en ciertos pasos (steps), es por esto que se tiene dos subclases que implementan los mismos métodos, implementados de diferente manera, para determinados pasos (steps), dependiendo de la elección del cliente se ejecutará un método contenido en una o en la otra, esto en la clase Noticias que es donde se esta implementando el algoritmo de publicación de la noticia.

## Chain of Responsability



El patrón Chain of Responsability fue implementado para resolver el problema de la aprobación o el rechazo de la publicación de una noticia. El planteamiento del problema indica que quien decide si la notica es aprobada o no va a variar dependiendo de la cantidad de palabras que tiene la nota, por lo que no se tiene un conocimiento explícito de quien iba a resolver la cuestión. Como hay mas de un objeto que puede manejar esta petición y esto se tiene a determinar automáticamente, el patrón de Chain of Responsability resuelve de manera perfecta el problema que se planteó.

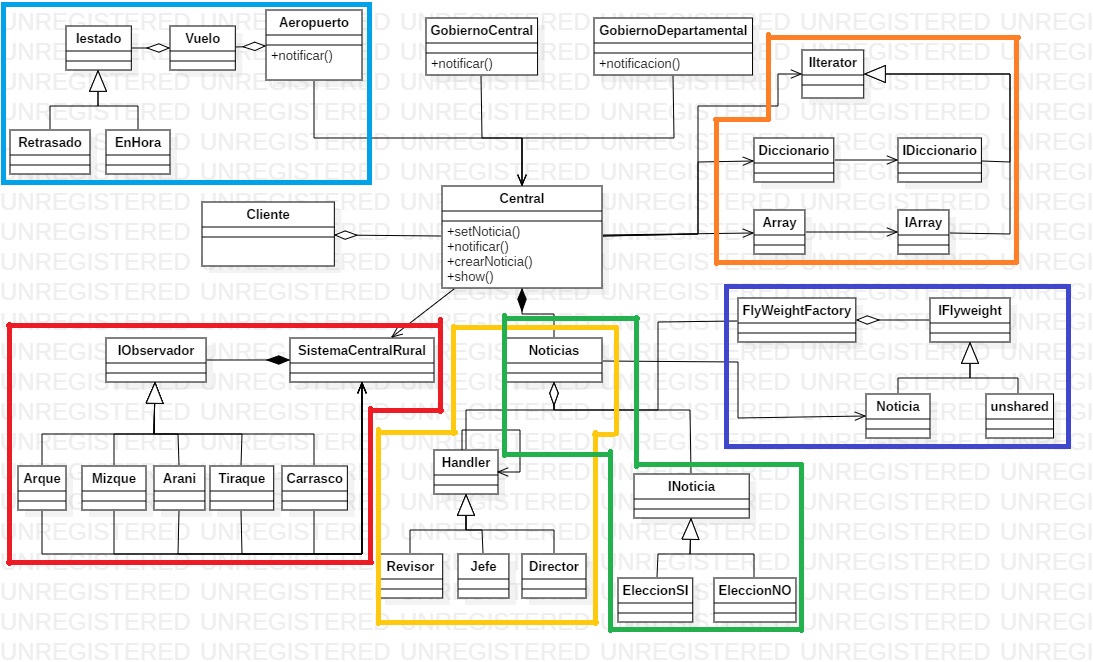
## Observer



El patrón Observer se implementó para resolver el problema de la notificación a las provincias de la zona rural cuando una noticia nueva era agregada, el problema igual plantea que los que reciben la notificación pueden escoger si desean ver la noticia completa o solo la notificación. El patrón Observer utiliza una dependencia de muchas clases a una, así cuando esta última clase cambia, todos los objetos dependientes de esta son notificados automáticamente, este comportamiento encaja perfectamente con lo que se quiere implementar. También cuenta con un modelo de extracción llamado pull en el que el observado solo envía nada que mas que la notificación mínima y ya de ahí los observadores preguntan de forma explicita los detalles, este modelo de extracción cumple con lo que pide el problema de manera perfecta igualmente.

Nota: Las dos representaciones en la imagen serían básicamente la misma, solo que por propósitos de que sea una representación mas intuitiva se uso la de la izquiera.

# Diagrama de Clases Global



* **Patrón State \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***
* **Patrón Iterator \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***
* **Patrón Flyweight \*\*\*\*\*\*\*\*\*\*\*\*\*\***
* **Patrón Template \*\*\*\*\*\*\*\*\*\*\*\*\*\***
* **Patrón Chain of Responsability \*\***
* **Patrón Observer \*\*\*\*\*\*\*\*\*\*\*\*\*\***