Concepción del diseño: un caso práctico

Listas de Correo

por
Leonardo Gassman

Aportes de
Fernando Dodino

Versión 2.0
Abril 2017

Indice

1 Introducción

El proceso de construir un sistema no es una actividad formal. Es decir, no generamos un modelo matemático que se convertirá en el software. Esta característica necesita encontrar otros caminos que le den un soporte al ingeniero para lograr un buen diseño. La experiencia ha llevado a la ciencia a formar distintas metodologías que describen y guían todo el proceso (desde la captura de requerimientos hasta el mantenimiento del mismo). Las metodologías se basan en la experiencia que ayudan al ingeniero a tener herramientas y un marco en el cual generar el diseño, pero nada dicen acerca de cómo debe hacerse, sino que se confía en el criterio del experto para tal fin. Como podrán adivinar tampoco es una ciencia formal. Otra vez debemos valernos de la experiencia para poder encontrar un proceso que nos ayude en esta tarea. Esta experiencia se transmite en un conjunto de heurísticas, es decir, en muchas buenas ideas que son aplicables en determinados contextos, pero que no componen una receta procedural que pueda ser aplicada. Por este motivo el estudio del mismo desde un punto de vista puramente teórico representa un desafío complejo para nuestra ciencia y preferimos utilizar un ejemplo práctico en el cual se pueda ver no sólo las heurísticas sino también un marco de aplicación.

2 Caso de prueba: Lista de correos

Nuestro caso de ejemplo es el de la lista de correo. Tanto el enunciado como el análisis de requerimientos está explicado en el documento: Las entradas del diseño

3 Inicio

¿Por dónde arrancar a trabajar? Ante esta pregunta existen varias respuestas comunes: diagrama de clases, objetos candidatos, test cases, código, diagramas de secuencia, etc. Existen tantas opciones que a menudo uno se encuentra perdido y no sabe cuál elegir. El exceso de opciones es a menudo igual a no tener ninguna opción. Empezar por cualquiera de estos caminos nos lleva a trabajar de lleno en el “cómo”, pero a veces eso es demasiado apresurado. La primer heurística a tener en cuenta es la de: **trabajar basado en el requerimiento.** Esto significa que cualquier decisión que tomemos tiene que estar basado en un requerimiento. Esto evita comenzar a llenar la hoja con posibles objetos que surgen de la simple intuición. A menudo el conocimiento del negocio nos lleva apresuradamente a postular objetos, agregarle atributos, métodos, sólo por el hecho de que tenemos información. Por ejemplo, un inexperto puede querer generar una clase Usuario que tenga los datos del usuario como nombre, apellido, dirección de mail, dirección personal, fecha de nacimiento, etc. ¿Es todo esto necesario? ¿dónde necesitamos la fecha de nacimiento? y... a priori

en ningún lugar, lo buscamos en los requerimientos y no hay nada que nos haga suponer que lo necesitamos. Entonces, ¿por qué lo estamos haciendo en este momento? Porque tomamos un mal camino y no seguimos la primer heurística.

Frenemos un poco y repasemos los requerimientos. Al hacer el análisis detectamos los casos de uso, y luego hablamos de casos de uso de negocio y de sistema. Los que nos interesa en este caso son los de sistema[1].

Anotamos los CU detectados en una lista que nos sirva como guía de lo que tenemos que hacer. Iremos atacando cada caso de uso en forma independiente. De esta manera, cuando pienso en lo que tengo que resolver, me restrinjo a un caso de uso particular y no me voy por las ramas.

Una aclaración importante es que la suscripción de una lista de correo es un caso de uso de negocio que se debe implementar con varios casos de uso de sistema, ya que hay dos momentos diferentes:

1. En un primer momento un usuario sin registración solicita unirse a la lista de suscripción cerrada
2. En otro momento, un administrador revisa las aprobaciones pendientes de los usuarios, y confirma o rechaza la suscripción.

Para el negocio es una gran operación (que dura *n* horas, o incluso varios días). Para el sistema son dos operaciones diferentes resueltos por dos roles diferentes (usuario sin registrar y administrador).

En general cuando hay que sincronizar operaciones que ocurren en diferentes tiempos o que involucran diferentes actores, debemos partir ese caso de uso de negocio en varios casos de uso de sistema: solicitar suscripción + aprobar/rechazar suscripción[2].

Pasamos en limpio los requerimientos:

- Solicitar suscripción
    - lista abierta
    - lista cerrada

[1] Esto es debido a la naturaleza de la mayorías de las tecnologías: el dominio está inserto en una arquitectura que es usada desde la aplicación, y no tiene capacidad técnica de solicitar información al usuario. Por eso partir el caso de uso de negocio en diferentes casos de uso de sistema permite luego de cada interacción del usuario devolverle el control para que invoque el siguiente.
[2] Para más información recomendamos leer el apunte Las Entradas del Diseño

- Aprobar / Rechazar suscripción
- Enviar mail
    - lista libre
    - lista restringida

Debemos elegir de esa lista por dónde empezar. A veces es difícil saber cuál es el mejor caso para arrancar y uno elige el que intuya que más lo va a ayudar a armar una base para los CU siguientes. En aquellos casos de uso que tienen grandes bifurcaciones, conviene comenzar por la rama más simple. En este caso optamos por la solicitud de suscripción.

4 Solicitar suscripción

4.1 realidad vs modelo

Entendiendo ya el requerimiento, tratamos de acomodar los conceptos que tenemos en la cabeza. Lo primero que debemos hacer, es separar “la realidad” con respecto a nuestro dominio. Un usuario quiere suscribirse a una lista, eso es la realidad. Ahora, eso no significa que tenga que trasladar directamente eso a nuestro modelo haciendo

```
usuario.suscribir(lista)
```

Existe todo un componente de software que se encarga de transformar las acciones que realiza la persona usuario a envíos de mensajes a nuestro dominio (UI). En el dominio nos independizamos de esta problemática. Vamos a modelar las reglas del dominio sin importar si efectivamente es la persona usuario de la lista la que solicita la suscripción, si es un administrador que está agregándolo a pedido de un usuario o si es un proceso batch que agrega a partir de un archivo de texto. Lo que nosotros llamamos “usuario” aquí es un término del dominio, y tendrá la información y el comportamiento que nos quede bien para el dominio. Muchas veces pasa que un objeto del dominio está relacionado directamente con un usuario del sistema. En otro momento, cuando definamos la arquitectura de nuestra aplicación nos preocuparemos por ello. **Usuario de lista es distinto de usuario del sistema**. Para evitar la confusión vamos a cambiarle el nombre de nuestra abstracción de negocio: llamaremos “miembro” a lo que del análisis llega como “usuario”.

4.2 Interactuando con el dominio

Si bien no nos preocupamos por cómo la UI convierte las acciones del usuario en llamadas al dominio, ni como un objeto del dominio es presentado al usuario, nosotros tenemos que ser conscientes de nuestra interfaz. El envío del mensaje inicial

al caso de uso debe ser posible por la UI. Si decido que tengo un objeto lista que conoce sus miembros, y para agregar un nuevo miembro debo invocarlo de la siguiente forma

```
lista.agregar(miembro)
```

tengo que asegurarme que dicho componente pueda obtener los objetos lista y miembro. No sería correcto que tenga que instanciarlos ya que -en este caso- son objetos que tienen una identidad y mantienen un estado.

¿Es el dominio el encargado de proveer los objetos lista y miembro? Esa es una decisión de arquitectura. Nosotros asumiremos que la UI puede interactuar con otro componente diferente al dominio para obtenerlos. Pero es una buena práctica conocer la interfaz de nuestro dominio, pensar en ello nos lleva a diseños implementables.

Los test unitarios son otra manera de interactuar con nuestro dominio. Podemos en un test armar un escenario en el cual el envío del mensaje que inicia el caso de uso es posible.

Mensaje Inicial

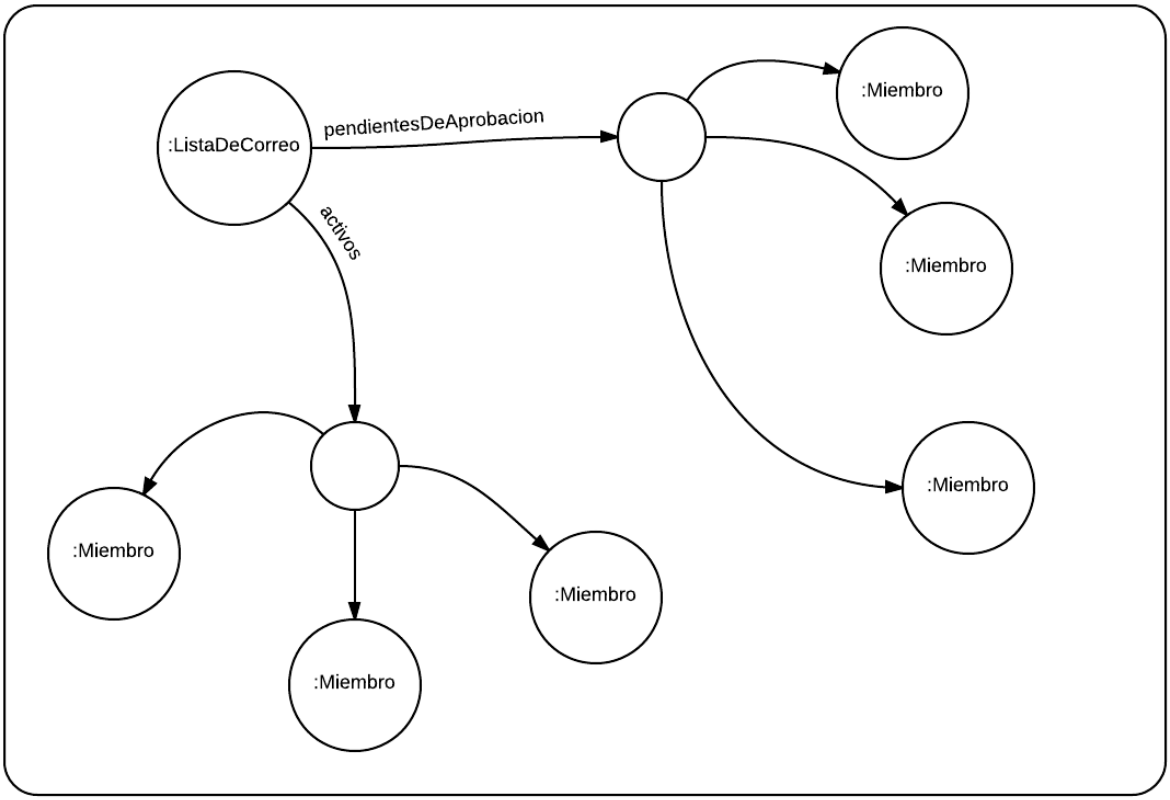
¿Quién debe entender el mensaje suscribir? Si elegimos empezar desde el miembro enseguida caemos en que el comportamiento del sistema depende del tipo de lista. Es entonces la lista quien puede decidir cómo contestar el mensaje en lugar del miembro. Así que nuestra mejor opción es:

```
lista.suscribir(miembro)
```

Decidimos además, que el conocimiento sobre si la lista es abierta o cerrada debe estar en la lista y no hay necesidad de decirlo al momento de suscribir.

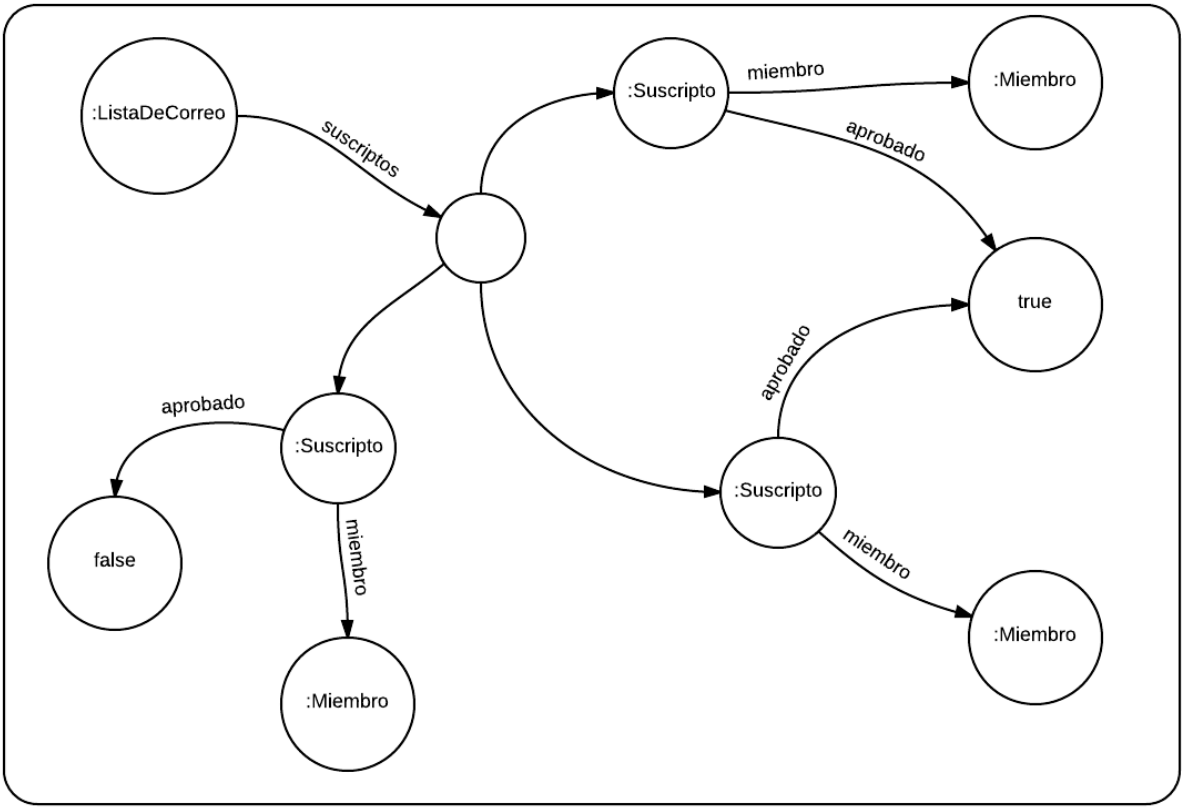
4.3 Cómo modelar la relación entre miembro y lista

Un camino posible es que dentro del mismo objeto lista mantengamos separados dos conjuntos de miembros: los activos y los pendientes de aprobación. En la aprobación se movería el miembro entre los conjuntos.



<<diagrama de objetos: 2 listas para modelar los usuarios suscriptos y los pendientes de aprobación>>

Otro camino alternativo es modelar una abstracción que nos indique qué es ese miembro para nuestra lista. Le podemos poner un nombre: Suscripto



<<diagrama de objetos: 3 usuarios suscriptos, dos aprobados y uno pendiente>>

En principio para lo único que necesitamos el suscripto es para armar el conjunto de miembros activos

```
@Accessors
class Suscripto {
	Boolean aprobado
	Miembro miembro
}

@Accessors
class Lista {
	Set<Suscripto> suscriptos

	def Set<Miembro> miembrosActivos() {
	    suscriptos.filter [ aprobado ].map [ miembro ].toSet
	}
}
```

La aprobación de un miembro implica modificar la referencia de la variable aprobado

de *false* a *true* en el objeto suscripto correspondiente.

Ambas soluciones cumplen el requerimiento, pero como toda solución, agrega ciertas características a nuestro diseño. Dependiendo el contexto éstas pueden ser consideradas como positivas, negativas o neutras. En todo caso, el ingeniero debe ser consciente de qué características le aportan a su diseño las decisiones tomadas.

En nuestro caso vemos que la segunda solución presenta cierta complejidad adicional, ya que necesitamos una nueva abstracción, mantener la nueva clase, y tener que hacer cierta transformación para llegar a la lista de miembros activos. Esta complejidad podría estar justificada si es que necesitáramos comenzar a agregar comportamiento a nuestra nueva abstracción. En nuestro caso, es todavía muy temprano para saber si ocurrirá. No aparecen a priori otras características que nos ayuden a elegir entre ambas soluciones. Vamos a aplicar en este caso otra de las heurísticas: **elegimos la solución más simple**, que va de la mano de: **no nos olvidamos de la alternativa descartada ni el motivo**, ya que en el futuro podemos detectar la necesidad de ir por esta solución.

4.4 Resolviendo el polimorfismo

Es evidente que el comportamiento del sistema debe variar según el tipo de la lista (abierta o cerrada). Para que un sistema orientado a objetos pueda comportarse de manera distinta según el caso particular tenemos distintas herramientas: usar una estructura de control para generar la bifurcación (condicional), herencia y composición.

4.4.1 Condicional

Nuestra primera versión se basa en usar una estructura de control.

```
class Lista {
    boolean abierta
    Set<Miembro> miembrosActivos = newHashSet
    Set<Miembro> miembrosPendientesDeAprobacion = newHashSet

    def suscribir(Miembro miembro) {
        if (abierta)
           suscribirAbierta(miembro)
        else
           suscribirCerrada(miembro)
    }
```

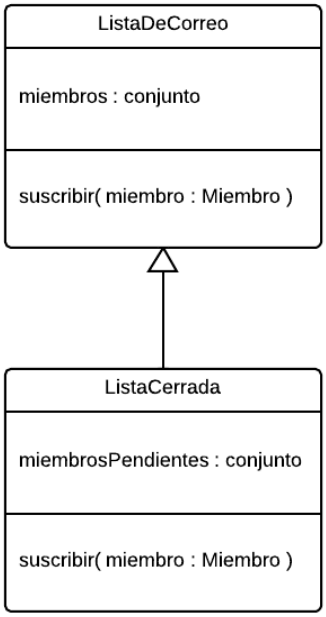
```
def suscribirAbierta(Miembro miembro) {
     miembrosActivos.add(miembro)
}

def suscribirCerrada(Miembro miembro) {
    miembrosPendientesDeAprobacion.add(miembro)
}
```

Características:

- En una lista abierta tengo código y estado innecesario (necesaria para las listas cerradas). Esto atenta contra la claridad del código.
- La clase Lista tiene la lógica de las listas abiertas y cerradas, por lo tanto al hacer más cosas es menos cohesiva que otras soluciones posibles.
- Agregar un nuevo tipo de lista requiere modificar código que es usado para las listas abiertas y cerradas (agrega acoplamiento indeseado).
- Necesito una clase sola para mantener todas las listas, si el código específico de cada tipo de lista dista muy poco respecto al código común, podría ser relativamente más fácil de mantener con respecto a distribuir el código entre distintas clases.
- Este modelo soporta el cambio del tipo de una lista luego de ser construida. Ojo que esto no es un requerimiento surgido del enunciado, pero pensar en esto me puede llevar a iterar sobre el análisis. Luego de esta iteración consideramos dicho requerimiento como “deseable”

4.4.2 Herencia

La herencia es un mecanismo que permite separar el código específico de cada tipo de lista en distintas clases, de manera que aumente la cohesión dentro de cada clase. Para armar un buen modelo basado en herencia debemos discernir qué es lo que tienen en común ambos tipos (para ubicarlo en una superclase) y qué tienen de particular (para ubicarlo en las subclases). Si no hubiera nada en común, entonces no necesitamos utilizar herencia, en ese caso podríamos hacer objetos polimórficos que respeten la misma interfaz de uso. Si algún tipo no tiene nada de particular, podría usarse la superclase como clase concreta.



En nuestro caso sabemos que ambos tipos de listas tienen el conjunto de miembros activos. Y las listas cerradas necesitan tener el conjunto de miembros pendiente de aprobación de manera separada. No aparece nada particular en las listas abiertas:

```
class Lista {
    Set<Miembro> miembros

    def suscribir(Miembro miembro) {
        miembros.add(miembro)
    }
}

class ListaCerrada extends Lista {
    Set<Miembro> miembrosPendientes = newHashSet

    override suscribir(Miembro miembro) {
        miembrosPendientes.add(miembro)
    }

}
```

Características:

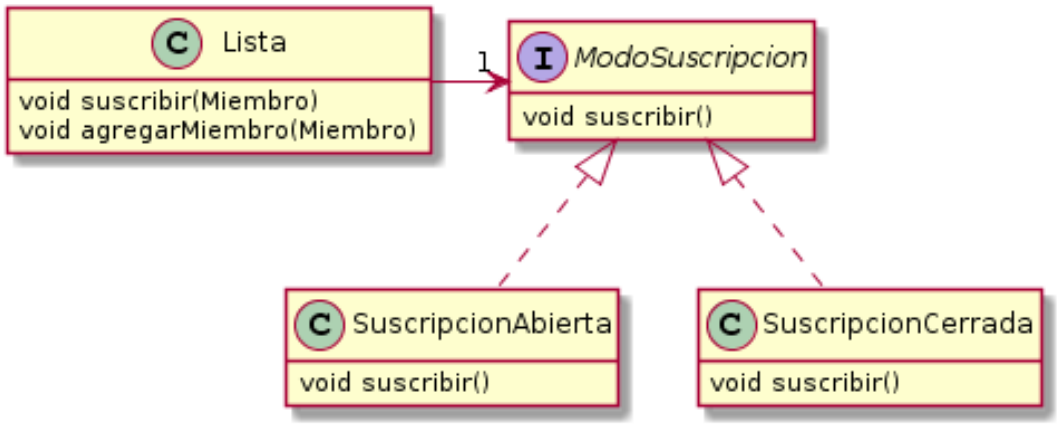
- Mejora la cohesión y el acoplamiento con respecto a la solución anterior. Las listas abiertas nada saben de las cerradas.
- No cumple el requerimiento *deseado* de dinamismo.
- En la mayoría de las tecnologías la herencia es un arma de un solo tiro, si la uso para el criterio de suscripción no la puedo usar para ningún otro criterio (como por ejemplo el modo de envío de mails)

4.4.3 Composición

Toda solución basada en herencia puede ser replanteada utilizando composición, extrayendo el comportamiento específico de cada tipo de lista en una jerarquía aparte y delegando el mensaje. Este tipo de solución tiene nombre de patrón de diseño: **Strategy**. Los objetos en los cuales se delega son “estrategias” del primero. La estrategia configurada determina cómo es el comportamiento del objeto.

```
class Lista {
    Set<Miembro> miembros
    ModoSuscripcion suscripcion
```

```
    def suscribir(Miembro miembro) {
        suscripcion.suscribir(miembro, this)
    }

    def agregarMiembro(Miembro miembro) {
        miembros.add(miembro)
    }
}

interface ModoSuscripcion {
    def void suscribir(Miembro miembro, Lista listaCorreo)
}

class SuscripcionAbierta implements ModoSuscripcion {
    override suscribir(Miembro miembro, Lista listaCorreo) {
        listaCorreo.agregarMiembro(miembro)
    }
}

class SuscripcionCerrada implements ModoSuscripcion {
    Set<Miembro> miembrosPendientes = newHashSet

    override suscribir(Miembro miembro, Lista listaCorreo) {
        miembrosPendientes.add(miembro)
    }

}
```



Características:

- Me libera el uso de la herencia para utilizar otro criterio, si es que la tecnología tenía esta restricción.
- Delego la responsabilidad de la suscripción de una lista en otra abstracción. Solo me quedaría estado y código *pegamento* con respecto a este requerimiento en la lista. Lo cual puede ser útil si la lista empieza a tener que recibir otras responsabilidades.
- Si la lista no adquiere nuevas responsabilidades, me genera una solución más compleja con respecto a la herencia.
- La construcción de los objetos se complejiza también.
- Soporta el requerimiento deseado de dinamismo.
- Si más cosas empiezan a depender del modo de suscripción tengo la abstracción justa dónde delegar la responsabilidad (por ejemplo, para aceptar/rechazar).

5 Aprobar/rechazar suscripción

La solución de este requerimiento está directamente relacionada con la solución anterior.

5.1 Solución con condicional

Parece natural que sea la misma lista quien entienda el mensaje aprobar y el mensaje rechazar:

```
class Lista {
    ...

    def aprobarSuscripcion(Miembro miembro) {
        miembrosPendientesDeAprobacion.remove(miembro)
        miembrosActivos.add(miembro)
    }

    def rechazarSuscripcion(Miembro miembro) {
        miembrosPendientesDeAprobacion.remove(miembro)
    }

}
```

5.2 Solución con herencia

En este caso sólo las listas cerradas deben entender los mensajes aprobar y rechazar.

5.3 Solución con composición

Una primer solución es que la lista entienda los mensajes aprobar/rechazar, y que éstos deleguen en su modo siguiendo la misma estrategia utilizada en el requerimiento anterior. Sin embargo ésta presenta la siguientes características:

- Le agrega el concepto de rechazar y suscribir a las listas abiertas, las cuales en un principio no tienen relación con este requerimiento. Parece que de cierta forma estamos forzando al modelo a que calce con nuestra idea de diseño. Generalmente ocurre que cuando un modelo es forzado cuesta más realizarle cambios, ya que el trabajo de refactor que debemos hacer previo al agregado de un requerimiento suele ser mayor a que si el modelo no está forzado.
- Estamos soportando un polimorfismo entre las listas abiertas y cerradas que no es necesario, ya que aquel que nos envíe el mensaje sabe que la lista con la cual está trabajando es cerrada. Esto no es un problema en sí, pero es bueno tenerlo en cuenta para no hacer ningún esfuerzo adicional en soportar algo innecesario.

La otra alternativa es que quien invoca al dominio hable directamente con un objeto de la clase *Cerrada*.

```
val Cerrada suscripcion = // de alguna forma lo obtiene
val Miembro miembro =     // de alguna forma obtiene el miembro
suscripcion.aprobar(miembro)
```

o bien

```
suscripcion.rechazar(miembro)
```

En este caso, estamos haciendo que la clase Cerrada sea parte de la interfaz de nuestro dominio. Es importante tener identificado para cada caso de uso, cuál es el objeto y el mensaje que se debe enviar para iniciarlo.

Y la forma en que la clase cerrada resuelve los mensajes serían:

```
class Cerrada implements ModoSuscripcion {
    ...

    def aprobar(Miembro miembro, ListaCorreo listaCorreo) {
        listaCorreo.agregarMiembro(miembro)
        miembrosPendientes.remove(miembro)
```

```
    }

    def rechazar(Miembro miembro, ListaCorreo listaCorreo) {
        miembrosPendientes.remove(miembro)
    }
}
```

6 Enviar un Mail

Abordamos ahora el último requerimiento, el envío de un mail.

Debemos elegir un objeto al cual enviarle un mensaje, y cuáles son los parámetros a recibir. Por un lado sabemos que una lista va a estar involucrada, y que además, dependiendo de si la lista es libre o restringida tendrá que actuar de forma diferente. La información que recibe de entrada es el email origen, el título y el texto.

Una primer decisión es si queremos recibir los parámetros por separado o construir un objeto que represente el envío de un mail y tenga adentro la información necesaria.

pensamos las siguientes opciones:

- `lista.enviar(origen, titulo, contenido)`
- `lista.enviar(mail)`

Es una buena práctica no tener parámetros dependientes entre sí. En nuestro caso origen, título y contenido están relacionados, por lo tanto queda una interfaz más sencilla agruparlos en un mail. De esta forma aparece una clase mail que es parte de la interfaz del dominio, a la cual es instanciado por fuera. Otra razón para agregar el objeto mail es que encapsula todos los atributos del mail y en caso de agregar o modificar uno de ellos no se altera la firma de los mensajes.

Al tener otro objeto, podemos elegir a cuál de ellos enviarle el mensaje para iniciar el caso de uso:

- `lista.enviar(mail)`
- `mail.enviarseA(lista)`

Un buen criterio para elegir cuál es el receptor del mensaje y cuál parámetro es pensar si existen comportamientos distintos dependiendo del receptor. Esto nos ayuda a acomodar nuestro diseño para aprovechar el polimorfismo. Como sabemos

que hay comportamiento distinto dependiendo de la lista, vamos a elegir esa opción.

Por otro lado, así como cambiamos la palabra “usuario” por “miembro” para evitar problemas de comunicación, vamos a hacer algo similar con la palabra “mail”. Ya que un “mail” enviado a la lista se convierte en muchos “mails” enviados (uno para cada miembro). Podríamos llamar “mensaje” al mail entrante y “mail” a los salientes. Pero para evitar confundir el “mensaje” del dominio con la palabra “mensaje” perteneciente al paradigma de objetos, utilizaremos “post”

```
lista.enviar(post)
```

6.1 Enviar mensaje a una lista libre

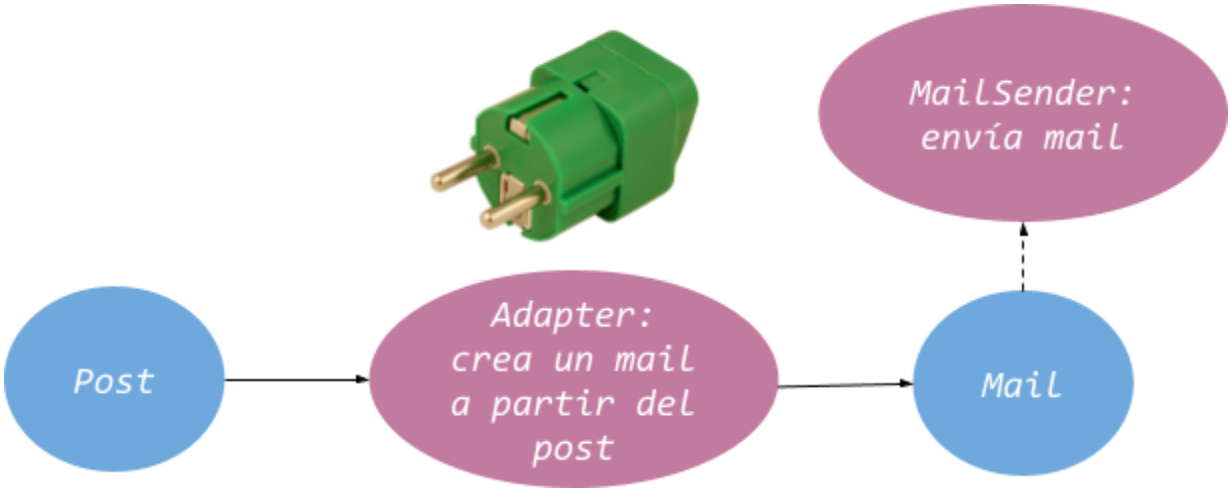
Empecemos a resolver la cuestión para lista libre ya que es el caso más sencillo, luego ampliaremos la solución para agregar la validación correspondiente. Nuestro requerimiento es bastante grande, hay varias cosas en las cuales pensar para resolverlo. Vamos a enumerar estas cuestiones para asegurarnos de atacar a todas:

- Construir un mail a partir de un post y un miembro
- Enviar el mail a todos los miembros
- Si el remitente del mensaje es un miembro no se le debe enviar el mail
- El envío de un mail está resuelto en un componente externo a nuestro dominio
    - ¿Quién define la interfaz con ese componente?
    - ¿Cómo le llega a nuestro dominio la instancia perteneciente a ese componente para poder ser usado?

6.2 Interface con el MailSender

Nuestro dominio tiene que utilizar a un componente externo para solicitar el envío de un mail. Es probable cuando diseñemos nuestro dominio que ese componente ya exista, en cuyo caso no tendremos poder de decisión para definir la interfaz (aunque si la interfaz es muy mala podamos implementar un Adapter -otro patrón de diseño- para no sufrir tanto). Por el momento vamos a asumir que es nuestra responsabilidad esta definición.

```
mailSender.send(mail)
```



Esta línea es una forma posible de usarlo, en cuyo caso el objeto mail pasa a ser parte de la interfaz del componente externo y no parte de nuestro dominio. Esto nos obliga a que la clase Email no tenga referencia a clases de nuestro dominio como Lista o Miembro. Debemos respetar que la clase Mail use nociones referentes a su propio dominio (los emails)

```
interface MailSender {
     def void send(Mail mail)
}

@Data[3]
class Mail {
    String to
    String from
    String subject
    String content
}
```

El mail sender es para nuestro dominio un “contrato”: no sabemos cuál es la clase concreta que entiende el mensaje, ni toda la configuración que puede necesitar para funcionar. De alguna forma nos van a proveer la instancia configurada lista para usar.

En cambio la clase Mail no tiene sentido que sea tratada a nivel tan abstracto, puede ser instanciada y configurada sin problemas por nuestro dominio.

[3] La annotation @Data de Xtend permite construir un objeto inmutable, que se configura únicamente en la inicialización

6.3 Obtención del MailSender

De qué forma le llega el objeto mailSender al dominio puede depender de la arquitectura y la tecnología. En todo caso lo importante es saber cómo se produce la inyección de la instancia y dejar el dominio preparado para eso. Si hago que todas las listas tengan un atributo mailSender,

```
class Lista {
     MailSender mailSender

     ...
```

- Al momento de instanciar una lista le tiene que llegar un mailSender por parámetro.
- Si quiero cambiar la implementación del MailSender que usan las listas, tengo que tocar en todas las listas.

Una alternativa es usar una variable de clase (*static*) en la Lista, la cual debe configurarse al inicio del sistema. Esto queda bien si los únicos objetos que utilizan al mailSender son las listas.

Otra alternativa es que las listas no tengan dicho atributo y que se pueda acceder al estilo “Singleton” -otro patrón más-:

```
MailSenderProvider.setInstance(mailSender) //configuracion
MailSenderProvider.getInstance() //uso
```

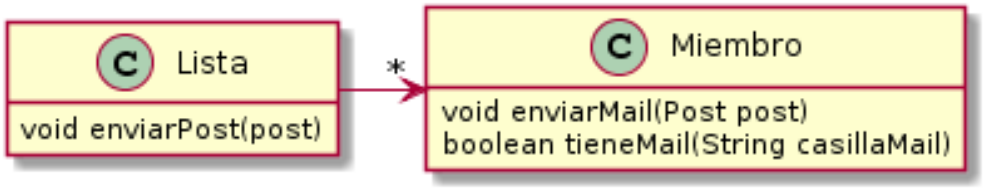
Esta alternativa nos permite ser más independiente de las listas y decidir que el uso esté en cualquier objeto del dominio, como por ejemplo en los miembros.

Lo más importante de esta decisión está en tener en cuenta que el objeto es inyectado. Tener una forma fácil de configuración nos permite poder probar nuestro dominio utilizando una implementación “Mock” o “Dummy” del mailSender.

6.4 Usando el mailSender

La lista tiene un conjunto de miembros, a cada miembro quiere enviarle un mail. Por otro lado se debe ignorar al emisor del mensaje. Parece cómodo que sea el miembro quien tenga la responsabilidad de resolver estos problemas y que la lista colabore con sus miembros para lograr el envío del mail.

```
class Lista {
   ...

   def enviar(Post post) {
       miembros
          .filter [ miembro | miembro.tieneMail(post.remitente) ]
          .forEach [ miembro | miembro.enviarMail(post) ]
   }

}

class Miembro {
   String mailDefault
   Set<String> direccionesMails = newHashSet

   def enviarMail(Post post) {
       MailSenderProvider.getInstance().enviar(
           new Mail(mailDefault,
                    post.lista.direccionMail,
                    post.titulo,
                    post.texto)
       )
   }

   def tieneMail(String casillaMail) {
       mailDefault.equals(casillaMail)
           || direccionesMails.contains(casillaMail)
   }

}
```



6.5 Agregando la validación para la lista restringida

Vamos a explorar las distintas opciones para cumplir el requerimiento, pero antes

debemos repasar lo que se espera. Las opciones son:

- ¿Se quiere ignorar el post en forma silenciosa?
- ¿Se espera que el dominio salga con un error?

Generalmente ignorar los problemas en forma silenciosa generan muchos problemas. La regla general es que **si un objeto no puede cumplir con la responsabilidad que está detrás del mensaje debe lanzar un error.** Salvo que la tarea de análisis diga lo contrario con un justificado argumento lanzaremos un error.

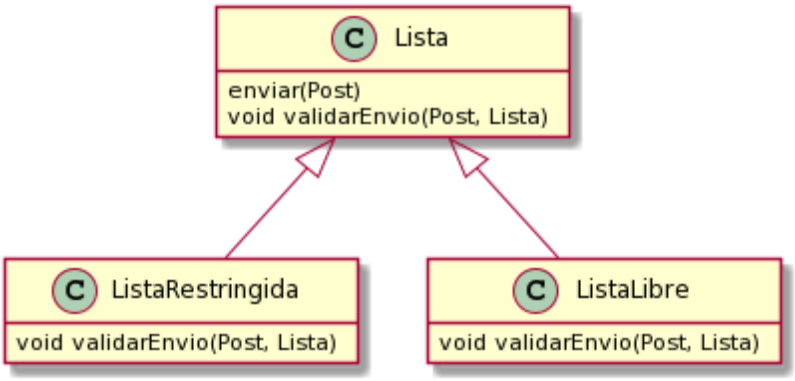
6.5.1 Condicional

Al igual que para la suscripción, el simple condicional presenta una solución posible que gana en simplicidad pero pierde contra la cohesión/acoplamiento. Dependiendo de la dimensión del problema podemos arriesgar si es una buena o mala solución.

```
class Lista {
    boolean listaRestringida

    def enviar(Post post) {
        this.validarEnvio(post)
        ...

    def validarEnvio(Post post) {
        if (listaRestringida && !esDeUnMiembro(post)) {
           throw new BusinessException("El post " + post + " no
corresponde a esta lista")
        }
    }

    def esDeUnMiembro(Post post) {
        miembros.exists [ miembro | miembro.tieneMail(post.remitente) ]
    }
```

6.5.2 Herencia

Esta solución la podemos usar si no optamos heredar en la suscripción, (o si estamos en una tecnología con herencia múltiple)




```
class ListaRestringida inherits Lista {
    ...

    def enviar(Post post) {
        this.validarEnvio(post)
        super.enviar(post)
    }

    def validarEnvio(Post post) {
        if (!esDeUnMiembro(post)) {
           throw new BusinessException("El post " + post + " no
corresponde a esta lista")
        }
    }

    def esDeUnMiembro(Post post) { ...
```

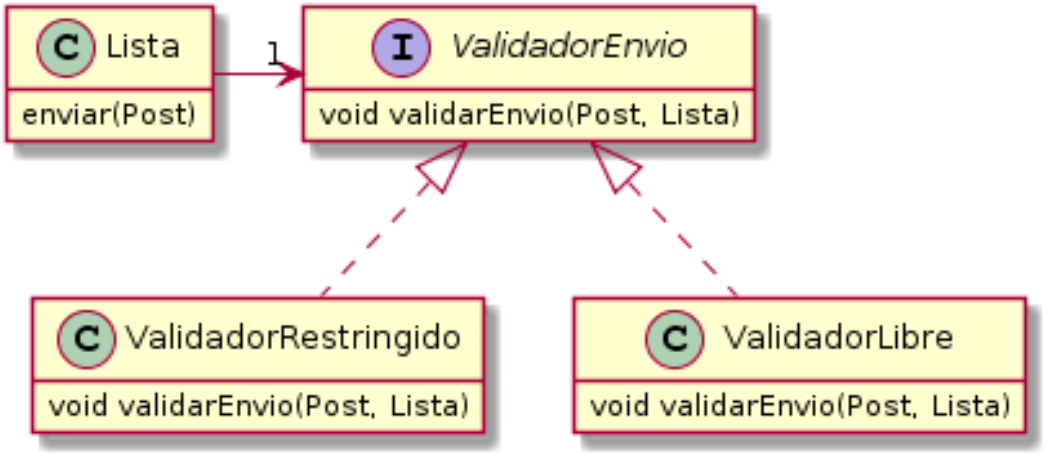
Al igual que en la suscripción, ganamos en cohesión/acoplamiento, pero perdemos dinamismo: no podemos cambiar el tipo de envío a la lista luego de instanciada. Y pagamos con respecto a la flexibilidad, ya que no podemos usar la herencia para otra cosa.

6.5.3 Composición

Si se puede resolver con herencia también con composición. En este caso no vamos a delegar completamente el mensaje enviar mail, si no que solo dejaremos la responsabilidad de la validación en otro objeto. En el caso de la lista libre podemos utilizar un validador nulo (NullObject)

```
class Lista {
```

```
    ValidadorEnvio validadorEnvio

    def enviar(Post post) {
        validadorEnvio.validarEnvio(post, this)
        ...

interface ValidadorEnvio {
    def void validarEnvio(Post post, Lista listaCorreo)
}

class ValidadorLibre implements ValidadorEnvio {
    override validarEnvio(Post post, Lista listaCorreo) {
        // No validamos
    }
}

class ValidadorRestringido implements ValidadorEnvio {
    override validarEnvio(Post post) {
        if (!esDeUnMiembro(post)) {
           throw new BusinessException("El post " + post + " no
corresponde a esta lista")
        }
    }

    def esDeUnMiembro(Post post) {
        listaCorreo.miembros
           .exists [ miembro | miembro.tieneMail(post.remitente) ]
    }
```



Características

- No tiene los problemas típicos de la herencia con respecto al dinamismo y a definir una visión única (solo tengo una bala para disparar, la que determina cómo se genera la jerarquía)
- Gana en claridad de código al despreocuparse la lista en cómo se valida.
- Ante un crecimiento en la complejidad o en la cantidad de reglas a evaluar para que validar un post, el problema recae en el lado de los validadores.
- Construir una lista es más complejo
- Las listas libres tienen algo de código innecesario, ya que no tienen el concepto de validación
    - Por otra parte el validador libre implementa el *NullObject*, lo que evita preguntas por valores nulos (la indirección permite evitar el condicional)

6.5.4 Decorator

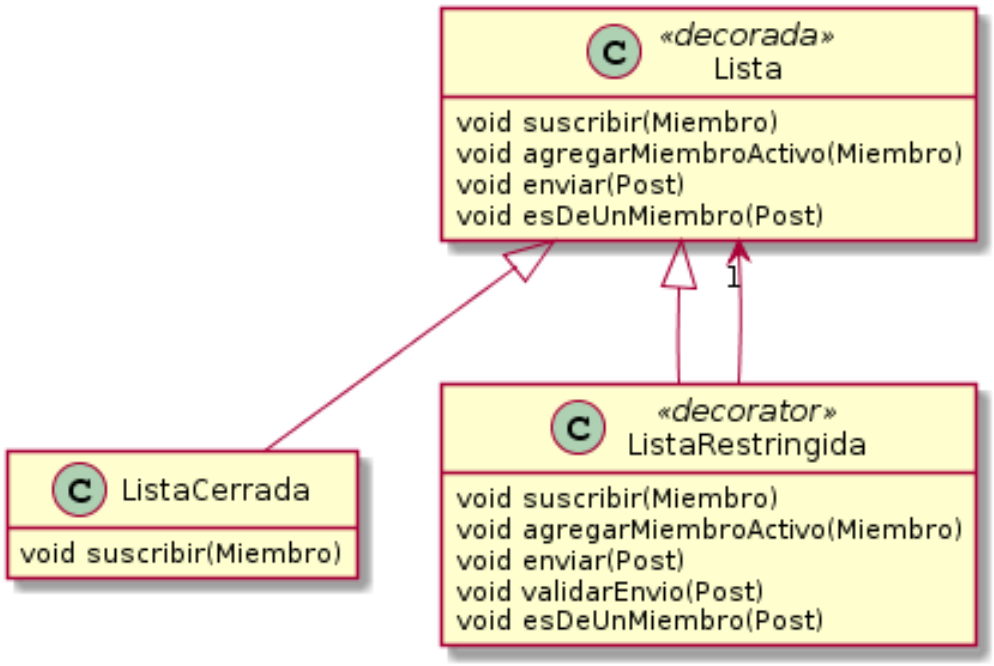
El decorator es otro patrón de diseño, que utiliza la composición/delegación como forma de compartir comportamiento, pero utiliza polimorfismo entre el objeto compuesto y su componente. La estrategia es simple, Una *ListaDecorator* se presenta al mundo como una lista, es decir, entiende los mensajes de las listas de correo, pero en su estructura interna no sabe cómo se comportan, por eso delega todos los mensajes que recibe en otra lista interna llamada decorada. En algún método en particular puede agregar comportamiento antes o después de delegar.

La mejor forma de entenderlo es viendo un bosquejo de cómo se implementaría

```
class Lista {
    Set<Miembro> miembros = newHashSet
```

```
    def void suscribir(Miembro miembro) {
        agregarMiembro(miembro)
    }

    def agregarMiembro(Miembro miembro) {
        miembros.add(miembro)
    }

    def void enviar(Post post) {
        miembros.filter [ miembro | miembro.tieneMail(post.remitente) ]
           .forEach [ miembro | miembro.enviarMail(post) ]
    }

    def esDeUnMiembro(Post post) {
        miembros.exists [ miembro | miembro.tieneMail(post.remitente) ]
    }

}

class ListaCerrada extends Lista {
   Set<Miembro> miembrosPendientes = ...

   override suscribir(Miembro miembro) {
      miembrosPendientes.add(miembro)
   }
}

class ListaRestringida extends Lista {
   Lista listaDecorada

   override suscribir(Miembro miembro) {
      listaDecorada.suscribir(miembro)
   }

   override agregarMiembro(Miembro miembro) {
       listaDecorada.agregarMiembro(miembro)
   }

   override enviar(Post post) {
      this.validarEnvio(post)
      listaDecorada.enviar(post)
```

```
    }

    override esDeUnMiembro(Post post) {
        listaDecorada.esDeUnMiembro(post)
    }

    def validarEnvio(Post post) {
        if (!this.esDeUnMiembro(post)) {
                throw new BusinessException("El post " + post + " no
corresponde a esta lista")
        }
    }
}
```



Las características de esta solución suelen ser comunes entre todas las soluciones basadas en decorator

- Se puede utilizar la herencia como criterio para la suscripción y agregar un decorator como criterio de validación (o al revés)
- Es la solución más compleja de las propuestas.
- Presenta un grado de dinamismo mayor a la herencia, pero menor a la composición por dentro del objeto (strategy).
- Hay que pensar cuidadosamente en qué lugar ocurre la decoración.
- Generalmente a la hora de programar un decorator pueden surgir problemas que no son fácilmente previsibles en el diseño:

- Hay una ruptura en la identidad del objeto (¿quién es this? a veces es el decorado y a veces el decorator),
- Tiene complicaciones cuando el objeto decorado implementa un *template method* (por el motivo anterior)
- En tecnologías con chequeo de tipos explícito y nominal puede requerirse una clase por cada subclase de la jerarquía a decorar que agregue mensajes a la interfaz. (En nuestro caso podríamos necesitar una ListaAbiertaRestringida y una ListaCerradaRestringida para dar soporte a los métodos aprobar y rechazar)
- En tecnologías como java puede traer cierta complicación si los objetos decorados hacen uso excesivo de lógica en los constructores, debido a que los mismos no pueden ser redefinidos.

7 Conclusiones

El camino para generar un diseño no es una receta. Nuestra guía son una serie de heurísticas y nuestra capacidad de comprensión que nos lleva a ir realizando ciertas preguntas que nos ayudan a tomar decisiones. Todas las decisiones son susceptibles de ser modificadas a medida que el proceso sigue su marcha. Esto nos lleva a que existen muchos modelos distintos que pueden dar solución a un problema. Cada modelo le aporta características/cualidades al diseño. A veces al estudiar estas características se puede determinar fácilmente qué modelo es mejor, pero en la mayoría de los problemas de mediana/alta complejidad la respuesta no es fácil. Está en la habilidad del ingeniero y en su capacidad de interactuar con el contexto determinar qué cualidad es preferible en cada caso particular, las limitaciones de su modelo y como impacta en el sistema las decisiones tomadas.