Manejo del cambio
Expansión y Refactoring

Versión 2.2
Marzo 2022

por Fernando Dodino

Índice

"Cambia lo superficial,
Cambia también lo profundo
Cambia el modo de pensar
Cambia todo en este mundo

Cambia el clima con los años
Cambia el pastor su rebaño
Y así como todo cambia
Que yo cambie no es extraño

Cambia, todo cambia"
tema de Julio Numhauser , cantado por Mercedes Sosa

1 El diseño en tiempos de cambio

Hasta el momento hemos diseñado nuestros modelos partiendo desde cero. Pero esta vez nos interesa comprender cómo se trabaja el diseño cuando tenemos una solución existente, ya sea que hayamos participado previamente en el diseño o no.

2 Recordemos algunos conceptos

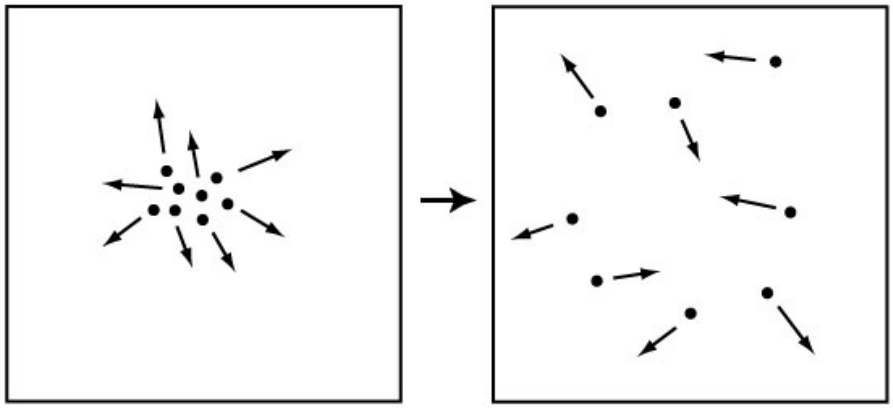
de la Teoría General de Sistemas

- ¿Qué es un sistema? Un conjunto de partes que se relacionan para un objetivo común
- ¿Cómo pueden ser los sistemas? Abiertos o cerrados. Nos interesa trabajar los primeros.
- Con el tiempo los sistemas abiertos se ven afectados por estímulos externos, tanto los componentes como sus relaciones.

2.1 Entropía

La entropía es una medida del cambio que se va produciendo en los sistemas, la tendencia natural de los sistemas hacia el desorden, el caos, que rompe el equilibrio (homeostasis[1]) que originalmente tiene un sistema.

[1] Ejemplos de homeostasis o el equilibrio que hace nuestro cuerpo para funcionar correctamente: el cuerpo suda para disipar el exceso de calor cuando aumenta la temperatura, los riñones equilibran el volumen de líquidos y minerales del cuerpo, el páncreas regula la concentración de glucosa en la sangre, el sistema nervioso y el sistema cardiovascular regulan la presión arterial. Variables que afectan esa homeostasis, la principal: el paso del tiempo (a mayor edad tenemos más dificultades para alcanzar ese equilibrio). Y por supuesto, nuestro estilo de vida: sedentarismo, vida social, químicos que tomamos, etc.



¿Cómo podemos llevar esto que parece tan abstracto al terreno de los sistemas de información?

- En un sistema de liquidación de sueldos la retención de sueldo básico para la jubilación de un empleado se modifica del 11% al 7% (cosa que ocurrió en Argentina en el 2002) y viceversa (a partir del 2007)
- Construimos un sistema de facturación que informa mensualmente los comprobantes emitidos por nuestro cliente a la AFIP y el organismo decide modificar el formato de la presentación (incorporando nuevos campos)
- Diseñamos un sistema de telefonía celular donde los clientes son individuos que deben tener más de 18 años pero luego sacan una promoción para inscribir a los hijos de los titulares y se genera el concepto de grupo familiar
- Hicimos un sistema para una empresa que arregla impresoras a domicilio y el cliente decide modificar el circuito para reparar unidades en un local propio que piensa inaugurar

En todos estos casos la entropía es la distancia entre el modelo original y el nuevo modelo que sigue evolucionando, una fuerza que demanda adaptar los componentes del sistema para aceptar esos cambios.

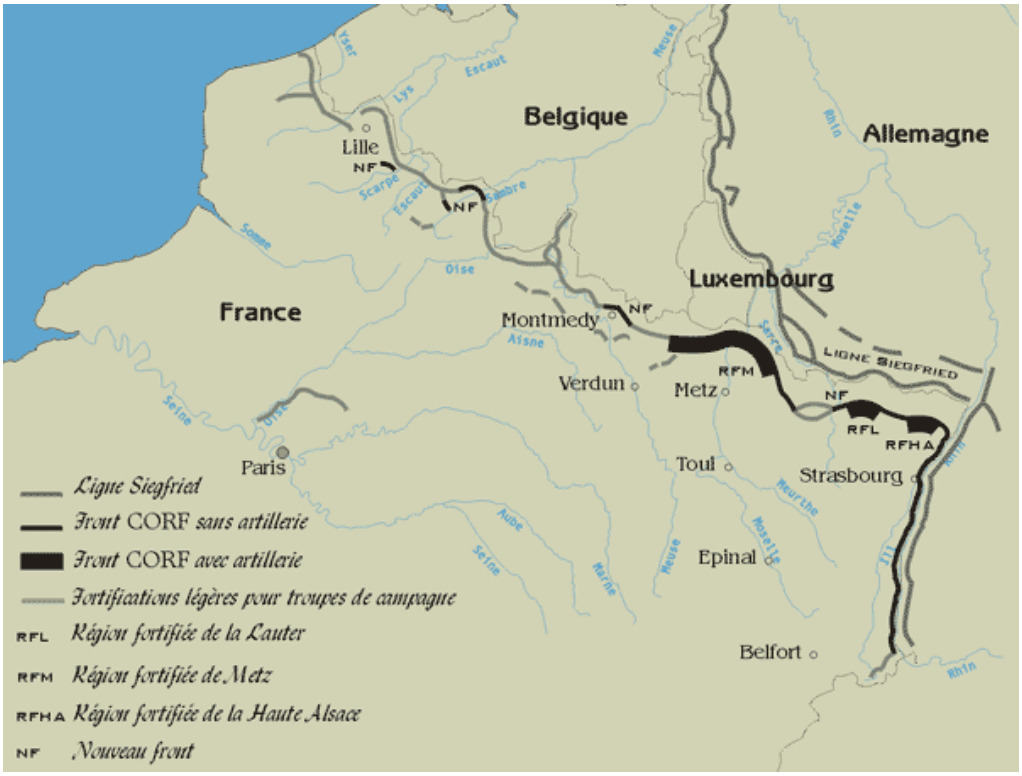
3 Estrategias de manejo del cambio

3.1 Línea Maginot o la resistencia al cambio

La primera reacción natural ante un cambio es la resistencia o negación: “dejámelo que lo vea”, “hay que analizarlo en detalle”, “técnicamente no se puede resolver”, o “el modelo no lo permite” han sido frases de cabecera para recibir al usuario.

La línea Maginot fue un sistema de fortificaciones ideado por el ministro de defensa francés André Maginot durante la segunda guerra mundial, que consistió en levantar

una columna de fuertes que cruzaban la frontera de Francia con Italia, Alemania y posteriormente Luxemburgo, Bélgica y Suiza[2].



La estrategia no dio resultado:

1. Francia no pudo afrontar los altísimos costos de armar una estructura de defensa efectiva,
2. y además los alemanes modificaron el paradigma de ataque: atravesaron Bélgica, rodearon el extremo occidental de la línea donde estaban las fortificaciones más débiles e invadieron París con éxito.

¿A qué nos lleva este cuento bélico? **No podemos evitar el cambio.** Tarde o temprano nos alcanza, tenemos que adaptar el sistema para que pueda seguir dándole valor agregado al cliente. De otra manera, estamos acelerando su fecha de obsolescencia.

El lector podrá preguntarse por qué la línea Maginot está asociada a la idea de negación o resistencia al cambio, cuando en realidad hubo una enorme inversión en dinero y esfuerzo para prepararse ante el inminente ataque alemán. El problema es que este esfuerzo estaba dirigido a tratar de preservar el estado de las cosas, negando el comienzo de la segunda guerra mundial. El cambio *ya estaba en marcha.* De la misma

[2] La idea era simular una “muralla china” armada. Véase http://www.exordio.com/1939-1945/militaris/armamento/lineas.html

manera,

- cuando pensamos que si documentamos adecuadamente los requerimientos y prevemos todos los escenarios posibles la construcción (y el posterior mantenimiento) será trivial
- cuando pensamos que el software puede tener la misma perspectiva que la construcción de materiales sólidos, en donde los cambios están controlados por limitaciones físicas y mecánicas de los elementos con los que trabajamos
- cuando decimos que estamos atacando los cambios cuando en realidad los estamos posponiendo tanto como sea posible, o generando soluciones “de compromiso” que obligan a enormes esfuerzos de mantenimiento (casi tanto como construir una muralla)

estamos negando la naturaleza misma de los sistemas, que están en continuo cambio y readaptación.

3.2 La defensa Suiza o aceptación del cambio

Suiza durante la segunda guerra mundial se declaró país neutral entre los bandos en disputa, aceptando visitantes de cualquier país, raza o religión. Podemos analizar largamente si esta es una estrategia válida en cualquier conflicto armado, de todas maneras lo que nos interesa rescatar es la actitud de aceptación y adaptación al cambio:

- los requerimientos pueden variar
- las definiciones no son constantes y están sujetas a error
- nuestro diseño no será perfecto de entrada, tenemos que permitirnos mejorarlo poco a poco cuando tengamos un mejor conocimiento del dominio que estamos modelando
- también es natural que haya desvíos entre lo que se modela y cómo se construye, y debemos actuar tan pronto los encontremos.

4 Los momentos del cambio: Expansión...

Es la etapa en la cual aceptamos nuevos requerimientos. Aquí vienen

- mejoras o cambios a requerimientos existentes
- nuevas funcionalidades

Lo importante es encontrar una metodología que permita relevar modificaciones o agregados a nuestro software, y considerar que cada requerimiento va a tener una prioridad.

4.1 Cómo diseñar en el contexto de un cambio

Que los componentes actuales acepten nuevas funcionalidades no siempre es fácil de lograr. *Ejemplo*: en una aplicación de telefonía, las solicitudes de reparación de una línea se resuelven en 2 días para clientes que viven en Capital Federal ó 5 en caso contrario. Definimos un objeto que modela la solicitud de reparación, que toma la responsabilidad de determinar si se excedió el tiempo previsto de reparación

```
class SolicitudReparacion(val cliente: Cliente) {
   var fechaInicio: LocalDate = LocalDate.now()

   fun excedioTiempoDeReparacion() =
       ChronoUnit.DAYS.between(fechaInicio, LocalDate.now())

   fun maximoDiasResolucion() = if (cliente.viveEnCapital) 2 else 5
}
```

Nos piden que ahora modifiquemos la cantidad de días máxima para resolver la reparación según el siguiente criterio:

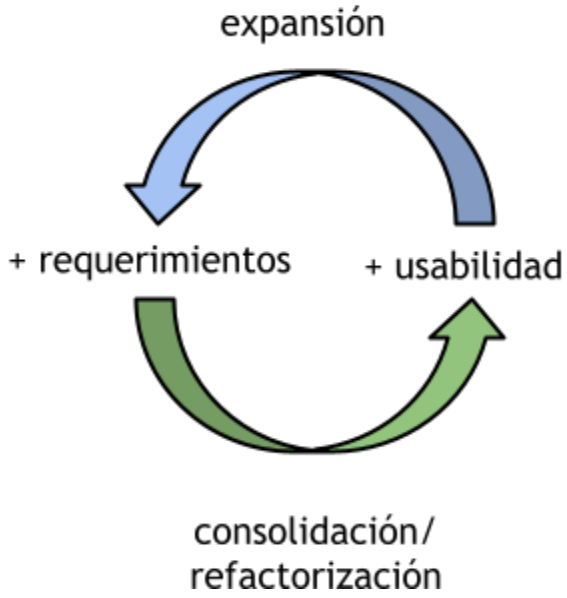
- 3 días para clientes en zonas residenciales
- para grandes clientes: 2 días ó 1 día si el cliente tiene un convenio especial con la compañía
- 8 días para clientes que vivan en countries
- no hay límite para el resto de los clientes

¿Qué componentes necesitamos adaptar? ¿Tenemos toda la información que el sistema requiere? ¿Cómo vamos a calcular la información histórica? Son algunas de las preguntas que podemos hacernos.

Pero aquí se presenta una oportunidad adicional, previa a la expansión: el proceso de **refactorización** de nuestros componentes.

5 Refactorización

¿Qué es refactorizar?



- Es cambiar la estructura del código para hacerlo más simple y extensible. Esto involucra muchas acciones posibles, algunas simples como cambiar el nombre de una clase o un método, o cosas más complejas como reemplazar condicionales (ifs) por objetos polimórficos o bien separar una clase en varias.
- **Refactoring no es incorporar funciones nuevas:** si bien el refactor suele surgir cuando hay que incorporar nuevos requerimientos a una aplicación, debemos separar claramente dos momentos: en uno nuestro objetivo es simplificar el código (refactoring), en otro aprovechar ese código simplificado para poder agregar funcionalidad (expansión).
- **Refactorizar vs. optimizar:** son cosas distintas porque tienen objetivos distintos. En un desarrollo lo aconsejable es refactorizar primero y optimizar sólo cuando necesite (y no siempre necesito pensar en performance). Cuando refactorizo el código queda más claro y mantenible, cuando optimizo no necesariamente pasa eso.

6 Refactoring y OO

En muchos casos el entorno de desarrollo (IDE) nos ayuda en esta tarea. Algunas cosas que damos por sentadas pero que conviene valorarlas son:

- **highlighting:** colorear lo que son palabras reservadas, comentarios, argumentos, variables, etc.
- **herramientas analizadoras de código:** muchas veces el entorno nos avisa cuando hay estructuras o algoritmos que no son utilizados, o cuando hay definiciones redundantes. También tener un sistema de tipos robusto ayuda a

detectar tempranamente errores en la implementación. Por último algunas herramientas van más allá y detectan diseños inconsistentes.
- **formatters:** tener un formato común para la indentación, para estructuras de selección (if), para manejar errores, también ayuda tanto en el mantenimiento como en la homogeneidad del código que se escribe.
- **manejo de dependencias**: tanto a nivel clase (imports) como a nivel proyecto
- contar con un buen **menú de refactors automáticos desde el IDE**

son cosas que favorecerán la actividad de mejorar nuestro código.

Los libros sugieren que el diseño orientado a objetos facilita la reutilización de código. Claro, eso requiere de codificar siguiendo **buenas prácticas**:

- Una consiste en programar pensando en la interfaz (qué necesito/qué ofrezco) y menos en la implementación. Por eso es preferible pensar primero en el mensaje y luego en el método, o definir las variables de tipo List y después ver si lo resuelvo con una colección de tipo ArrayList o LinkedList
- Otra es minimizar el acoplamiento entre objetos: no conocer cosas de más, relacionado con el punto anterior
- Otra es subir la cohesión de los métodos y las clases: métodos cortos que tengan un objetivo claro y definido, y que eso me lleve a entender cuál es el objetivo principal de la clase, si tengo muchos objetivos puedo perder foco en lo que quiero hacer
- Y por supuesto, el polimorfismo es una herramienta esencial para poder enviar mensajes a objetos intercambiables sin que el que envía el mensaje necesite saber a qué clase pertenecen
- La herencia, en menor medida, también colabora en la reutilización al permitir extender comportamiento existente de las superclases.

Si quiero refactorizar es <u>imprescindible</u> contar con una herramienta de testeo unitario automatizado, para hacer pruebas de regresión que garanticen que el refactoring no alteró el comportamiento del sistema. Mientras mejores tests tengamos, tanto más seguridad podemos brindar de que el refactor cumplió su objetivo. Sin herramientas de testeo unitario el miedo se apodera de los programadores, un miedo que es natural, pero que paraliza las posibilidades de mejorar el código ("no tocar lo que anda" vuelve a la idea de negar el cambio).

6.1 ¿Qué busca el refactor? ¿Cómo es "más simple"?

- Métodos cortos y con nombres bien definidos (*intention revealing*, que revelen exactamente el propósito para el que fueron creados)

- Métodos y clases con responsabilidades claras y bien definidas (mayor cohesión)
- Respecto a las variables de instancia: hay que evaluar el costo-beneficio de tener en variables valores que puedan calcularse, como el total de deuda de un cliente, o la cantidad de hijos de un empleado
- No tener *"god objects"* ni *"managers"*, objetos que roban responsabilidades que les corresponden a otros objetos, que quedan anémicos, con poco comportamiento y se parecen a estructuras de datos
- Es preferible tener objetos chicos antes que un objeto grande con muchas responsabilidades (sobre todo si los objetos chicos pueden intercambiarse)
- Evitar ciclos de dependencia entre objetos si no los necesito. Ej: cada cliente conoce a sus facturas, la factura ¿necesita conocer al cliente? muchas veces, por las dudas, tenemos relaciones innecesarias. Esto tiene un nombre: *"The Acyclic Dependencies Principle"*. Como dice Opdyke: "One asymmetric aspect of an aggregation is that the aggregate object usually needs to know about its components, but less often does a component need to know either about the aggregate object that contains it, or about other components."

6.2 Code smells

"If it stinks, change it"
La abuela de Kent Beck acerca de cuándo cambiar los pañales

¿Cuándo nos damos cuenta de que necesitamos mejorar nuestro diseño? ¿Cómo saber si nuestro código necesita un "service" de refactoring? Vale la pena la advertencia de Martin Fowler y Kent Beck en su libro Refactoring: "One thing we won't try to do here is give you precise criteria for when a refactoring is overdue. In our experience no set of metrics rivals informed human intuition"[3].

Lo que vamos a ver a continuación son algunos ejemplos de diseños (y sus correspondientes implementaciones) que presentan señales, "olores". En cada caso modelaremos una solución alternativa que puede (o no) ser comparativamente mejor. Por eso es importante una aclaración previa: **los smells no implican necesariamente un mal diseño o implementación, sino que son señales de algo que se puede mejorar (no hay que ser fanático ni terminante, muchos code smells se entrecruzan al tratar de solucionarlos).**

6.2.1 Misplaced methods

[3] *Refactoring, Improving the Design of existing code*, Martin Fowler, Addison-Wesley, 1999, p.76

Ejemplo: tenemos una aplicación para Papá Noel donde debemos calcular la cantidad de maldad de una acción. En la clase PapaNoel codificamos:

```
class PapaNoel {
   fun getCantidadMaldad(accion: Accion) = accion.getCantidadMaldad()
}
```

Si no accedemos a ninguna variable ni tampoco utilizamos comportamiento interno, posiblemente estamos ubicando mal el método: ¿por qué no enviar directamente el mensaje a acción?

Es frecuente encontrarnos con muchos métodos “secretarios” al implementar el Façade pattern.

6.2.2 Duplicated Code

Missing inheritance or delegation

No queremos duplicar la misma idea en el código. Dos lemas que nos acompañan son:

- Once and only once: hacer las cosas una sola vez
- Don't repeat yourself (DRY)

y esto no sólo vale para el código, también aplica para el diseño. Tengo herramientas para esto:

- la composición (y consecuente delegación)
- y la herencia (como segunda opción, si seguimos el consejo del libro Gang of Four)

6.2.3 Long Method

Inadequate decomposition

Un método largo podemos descomponerlo en varias partes.

- cada parte supone
    - identificar el objetivo que cumple
    - implementar esa abstracción: ponerle un nombre representativo y encontrar qué objeto es responsable de ese objetivo.
- los métodos generados pueden ser utilizados en otro contexto
- delegar el método original en varios submétodos nos permite entender mejor qué es lo que hace

Esta técnica no es excluyente del paradigma de objetos, de hecho es la técnica de “divide y vencerás”[4] aplicable a cualquier paradigma.

[4] El lector puede ver la siguiente clase del MIT sobre este tema:

6.2.4 Large Class / God Class

Too many responsibilities

Una clase con muchas responsabilidades conoce información de muchos objetos, y se ve rápidamente impactada ante cualquier cambio. Cualquier agregado o corrección de estas clases suele representar un dolor de cabeza. La solución no es trivial, dado que requiere repensar el diseño (eso implica poner responsabilidades en distintas clases, o crear clases nuevas).

6.2.5 Long Parameter List

Object is missing

Cuando un objeto ofrece un servicio hacia los demás, los publica a partir de una interfaz. Y esa interfaz está bueno cambiarla lo menos posible. La interfaz de un método está dado por:

- el nombre del método
- lo que devuelve (en lenguajes con chequeo estático incluye el tipo de lo que devuelve o void)
- los parámetros (si el lenguaje tiene chequeo estático esto incluye los tipos de esos parámetros)
- las excepciones que puede devolver (si estás en Java y las excepciones son chequeadas)
- incluso si el método tiene efecto colateral o no (modificar eso también impacta en los objetos que lo usan)

Entonces, este método en Cliente

```
fun getVentas(
   fechaDesde: LocalDate, fechaHasta: LocalDate,
   incluyeVentasPendientes: Boolean,
   nombreVendedorContiene: String, productoEmpiezaCon: String,
   montoMinimo: BigDecimal, montoMaximo: BigDecimal
   // ...
): List<Venta> {
   // ...
}
```

es muy probable que sufra cambios el día de mañana, porque estamos pasando una estructura bastante importante como parámetro. Una solución posible: generar un

---

http://www.youtube.com/watch?v=GKncNr2M8G4

objeto que cumpla esta misión, en principio como agrupador de los parámetros de búsqueda:

```
fun getVentas(busquedaVentas: BusquedaVentas): List<Venta> {
    ...
```

Esto permite ampliar, modificar o quitar parámetros de búsqueda sin afectar la interfaz del objeto.

6.2.6 Type Tests

Missing polymorphism

Preguntarle a los objetos "de qué tipo sos" es señal de no haber utilizado correctamente objetos polimórficos.

6.2.7 Message Chains

Coupled classes, internal representation dependencies

Si un objeto envía un mensaje de la forma:

```
a.b().c().d();
```

esto va en contra del consejo Tell, don't ask y de la Ley de Demeter ("don't talk to strangers", "only talk to friends"), donde un objeto sólo debería enviar mensajes:

- a sí mismo
- a objetos que conoce (como variables de instancia)
- a objetos que recibe como parámetro
- a objetos que instancia

De todas maneras, la ley de Demeter no siempre puede cumplirse a rajatabla y hay que tener cuidado con llevarlo a un pensamiento purista y dogmático (por ejemplo, con el manejo de colecciones).

6.2.8 Data Clumps

Data always used together

Ejemplos:

- x, y está representando un objeto Point
- calle, altura, piso, departamento, etc. forman parte de un objeto Direccion
- dni, nombre puede representar a una Persona

Si estamos trabajando un conjunto de datos que está relacionado nos estamos perdiendo una abstracción: es importante reconocerla, darle un nombre, una entidad. Eso nos permite asignarle responsabilidades y vuelve al sistema menos permeable a los cambios. Refactor asociado: Extract class

6.2.9 Temporary Field

Attributes only used partially under certain circumstances

Ejemplo: Una clase Factura que tiene una variable privada double totalConIVA, otra double totalSinIVA y además una double total (que es la suma del total con IVA y sin IVA), o un Cliente que tiene una variable privada fechaUltimoPago cuando también tiene una colección de pagos cada uno con su correspondiente fecha. Es decir que "cachea" valores que podría calcular. De todas maneras esto no constituye un problema *per se*, uno puede definir variables calculadas basándose en:

- que es costoso resolver el valor y se necesita conocer ese valor muchas más veces que las veces que se actualiza
- temas de implementación (como la necesidad de guardar ese valor en una base de datos relacional para luego ejecutar queries de consulta, así estaríamos evitando duplicar la lógica de negocio en el SQL)

Ejemplo: tenemos una clase Guerrero que contiene una colección de armas, y queremos saber el arma más poderosa: ¿necesitamos guardarlo en un atributo aparte o lo podemos calcular?

6.2.10 Data Classes

Only accessors

Una clase que sólo representa una estructura de datos, es fácil reconocerla cuando tiene sólo getters y setters. Un contraejemplo son los *Value Objects* (como un Mail, o un punto en un eje de coordenadas), y los objetos que modelan los parámetros que enviamos a un objeto (relacionado con Long Parameter List). Pero sí constituye una mala práctica de diseño pensar que es bueno separar un objeto en

- atributos
- y comportamiento

solo por el hecho de separar la estructura de datos y los procesos que acceden a esa estructura.

¿Por qué es una mala práctica? Porque

- descree del principio del paradigma: un objeto agrupa atributos y

comportamiento

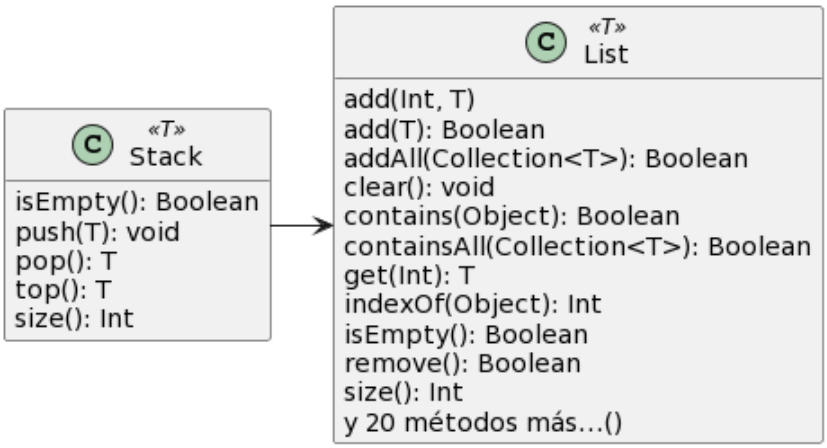
- los objetos resultantes tienen muy poca cohesión y un alto acoplamiento entre sí: es fácil darse cuenta de que cualquier cambio en la estructura de datos causa un inmediato impacto en los objetos proceso
- el corolario de lo expuesto anteriormente es que no hay encapsulamiento: la implementación de cada atributo está expuesta en los getters y setters del objeto
- es difícil trabajar con clientes polimórficos: el comportamiento está puesto fuera del objeto que representa al cliente
- es difícil no repetir ideas de código si tenemos varios objetos que representan funcionalidades del cliente: aplicar descuentos, calcular deuda, facturar, etc.

6.2.11 Primitive Obsession

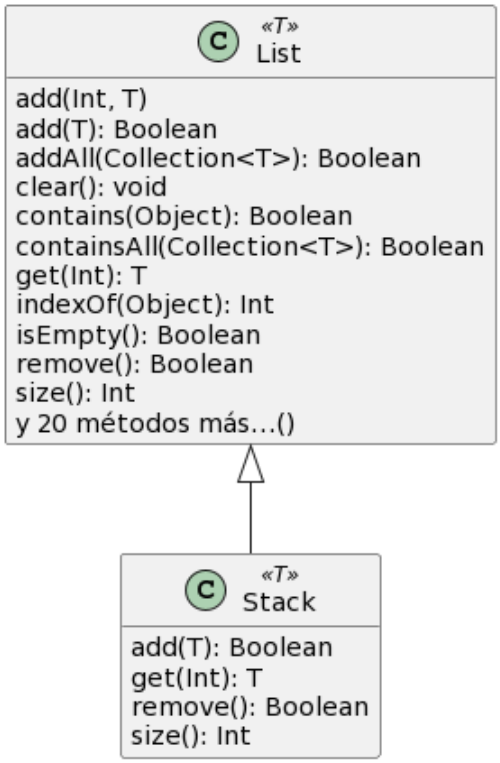
Representar con ints, booleans, Strings o enumeraciones cosas que podrían ser objetos con comportamiento. Las enumeraciones nos llevan a tener sentencias condicionales en lugar de trabajar con objetos polimórficos. O bien preferir el Array [] en lugar de tener objetos que modelen una colección, basta con ver la rica interfaz de Collection en contraposición a la acotada cantidad de cosas que se puede hacer con un array.

6.2.12 Refused Bequest

Utilizar herencia cuando se puede obtener lo mismo con delegación. Ejemplo típico: ¿una Pila hereda de List o tiene un List?



Caso 1: Pila tiene una Lista



Caso 2: Pila hereda de Lista

La herencia es rígida:

- obliga a definir más métodos que el necesario: la Pila tiene que redefinir o heredar containsAll, clear y muchos métodos que quizás no apliquen para el concepto Pila
- en la mayoría de las implementaciones sólo puedo heredar una vez, eso es una limitante (permite un solo punto de vista)

6.2.13 Inappropriate Intimacy (Subclass Form)

Cuando una subclase accede directamente a las variables de su superclase en lugar de utilizar los getters (acceso indirecto). Este bad smell es el corolario de la frase "la herencia viola el encapsulamiento", o bien de que la herencia me lleva a un acoplamiento mayor que la delegación. De todas maneras:

- es lógico suponer que cuando uno modifica una superclase es muy probable que las subclases se vean afectadas
- trabajar con accesors mitiga pero no resuelve este problema, el acoplamiento sigue existiendo.

Ejemplo: existe una superclase Cliente con un atributo saldo y una subclase

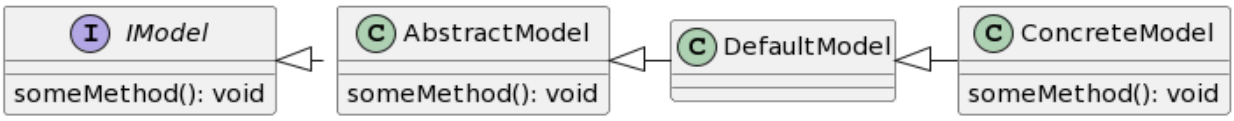
ClienteComun. Un cliente común es moroso (deudor) si su saldo es mayor a 0. Implementamos el comportamiento en la clase ClienteComun:

```
open class Cliente {
   var saldo = 0
}

class ClienteComun : Cliente() {
   fun esMoroso() = saldo > 0
}
```

Si decidimos dejar de guardar el atributo saldo (que es un atributo calculado, visto en el smell temporary field) entonces la subclase ClienteComun se va a ver afectada. Si creamos una propiedad saldo (es decir, un método getSaldo()) que sume los saldos pendientes de cada factura y calcule el saldo para ese cliente, la subclase ClienteComun no se verá afectada por este cambio.

6.2.14 Lazy Class

Suele ocurrir al sobrediseñar jerarquías para uso futuro: armamos la interfaz IModel, tenemos luego una clase abstracta AbstractIModel con una implementación default: DefaultModel concreta que hereda de AbstractIModel. Pero la clase DefaultModel no define comportamiento ni atributos, está PLD (por las dudas).



En ese caso el consejo "no programar para el cambio" de XP resulta útil: pensar en lo más simple nos llevaría a primero definir ConcreteModel1. Luego, a medida que lo necesitemos, podemos hacer aparecer nuevas abstracciones.

Otro acrónimo asociado, es YAGNI "You aren't gonna need it": no agregar funcionalidad hasta que sea necesario. Trabajar con una metodología TDD ayuda para que esto no pase: sólo se definen los métodos necesarios para resolver un test.

6.2.15 Feature Envy

Method needing too much information from another object

Esto ocurre cuando una clase A envía demasiados mensajes a otra clase B, quizás porque la clase B no esté ofreciendo el servicio que la clase A necesita. Un ejemplo

extraído de http://c2.com/cgi/wiki?ManifestResponsibility es cuando tenemos una clase Carta, que conoce a una persona y definimos un método

```
class Carta(val persona: Persona) {
   fun destinatario() = persona.nombre + " " + persona.direccion + "
" + persona.edad
}
```

La clase Carta está pidiendo demasiada info a la persona, entonces debería ser responsabilidad de la persona devolver el domicilio completo:

```
class Persona {
   fun destinatario() = nombre + " " + direccion + " " + edad
}

class Carta(val persona: Persona) {
   fun destinatario() = persona.destinatario()
}
```

6.2.16 Middle Man

Class with too many delegating methods

```
class MiddleMan {
   var delegate: DelegateClass = DelegateClass()

   fun f() {
       delegate.f()
   }

   fun g() {
       delegate.g()
   }

   fun h() {
       delegate.h()
   }
}
```

Una clase "recepcionista" que en realidad termina delegando a otro objeto toda la responsabilidad. Puede ocurrir en los objetos fachada si hay uno o muy pocos objetos a los cuales se delega y poco valor agregado del Façade (simplemente redespachar el mensaje). También en los decorators, si el objeto decorado tiene muchos métodos que no agregan ninguna funcionalidad. Por otra parte, también puede ocurrir al tratar de

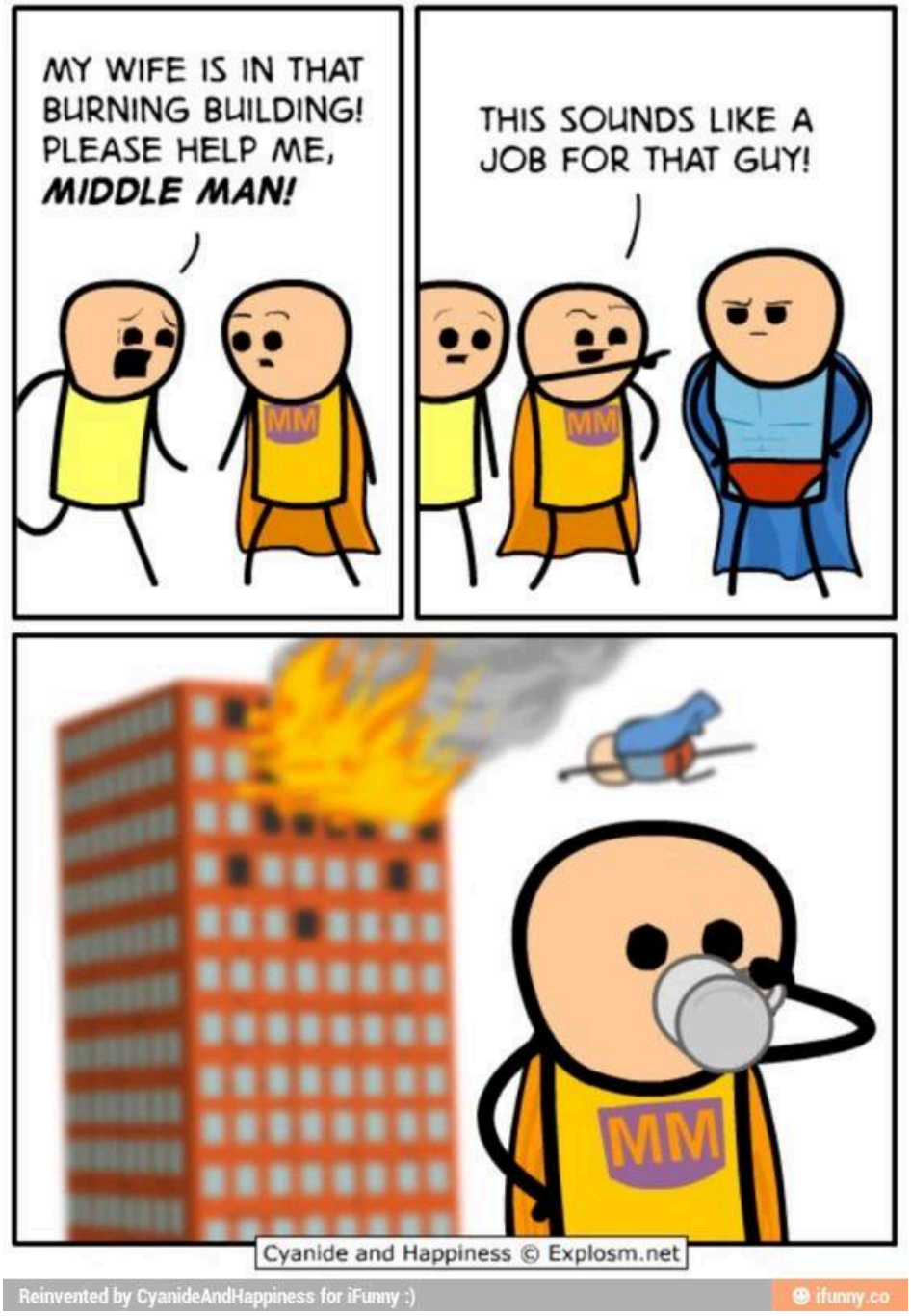
evitar el smell Message Chain (ver más abajo), ejemplo:

```
a.b().c().d()
```

En lugar de eso, hacemos:

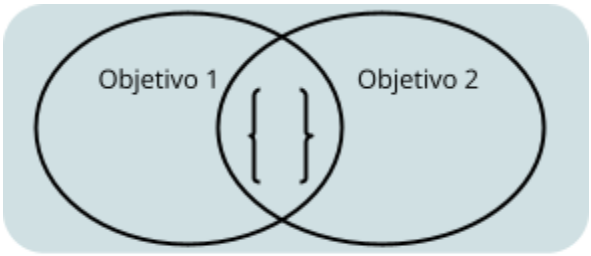
```
a.d()

class A {
   lateinit var b: B
   fun d() {
       b.d()
   }
}

class B {
   lateinit var c: C
   fun d() {
       c.d()
   }
}

class C {
   fun d() {
       // implementación final
   }
}
```

Alguna de las clases puede no tener demasiado comportamiento y quedar simplemente como un intermediario de las otras clases. La señal de un posible problema del Middle Man es que la clase intermediaria no juega ningún papel, sólo sirve como pasamanos: "too much simple delegation instead of really contributing to the application".



6.2.17 Divergent Change

Same class changes differently depending on addition



Objeto

Este bad smell ocurre cuando tenemos una clase que tiene más de un objetivo:

- objetivo 1: 2 atributos y 3 métodos
- objetivo 2: 3 atributos y 4 métodos

Pero esos atributos y métodos no se intersectan entre objetivos, son disjuntos. La clase tiene entonces una baja cohesión, y deberíamos separar esa clase en dos, porque si no cada modificación me llevará a actualizar los métodos y atributos del objetivo 1 ó del 2. La solución está en el refactor Extract class.

6.2.18 Shotgun Surgery

Small changes affect too many objects

Este ejemplo de Nick Harrison extraído de esta página ilustra lo que sucede cuando queremos llamar a un stored procedure para recuperar los clientes de una empresa. Si tenemos el siguiente método:

```
public ResultSet getClientes() {
      SqlConnection con = new SqlConnection(this.connectionString);
      SqlCommand cmd = con.createCommand();
      con.open();
      cmd.commandType = CommandType.StoredProcedure;
      cmd.commandText = "spGetClientes";
      return cmd.getResults(CommandBehavior.CloseConnection);
}
```

(el código es Java-C#-like, no nos interesa que compile sino el concepto)

- Cuando necesitemos llamar a otro stored procedure para traer la información de un cliente específico, seguramente habrá mucho en común con el método getClientes
- Cualquier cambio cross-aplicación tiene un altísimo impacto en nuestro sistema porque hay que modificar todas las llamadas a los stored procedures. Algunos

ejemplos:

- queremos medir la performance de los stored procedures para detectar cuellos de botella
- si la conexión a la base original está caída debemos conectarnos a otra base (que está en otro servidor)
- el área de QA recomendó eliminar el prefijo "sp" de todos los stored procedures y para salir a producción necesitamos hacer ese cambio (evidentemente se trata de un tema político pero que afecta a la implementación)

En general, cuando tenemos cuestiones arquitecturales aparece este *bad smell*, por eso su naturaleza *cross*. Algunos ejemplos

- manejo de la persistencia, y en general llamadas a componentes externos (como una base de datos o un servicio)
- manejo de errores: ¿qué queremos hacer cada vez que ocurra un error?
- activar o desactivar niveles de debug/trace/logueo por temas de performance, errores, etc.
- configuraciones de una aplicación
- en general donde necesitamos hacer algo en muchos lugares, ese algo involucra varios pasos (al menos dos) y queremos tener un comportamiento uniforme.

En esos casos, lo ideal es concentrar en un único punto esos pasos para que cuando nos pidan un cambio aparentemente pequeño eso no nos impacte en una gran cantidad de objetos.

Refactor que permite resolver este bad smell: Extract Class + Move Method

```
public ResultSet getClientes() {
      new StoredProcedure().getResults("GetClientes");
}

>>StoredProcedure (clase creada por nosotros)
public StoredProcedure() {
      SqlConnection con = new SqlConnection(this.connectionString);
}

public ResultSet getResults(String storedProc) {
      SqlCommand cmd = con.createCommand();
      con.open();
```

```
      cmd.commandType = CommandType.StoredProcedure;
      cmd.commandText = "sp” + storedProc; // volamos el prefijo
      return cmd.getResults(CommandBehavior.CloseConnection);
}
```

6.3 ¿Cómo hacer refactoring? Refactors más comunes

6.3.1 Comentarios

Cualquier comercial de Sprayette debería incluir deshacerse del código comentado que lleva años juntando polvo. Código que uno lee release tras release y que sólo deja la sensación de que lo comentado algún día servirá para arreglar un problema que todavía no apareció. *Moraleja*: no comentar código y de existir, eliminarlo. Utilicemos un repositorio para manejar el versionado de nuestros fuentes.

6.3.2 Falta de polimorfismo

- "Tell, don't ask": pasamos de sentencias case (if múltiples)

```
class Empleado {
   fun porcentaje() =
      when (tipoEmpleado) {
          "E" -> porcentajeEfectivo()
          "J" -> porcentajeJerarquico()
          else -> porcentajeIndependiente()
      }
```

a delegar en objetos que comparten la misma interfaz, donde las implementaciones de cada porcentaje pueden corresponder a subclases de Empleado (Efectivo, Jerarquico e Independiente) o bien a strategies Efectivo, Jerarquico e Independiente de tipo de empleado.

- State/Strategy/Null Object patterns evita preguntas

En el State, estados que derivan en distintos comportamientos se vuelven polimórficos.
En el Strategy, los algoritmos se intercambian.
En el Null Object no queremos distinguir el caso particular de cuando una variable no tiene valor, entonces para no estar preguntando todo el tiempo ésto

```
fun someMethod() {
   if (seleccionado != null) {
```

```
        seleccionado.someMethod()
    } else {
        // ...
    }
}
```

trabajamos con un objeto que maneja el caso nulo de manera polimórfica:

```
fun someMethod() {
    seleccionado.someMethod()
}
```

6.3.3 Código duplicado

- Extract Method dentro de la misma clase.

Ejemplo: para un sistema que necesita Papá Noel calculamos la cantidad de maldad de una acción que hace un chico, que es "el resultado de multiplicar la gravedad de dicha acción por la cantidad de afectados que tuvo (en el caso de que el responsable se haya arrepentido, sólo se usa la mitad de la gravedad)"

En la clase Acción:

```
class Accion {
    fun getCantidadMaldad() =
        if (seArrepintio) {
            gravedad * 0.5 * cantidadAfectados()
        } else {
            gravedad * cantidadAfectados()
        }
}
```

Si proponemos llevar afuera la pregunta de si se arrepintió, nos queda en la misma clase Acción:

```
class Accion {
    fun getCantidadMaldad() = gravedad * cantidadAfectados() *
coeficienteAjuste()

    fun coeficienteAjuste() = if (seArrepintio) 0.5 else 1.0
}
```

De esta manera se elimina el código duplicado (multiplicación de cantidad de afectados por gravedad)

- Entre clases hermanas se puede extraer el comportamiento y ubicarlo en una superclase común a ambas

Ejemplo: "Las acciones buenas y las malas tienen personas que se ven afectadas por dicha acción", esto se implementa con una variable de instancia

```
val afectadas = mutableListOf<Persona>()
```

en AccionBuena y AccionMala respectivamente. Para filtrar las personas malas afectadas disponemos de un método:

```
fun malasPersonasAfectadas() = afectadas.filter { !it.esBuena() }
```

El mismo código está en AccionMala y en AccionBuena. Podemos pararnos sobre el método, botón derecho: Refactor > Pull-up.

- Si tenemos dos clases hermanas cuyo comportamiento es el mismo, pero una tiene una ligera especialización, una opción es ubicar el comportamiento común en la superclase y desde las subclases invocarlas mediante super.

Ejemplo: el volumen que necesitan los productos para ser almacenados se mide en base al alto, ancho y largo. El volumen que necesitan los productos especiales es el doble de los productos comunes. Tenemos la clase Producto con el siguiente método:

```
fun volumen() = alto() + ancho() + largo()
```

Para evitar que ProductoEspecial haga lo mismo pero multiplicado por dos podemos utilizar super:

```
class ProductoEspecial : Producto() {
   override fun volumen() = super.volumen() * 2
}
```

Esto delega el comportamiento default a la superclase y permite no verse impactado por un cambio en el cálculo del volumen default (puedo modificar las tres variables por una variable volumen en metros cúbicos sin tener que tocar la subclase).

- También el pattern **template method** funciona como un mecanismo para evitar duplicación de código. En el mismo ejemplo del volumen, podemos tener una clase abstracta Producto con un método default que sea getVolumen(), implementado de la siguiente manera:

```
abstract class Producto {
   fun alto() = 0
   fun ancho() = 1
   fun largo() = 1

   fun volumenBase() = alto() + ancho() + largo()
   fun volumen() = coeficienteAjuste(volumenBase())
   abstract fun coeficienteAjuste(volumenBase: Int): Int
}

class ProductoComun : Producto() {
   override fun coeficienteAjuste(volumenBase: Int) = volumenBase

class ProductoEspecial : Producto() {
   override fun coeficienteAjuste(volumenBase: Int) = volumenBase * 2
```

(el lector puede buscar soluciones alternativas)

- Entre clases no relacionadas, bien podemos crear una superclase en común, o bien crear un componente nuevo y referenciar a él (un Helper, un Util o bien un objeto al que tenga sentido darle un nombre)

7 Limitaciones del Refactoring

- Sistemas externos que tienen una interfaz incómoda donde no tenemos control limitan nuestras ideas de diseño, aun así existe la posibilidad de construir wrappers (adapters o decorators, que veremos más adelante) para mitigar este problema.
- Los cambios internos de interfaz requieren consenso entre equipos de trabajo (lo que produce una inevitable tensión), y hay que implementarse gradualmente: deprecar un método o clase y ofrecer una nueva manera de resolver un requerimiento y planear el reemplazo de manera escalonada.
- El código *legacy* (heredado), sin testear o con un porcentaje muy bajo de cobertura nos pone en un aprieto grande. Cuando no sabemos por dónde empezar quizás hay que buscar las partes más relevantes o las más utilizadas para hacer el cambio paulatino.

8 Conclusión

Para mitigar los efectos de la entropía de los sistemas, el refactoring es un proceso que ayuda a permitir que los componentes se adapten para soportar nuevas funcionalidades, o bien facilita la detección de errores en nuestro modelo. Entrenar al equipo de desarrollo de software en las técnicas de refactoring es un paso significativo para establecer un circuito iterativo de un sistema que mejore continuamente, que debe ir acompañado de

- una práctica profesional y responsable
- entornos tecnológicos que promuevan refactorizaciones
- y por sobre todo, la automatización de las pruebas unitarias, para comprobar que la refactorización no se entremezcle con el agregado de nuevos requerimientos.