Reificando Comportamiento

Versión 0.1
Mayo 2020

Versiones

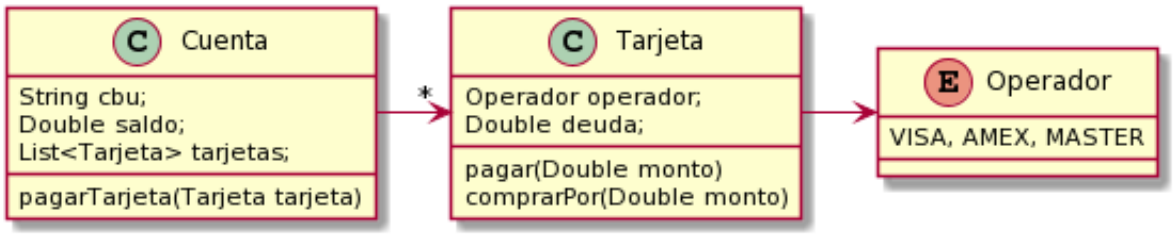
Autores principales	Versión	Fecha	Observaciones
Rodolfo Caputo	0.1	2020	Versión original

Introducción

"*Reifi-qué?*" Clásicamente estamos acostumbrados a separar nuestra lógica de programación en dos grandes grupos: Datos y Comportamiento. Los *datos* son las variables que almacenan la entrada, los estados intermedios y la salida de nuestro *comportamiento*, el cual procesa esas entradas para generar sus correspondientes salidas en forma de información.
Si bien el paradigma Orientado a Objetos nos propone encapsular estas dos ideas dentro de un todo coherente - una **cosa**, un *objeto* -, dentro del objeto mismo esta diferenciación sigue vigente: atributos internos (que apuntan a otros *objetos*) vs. métodos (que operan sobre otros *objetos*).
El proceso por el cual una abstracción de nuestro dominio (como un Usuario, una Venta, un Contrato, etc) se constituye en un *objeto* -una **cosa**- en nuestro modelo de objetos, es conocido como ***cosificación***, o ***reificación***[1]***.***
Los datos tienen muchas ventajas sobre las operaciones, como poder ser almacenados, pasados por parámetros, etc. Por ello, nos resulta interesante poder ver a las operaciones como datos, para aprovechar esas ventajas. Sin embargo, no queremos perder su primigenia calidad de "comportamiento". Entonces, podemos decir que nos interesa poder analizar a las operaciones como conjunciones de comportamiento + datos, es decir: objetos -o ***cosas***.
En este texto analizaremos cómo modelar operaciones como objetos y qué ventajas puede esto proporcionarnos.

El caso del Banco Valencia

Banco Valencia es una compañía líder en el mercado bancario que cuenta con un modelo muy simple: Cada usuario puede tener una cuenta corriente y una o varias tarjetas de crédito de distintas categorías las cuales acumulan una deuda al realizar compras y, a fin de mes, pueden ser canceladas en su totalidad con el saldo en cuenta.



[1] El término *reificar* tiene su origen en el vocablo latino *res* que significa "cosa" (Ej, la "República" era, en la Antigua Roma, la *res publicae,* la "cosa pública") y llega a nuestro español tras pasar por varias adaptaciones en otros idiomas. La nuestra, viene del inglés *reify.*

```
class Cuenta {
  String cbu;
  Double saldo;
  List<Tarjeta> tarjetas;

  void pagarTarjeta(Tarjeta tarjeta){
    Double montoAPagar = Math.min(this.saldo, tarjeta.getDeuda());
    tarjeta.pagar(montoAPagar)
    saldo -= montoAPagar
  }

}

class Tarjeta {
  Operador operador;
  Double deuda;

  void pagar(Double monto){
    deuda -= monto;
  }

  void comprarPor(Double monto){
    deuda += monto;
  }
}
```

Cuando un usuario desea pagar una tarjeta, la selecciona, indica la operación de pago y el sistema automáticamente descuenta de su saldo el total del resumen y reduce la deuda de la tarjeta. De no haber saldo suficiente, se cancela la deuda por el máximo monto disponible.

Pague ya! (pero no tan ya)

Si bien a los ejecutivos de Banco Valencia les encanta recibir el dinero de sus clientes, comenzaron a encontrarse con un inconveniente: El sistema de homebanking les permite a los usuarios pagar literalmente *cuando quieran.* Y muchas veces eso implica tener usuarios enviando pagos a altas horas de la noche, mientras se corre el *clearing*[2] bancario.

Dado que el clearing consolida balances, cuentas y deudas del banco, es un momento durante el cual no se pueden hacer operaciones que afecten esos mismos balances, cuentas y deudas. Es por ello que durante el horario del clearing, el banco decide restringir los pagos de deudas de tarjetas:

[2] El “clearing” es un proceso que siguen las entidades bancarias para cruzar operaciones entre ellos, como depósitos de cheques y cancelaciones de deudas mutuas.

```
class Cuenta{
  (...)

  boolean esHorarioDeClearing(){
    return new LocalDateTime().getHours() > 0
    && new LocalDateTime().getHours() < 3
  }

  void pagarTarjeta(Tarjeta tarjeta){
    if(esHorarioDeClearing()){
      throw new ClearingEnCursoException(
        "No es posible procesar su pago en este horario"
      )
    }else{
      Double montoAPagar = Math.min(this.saldo, tarjeta.getDeuda());
      tarjeta.pagar(montoAPagar)
      saldo -= montoAPagar
    }
  }
}
```

Al principio, todo parece funcionar bien… hasta que comienzan a llegar innumerables quejas de usuarios noctámbulos. Los usuarios no pueden pagar su tarjeta, luego se olvidan y la tarjeta comienza a acumular deuda impaga. Al parecer, pagar a las 2 am era mas importante de lo que parecía…

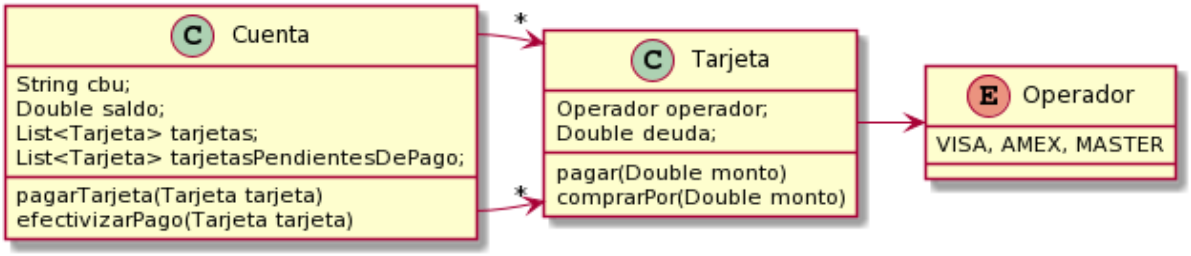
Programando el pago de tarjetas

Dada la clara inconveniencia de continuar impidiendo los pagos de madrugada, la dirección le pide al equipo de sistemas una solución al problema:

> *“Como tenedor de tarjetas de crédito, quiero poder ‘programar’ los pagos que se realicen de trasnoche, para que sean impactados al dia siguiente”*

De esta forma, si un usuario desea pagar su tarjeta a las 2 de la mañana, el pago no debe -como antes- realizarse en el momento, pero debe tomarse nota de que esa tarjeta fue pagada por el usuario, para que a la mañana siguiente un operador del banco pueda terminar de efectivizar dicho pago.

Es entonces que el equipo de desarrollo se lanza a codificar una solución bastante simple:




```
class Cuenta{
  String cbu;
  Double saldo;
  List<Tarjeta> tarjetas;
  List<Tarjeta> tarjetasPendientesDePago;

  (...)

  void agregarTarjetaPendientePago(tarjeta){
    this.tarjetasPendientesDePago.add(tarjeta)
  }

  void quitarTarjetaPendientePago(tarjeta){
    this.tarjetasPendientesDePago.remove(tarjeta)
  }

  void pagarTarjeta(Tarjeta tarjeta){
    if(esHorarioDeClearing()){
      agregarTarjetaPendientePago(tarjeta)
    }else{
      impactarPago(tarjeta)
    }
  }

  void efectivizarPago(Tarjeta tarjeta){
    quitarTarjetaPendientePago(tarjeta)
    impactarPago(tarjeta)
  }

  void impactarPago(Tarjeta tarjeta){
    Double montoAPagar = Math.min(this.saldo, tarjeta.getDeuda());
    tarjeta.pagar(montoAPagar)
    saldo -= montoAPagar
  }
}
```

De esta forma, cuando un usuario quiere pagar fuera del horario permitido, la tarjeta se agrega a una lista de tarjetas pendientes de pago y a la mañana siguiente un operador puede ver esa lista y efectivizar el pago de esas tarjetas, ya dentro del horario permitido. Si el pago es dentro del horario permitido, la efectivización es instantánea.

¿Pago mínimo o total?

¡Enhorabuena! Durante meses el sistema anda de maravilla: Los usuarios pueden pagar sus tarjetas a la hora que deseen, la tasa de olvidos está en mínimos y el clearing se desarrolla sin intervenciones molestas.
Sin embargo, como todo lo bueno en esta vida, esta armonía tiene sus días contados. Resulta que, habiendo minimizado los casos de olvido, el banco sigue notando que un porcentaje importante de tarjetas siguen quedando impagas al final de cada período. Y tras realizar una encuesta a sus clientes descubren la razón: Muchos usuarios no llegan todos los meses a cubrir el total de sus deudas y deciden no pagar dado que se quedarían sin saldo en sus cuentas.

Es por ello el equipo de desarrollo es nuevamente citado por el área de Producto y se les comunica el siguiente requerimiento:

> *"Como tenedor de tarjetas de crédito, quiero poder elegir el monto a pagar para poder cubrir parte de la deuda -evitando que genere intereses- sin quedarme sin saldo en la cuenta"*

"¡Pero claro que sí! Nosotros nos encargamos" Contesta el líder del equipo (El resto lo mira de reojo). Total… ¿qué tan complicado puede ser? Es agregar un parámetro...

Los desarrolladores se lanzan entonces a "agregar el parámetro" en el método pagar cuando de repente…

```
class Cuenta{

  void pagarTarjeta(Tarjeta tarjeta, Double monto){
    if(esHorarioDeClearing()){
      agregarTarjetaPendientePago(tarjeta)
    }else{
      impactarPago(tarjeta, monto)
    }
  }

  void impactarPago(Tarjeta tarjeta, Double montoAPagar){
    Double montoPosible = Math.min(this.saldo, montoAPagar);
    tarjeta.pagar(montoPosible)
    saldo -= montoPosible
  }
```

```
  void efectivizarPago(Tarjeta tarjeta){
    quitarTarjetaPendientePago(tarjeta)
    impactarPago(tarjeta)  ← COMPILATION ERROR!
  }

}
```

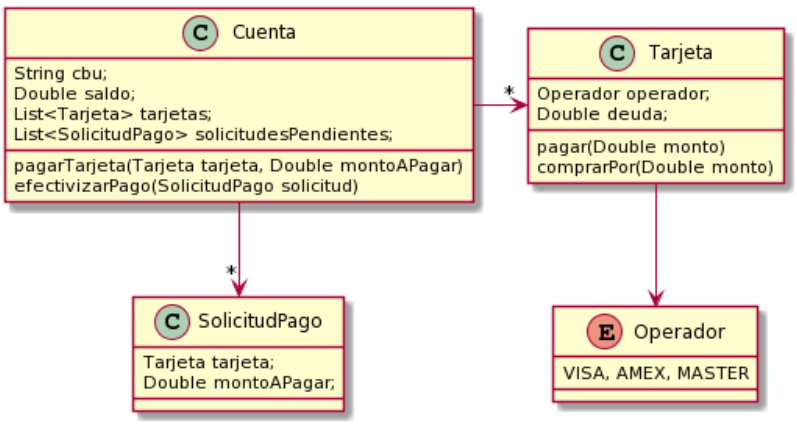
¡¿Y ahora?! ¿De dónde sacamos este monto? ¡Hay que guardarse el monto que el usuario decidió pagar cuando programó el pago! Bueno… Si lo pensamos bien, es el mismo problema que ya resolvimos exitosamente antes con una simple lista… ¿no?

El equipo comienzan a analizar la alternativa de, por imitación, agregar otra lista:

```
class Cuenta{
  String cbu;
  Double saldo;
  List<Tarjeta> tarjetas;
  List<Tarjeta> tarjetasPendientesDePago;
  List<Double> montosPendientesDePago;
```

Sin embargo, algo no parece ir del todo bien:

- Pero… y cómo *matcheamos*[3] esos montos con sus correspondientes tarjetas?-, dice uno.
- ¿Por orden? - esboza otro, más joven.
- Podía andar… pero no sé, algo no me cierra del todo… - retruca el primero.
- Es que hay un acoplamiento implícito - tira un tercero -. Si te fijás, estamos descansando mentalmente en que nunca nadie va a agregar una tarjeta sin monto, o un monto sin tarjeta. Y eso no está evidente en ningún lado… - completa.
- Mmm.. sí. Aparte como que cada tarjeta con su monto son un *algo*. Están asociadas en una especie de… “Solicitud de Pago”, o algo así... - acota el más joven.
- Tienen razón - reconoce el primero -. Y bueno, ¡modelemos eso! - cierra con entusiasmo.



[3] “Asociamos”, del inglés “match”

```
class SolicitudPago{
  Tarjeta tarjeta;
  Double montoAPagar;
}

class Cuenta{
  (...)
  List<SolicitudPago> solicitudesPendientes;

  void agregarSolicitudPago(SolicitudPago solicitud){
    this.solicitudesPendientes.add(solicitud)
  }

 void quitarSolicitudPago(SolicitudPago solicitud){
    this.solicitudesPendientes.remove(solicitud)
  }

  void pagarTarjeta(Tarjeta tarjeta, Double montoAPagar){
    if(esHorarioDeClearing()){
      agregarSolicitudPago(new SolicitudPago(tarjeta,montoAPagar))
    }else{
      impactarPago(tarjeta, montoAPagar)
    }
  }

  void efectivizarPago(SolicitudPago solicitud){
    quitarSolicitudPago(solicitud);
    impactarPago(solicitud.getTarjeta(), solicitud.getMontoAPagar())
  }

  void impactarPago(Tarjeta tarjeta, Double montoAPagar){
    Double montoPosible = Math.min(this.saldo, montoAPagar);
    tarjeta.pagar(montoPosible)
    saldo -= montoPosible
  }
}
```

De esta forma, agregamos una nueva abstracción a nuestro sistema: la Solicitud de Pago. Esta nueva abstracción nos permite: a) Asociar tarjetas a pagar con su correspondiente monto de una forma explícita y b) Reflejar en el sistema el concepto de “pago programado” que venía arrastrándose desde hacía un tiempo.

Ahora, el operador podrá acceder a la lista de solicitudes y, al efectivizarla, impactar el pago con de la tarjeta solicitada por su correspondiente monto.

Compras trasnochadas

Banco Valencia está más que conforme: Sus clientes pueden pagar en el horario que desean, el clearing funciona perfecto y casi nadie deja de pagar al menos parcialmente las tarjetas ahora que pueden elegir cuánto pagar.
No obstante, la modernidad acecha a la vuelta de la esquina y la necesidad de nuevos requerimientos se hace presente una vez más.

Resulta que cada vez que se procesa una compra, se ejecuta el siguiente código, similar al de los pagos:

```
class Cuenta{
  void procesarCompra(Tarjeta tarjeta, Double monto){
    if(esHorarioDeClearing()){
      throw new ClearingEnCursoException(
        "No es posible procesar su compra en este horario"
      )
    }else{
      tarjeta.comprarPor(montoAPagar)
    }
  }
}

class Tarjeta {
  (...)

  void comprarPor(Double monto){
    deuda += monto;
  }
}
```

Henos aquí que, de forma similar a los pagos, es imposible realizar compras durante el horario del *clearing*. Y esto originalmente no parecía ser un problema, dado que prácticamente ningún negocio en el que uno fuera a pagar con una tarjeta de crédito estaría abierto tan tarde… hasta que las compras con tarjeta de crédito comenzaron a masificarse. El efectivo fue desapareciendo, y de estar reservadas para compras grandes, los plásticos comenzaron a ser utilizados para todo tipo de compras, hasta las más triviales, en cualquier horario. Y ni hablar de las compras por internet.

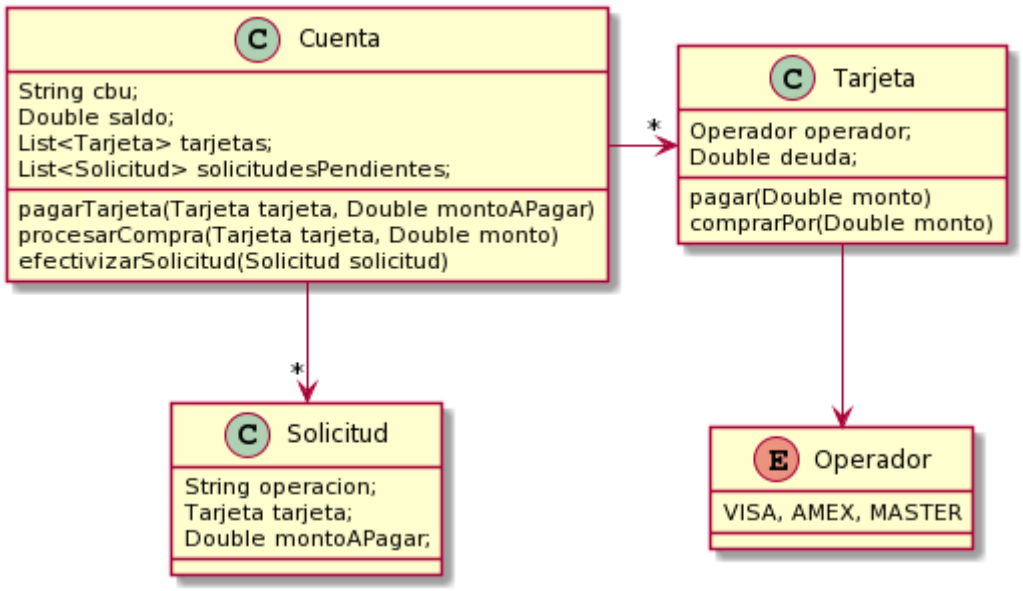
Es entonces que la restricción horaria comienza a generar problemas con los usuarios, quejas, y a causar que los clientes de Banco Valencia pierdan oportunidades de utilizar su tarjeta. Es cuestión de tiempo hasta que el equipo de Producto solicita el próximo requerimiento:

“Como tenedor de tarjetas de crédito, quiero poder realizar compras de trasnoche y que estas sean impactados al dia siguiente”

Y entonces el equipo comienza a discutir:

- Bueno, hacemos lo mismo que antes: ¡“Solicitud de compra”! - lanza uno.
- Si, ya fue, ¿para qué complicarnos? - apoya otro.
- Mmm, pero… ¿No va a haber como un montón de logica medio repetida? Onda… Lo de meterlo y sacarlo de la lista, la estructura… - contraargumenta el más joven.
- Sí… eso es cierto. - admite el primero,
- Aparte, solicitud de pago, solicitud de compra, todo porque mientras está el clearing no se puede hacer nada… ¿No ven un *patrón* que se repite? - se entusiasma el más joven.
- Sí, es verdad… Siempre que tenemos que hacer algo que toque saldos, de repente hay que poder patearlo en el tiempo para que se impacte a la mañana - se acopla el segundo.
- ¡Claro! “Solicitud de A”, “Solicitud de B”... Al paso que vamos vamos a terminar con quichicientas listas para que el operador las mire a la mañana y las impacte.
- Entonces podríamos modelar la “Solicitud” como algo genérico que se guarda para ejecutarse más adelante!- propone el primero.

Y se lanzan a implementar lo discutido:

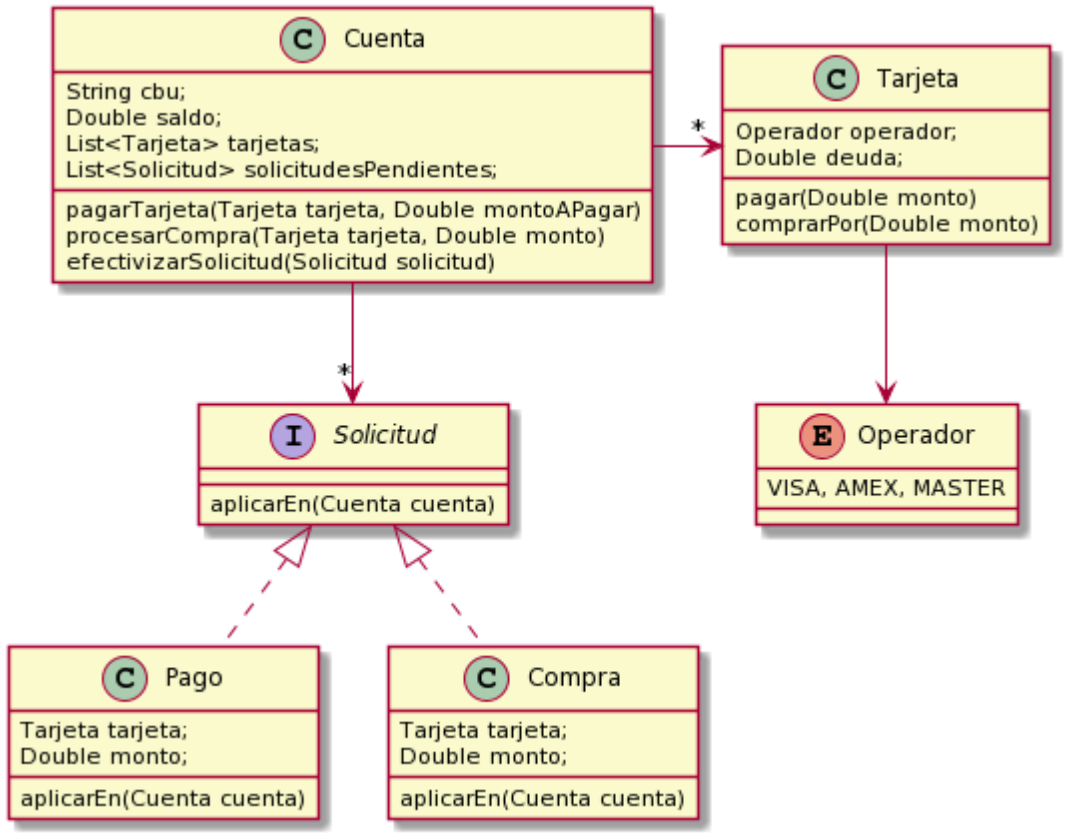


```
class Cuenta{

  void pagarTarjeta(Tarjeta tarjeta, Double montoAPagar){
    if(esHorarioDeClearing()){
      agregarSolicitud(new SolicitudPago("PAGAR",tarjeta,montoAPagar))
    }else{
      impactarPago(tarjeta, montoAPagar)
    }
  }

  void procesarCompra(Tarjeta tarjeta, Double monto){
    if(esHorarioDeClearing()){
      agregarSolicitud(new SolicitudPago("COMPRAR",tarjeta,monto))
    }else{
      tarjeta.comprarPor(montoAPagar)
    }
  }

  void efectivizarSolicitud(SolicitudPago solicitud){
    quitarSolicitudPago(solicitud);
    if(solicitud.getOperacion() == "PAGAR"){
      impactarPago(solicitud.getTarjeta(), solicitud.getMonto())
    }
    if(solicitud.getOperacion() == "COMPRAR"){
      solicitud.getTarjeta().comprarPor(solicitud.getMonto())
    }
  }

  void impactarPago(Tarjeta tarjeta, Double montoAPagar){
    Double montoPosible = Math.min(this.saldo, montoAPagar);
    tarjeta.pagar(montoPosible)
    saldo -= montoPosible
  }
}

class Solicitud{
  String operacion;
  Tarjeta tarjeta;
  Double monto;
}
```

Todo está perfectamente implementado y parece funcionar. Aún así, hay algo que no cierra del todo:

- Che… - tira el más joven, dubitativo - Me hace ruido el *if* ese con “Comprar” y “Pagar” - completa.
- Mmm… sí, no está muy lindo, podríamos usar *enums* para no pifiarle al string… - responde el mas senior.
- No, pero *aparte…* - completa - No es tremendo *Type Test*[4] eso?
- Ahhh, eso decías! Sí, la verdad que nos sacamos el problema de tener varias listas con solicitudes diferentes pero igual nos arriesgamos a tener un *if* interminable.
- Cierto. Y encima el método efectivizar hace muchas cosas distintas… no es muy cohesivo que digamos… - acota el tercero.
- Tal vez deberíamos *refactorizar* ese código para que delegue en objetos polimórficos - sugiere el más experimentado.
- ¡Me parece una genial idea! - finaliza el más joven.

Y entonces llegan a lo siguiente:



[4] Uso de ifs concatenados en detrimento del polimorfismo de objetos. Ver apunte de Code Smells

```
class Cuenta{
  (...)
  List<Solicitud> solicitudesPendientes;

  void pagarTarjeta(Tarjeta tarjeta, Double montoAPagar){
    if(esHorarioDeClearing()){
      agregarSolicitud(new Pago(tarjeta,montoAPagar))
    }else{(...)}
  }

  void procesarCompra(Tarjeta tarjeta, Monto monto){
    if(esHorarioDeClearing()){
      agregarSolicitud(new Compra(tarjeta,monto))
    }else{(...)}
  }


  void efectivizarSolicitud(SolicitudPago solicitud){
    quitarSolicitudPago(solicitud);
    solicitud.aplicarEn(this)
  }
}

interface Solicitud{
  aplicarEn(Cuenta cuenta){
}

class Pago implements Solicitud{
  Tarjeta tarjeta;
  Double monto;

  aplicarEn(Cuenta cuenta){
    cuenta.impactarPago(this.getTarjeta(), this.getMonto())
  }
}

class Compra implements Solicitud{
  Tarjeta tarjeta;
  Double monto;

  aplicarEn(Cuenta cuenta){
    this.getTarjeta().pagar(this.getMonto())
  }
}
```

Analizando lo que nos quedó

Tomémonos un segundo para reflexionar sobre los diferentes requerimientos que enfrentó el equipo de Banco Valencia, qué puntos en común tenían y qué conclusiones nos dejan.

Difiriendo la ejecución

Comencemos por analizar el primer problema que enfrentaron: Existía una **operación**, el *pago de tarjeta,* que -como toda operación- se veía reflejada un *método*: `pagarTarjeta(Tarjeta tarjeta)`.
Al igual que cualquier operación, su ejecución era inmediata: Si un usuario seleccionaba esa opción, el método se ejecutaba al instante, de forma *síncrona.* Esto significa que el sistema realizaba la operación y no le devolvía el control al usuario hasta no haberla completado.
Sin embargo, el primer requerimiento disruptivo fue el de "*programar*" los pagos. ¿Por qué? Porque lo que este requerimiento nos propuso es que, conceptualmente, la operación de pago dejará de ejecutarse de forma síncrona y pasará a hacerlo de forma *asíncrona*. Es decir, al disparar su ejecución, el sistema devolverá el control inmediatamente al usuario y la operación se ejecutará en un momento posterior, totalmente independiente del flujo de trabajo del usuario. La ejecución será *diferida* -pospuesta- en el tiempo

Podemos entonces decir que:

> *Las operaciones **síncronas** son aquellas cuya ejecución está **temporalmente acoplada** a su invocación, dándose inmediatamente después de esta última y **tomando el control del flujo de trabajo hasta finalizar**.*

> *Las operaciones **asíncronas** son aquellas cuya ejecución está t**emporalmente desacoplada** de su invocación, pudiendo comenzar a ejecutarse posteriormente y **retornando el control del flujo de trabajo** apenas son invocadas, **sin esperar a finalizar**.*

Métodos y sincronismo

Ahora bien, ¿por qué fue disruptivo este cambio de *síncrono* a *asincrono*? Esto ocurrió porque la ejecución clásica de los métodos en los lenguajes de programación Orientados a Objetos es siempre, naturalmente e *inevitablemente*, **síncrona.** Siempre que se envíe un mensaje a un objeto y el correspondiente método sea invocado, éste se ejecutará inmediatamente y tomará el control del flujo hasta finalizar.

Parámetros de las operaciones diferidas

Entonces, ¿cómo podemos lograr esa *ejecución diferid*a? Si observamos el caso de Banco Valencia, la respuesta fue bastante simple: no llamar al método que realizaba el trabajo, en este caso `impactarPago(Tarjeta tarjeta)`, hasta tanto no fuera estrictamente necesario (a la mañana siguiente). Pero para poder diferir esa ejecución hasta el día siguiente, había una pieza de información que era clave y necesaria: La tarjeta.
Esto ocurre porque, durante la ejecución del método, los parámetros pasados al mismo actúan como variables locales y contienen los datos necesarios para realizar la operación. No obstante, al posponer la ejecución perdemos ese "depósito de datos temporal" que son los parámetros.
La solución de Banco Valencia, entonces, fue guardar la tarjeta en un "depósito" mas duradero: atributos del objeto.

> *El primer problema a la hora de diferir la ejecución es: ¿Dónde almacenar los parámetros de la operación hasta tanto sea esta ejecutada?*

El siguiente problema surgió cuando los parámetros de la operación diferida se volvieron más complejos: En vez de solamente la tarjeta, se agregó el monto a pagar. Esto generó que una simple referencia no fuera suficiente para contener la complejidad de los parámetros de la operación, obligándonos a ver que estos parámetros no podían ser tratados como meros valores independientes: Estaban interrelacionados por medio la operación misma. De esa forma, los convertimos en atributos de un nuevo objeto que representara, indirectamente, esa operación: la Solicitud de Pago.

> *Al diferir una operación, sus parámetros pueden ser almacenados hasta tanto se ejecute en un objeto que la represente.*

La operación como objeto

Finalmente, el último requerimiento presentado consiste en una nueva operación, `procesarCompra`, de igual naturaleza que la anterior: Debe ser diferida desde la noche hasta la mañana siguiente.

Nuevamente tenemos un método cuya ejecución debe ser pospuesta, por lo que deberemos almacenar sus parámetros de entrada en un objeto dedicado mientras tanto.

Más aún, son tantas las similitudes entre la naturaleza diferida de la compra y el pago que los desarrolladores de Banco Valencia comienzan a ver un *patrón:* Una serie de conceptos similares, con la misma estructura, que se repiten. Y, en consecuencia, previendo futuras nuevas ocurrencias del mismo, empiezan a pensar una solución que se ajuste a este patrón genérico, en lugar de a cada caso particular.

Hasta este momento, tienen casi todo lo que necesitan para diferir una operación: objetos capaces de almacenar los parámetros de las mismas. Sin embargo, les falta el detalle mas importante: *la operación en sí.* Dada una solicitud con los parámetros de entrada correctos, ¿qué se debe hacer con ellos una vez que deba concretarse la ejecución?

Es en esta instancia cuando la respuesta surge casi naturalmente: Si ya tenemos una abstracción, un *objeto*, que representa indirectamente a la operación y que almacena los parámetros necesarios para ejecutarla, ¿por qué no agregarle también el comportamiento en sí? ¿Y cómo agregar comportamiento a un objeto? Naturalmente: en un método. Un método que representará la ejecución de lo que sea que esa operación haga, dependiendo de qué operación es.

De este modo, nuestro objeto representa completamente la operación como un conjunto de datos de entrada y comportamiento. Y, de forma similar a una función, su comportamiento es “ser aplicado”, haciendo para cada caso lo sea que la operación defina.

> *Al representar una operación como un objeto, sus parámetros podrán ser atributos de este objeto y su comportamiento será simplemente el de “ejecutarse” o “ser aplicado”*

Además, este objeto podrá implementar una interfaz, si se quisiera tratar diferentes operaciones de forma polimórfica.

Un paso más allá

Un detalle interesante a comentar es que en los ejemplos vistos los parámetros de las operaciones eran Tarjeta y Monto, pero esto se debe a una circunstancialidad del enunciado. Si el último requerimiento hubiera sido el segundo, inicialmente la operación ‘pagar’ solo habría necesitado almacenar la tarjeta, sin el monto. De la misma forma, si una operación futura necesitara parámetros completamente diferentes, esto no rompería el polimorfismo entre ellas, dado que los atributos son internos a cada objeto. Cabe destacar, además, que hubiera sido una decisión cuestionable constituir a la interfaz “Solicitud” como una superclase abstracta, por lo anteriormente expuesto.

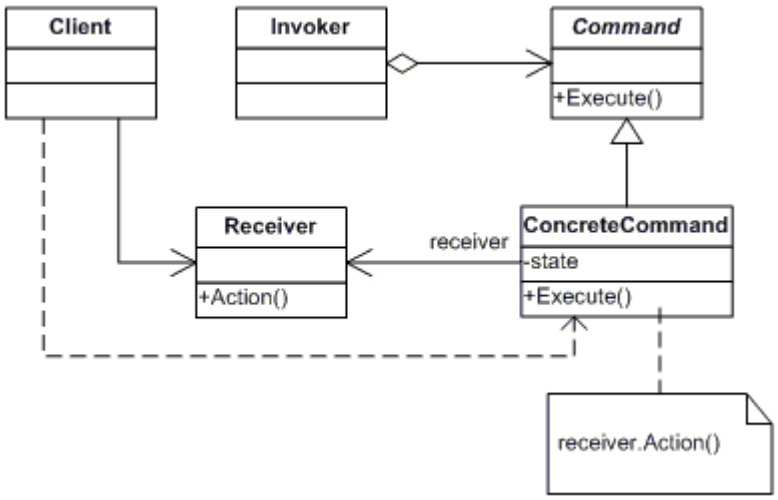
Otro detalle interesante para analizar es que lo que modelan nuestros objetos son “una operación”, por lo que el comportamiento propio de estos objetos (los mensajes que entienden) suelen ser del estilo de “aplicar”, “ejecutar”, “correr”, etc.

Finalmente, vale la pena mencionar que la idea de comportamiento como objeto -y como ciudadano de primera clase[5]- si bien es una noción funcional, existe hoy en día en la mayoría de los lenguajes de programación orientados a objetos en diversas formas (bloques de código en Smalltalk, clases anónimas en Java, *lambda expressions* o *arrow functions* en Java, Scala, Javascript y otros…). Todos estos son implementados como objetos genéricos que entienden el método “aplicar” o “ejecutar”, siguiendo las ideas anteriormente expuestas. Sin embargo, es interesante destacar que, debido a su carácter altamente genérico, tienen algunas limitaciones respecto de modelar objetos dedicados:

- Son valores sin nombre que no permiten representar una abstracción propia del sistema (a diferencia de clases como “Pago” o “Compra”)
- No son fáciles de almacenar en medios persistentes (bases de datos)
- En la mayoría de los casos tienen un comportamiento enteramente limitado a ejecutarse, mientras que modelando objetos dedicados pueden implementarse operaciones como “deshacer”, “componer”(ejecutar en cadena), etc.

Relación con el patrón Command

Esta idea suele aparecer tanto que está plasmada en el patrón Command, el cual es un patrón que permite reificar las operaciones para que un tercero pueda decidir qué hacer con ellas (invocarlas, cancelarlas, deshacerlas, etc). En el patrón de libro se definen las siguientes responsabilidades:



[5] Se conoce como “ciudadano de primera clase” o *first class citizen* a todo aquel valor que, en un determinado lenguaje de programación, puede ser asignado a una variable o pasado como parámetro. Los casos típicos son los valores primitivos y punteros en lenguajes procedurales; objetos en lenguajes orientados a objetos; funciones, tuplas en lenguajes funcionales...

Command: es la operación genérica, en nuestro caso la interfaz Solicitud y el mensaje es #aplicarEn(cuenta)

ConcreteCommand: son cada una de las operaciones concretas que vamos a cosificar, en nuestro caso clase Pago, clase Compra

Receiver: es el objeto que tenía la operación concreta que estaba modelada como un método, en nuestro caso

- para la clase Pago es Cuenta y la acción es #impactarPago(tarjeta, monto)
- para la clase Compra es Tarjeta y la acción es #pagar(monto)

Client: es el que conocía la operación original y sabe instanciar al concrete command, en nuestro caso la clase Cuenta.

Invoker: es el que se encarga de ejecutar la operación, en nuestro caso la interfaz gráfica que usa el operador del banco, es la encargada de ejecutar la operación[6].

Conclusión

Comenzamos analizando cómo las operaciones son naturamente modeladas como métodos. Vimos que esto nos impone algunas restricciones, como el sincronismo y la incapacidad de almacenar operaciones a posteriori.
Propusimos solucionar esto generando objetos que representen a una operación, siendo usualmente sus atributos los parámetros naturales de la misma y exhibiendo un comportamiento de naturaleza “ejecutar”que realizara el trabajo en cuestión.

Al implementar estas ideas, obtuvimos varias ventajas. Algunas de ellas pueden ser:

- Diferir la ejecución de acciones en el tiempo
- Almacenar acciones pendientes hasta tanto deban ser ejecutadas
- Cancelar una acción pendiente y nunca ejecutarla
- Implementar sistemas de aprobación y rechazo de acciones antes de ejecutarlas
- Deshacer acciones realizadas
- Desacoplar la ejecución del flujo de control (asincronismo)
- Hacer ‘viajar en el tiempo’ a un objeto al desaplicar o reaplicar operaciones

[6] Si, por ejemplo, esta acción se automatizara, el proceso automático que corriera estas operaciones cumpliría este rol