Problemas Resueltos – Patrones de Diseño

Maximiliano Cristiá

Ingeniería de Software 2
LCC – FCEIA – UNR
Rosario – Argentina
2021

Los problemas que presentamos en este apunte son simples problemas de aplicación de patrones de diseño.

1. Composite

El Departamento de Personal de una empresa necesita una aplicación para realizar diversos análisis sobre el organigrama de la empresa[1]. La estructura de personal de la empresa se divide en gerencias, subgerencias, departamentos y equipos. Los equipos están formados por empleados. Algunos análisis posibles son: determinar el número de empleados de una división; determinar la lista de empleados de una división; determinar el sueldo promedio de una división; etc. Notar que el Departamento de Personal no está tan interesado en la representación gráfica del organigrama.

Una de las cuestiones fundamentales que debemos definir es la estructura de datos que vamos a usar para mantener el organigrama de la empresa. Como el organigrama se representa como un árbol es razonable usar el patrón Composite para mantenerlo. Claramente los Compuestos de este Composite son `Gerencia`, `Subgerencia`, `Departamento` y `Equipo`; mientras que las Hojas son objetos de tipo `Empleado`; y el Componente lo denominaremos `División`. Por otra parte, vamos a definir como Operación al método `trabajaEn(i Empleado)` que retorna la `División` de menor nivel en la cual trabaja el `Empleado`; y al método `empleados()` que retorna la lista de `Empleados` que pertenecen a la `División` en cuestión.

Tengan en cuenta que si bien la descripción genérica de Composite menciona una única Operación, en realidad se pueden incluir en la interfaz tantas como se necesiten. Esto es algo que tienen que tener presente cuando estudien los patrones de diseño del libro de Gamma. Los autores usan una notación que requiere una cierta interpretación de cada descripción más allá de lo que está efectivamente escrito. Por ejemplo, en el caso de Composite, Operación representa una o más operaciones que se implementan de modo recursivo sobre la estructura de árbol del Composite. Que pueda haber más de una Operación no está dicho en la descripción ni lo implica la notación que ellos usan.

Entonces definimos el módulo `División` de la siguiente forma. La interfaz podría tener más

[1] Un organigrama es la representación gráfica de la estructura de una empresa o cualquier otra organización, que incluye las estructuras departamentales y, en algunos casos, las personas que las dirigen, hacen un esquema sobre las relaciones jerárquicas y competenciales de vigor. Ver más detalles en https://es.wikipedia.org/wiki/Organigrama.

métodos tipo *setters* y *getters* dependiendo de los datos que definen a una `División`.

MODULE	`División`	
EXPORTS	`setNombre(i String)`	
	`getNombre():String`	
	`setJefe(i Empleado)`	jefe de la `División`
	`getJefe():Empleado`	
	`trabajaEn(i Empleado):División`	
	`empleados():List(Empleado)`	
	`alta(i División)`	gestión de hijos
	`baja(i División)`	gestión de hijos
	`división(i Int):División`	gestión de hijos
	`parteDe():Division`	`División` padre

¿Por qué no incluimos en la interfaz de `División`, por ejemplo, un método para determinar la cantidad de empleados de una cierta `División` (puesto que es uno de los análisis requeridos)? La respuesta es la misma de siempre: si lo hiciéramos estaríamos incluyendo más de un ítem de cambio en ciertos módulos. En efecto, si incluyéramos ese método, el módulo implementaría la estructura de datos para el organigrama más el algoritmo para determinar la cantidad de empleados de una cierta `División`. Más aun, ¿por qué no incluiríamos el método que permite calcular el gasto en sueldos de una cierta `División`? ¿Y por qué no el que permite calcular el promedio de los sueldos de una `División`? Y si nos hubieran solicitado más análisis, ¿deberíamos continuar agregando métodos a la interfaz de los módulos que definen el COMPOSITE? ¿Cuál sería el resultado final de tal diseño? Claramente, terminaríamos teniendo un grupo de clases muy grandes que implementan una cantidad de funciones relativamente independientes entre ellas, y que podrían estar en ciertas versiones del sistema pero no en otras. Por ejemplo, podríamos implementar todas esas funciones en la versión comercial pero solo algunas en la versión gratuita. Pero entonces, ¿cómo haríamos para compilar la versión gratuita solo con algunas de estas funciones? ¿Tendríamos que tener dos versiones de cada módulo del COMPOSITE, una para la versión comercial y la otra para la gratuita? ¿Y si hubiera más versiones (premium, PyMES, etc.)? Claramente este diseño no escala.

Pueden ir pensando cómo implementar todas esas funciones.

Pero entonces, ¿por qué incluimos en la interfaz los métodos `trabajaEn()` y `empleados()`? Porque son las funciones esenciales que define el organigrama. Es decir, un organigrama determina la jerarquía de divisiones laborales de la empresa y quién trabaja en cada una. Si sacamos `trabajaEn()` o `empleados()` de la interfaz, el módulo no cumpliría con su especificación.

Notar que `División` incluye métodos que no están presentes en el patrón COMPOSITE pues no corresponden a un método tipo OPERACIÓN ni a métodos para gestionar los hijos. Esto es perfectamente normal en tanto y cuanto estos métodos sean esenciales para que el módulo cumpla con su especificación. Por ejemplo, si sacáramos `setNombre()` o `getNombre()` no podríamos darle un nombre a la `División` lo cual sería una falta respecto a los requerimientos.

Los otros módulos del diseño que representan divisiones del organigrama son los siguientes.

MODULE	`Gerencia` INHERITS FROM `División`

MODULE	`SubGerencia` INHERITS FROM `División`

MODULE	`Departamento` INHERITS FROM `División`

MODULE	`Equipo` INHERITS FROM `División`

Como pueden ver ninguno agrega métodos a la interfaz de `División`. En lo posible se trata de que todos los módulos de un COMPOSITE que representan grupos de elementos (e incluso los que representan elementos individuales) tengan exactamente la misma interfaz que el COMPONENTE. De esta forma el cliente del COMPOSITE no tiene que tener código que dependa del tipo de grupo de elementos. En nuestro ejemplo, las `Gerencias`, `SubGerencias`, `Departamentos` y `Equipos` se tratan de la misma forma (o sea uniformemente, o sea el cliente no tiene sentencias condicionales que dependen del tipo de agrupamiento). Muchas veces esto lleva a que sea necesario incluir métodos en algunas interfaces que no tienen mucho sentido o que no son necesarios. Por ejemplo, un `Equipo` podría no tener un jefe en cuyo caso `setJefe()` y `getJefe()` no deberían estar. Pero si no los ponemos el cliente deberá considerar con qué tipo de grupo de elementos está trabajando. Es decir, o el cliente tiene sentencias condicionales para tratar distintos grupos de elementos o cierto grupos de elementos tienen métodos que no tienen mucho sentido. En general se prefiere la última opción. En ese caso, los métodos que no deberían estar levantarán una excepción si son invocados o simplemente no harán nada.

Finalmente tenemos el módulo `Empleado`, que sí extiende la interfaz de `División`.

MODULE	`Empleado` INHERITS FROM `División`
EXPORTS	`setSueldo(i Sueldo)`
	`getSueldo():Sueldo`
	`setDomicilio(i Domicilio)`
	`getDomicilio():Domicilio`
	`... posiblemente más métodos ...`

Aquí se rompe la uniformidad de la interfaz de COMPOSITE. Como dijimos más arriba, en lo posible hay que evitar romper la uniformidad pero en ocasiones no quedan más alternativas. La interfaz de `Empleado` es necesariamente muy distinta a la de cualquiera de los niveles del organigrama por lo que es casi imposible lograr uniformidad en todo el COMPOSITE. De todas formas, `Empleado` hereda los métodos para la gestión de los hijos, aunque son totalmente innecesarios. Estos métodos tendrán implementaciones particulares que de alguna forma sean consistentes con las implementaciones de los otros módulos del COMPOSITE. Por ejemplo, `alta()`

y `baja()` no harán nada o lanzarán una excepción.

Como ya mencionamos Composite está pensado para brindar uniformidad al cliente. Esto se hace, en general, a expensas de la *seguridad de tipos*[2]. Es decir, uno podría dar de alta una `Gerencia` dentro de un `Equipo` lo que claramente no debería ocurrir. La alternativa es romper la uniformidad mediante seguridad de tipos. En este caso el método `alta()` de cada heredero de `División` debería esperar el tipo adecuado. Por ejemplo, `Equipo.alta(i Empleado)` y `Gerencia.alta(i SubGerencia)`.

Entonces, ¿cómo hacemos para evitar que se añada una `Gerencia` dentro de un `Equipo`? Una forma de hacerlo es que la implementación de `Equipo.alta()` controle (dinámicamente) el tipo del parámetro que recibe. La otra posibilidad es confiar o delegar en el cliente esta responsabilidad. Un ejemplo de control dinámico es el siguiente.

```
Equipo.alta(División d) {
  if type_of(d) = Empleado
  then agregar d como parte del equipo
  else lanzar una excepción
}
```

En general los lenguajes de programación orientados a objetos proveen funciones como `type_of()` que permiten determinar el tipo de un objeto en tiempo de ejecución. Ciertamente la solución no es muy elegante porque requiere predicar sobre el sistema de tipos de forma explícita (o *programática*). Por ejemplo, si en algún momento decidiéramos cambiar el nombre del módulo `Empleado` el compilador no nos daría un error de tipos en `Equipo.alta()` y la rama `then` nunca sería ejecutada.

Recuerden que cuando decimos que `trabajaEn()` y `empleados()` corresponden a Operación, estamos dando pistas de cómo deberían ser sus implementaciones. Más precisamente, *debemos* o *es muy aconsejable* implementar este tipo de métodos como se indica en la descripción del patrón. Les doy un ejemplo, asumiendo que todos los módulos del Composite declaran la variable `miembros` para almacenar los objetos `División` que pertenecen a una `División` dada[3].

```
División Empleado.trabajaEn(Empleado e) {return null;}

División Equipo.trabajaEn(Empleado e) {
  i = 1;
  while e != miembros[i] do i = i + 1;
  if miembros[i] == e
  then return this;
  else return null;
}

División Departamento.trabajaEn(Empleado e) {
  i = 1;
  while miembros[i].trabajaEn(e) == null do i = i + 1;
  return miembros[i].trabajaEn(e);
}
```

[2] 'type safety', en inglés.

[3] La implementación que les muestro no tiene en cuenta muchos casos de borde que deben ser correctamente programados.

Las implementaciones de `trabajaEn()` en `Gerencia` y `SubGerencia` son idénticas a la de `Departamento`.

Una cuestión de conceptual que tienen que tener presente cuando aplican Composite es que en general este patrón de diseño no está pensado para implementar un ítem de cambio en particular sino más bien brinda una solución con buenas propiedades cuando se trata de implementar estructuras de árbol. De todas formas, al usar Composite es muy simple añadir y quitar Compuestos y Hojas. Por ejemplo, sería muy simple añadir la división `Área` definiéndola como heredero de `División`.

Para pensar, discutir y analizar

1. ¿Qué ocurre si se le pasa un `Empleado` inexistente a `trabajaEn()`?
2. ¿Se puede implementar `trabajaEn()` en términos de `empleados()`?
3. ¿Qué ocurre si se le pasa a `división()` un entero que excede el número de divisiones de esa `División`? ¿Cómo sería una solución más robusta para recorrer los elementos de una `División`?
4. ¿Se puede implementar el método `empleados()` en términos de `división()`? ¿Y `división()` en términos de `empleados()`? Si alguna de las dos fuera posible, ¿qué regla de diseño se estaría violando?
5. ¿Dónde incluiría un método que retorna la lista de `Empleados` de una `División` en particular? ¿Cómo sería la implementación de ese método?
6. ¿Dónde incluiría un método que determina si dos `Empleados` trabajan en el mismo `Equipo`? ¿Cómo sería la implementación de ese método?
7. ¿Tendría sentido cambiar el tipo del parámetro de `trabajaEn()` a `División`? ¿Se podría usar para determinar la división de menor nivel a la que pertenece el parámetro? Si es posible muestre una implementación esquemática.
8. Completar los diseños con las cláusulas Imports.
9. Use las cajas Pattern para documentar el diseño visto en esta sección.

2. Strategy

La atención al público de una empresa se realiza en parte por medio de un *chat* donde los usuarios plantean sus problemas y los empleados de atención al público los atienden con el objetivo de resolver esos problemas. Uno de los problemas a resolver en este tipo de sistemas es la asignación de *chats* a los empleados. Es decir, cuando un usuario abre un *chat* para plantear su problema el sistema debe avisarle a uno de los empleados para que responda. Cada empleado puede atender varios usuarios al mismo tiempo y tiene una cola de pedidos que debe atender en cuanto uno de los problemas se resuelve o queda en espera por un tiempo considerable.

Existen varias políticas para asignar *chats* a empleados:

1. Lineal. Se organiza a los empleados en una suerte de lista circular de manera tal que cada vez que entra un nuevo pedido de ayuda se le asigna al siguiente en la lista.

2. Equitativa. El sistema asigna el nuevo pedido de ayuda al empleado que tiene la menor cantidad de *chats* asignados.
3. Temática. Cada pedido de ayuda se relaciona con un tema (por ejemplo: no me puedo registrar en el sistema; no puedo cambiar la contraseña; mi cuenta está bloqueada; etc.). Cada empleado está capacitado para atender pedidos relacionados con ciertos temas. Entonces el sistema asigna los pedidos según el tema del que se traten.

La idea es que si nosotros vamos a proveer el sistema de atención al público debemos diseñarlo de forma tal que:

1. Se puedan agregar, modificar y eliminar políticas de asignación de *chats*.
2. Los administradores del sistema puedan configurar el sistema con la política que mejor se ajusta a sus necesidades, si necesidad de detener y reiniciar el sistema.
3. Se puedan generar distintas versiones del sistema con más o menos políticas disponibles.

Vamos a focalizar el problema en el o los componentes que deben distribuir los pedidos de ayuda entre los empleados. En este sentido suponemos que existe el módulo `Chat` que recibe y envía mensajes entre un usuario y un empleado. Por otro lado, existe un objeto de tipo `ChatsAtender` asociado a cada empleado que contiene las instancias de tipo `Chat` que debe atender el empleado. Una interfaz simple para `ChatsAtender` puede ser la siguiente.

```
MODULE   ChatsAtender
EXPORTS  ChatsAtender(i Empleado)
         empleado():Empleado
         nuevoPedido(i Chat)
         atenderPedido():Chat
         pedidoResuelto(i Chat)
         cantidad():Int
```

Luego tenemos el módulo que más nos interesa que es el encargado de gestionar los pedidos a procesar. Una interfaz básica sería la siguiente.

```
MODULE   GestorPedidos
EXPORTS  nuevoChat(i Chat)
         nuevoEmpleado(i ChatsAtender)
         bajaEmpleado(i ChatsAtender)
         empleadosAtendiendo():List(ChatsAtender)
```

La idea es que el método `nuevoChat()` es invocado cada vez que se crea un nuevo objeto de tipo `Chat`. Este, a su vez, decide a cuál de los objetos `ChatsAtender` le asigna el pedido (invocando al método `nuevoPedido()` correspondiente).

Entonces, una primera solución para implementar las políticas de distribución de pedidos podría ser usar el constructor de `GestorPedidos` para configurar la política a usar y un selector en el método `nuevoChat()`.

```
GestorPedidos(Política p) {pol = p;}       // Política es un enumerado
```

```
nuevoChat(Chat c) {
  case pol is:
    lineal: lineal(c);
    equitativa: equitativa(c);
    temática: temática(c);
  endcase
}
```

Donde `lineal()`, `equitativa()` y `temática()` son los métodos privados del módulo `GestorPedidos` que implementan cada una de las políticas.

Claramente la solución es simple, intuitiva y funciona. Pero está mal desde el punto de vista del diseño. A esta altura deberían tener claro el porqué.

La solución más correcta pasa por aplicar el patrón de diseño Strategy entendiendo que cada política de distribución de pedidos de ayuda es una EstrategiaConcreta distinta y que el `GestorPedidos` es el Contexto sobre el cual actúan las estrategias (el Contexto es el conjunto de datos con el cual tienen que trabajar cada Estrategia). Entonces, tenemos que definir una interfaz para las políticas de distribución de pedidos de ayuda (Estrategia) de manera tal que cada política concreta sea un heredero de ella.

Module	`PolíticaDistribución`
exports	`PolíticaDistribución(i GestorPedidos)`
	`distribuir(i Chat)`

Como se puede ver usamos el constructor para pasar el Contexto a la `PolíticaDistribución`. Además, el método `distribuir()` corresponde a InterfazAlgoritmo. Ahora podemos declarar las políticas concretas.

Module	`Lineal` inherits from `PolíticaDistribución`

Module	`Equitativa` inherits from `PolíticaDistribución`

Module	`Temática` inherits from `PolíticaDistribución`

Finalmente, en lugar de usar el constructor de `GestorPedidos` para pasar un valor de un tipo enumerado, lo usamos para pasar un objeto de tipo `PolíticaDistribución` y `nuevoChat()` invoca a `distribuir()` cada vez que llega un nuevo pedido de ayuda.

```
GestorPedidos(PolíticaDistribución p) {pol = p;}

nuevoChat(Chat c) {pol.distribuir(c);}
```

Tengan en cuenta que `distribuir()` invocará a `empleadosAtendiendo()` para obtener la lista

de ChatsAtender que están activos (es decir los datos del Contexto) y seleccionar uno de ellos al cual asignarle c. Por ejemplo:

```
Lineal(GestorPedidos g) {
  gestor = g;
  último = 0;                    // variable de estado
}

Lineal.distribuir(Chat c) {
  List<ChatsAtender> procesadores = gestor.empleadosAtendiendo();
  procesadores.get(último+1).nuevoPedido(c);
  último = último + 1;
}
```

En main() configuramos la PolíticaDistribución que hayan seleccionado los administradores con el GestorPedidos adecuado.

```
main() {
  PolíticaDistribución p;
  GestorPedidos g;
  ......
  p = new Lineal(g);     // los admin seleccionaron Lineal
  ......
}
```

Para pensar, discutir y analizar

1. Explique cómo agregaría una nueva política de asignación de pedidos de ayuda.
2. Muestre una implementación esquemática del método Equitativa.distribuir().
3. Muestre una implementación esquemática del método Temática.distribuir().
4. Muestre una implementación esquemática del código que permitiría que los administradores cambien en tiempo de ejecución la política de asignación de pedidos de ayuda.
5. Explique cómo haría para generar dos versiones del sistema con distintas políticas de asignación de pedidos de ayuda. Puede pensar la solución a nivel de la gestión del código fuente (control de versiones, e.g. Git) o a nivel de directivas al compilador (automatización de binarios, e.g. Make).
6. Describa una versión del diseño visto más arriba pero donde el Contexto le pasa los datos a la Estrategia.
7. Completar los diseños con las cláusulas Imports.
8. Use las cajas Pattern para documentar el diseño visto en esta sección.

3. Decorator

Volvemos sobre el problema de las cuentas bancarias visto en el apunte "Problemas resueltos – Diseño". Consideramos el caso en que tenemos dos tipos de cuentas bancarias: cajas de ahorros y cuentas corrientes. Además consideramos el requisito que impide hacer extracciones de más de 10000 pesos de cajas de ahorros.

Antes de aplicar el patrón de diseño Decorator vamos a analizar una solución basada únicamente en usar herencia. Asumimos que ya hemos definido los módulos del tercer problema que analizamos en "Problemas resueltos – Diseño".

Module	`CuentaBancaria`
imports	`Monto`
exports	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`

Module	`CajaAhorros` inherits from `CuentaBancaria`

Module	`CuentaCorriente` inherits from `CuentaBancaria`

Ahora queremos ver cómo incorporamos el requerimiento:

- *No se pueden realizar extracciones de más de 10000 pesos*

Como dijimos, vamos a analizar una solución basada únicamente en usar herencia. Entonces, definimos el siguiente módulo:

Module	`CajaAhorros10000` inherits from `CajaAhorros`

Este módulo implementaría el requisito mencionado en su método `extraer()` mientras que la implementación de todos los otros métodos sería idéntica a la de `CajaAhorros`. Este diseño está bastante bien porque mantiene separados los ítems de cambio pero tiene un problema que aun no se manifiesta pero está implícito. Imaginemos que aparece el siguiente requerimiento sobre las cajas de ahorro:

- *No se pueden realizar más de 4 extracciones mensuales*

Para implementarlo, podríamos volver a aplicar la misma idea y en consecuencia definimos el siguiente módulo.

Module	`CajaAhorros4` inherits from `CajaAhorros`

Este módulo implementaría el último requisito, manteniendo los ítem de cambio separados. Sin embargo, ¿cómo deberíamos proceder si se deben combinar los dos requisitos (o sea no se

pueden hacer extracciones de más de 10000 pesos y no se pueden hacer más de 4 mensuales)? Podríamos definir el siguiente módulo.

Module	CajaAhorros10000-4 inherits from CajaAhorros

Ahora tendríamos tres herederos de `CajaAhorros`. Pero supongamos que aparece un nuevo requerimiento:

- *Se deben informar los depósitos de más de 50000 pesos*

Siguiendo la misma estrategia definimos otro heredero de `CajaAhorros`.

Module	CajaAhorrosInfDep inherits from CajaAhorros

Acá podemos ver que la cosa se empieza a complicar. ¿Qué pasa si hay que combinar los requerimientos de a pares (por ejemplo, no más de 4 extracciones mensuales e informar los depósitos grandes; o extracciones de menos de 10000 e e informar los depósitos grandes)? ¿Y si tenemos que combinar los tres requerimientos? ¿Y si importa el orden en que se ejecuta cada requerimiento? ¿Y si aparece un cuarto requerimiento que debe ser combinado de múltiples formas con los otros tres? ¿Y un quinto? Claramente terminaríamos teniendo una cantidad enorme de módulos. A esto se le suele llamar *explosión de módulos*. Lamentablemente es la segunda forma de diseño más común elegida por los programadores, luego de la que implementaría todos los requerimientos en el mismo módulo mediante sentencias condicionales.

> *La explosión de módulos es un fenómeno muy frecuente en el diseño de software. Se da por la combinación geométrica de requisitos (o* features *en inglés). La forma correcta de evitar la explosión de módulos es aplicando el patrón* Decorator.

Entonces apliquemos el patrón Decorator para implementar todos los requisitos que hemos mencionado *y todas sus combinaciones posibles*. En esta aplicación del patrón el módulo `CajaAhorros` corresponde al Componente. Ahora definimos el siguiente módulo que corresponde al Decorador:

Module	ControlesCA inherits from CajaAhorros

Este módulo será solo una interfaz (no tiene implementación). De él hereda cada uno de los módulos que implementa cada uno de los requerimientos (o sea los DecoradoresConcretos); ninguno de los herederos extiende la interfaz de `CajaAhorros`.

Module	CajaAhorros10000 inherits from ControlesCA

Module	CajaAhorros4 inherits from ControlesCA

MODULE	CajaAhorrosInfDep INHERITS FROM ControlesCA

El constructor de cada uno de estos módulos espera un objeto de tipo `CajaAhorros`.

```
CajaAhorros10000(CajaAhorros c) {miCA = c;}

CajaAhorros4(CajaAhorros c) {miCA = c;}

CajaAhorrosInfDep(CajaAhorros c) {miCA = c;}
```

Los métodos de los `ControlesCA` operan sobre el objeto `miCa` mediante delegación o implementando lo que cada uno tenga que implementar. Muestro solo algunos ejemplos representativos.

```
CajaAhorros10000.extraer(Monto m) {
  if m =< 10000
  then miCa.extraer(m)
  else no hacer nada o devolver error
}

CajaAhorros10000.depositar(Monto m) {miCA.depositar(m);}

CajaAhorros4.extraer(Monto m) {
  if numExt < 4 then               // numExt variable de estado de CajaAhorros4
  then miCa.extraer(m); numExt = numExt + 1;
  else no hacer nada o devolver error
}

Monto CajaAhorros4.saldo() {return miCA.saldo();}

CajaAhorrosInfDep.depositar(Monto m) {
  miCA.depositar(m);
  if m > 50000 then informarDep(m);   // informarDep() es un método privado
}

CajaAhorrosInfDep.extraer(Monto m) {miCA.extraer(m);}
```

En tiempo de ejecución se compone una `CajaAhorros` con todos los controles que sean necesarios para esa instancia.

```
main() {
  CajaAhorros ca1, ca2;
  ControlesCA ctrl1, ctrl2, ctrl3, ctrl4, ctrl5;

  // ca1 necesita todos los controles
  ctrl1 = new CajaAhorros10000(ca1);
  ctrl2 = new CajaAhorros4(ctrl1);
  ctrl3 = new CajaAhorrosInfDep(ctrl2);
```

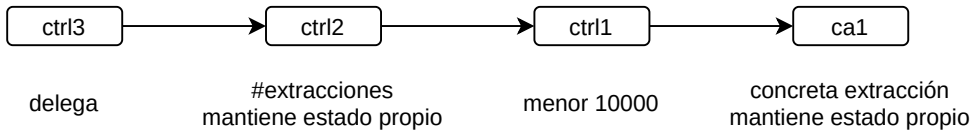

Figura 1: Representación gráfica de la secuencia de invocaciones

```
  // ca2 solo dos de ellos
  ctrl4 = new CajaAhorros4(ca2);
  ctrl5 = new CajaAhorrosInfDep(ctrl4);

  // las cajas de ahorro se manejan desde afuera
  ctrl3.extraer(8000);
  ctrl5.depositar(65000);
}
```

De esta forma, por ejemplo, cuando se ejecuta `ctrl3.extraer(8000)` primero se invoca `CajaAhorrosInfDep.extraer()` que simplemente delega la petición en su `miCA`; pero esa `miCA` es de tipo `CajaAhorros4` por lo que se invoca `CajaAhorros4.extraer()` que controla si hubo o no 4 extracciones; si no las hubo la petición vuelve a delegarse en su `miCA` (que es una instancia distinta de la de `CajaAhorrosInfDep`); pero esa `miCA` es de tipo `CajaAhorros10000` por lo que se invoca `CajaAhorros10000.extraer()` que controla si la extracción es de más de 10000 pesos; como en este caso no lo es, la petición se vuelve a delegar en su `miCA` la cual es de tipo `CajaAhorros` por lo que se concreta la extracción. Este proceso se describe gráficamente en la Figura 1.

Decorator es una alternativa a la herencia. Es decir, en lugar de usar solo herencia, Decorator abre la puerta a una nueva forma de estructurar el código. Mientras la herencia vincula módulos en tiempo de compilación, Decorator vincula objetos en tiempo de ejecución. Representa un mecanismo general de composición de objetos. Permite añadir funcionalidad a objetos específicos y no a todo un tipo. Por este motivo es una idea casi tan poderosa como la de la herencia.

Noten que es muy simple eliminar del sistema uno de los requisitos. Simplemente no creamos objetos del módulo que implementa ese requisito. De la misma forma es muy simple añadir otro requerimiento.

Para pensar, discutir y analizar

1. Analice cómo sería el proceso para modificar en tiempo de ejecución una cadena de composición como la efectuada entre `ctrl1`, `ctrl2`, `ctrl3` y `ca1`. Por ejemplo, si en algún momento de la ejecución tuviéramos que sacar de la cadena a `ctrl2`.
2. La implementación de `CajaAhorros4.extraer()` no es correcta porque no tiene en cuenta que al inicio de cada mes se debe reiniciar el contador de extracciones.
3. Nosotros consideramos que `CajaAhorros` corresponde al Componente. ¿Podría el

Componente ser `CuentaBancaria`? En ese caso, ¿cuál o cuáles serían los ComponentesConcretos? ¿Es mejor o peor que el Componente sea `CuentaBancaria` en lugar de `CajaAhorros`?

4. Podría darse el caso que en una parte del programa debamos construir una cadena de composiciones pero en otra parte la cadena sea distinta. Más aun, podría ocurrir que estas formas de construcción se vayan añadiendo y eliminando con el tiempo, a raíz de la aparición y desaparición de requerimientos. Le sugerimos estudiar los patrones de diseño Builder y Factory Method.
5. Hasta el momento todas las cadenas de composición representan *conjunciones* de funcionalidad (requerimientos). ¿Cómo debería ser el diseño si fuera necesario implementar disyunciones?
6. Completar los diseños con las cláusulas Imports.
7. Use las cajas Pattern para documentar el diseño visto en esta sección.

4. Abstract factory

En este caso volvemos sobre el problema visto en el apunte "Problemas resueltos – Diseño", referido al formulario para ingresar nombre y DNI, que tiene un título y el botón `Aceptar`. El problema que vamos a resolver aquí se vincula con el requerimiento no funcional denominado *internacionalización*. Es decir, la capacidad de un programa de adaptarse al país donde está ejecutando o de donde proviene el usuario; y más en general, la capacidad de configurar esas cuestiones en tiempo de ejecución.

La idea es que además del título el formulario debe imprimir las etiquetas de los campos para ingresar el nombre y el DNI y la etiqueta del botón `Aceptar`. Pensamos, por ejemplo, en un aplicación Web o móvil que el usuario descarga en un cierto país, la aplicación detecta el idioma local y presenta el formulario con las etiqueta en el idioma correspondiente. Es decir, no vale que el programador compile la aplicación para ese idioma sino que la aplicación se adapte dinámicamente al idioma local.

Vamos a generalizar la solución presentada en el sexto problema de diseño (apunte "Problemas resueltos – Diseño")[4]. En esa solución el módulo `FormNombreDNI` recibía en el constructor un objeto de tipo `TituloFormNombreDNI`. Este, a su vez, tenía un heredero por cada idioma en que se podía escribir el título del formulario. Entonces, `FormNombreDNI` le pide a `TituloFormNombreDNI` el título a imprimir el cual dependerá del tipo (idioma) de la instancia que se le haya pasado al crear el formulario.

La limitación de ese diseño es que solo imprime el título del formulario cuando nosotros debemos imprimir además las etiquetas de los campos nombre y DNI y la del botón `Aceptar`. La solución que surge de aplicar el patrón Abstract Factory consiste, entre otras cosas, en pasarle a `FormNombreDNI` (el Cliente) una "fábrica de etiquetas" que imprima etiquetas en un idioma particular. O sea que esta fábrica "fabricará" las etiquetas que necesita el formulario en un idioma particular. Tendremos una fábrica para cada idioma porque ese es el ítem de cambio:

[4]Les sugiero repasar esa solución antes de continuar.

el idioma de cada texto que hay que mostrar en el formulario. La fábrica tendrá un método para fabricar cada etiqueta, como lo indica el siguiente módulo.

MODULE	EtiquetasFormNombreDNI
EXPORTS	título():String
	nombre():String
	dni():String
	botón():String

EtiquetasFormNombreDNI es la interfaz que corresponde a FÁBRICAABSTRACTA. Si bien en este caso todos los métodos devuelven algo de tipo String, en otros casos cada método puede devolver un tipo diferente. Entonces esta aplicación del patrón no tiene PRODUCTOSABSTRACTOS y los PRODUCTOS son siempre de tipo String.

Luego definimos un heredero de EtiquetasFormNombreDNI para cada idioma; los herederos no extienden la interfaz. Estos corresponden a las FÁBRICASCONCRETAS.

MODULE	EtFormNombreDNI_SP INHERITS FROM EtiquetasFormNombreDNI

MODULE	EtsFormNombreDNI_EN INHERITS FROM EtiquetasFormNombreDNI

MODULE	EtFormNombreDNI_IT INHERITS FROM EtiquetasFormNombreDNI

La implementación de los métodos de cada heredero es muy simple (similar a la que tenemos en el sexto problema de diseño). Mostramos solo algunos ejemplos.

```
String EtFormNombreDNI_SP.título() {return 'Ingrese su nombre y DNI';}

String EtFormNombreDNI_EN.título() {return 'Enter your name and ID number';}

String EtFormNombreDNI_SP.nombre() {return 'Nombre y apellido';}

String EtFormNombreDNI_SP.nombre() {return 'Name and surname';}

String EtFormNombreDNI_IT.nombre() {return 'Nome e cognome';}
```

El formulario invoca los métodos de la fábrica recibida en el constructor para imprimir las etiquetas correctas.

```
FormNombreDNI(EtiquetasFormNombreDNI et) {misEts = et;}

dibujar() {
  print(misEts.titulo());
  ......
```

```
  // creamos un botón y le pasamos la etiqueta correcta
  Button aceptar = new KDEButton(misEts.aceptar());
  ......
  // creamos un campo de texto y le pasamos la etiqueta correcta
  InputField nombre = new KDEInputField(misEts.nombre());
  ......
}
```

En `main()` configuramos adecuadamente el formulario en tiempo de ejecución creando la fábrica correspondiente al idioma local.

```
main() {
  EtiquetasFormNombreDNI et;
  FormNombreDNI form;
  case LANG is
    'sp': et = new EtFormNombreDNI_SP;
    'en': et = new EtFormNombreDNI_EN;
    'it': et = new EtFormNombreDNI_IT;
    // más idiomas
  endcase
  form = new FormNombreDNI(et);
  .....
}
```

Con este diseño es muy simple agregar soporte para nuevos idiomas: basta con implementar un heredero de `EtiquetasFormNombreDNI` y agregar una línea a la sentencia `case` de `main()`. Además es muy simple generar versiones de la aplicación con soporte para uno, dos o muchos idiomas.

Para pensar, discutir y analizar

1. ¿Se usa la composición de objetos en este diseño? ¿Dónde?
2. Suponga que la aplicación tiene varios formularios y como es de esperarse varias de las etiquetas son compartidas por esos formularios pero otras no. Por ejemplo, varios formularios tendrán el botón `Aceptar` pero no todos pedirán el DNI. ¿Cómo sería el diseño en este caso?
3. ¿Se puede evitar la sentencia `case` en el `main()`? ¿Por qué sería bueno poder evitarla? ¿Es tan importante evitarla?
4. El diseño analizado, ¿respeta el principio de diseño conocido como Diseños Abiertos y Cerrados?
5. Completar los diseños con las cláusulas Imports.
6. Use las cajas Pattern para documentar el diseño visto en esta sección.

5. Bridge

Una empresa del área de servicios financieros[5] tiene clientes que depositan dinero en las cuentas bancarias de la empresa. Esta, a su vez, actualiza el saldo de las cuentas de sus clientes con esos depósitos (o sea que los clientes de la empresa tienen una cuenta en la empresa donde se guarda su dinero). Además la empresa usa el dinero de sus cuentas bancarias para pagar sueldos, insumos, impuestos, etc. Por lo tanto la empresa necesita ver periódicamente los movimientos ocurridos en cada una de sus cuentas bancarias (estas empresas suelen tener varias cuentas en distintos bancos). Con este fin los bancos ofrecen APIs[6] que sus clientes (o sea nuestra empresa de servicios financieros) pueden usar para interactuar de forma remota y automatizada con el banco. Entonces nuestra empresa debe programar una aplicación que se comunique con cada banco para solicitar los movimientos (depósitos, retiros, transferencias, etc.) que ocurren u ocurrieron en sus cuentas.

Observen que hay, implícitos, dos ítems de cambio:

1. La empresa puede abrir cuentas con otros bancos o cerrar cuentas con ciertos bancos.
2. Cada banco ofrece *su* API. Es decir no existe un estándar bancario para las APIs que los bancos ofrecen a sus clientes. Cada banco define su API como mejor le parece.

Les muestro la API del banco *X* y la del banco *Y*. Tengan en cuenta que está todo muy simplificado. El banco *X* ofrece dos subrutinas: `login()` para autenticar a sus usuarios; y `getTransacDía()` para que sus usuarios consulten los movimientos de sus cuentas. Fíjense que el *X* define los tipos `Cuenta` y `Transac`, y la forma en que se comunican los errores.

MODULE	`APIBancoX`
EXPORTS	`login(i String, i String)` `getTransacDía(i Cuenta, o List(Transac)):Int`
COMMENTS	`getTransacDía()` retorna en el segundo parámetro las transacciones (débitos y créditos) efectuadas durante el día en una `Cuenta`. El entero de retorno codifica los posibles errores.

MODULE	`APIBancoY`

EXPORTS	`login(i String, i String)`
	`setCuenta(i CtaBanco)`
	`getDébitos():List(Movi)`
	`getCréditos():List(Movi)`

COMMENTS	`getDébitos()` devuelve los débitos efectuados durante el día en la cuenta previamente establecida con `setCuenta()`. Los posibles errores se codifican en el primer elemento de la lista.

El banco *Y* ofrece cuatro subrutinas: una para autenticar a sus usuarios; otra para establecer la cuenta de la cual quieren consultar información; y dos subrutinas para consultar débitos y

[5] Se las conoce como '*fintech*', por ejemplo Ualá, Let'sBit, etc.

[6] *Application Programming Interface*. Simplificando la cosa una API es básicamente un conjunto de subrutinas (o sea una interfaz). https://en.wikipedia.org/wiki/API

créditos. El banco define el tipo `Movi` y una forma de comunicar errores distinta de la usada por el banco *X*.

Los tipos `Movi` y `Transac` permiten obtener detalles de cada movimiento. Por ejemplo, si el depósito fue en efectivo, en qué sucursal se hizo, el originante de una transferencia, el monto, la moneda, el número de operación interna del banco, etc. Uno de los problemas de estas APIs es que esos datos se codifican de maneras totalmente diferentes y arbitrarias.

Como ven las dos API son muy distintas. Esto no está exagerado, de hecho está simplificado.

Entonces el código de la empresa para, por ejemplo, actualizar el saldo de las cuentas de sus clientes tendría que solicitar los movimientos de cada banco usando cada API por lo que ese código sería muy dependiente de esa API. Básicamente sería un código condicional que depende de cada banco. Como siempre esa solución funciona y es muy intuitiva pero errada desde el punto de vista del diseño. La incorporación de un nuevo banco o la eliminación de un banco existente implicaría modificar el código que actualiza los balances. También es muy complicado si un banco cambia su API pues tenemos que modificar el código que actualiza los saldos debido a un cambio no-funcional.

Lo mismo vale para el código de la empresa que, por ejemplo, mantiene la contabilidad o realiza los pagos. Si un pago se realiza con un banco se debe usar esa API, si se realiza con otro se debe usar otra API. Nuevamente código condicional que se ve afectado por cada alta o baja de un banco.

Este diseño tiene altos costos de mantenimiento. El motivo es siempre el mismo: se está implementando más de un ítem de cambio en cada módulo. Por ejemplo, el módulo que implementa la actualización de los saldos de las cuentas de los clientes tiene código que actualiza los saldos tomando los depósitos (créditos de cierta clase) a través de la API de *cada* banco. Pero la API de cada banco es un ítem de cambio. Entonces ese módulo oculta varios.

Aplicando el patrón de diseño Bridge todos estos problemas se solucionan. Lo primero que hacemos es definir una API abstracta o de un banco ideal. Es la API que a nosotros más nos conviene. Tiene que tener métodos para acceder a toda la información que necesitamos de la forma que mejor nos convenga. Esta API puede ser totalmente distinta de las APIs de los bancos con los que opera la empresa. El módulo que provee esta API (`Banco`) corresponde al Implementador de la descripción de Bridge.

Module	`Banco`

exports	`login(i String, i String)`
	`débitos(o List(Débito)): Error`
	`créditos(o List(Crédito)): Error`

comments	Cada método devuelve las operaciones de ese tipo en el día de la fecha. El tipo `Error` se usa para comunicar los errores.

Ahora definimos los módulos que corresponden a los ImplementadorConcreto del patrón.

Module	`BancoX` inherits from `Banco`

MODULE	BancoY INHERITS FROM Banco

Como ven estos herederos no extienden la interfaz del supertipo. Más aun, siguiendo la descripción del patrón, los métodos de los herederos se implementan en términos de las APIs de cada banco. Mostramos solo un ejemplo basado en `APIBancoX`.

```
Error BancoX.débitos(List(Débito) debs) {
  Cuenta cx = new Cuenta('12345/8');
  Transac t;     // tipo del banco X
  Débito d;
  Int errx = APIBancoX.getTrasacDía(cx,depsx);
  if errx = 0    // no hay error, definido por X !!
  then
    for t in depsx do {
      d = convertir_transac_deb(t);
      if d != null then debs.add(d);
    }
  else if errx = 1
       then return new Error('error conexión');
       else ...
  return new Error('ok');
}
```

Las funciones `convertir_cta()` y `convertir_transac_deb()` son privadas de `BancoX`. Observen que la implementación de `BancoX.depósitos()` usa las rutinas y tipos provistos por `APIBancoX`. Es decir este código depende fuertemente de la API del banco *X*. Pero toda esa dependencia está concentrada en los métodos de `BancoX`. Si el banco *X* decide cambiar su interfaz tendremos que modificar solo métodos de `BancoX`. `BancoX` oculta un único ítem de cambio.

Finalmente, veamos un par de módulos que corresponden a ABSTRACCIÓN y ABSTRACCIÓNREFINADA. En este caso ABSTRACCIÓN es el módulo `OperacionesBancos` que solo se configura con un `Banco`.

MODULE	OperacionesBancos
EXPORTS	setBanco(i Banco)

Luego definimos dos herederos completamente distintos. Uno para actualizar los depósitos.

MODULE	ActualizarSaldos INHERITS FROM OperacionesBancos
EXPORTS	actualizar(i CuentaCliente)
	actualizar(i List(CuentaCliente))

Otro para sumar los retiros diarios.

MODULE	`SumarRetiros` INHERITS FROM `OperacionesBancos`
EXPORTS	`sumarRetiroEft():Monto`
	`sumarDébitoAuto():Monto`
	`sumarTransfOut():Monto`

Entonces la implementación de estos métodos usa la interfaz de `Banco` (a través del objeto que se recibe en `setBanco()`) de forma tal que están ajenos a la complejidad y diversidad de las APIs. A su vez el resto del sistema puede usar las `OperacionesBancos` que representan conceptos del negocio de la empresa sin tener que preocuparse de cómo se obtienen esos datos desde los bancos.

En la Figura 2 pueden ver la representación gráfica de las relaciones entre los módulos de la solución basada en el patrón BRIDGE. Como pueden ver responde a la estructura mostrada en la descripción del patrón. En esta figura, además, incluimos los módulos de bajo nivel `APIBancoX` y `APIBancoY` que normalmente no forman parte del patrón.

Es interesante observar que BRIDGE usualmente organiza los módulos en tres niveles de abstracción. En el nivel inferior tenemos operaciones de bajo nivel que nos permiten comunicarnos con los bancos y obtener información de ellos; en el nivel intermedio tenemos una abstracción de la comunicación con los bancos unificada para la empresa; y en el nivel superior tenemos las operaciones del negocio bancario que es lo que más nos interesa (usualmente llamada *lógica de negocio*). Como lo muestra la figura, los servicios del nivel superior se implementan solo en términos de los servicios del nivel medio, y estos solo en términos de los servicios de bajo nivel. Idealmente ningún servicio de alto nivel debería invocar servicios de bajo nivel; y menos aun los servicios en niveles inferiores invocar servicios en los superiores. Recuerden que si un conjunto de módulos tiene esas propiedades Parnas lo llama *máquinas abstracta o virtual* y que él propone organizar los sistemas como pilas de máquinas abstractas. Lograr este tipo de estructuras es un síntoma de un diseño de buena calidad que tendrá bajo costo de mantenimiento.

Para pensar, discutir y analizar

1. ¿Cómo debería ser el diseño/implementación de `BancoX`.depósitos() si nuestra empresa tuviera varias cuentas en el banco *X* y no solo la 12345/8? ¿Dónde deberían estar esos números de cuenta? ¿Qué módulo debería indicar el número de cuenta? ¿Qué ocurre si se pasa un número de cuenta inexistente?
2. Muestre la implementación del método `BancoY`.depósitos().
3. Normalmente las APIs de los bancos ofrecen rutinas para efectuar transacciones tales como pagos, transferencias bancarias, etc. Imagine que las APIs de los bancos *X* e *Y* ofrecen esas rutinas. Diseñe y muestre una pseudo-implementación de un módulo que realice transferencias bancarias a nombre de la empresa.
4. Así mismo las APIs de los bancos permiten recuperar los movimientos entre dos fechas determinadas. Extienda el diseño en ese sentido.
5. Muestre una pseudo-implementación de un método que sume los retiros en efectivo

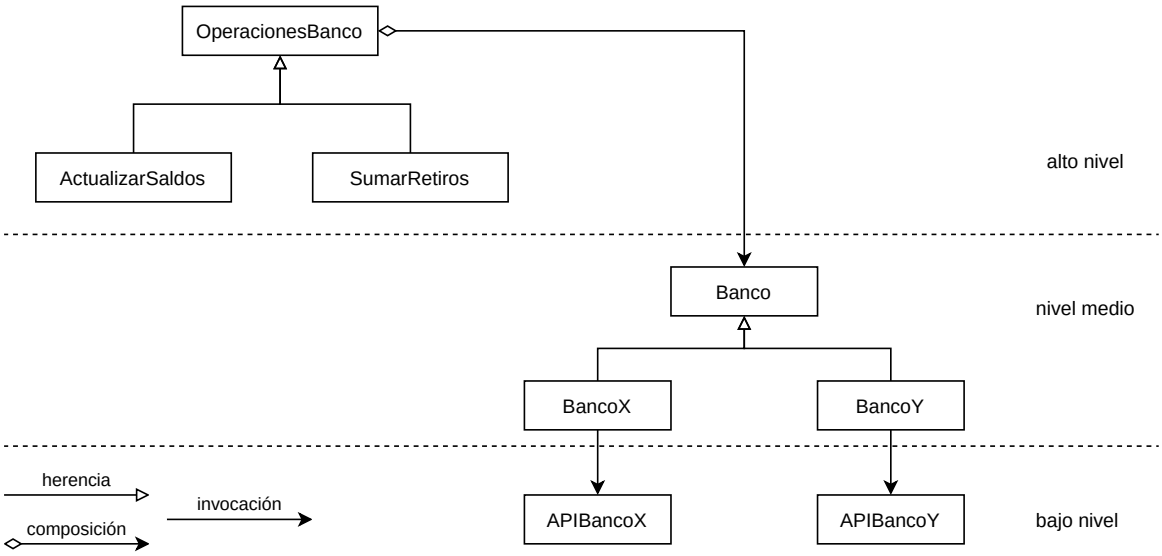

Figura 2: Representación gráfica de las relaciones entre módulos

de todas las cuentas de todos los bancos en los que la empresa tiene cuentas. ¿En qué módulo ubicaría este método?

6. Completar los diseños con las cláusulas IMPORTS.
7. Use las cajas PATTERN para documentar el diseño visto en esta sección.

6. Command

Vamos a aplicar el patrón de diseño COMMAND a la comunicación entre la máquina que recibe monedas y billetes (módulo `MáquinaMB`) en la estación de peaje y el módulo que implementa la lógica de pago en efectivo (`PagoEfectivo`). Reitero el módulo `MáquinaMB` el cual ocultaba la comunicación con el hardware de la máquina.

```
MODULE    MáquinaMB
IMPORTS   Valor
EXPORTS   nuevaMB(i *F)
          denominacion():Valor
          cilindroVacio(i *F(i Valor))
          capacidadCilindro(i Valor):Int
          bandejaRetirada(i *F)
          bandejaInsertada(i *F)
          entregarMoneda(i Valor)
          inicializar()
PRIVATE   esperarInterrupciones()
```

Los parámetros de tipo *F son punteros a función; si la función a la cual apunta el puntero debe tener parámetros estos se indican entre paréntesis luego del símbolo *F como en la signatura de cualquier función.

Más abajo tienen el módulo `PagoEfectivo`. Algunas de las subrutinas de `PagoEfectivo` son las funciones que se pasan como parámetro a las subrutinas que esperan punteros a función de `MaquinaMB`.

MODULE	`PagoEfectivo`
IMPORTS	`Valor, MaquinaMB, Ticket, TablaPrecios, Monto`
EXPORTS	`hayNuevaMB()`
	`noHayCambioDe(i Valor)`
	`noHayCambio()`
	`hayCambio()`
	`pagoEfectivo():Monto`
	`ticket():Ticket`
	`inicializar()`

Concretamente:

- `hayNuevaMB()` se pasa a `nuevaMB()`
- `noHayCambioDe()` se pasa a `cilindroVacio()`
- `noHayCambio()` se pasa a `bandejaRetirada()`
- `hayCambio()` se pasa a `bandejaInsertada()`

Estos parámetros se pasan cuando se configura el sistema o cuando se invoca `inicializar()` de `PagoEfectivo`.

Recuerden que la cuestión de usar punteros a función es para implementar *callbacks*. A su vez los *callbacks* evitan que la implementación de `MáquinaMB` haga invocaciones *explícitas* a los métodos de `PagoEfectivo`. Queremos evitar las invocaciones explícitas desde `MáquinaMB` a `PagoEfectivo` porque el subsistema de medios de pago está organizado como una pila de máquinas abstractas en la cual `PagoEfectivo` pertenece a una máquina más abstracta que `MáquinaMB`. Por lo tanto `MáquinaMB` no puede realizar invocaciones explícitas a los servicios de `PagoEfectivo`. Efectivamente, los punteros a función permiten que `MáquinaMB` invoque los servicios de `PagoEfectivo` pero de forma *implícita*. O sea hay invocaciones a `PagoEfectivo` pero en el código de `MáquinaMB` no hay ninguna referencia a los métodos de `PagoEfectivo`. Por lo tanto podemos usar `MáquinaMB` 'debajo' de una máquina abstracta con una interfaz distinta de `PagoEfectivo`.

Este diseño es correcto pero los *callbacks* (sobre todo cuando son implementados con punteros a función) son un mecanismo de muy bajo nivel que puede ser reemplazado por la aplicación del patrón COMMAND, que además es más versátil. La idea fundamental es que en lugar de pasar una función vamos a pasar un objeto que encapsula la función que debe ser invocada. En este caso el INVOCADOR mencionado en la descripción del patrón es `MáquinaMB`; cada ORDENCONCRETA encapsulará una de las funciones de `PagoEfectivo` que hay que invocar; y esté último es el RECEPTOR.

Comenzamos definiendo el módulo que corresponde a Orden.

```
Module   Manejador
exports  manejar()
         manejar(i Valor)
```

Luego tenemos las OrdenConcreta que mantienen la interfaz de su padre.

```
Module   NuevaMoneda inherits from Manejador
```

```
Module   CilindroVacío inherits from Manejador
```

```
Module   SinBandeja inherits from Manejador
```

```
Module   ConBandeja inherits from Manejador
```

Veamos un par de implementaciones de los herederos. Como pueden ver, el constructor de cada OrdenConcreta se usa para establecer el receptor que en este caso es un objeto tipo `PagoEfectivo`.

```
NuevaMoneda(PagoEfectivo pe) {receptor = pe;}

NuevaMoneda.manejar() {receptor.hayNuevaMB();}

CilindroVacío(PagoEfectivo pe) {receptor = pe;}

CilindroVacío.manejar(Valor v) {receptor.noHayCambioDe(v);}
```

Como pueden ver algunos herederos implementan `manejar()` y otros `manejar(v)`. Esto depende de si el manejador espera o no parámetros.

Nuestro Invocador necesita varias OrdenConcreta y no solo una como sugiere la descripción del patrón. Sea una o varias se puede usar el constructor del Invocador para establecer la composición o se puede usar un método tipo *setter*. Nosotros vamos a usar el constructor.

```
main() {
  PagoEfectivo pe;
  MáquinaMB maq;
  Manejador nm, cv, sb, cb;
  nm = new NuevaMoneda(pe);
  cv = new CilindroVacío(pe);
  sb = new SinBandeja(pe);
  cb = neq ConBandeja(pe);
```

```
  maq = new MáquinaMB(nm, cv, sb, cb);
  ......
}
```

Finalmente veamos cómo `MáquinaMB` usa los manejadores. Recuerden que este módulo está en contacto con el hardware de la máquina que recibe monedas y billetes. Por ejemplo, cuando el conductor inserta una moneda la máquina genera una interrupción la cual es capturada por la rutina privada `esperarInterrupciones()` de `MáquinaMB`. A su vez, `esperarInterrupciones()` invoca al manejador correspondiente. Suponemos que la variable `interrupt` vale 0 cuando no hay interrupción y que cada interrupción está asociada a un natural (1, 2, etc.); la variable `v` guarda el valor del cilindro de cambio que se vació. Entonces la implementación es más o menos la siguiente.

```
void esperarInterrupciones() {
  while (true) {
    case interrupt is {
      1: nm.manejar();       // código 1 --> moneda o billete insertado
         interrupt = 0;      // código 0 --> no hay interrupción
      2: cv.manejar(v);      // código 2 --> se vació un cilindro
         interrupt = 0;      // código 0 --> no hay interrupción
      .....
    }
  }
}
```

Para pensar, discutir y analizar

1. ¿De qué otra(s) forma(s) se podrían organizar/diseñar la Orden y las OrdenConcreta que esperan parámetros y las que no? Analice ventajas y desventajas respecto a la solución mostrada en el ejemplo.
2. Muestre una pseudo-implementación de `main()` cuando `MáquinaMB` usa uno o varios métodos *setter* para recibir los manejadores. ¿Conviene que sea un método que recibe todos los manejadores o uno por manejador?
3. Revea el segundo problema resuelto de diseño (el del formulario para ingresar DNI y nombre). Use Command para desacoplar `FormNombreDNI` de `ControlarNombreDNI`. ¿Por qué valdría la pena desacoplarlos?
4. Completar los diseños con las cláusulas Imports.
5. Use las cajas Pattern para documentar el diseño visto en esta sección.

7. Visitor

En la Figura 3 vemos las relaciones típicas entre clientes, pedidos y productos. Supongamos que el subsistema que procesa los pedidos tiene módulos `Cliente`, `Pedido` y `Producto` y conjuntos de ellos (`Clientes`, `Pedidos` y `Productos`). Un objeto `Cliente` se compone de un objeto

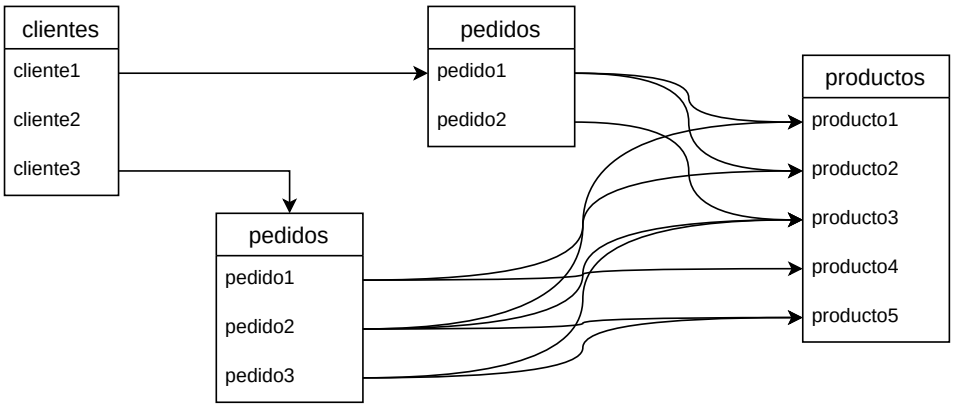

Figura 3: Los clientes hacen pedidos los cuales contienen productos

`Pedidos` el cual se compone de objetos `Pedido`, cada uno de los cuales se compone de un objeto `Productos`. Finalmente el sistema declara un instancia de `Clientes` para llevar los clientes de la empresa. Supongamos que todo esto está bien diseñado e implementado; hace tiempo que no se presentan errores; la gerencia de la empresa está conforme con este subsistema.

Ahora supongamos que llegan una serie de requerimientos nuevos:

1. *Cantidad de pedidos promedio por cliente*
2. *Producto más solicitado por los clientes*
3. *Cliente con la mayor facturación (suma de los precios de todos los productos de todos los pedidos).*

¿Cómo diseñamos/implementamos estos nuevos requerimientos? ¿Cuál es nuestra estrategia de cambio ante esta situación? Tengamos en cuenta que ahora llegaron estos tres nuevos requerimientos pero claramente podrían aparecer muchos otros del estilo. Analicemos algunas alternativas:

1. Podemos extender las interfaces de los módulos `Cliente`, `Pedido`, `Producto`, etc. con métodos que permitan implementar los nuevos requerimientos. Funciona y es intuitivo pero no va. ¿Por qué?
2. Podemos definir herederos de los módulos `Cliente`, `Pedido`, `Producto`, etc. de forma tal que implementen los nuevos requerimientos. Pero no va. ¿Por qué? ¿Cambia algo sustancial respecto a la alternativa anterior?
3. Claramente, aplicando Parnas y el principio de diseño Diseños abiertos y cerrados, deberíamos evitar por todos los medios modificar los módulos que ya tenemos, tratando de añadir los nuevos requerimientos en nuevos módulos.

El patrón de diseño Visitor es la solución correcta alineada con la última alternativa. En este caso requiere una pequeña modificación en las interfaces de los módulos que ya tenemos pero que una vez hecha permitirá añadir todos los requerimientos que queramos que sean similares a los tres que estamos considerando. La modificación que tenemos que hacer en estos módulos consiste en agregar a sus interfaces un método, que vamos a llamar `ejecutar()`, que espera un parámetro de tipo `OpExt`. De esta forma, `ejecutar()` corresponde al método Aceptar de la

descripción del patrón y `OpExt` corresponde a VISITANTE. En consecuencia `OpExt` se define de la siguiente forma.

MODULE	`OpExt`
EXPORTS	`analizarClientes(i Clientes)`
	`analizarCliente(i Cliente)`
	`analizarPedidos(i Pedidos)`
	`analizarPedido(i Pedido)`
	`analizarProductos(i Productos)`
	`analizarProducto(i Producto)`

Así, los métodos de la interfaz de `OpExt` corresponden a los métodos tipo VISITELEMENTOCONCRETO de la descripción del patrón. Es decir, hay un método por cada tipo de objeto que es necesario visitar.

Por lo tanto, la implementación de `ejecutar()` es la siguiente (mostramos solo algunos ejemplos).

```
Clientes.ejecutar(OpExt o) {o.analizarClientes(this);}

Cliente.ejecutar(OpExt o) {o.analizarCliente(this);}
```

Como podemos ver la modificación que teníamos que hacer era realmente muy simple y de riesgo virtualmente nulo en relación a introducir errores en código que sabemos que funciona bien.

Siguiendo la descripción del patrón, cada uno de los nuevos requerimientos se implementa en un VISITANTECONCRETO diferente. Por lo tanto definimos los siguientes módulos los que, siguiendo la descripción del patrón, no extienden la interfaz de `OpExt`.

MODULE	`PedidosPromedio` INHERITS FROM `OpExt`

MODULE	`ProductoTop` INHERITS FROM `OpExt`

MODULE	`ClienteTopFact` INHERITS FROM `OpExt`

VISITOR es un patrón de diseño muy general y potente pues permite agregar funcionalidad para analizar, explorar o consular estructuras de objetos muy complejas sin tener que modificar los módulos que implementan esas estructuras.

De esta forma, la implementación de `PedidosPromedio` es más o menos así[7].

[7] Asumimos que `misClientes()` es un método en la interfaz de `Clientes` que devuelve la lista de clientes. Similarmente para `misPedidos()`.

```
PedidosPromedio.analizarClientes(Clientes cs) {
  pedidos = 0;             // variable de estado
  for c in cs.misClientes() do c.ejecutar(this);
  promedio = pedidos / length(cs.misClientes());
  print(promedio);
}

PedidosPromedio.analizarCliente(Cliente c) {
  pedidos = pedidos + length(c.misPedidos());
}

PedidosPromedio.analizarPedidos(Pedidos ps) {;}

PedidosPromedio.analizarPedido(Pedido p) {;}
```

Entonces, si queremos ejecutar una de las operaciones externas lo hacemos de la siguiente forma.

```
main() {
  Clientes clis;
  OpExt pp;
  ......                    // clis se llena de clientes con sus pedidos
  pp = new PedidosPromedio;
  clis.ejecutar(pp);        // en pantalla debería verse el promedio
  ......
}
```

Para pensar, discutir y analizar

1. ¿Por qué cada uno de los nuevos requerimientos se implementa en un VisitanteConcreto diferente?
2. Dar la pseudo-implementación de los métodos `ejecutar()` que faltan.
3. ¿Cómo es la implementación de `PedidosPromedio.analizarProductos()` y `PedidosPromedio.analizarProducto()`?
4. Dar la pseudo-implementación de `ProductoTop` y `ClienteTopFact`. Suponga que en `Productos` tiene la cantidad de cada `Producto` y que cada uno de estos tiene un precio.
5. Los VisitanteConcreto que hemos definido deben devolver un valor. ¿Es correcto hacer un `print` como hicimos en `PedidosPromedio`? ¿Es una solución general? ¿Podrían los métodos del tipo VisitElementoConcreto ser funciones?
6. ¿Es correcta la forma en que *iteramos* sobre los clientes de `c` en `PedidosPromedio.analizarClientes()`? Justifique.
7. Suponga que quiere generar una versión del sistema solo con `ProductoTop` y `ClienteTopFact`. Analice cómo construir el ejecutable y compare la situación con

la que se hubiera dado de haber aumentado las interfaces de los módulos existentes con métodos que implementaran los nuevos requerimientos.

8. Describa la estrategia de cambio para incorporar nuevos requerimientos como los analizados en esta sección asumiendo el diseño basado en el patrón Visitor. Consultar la Sección 10.3 del apunte "Diseño de Software".
9. Completar los diseños con las cláusulas Imports.
10. Use las cajas Pattern para documentar el diseño visto en esta sección.