Problemas Resueltos – Diseño

Maximiliano Cristiá

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

Los problemas que presentamos en este apunte son simples problemas de diseño orientados a mostrar cuestiones conceptuales de la teoría de diseño de software vista en clase.

1. Primer problema

En una caja de ahorros se pueden hacer depósitos, extracciones y consultar el saldo. Supongamos que algún ingeniero de software definió el siguiente módulo para la caja de ahorros. El módulo `Monto` se asume definido en alguna parte.

MODULE	`CajaAhorros`
IMPORTS	`Monto`
EXPORTS	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`

Ahora supongamos que nos dan el siguiente requerimiento:

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

La pregunta es, ¿cómo hacemos para implementar este requerimiento dado el módulo que ya está definido? Analicemos dos soluciones.

La primera solución consiste en agregar una sentencia condicional en la implementación de `extraer()`.

```
extraer(Monto m) {
  if m =< 10000
  then realizar la extracción
  else no hacer nada o devolver error
}
```

Esta solución es simple y el programa funciona pero está mal desde el punto de vista del DBOI. ¿Por qué está mal? Porque la nueva implementación oculta dos ítems de cambio en el módulo `CajaAhorros` en lugar de uno. El módulo definido inicialmente oculta las estructuras de datos y algoritmos relacionados con el manejo básico de una caja de ahorros (un ítem de cambio). Al agregar el control del monto de extracción estamos incluyendo un segundo ítem de cambio porque este requisito puede ser modificado en el futuro. Además:

1. Deberíamos volver a testear (verificar) *todas* las subrutinas del módulo a pesar de que solo modificamos una porque, como comparten la implementación, la modificación podría afectar a las otras subrutinas.
2. Si aparece otro requerimiento (por ejemplo: la primera extracción del mes puede ser de más de 10000 pesos) la implementación se haría cada vez más compleja (básicamente porque implementaría varios ítems de cambio).
3. Si desaparece el requerimiento (es decir se vuelven a permitir extracciones sin límite) deberíamos volver a modificar el código y volver a testearlo (verificarlo).

La segunda solución consiste (simplemente) en aplicar la metodología de Parnas y el DBOI. Como el nuevo requerimiento implica un nuevo ítem de cambio deberíamos definir un nuevo módulo que lo oculte y lo implemente. Al mismo tiempo queremos reutilizar la implementación de `CajaAhorros` que ya tenemos. La solución es la siguiente. Definimos el siguiente módulo que tiene la misma interfaz que `CajaAhorros`.

MODULE	`ControlExtraccion`
IMPORTS	`CajaAhorros, Monto`
EXPORTS	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`

Ahora implementamos las subrutinas invocando a las subrutinas de `CajaAhorros`, agregando el código necesario para implementar el nuevo requerimiento. Esta forma de implementación se denomina *delegación*, porque las subrutinas de `ControlExtraccion` *delegan* su implementación en las de `CajaAhorros`.

```
depositar(Monto m) {CajaAhorros.depositar(m);}

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

Monto saldo() {return CajaAhorros.saldo();}
```

De esta forma evitamos los problemas mencionados más arriba: no tenemos que volver a testear (verificar) `CajaAhorros` ni hacemos más compleja su implementación. Al mismo tiempo la cantidad de líneas de código que escribimos en ambas soluciones es básicamente la misma. Ambos programas tienen casi la misma longitud. Además, cuando el requerimiento desaparece lo único que tenemos que hacer es no usar más el módulo `ControlExtraccion`.

¿Por qué `ControlExtraccion` tiene la misma interfaz que `CajaAhorros`? Como estamos implementando una extensión sobre código existente hay código en otras partes del programa que usa la interfaz de `CajaAhorros`. Este código se llama *cliente* de `CajaAhorros` porque usa sus servicios. Ahora queremos que ese código cliente use la interfaz de `ControlExtraccion` (es decir, deje de ser cliente de `CajaAhorros` y pase a ser cliente de `ControlExtraccion`). Si

`ControlExtraccion` tuviera una interfaz diferente a `CajaAhorros` deberíamos modificar todo el cliente para ajustarlo a la nueva interfaz.

Finalmente, observar que el diseño cumple con la definición de diseño que vimos en el apunte: dividimos el sistema en partes (`CajaAhorros` y `ControlExtraccion`); asignamos una funcionalidad a cada parte (aunque no lo hicimos explícitamente sabemos qué debe hacer cada subrutina); y establecimos las relaciones entre las partes (`ControlExtraccion` invoca las subrutinas de `CajaAhorros` pero no al revés).

2. Segundo problema

Debemos implementar un programa que le presente al usuario un formulario donde este debe ingresar su nombre y su DNI, y cuando pulsa el botón Aceptar debemos consultar en una base de datos si el DNI existe y si corresponde con el nombre ingresado.

Como estamos estudiando diseño de software en realidad no vamos a implementar nada sino que solo vamos a definir el diseño. Como en el problema anterior vamos a considerar dos soluciones.

La primera solución consiste en definir el siguiente módulo.

MODULE	`FormNombreDNI`
EXPORTS	`dibujar()`
	`leerEntrada()`

Donde `dibujar()` presenta el formulario y `leerEntrada()` espera la entrada del usuario y consulta en la base de datos si el DNI existe y si corresponde con el nombre ingresado.

Nuevamente esta solución es simple y el programa funciona pero está mal desde el punto de vista del DBOI. La razón es la misma que antes: el módulo `FormNombreDNI` oculta más de un ítem de cambio. En este caso oculta tres:

1. La interacción con el usuario. Es decir, qué tipo de formulario se le presenta al usuario. ¿Es un formulario gráfico o en modo texto? ¿En qué posición está cada campo de texto y el botón? ¿Se usa alguna biblioteca de componentes de interfaz gráfica? ¿Cuál?
2. La regla que pide controlar si el DNI existe y corresponde al nombre ingresado
3. El acceso a la base de datos. ¿Es una base de datos relacional? ¿Es un archivo de texto? ¿Está en la computadora local?

Entonces como son tres ítems de cambio debería haber tres módulos y no solo uno. De todas formas acá vamos a definir dos dejando el tercero como ejercicio. El primer módulo será `FormNombreDNI` pero ahora ocultará e implementará solo el primer ítem de cambio. O sea que cuando se pulse el botón Aceptar invocará las subrutinas del segundo módulo. El segundo módulo es el siguiente.

MODULE	`ControlarNombreDNI`
EXPORTS	`existe(i DNI)`
	`corresponde(i DNI, i String)`

La interfaz podría ser solo la subrutina `corresponde()` que controlaría ambas cosas. Poner una o dos subrutinas depende de cuestiones tales como eficiencia y reuso (por ejemplo si en alguna otra parte del sistema se quiere controlar solo si un DNI existe).

El módulo `ControlarNombreDNI` no debería acceder de forma directa a la base de datos. Por ejemplo si la base de datos es relacional la implementación de las subrutinas podría realizarse con sentencias SQL. Sin embargo esta implementación ocultaría los dos últimos ítems de cambio mencionados más arriba. Es decir que, si en el futuro la base de datos se cambia por un archivo de texto, habría que modificar el módulo `ControlarNombreDNI` sin que haya cambiado la regla (llamada *regla de negocio*) que él implementa (o sea, el DNI existe y el nombre corresponde).

Dejamos como ejercicio definir el tercer módulo y completar las interfaces con las cláusulas IMPORTS correspondientes.

3. Tercer problema

Los bancos suelen ofrecer (al menos) dos tipos de cuentas bancarias: cajas de ahorro y cuentas corrientes. En ambas se pueden hacer depósitos y extracciones y se puede consultar el saldo. Sin embargo, en las cuentas corrientes es posible realizar extracciones que dejen la cuenta con saldo negativo.

Supongamos que contamos con el módulo `CajaAhorros` definido en la Sección 1. ¿Cómo extendemos el diseño para proveer también cuentas corrientes? Vamos a analizar dos soluciones en el contexto del Diseño Orientado a Objetos (DOO).

La primera solución consiste en renombrar el módulo `CajaAhorros` a `CuentaBancaria` agregándole la posibilidad de establecer el tipo de cuenta. El tipo TipoCta puede ser un tipo enumerado cuyos elementos pueden ser ca y cc.

MODULE	`CuentaBancaria`
IMPORTS	`Monto`
EXPORTS	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`
	`tipoCta(i TipoCta)`

De esta forma cuando se crea una instancia del tipo `CuentaBancaria` se establece su tipo:

```
CuentaBancaria c;
c = new CuentaBancaria;
c.tipoCta(cc);
```

Luego, la implementación de `extraer()` realiza el control sobre el monto a extraer en relación al saldo o no dependiendo del tipo de la cuenta:

```
extraer(Monto m) {
  if tipo = cc || m =< saldo
  then saldo = saldo - m;              // el saldo puede ser negativo
  else print('Saldo insuficiente');
}
```

Claramente esta solución es simple y funciona pero está mal respecto a la metodología de diseño que hemos estado estudiando. El problema es que estamos incluyendo dos ítems de cambio en el mismo módulo. Por un lado, incluimos las estructuras de datos y algoritmos para gestionar cajas de ahorro y por el otro, lo mismo pero para las cuentas corrientes. Aun este pequeño ejemplo muestra la necesidad de incluir una sentencia condicional para poder distinguir los dos comportamientos.

Es necesario que entiendan que cada ítem de cambio implica más líneas de código. Si continuamos con este diseño el módulo terminará siendo muy complejo de entender y mantener. Además, al ser ítems de cambio *uno puede cambiar sin que cambie el otro. En ese caso deberíamos modificar una parte del código sin modificar la otra. Esa estrategia es muy propensa a errores.*

La segunda solución consiste en aplicar la metodología de diseño que culmina con el DOO. En este sentido, comenzamos por ocultar cada ítem de cambio en un módulo diferente aunque ambos tendrán la misma interfaz (porque, después de todo, se pueden hacer las mismas operaciones).

MODULE	`CajaAhorros`
IMPORTS	`Monto`
EXPORTS	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`

MODULE	`CuentaCorriente`
IMPORTS	`Monto`
EXPORTS	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`

De esta forma la implementación de `extraer()` puede ser diferente en cada módulo:

```
CajaAhorros.extraer(Monto m) {
  if m =< saldo
  then saldo = saldo - m;
  else print('Saldo insuficiente');
}

CuentaCorriente.extraer(Monto m) {saldo = saldo - m;}
```

En tanto que la implementación de las otras subrutinas es la misma para ambos módulos.

Queremos que presten especial atención al hecho de que esta solución y la primera que analizamos, tienen básicamente la misma cantidad de líneas de código. O sea, un buen diseño no implica (al menos necesariamente) un programa más grande.

Por otro lado, habrá código cliente de `CajaAhorros` y `CuentaCorriente` que hará exactamente lo mismo independientemente del tipo de cuenta. Por ejemplo, una función que calcula el saldo total de todas las cuentas del banco debería recibir un arreglo o lista con todas las cuentas y retornar la suma algebraica de sus componentes. Sin embargo, no podemos declarar ese arreglo porque no podemos asignarle un tipo. Si declaráramos `CajaAhorros cuentas[100]` no podríamos poner una instancia de `CuentaCorriente`; y, simétricamente, si declaráramos `CuentaCorriente cuentas[100]` no podríamos poner una instancia de `CajaAhorros`. Este tipo de problemas se solucionan recurriendo al concepto de *herencia (de interfaces)*. Entonces, definimos un módulo (abstracto, es decir, sin implementación) con la interfaz que venimos usando.

MODULE	`CuentaBancaria`
IMPORTS	`Monto`
EXPORTS	`depositar(i Monto)`
	`extraer(i Monto)`
	`saldo():Monto`

Luego hacemos que `CajaAhorros` y `CuentaCorriente` sean herederos de `CuentaBancaria`.

MODULE	`CajaAhorros` INHERITS FROM `CuentaBancaria`

MODULE	`CuentaCorriente` INHERITS FROM `CuentaBancaria`

Estos dos herederos se implementan como ya mencionamos: la implementación de `depositar()` y `saldo()` es la misma para ambos; mientras que la de `extraer()` es diferente, como mostramos más arriba.

Ahora podemos definir el arreglo o lista que recibiría la función que calcula el saldo total de todas las cuentas del banco: `CuentaBancaria cuentas[100]`. Este arreglo puede almacenar en cada componente una instancia de `CajaAhorros` o `CuentaCorriente`:

```
CuentaBancaria ca, cc, cuentas[100];
ca = new CajaAhorros;                // ligamos ca a una implementación
cc = new CuentaCorriente;            // ligamos cc a la otra implementación
cuentas[1] = ca;
cuentas[2] = cc;
print(cuentas[1].saldo() + cuentas[2].saldo);
```

Además, observen que de esta forma el tipo de cada cuenta se establece con primitivas del lenguaje en lugar de tener que hacerlo programáticamente (como hicimos en la primera solución).

Dijimos que la implementación de `depositar()` y `saldo()` es la misma para `CajaAhorros` y `CuentaCorriente`. Hasta ahora esto significa que debemos literalmente copiar la implementación de un módulo en el otro. Es decir tendremos dos copias idénticas de `depositar()` y `saldo()`. Pueden ir pensado cómo evitar esta repetición de esta parte del código, aunque más adelante vamos a ver cómo solucionarlo.

Otra cuestión que pueden analizar es el mensaje de error que imprimimos en la implementación de `extraer()` de `CajaAhorros`. ¿Es correcto imprimir un mensaje de error desde `extraer()`? ¿Por qué sí? ¿Por qué no? ¿Cuál es el criterio que determina si es correcto o no hacerlo? Si no es correcto, modifiquen el diseño apropiadamente.

4. Cuarto problema

Volvemos sobre el problema visto en la Sección 2. Supongamos que tenemos que imprimir un título en el formulario (por ejemplo, INGRESE SU NOMBRE Y DNI) *que depende del idioma configurado en la computadora donde ejecuta el sistema*. Nuevamente analizaremos dos soluciones.

La primera consiste en incluir una sentencia tipo `case` en el método `dibujar()` del módulo `FormNombreDNI`. Asumimos que la variable de entorno `LANG` guarda el idioma local.

```
dibujar() {
  case LANG is
    'sp': print('Ingrese su nombre y DNI');
    'en': print('Enter your name and ID number');
    // más idiomas
  endcase
  // dibujar el formulario con los campos para nombre y DNI
}
```

Obviamente esta solución es muy simple e intuitiva y funciona. Pero está mal desde el punto de vista del DOO. La razón es la misma de siempre pues ocultamos varios ítems de cambio en el mismo módulo: por un lado implementamos la forma específica en que se dibuja el formulario; y por el otro, implementamos cuestiones que dependen de la *internacionalización* del la aplicación[1]. Claramente podemos querer cambiar una sin cambiar la otra pero al tener todo en el mismo módulo corremos el riesgo de introducir errores en la parte que no queremos cambiar.

La segunda opción pasa por aplicar la metodología de Parnas de forma *liviana*. Es decir, no vamos a mostrar el diseño ideal sino una primera aproximación. En esta primera aproximación definimos un módulo por idioma que implementa todo el formulario, todos con la misma interfaz.

MODULE	`FormNombreDNI_SP`
EXPORTS	`dibujar()`
	`leerEntrada()`

[1]Sobre internacionalización pueden ver: https://en.wikipedia.org/wiki/Internationalization_and_localization.

MODULE	FormNombreDNI_EN
EXPORTS	dibujar()
	leerEntrada()

De esta forma tenemos la siguientes implementaciones de `dibujar()`:

```
FormNombreDNI_SP.dibujar() {
  print('Ingrese su nombre y DNI');
  // dibujar el formulario con los campos para nombre y DNI
}

FormNombreDNI_EN.dibujar() {
  print('Enter your name and ID number');
  // dibujar el formulario con los campos para nombre y DNI
}
```

Ahora cada módulo oculta *dos* ítems de cambio: el idioma del título del formulario y la implementación del formulario en sí. No es lo ideal pero hemos mejorado bastante. Además, notar que desaparece la sentencia `case`. Esto no es menor porque las sentencias condicionales son una de las causas principales que atentan contra la legibilidad del código.

Como hemos visto en otras oportunidades, tener varios módulos con la misma interfaz pero diferente implementación nos trae problemas cuando queremos pasar una instancia de alguno de ellos a funciones que hacen lo mismo independientemente de cuál sea la implementación de la instancia que reciben. Por ejemplo, podríamos tener una función que recibe varios formularios e implementa el flujo entre ellos (o sea, cuál es la secuencia de "pantallas" de la aplicación, lo cual es independiente del idioma). Si esa función tiene tipo:

```
workflow(FormNombreDNI_EN nomDni,...)
```

no le podremos pasar una instancia de `FormNombreDNI_SP` como primer parámetro. La solución a este problema pasa por definir una estructura de herencia donde la raíz es un módulo abstracto (sin implementación) y todos los herederos tienen la misma interfaz aunque diferentes implementaciones.

MODULE	FormNombreDNI
EXPORTS	dibujar()
	leerEntrada()

MODULE	FormNombreDNI_SP INHERITS FROM FormNombreDNI

MODULE	FormNombreDNI_EN INHERITS FROM FormNombreDNI

Con esta estructura de herencia el tipo de `workflow` es el siguiente:

```
workflow(FormNombreDNI nomDni,...)
```

Esto nos permite invocarla con formularios en inglés o castellano:

```
FormNombreDNI fsp, fen;
fsp = new FormNombreDNI_SP;
fen = new FormNombreDNI_EN;
workflow(fsp,...);
......
workgflow(fen,...);
```

De todas formas, la sentencia `case` vuelve a aparecer cuando queremos crear el formulario correcto para el idioma local:

```
main() {
  FormNombreDNI form;
  case LANG is
    'sp': form = new FormNombreDNI_SP;
    'en': form = new FormNombreDNI_EN;
    // más idiomas
  endcase
  .....
  workflow(form,...);
  .....
}
```

Sin embargo, el problema es menos grave que teniendo la sentencia `case` dentro de `dibujar()`. En efecto, en `dibujar()` no solo el título del formulario depende del idioma local sino también las etiquetas de los campos para ingresar el nombre y el DNI y la del botón Aceptar. Entonces, analizando el caso más en profundidad, vemos que en realidad con la primera solución de diseño tendríamos varias sentencias `case` dentro de `dibujar()`. En tanto que con la solución que aplica la metodología de Parnas (aun livianamente) tenemos un único `case` en `main()`.

Resta, no obstante, tener módulos que oculten solo un ítem de cambio cada uno (recuerden que hasta el momento `FormNombreDNI_SP` y `FormNombreDNI_EN` ocultan dos ítem de cambio cada uno). Esto lo resolveremos más adelante (sugerimos que lo vayan pensando).

> *¿Por qué es tan importante tener módulos que oculten un único ítem de cambio? ¿Por qué hacemos tanto énfasis en la cuestión del cambio en el diseño de software?*

5. Quinto problema

Volvemos sobre el problema de la Sección 3. Queremos evitar la repetición del código de `depositar()` y `saldo()` (recordar que el código de estos métodos es el mismo para las implementaciones de `CajaAhorros` y `CuentaCorriente`).

Desde el punto de vista de la teoría de diseño, la repetición de ese código no es un problema pues el diseño contempla correctamente los cambios posibles (en este caso, podemos modificar la implementación de estos métodos en `CajaAhorros` sin que impacte en `CuentaCorriente` y

viceversa). Sin embargo, la repetición de ese código puede traer problemas de mantenimiento. Por ejemplo, si vemos que hay un error en `CajaAhorros.depositar()` deberíamos recordar corregirlo también en la implementación de `CuentaCorriente`.

Este problema se resuelve mediante *composición de objetos*; o mejor, combinando adecuadamente herencia con composición. Lo que vamos a hacer es definir un módulo que reúna todo el código que es común a `CajaAhorros` y `CuentaCorriente`, y vamos a hacer que `depositar()` y `saldo()` deleguen sus implementaciones en ese nuevo módulo. Empezamos definiendo el módulo que implementa el código común.

MODULE	`CuentaImp`
IMPORTS	`Monto`
EXPORTS	`depositarImp(i Monto)`
	`extraerImp(i Monto)`
	`saldoImp():Monto`

Observen que la interfaz de `CuentaImp` es distinta de la de `CuentaBancaria`. La implementación de los métodos de `CuentaImp` es la siguiente.

```
depositarImp(Monto m) {saldo = saldo + m;}

extraerImp(Monto m) {saldo = saldo - m;}

Monto saldoImp() {return saldo;}
```

Ahora tenemos que componer un objeto `CuentaImp` dentro de objetos `CajaAhorros` y `CuentaCorriente`. Para esto definimos una variable de tipo `CuentaImp` como variable de estado de `CajaAhorros` y `CuentaCorriente` y usamos los *constructores* de estos módulos para realizar la composición.

```
CajaAhorros(CuentaImp imp) {miImp = imp;}

CuentaCorriente(CuentaImp imp) {miImp = imp;}
```

Finalmente, escribimos la implementación de los métodos de `CajaAhorros` y `CuentaCorriente` usando, siempre que corresponda, las implementaciones de `CuentaImp`.

```
CajaAhorros.depositar(Monto m) {miImp.depositarImp(m);}

CajaAhorros.extraer(Monto m) {
  if m =< miImp.saldoImp()
  then miImp.extraerImp(m);
  else print('Saldo insuficiente');
}

CajaAhorros.saldo() {return miImp.saldoImp();}

CuentaCorriente.depositar(Monto m) {miImp.depositarImp(m);}
```

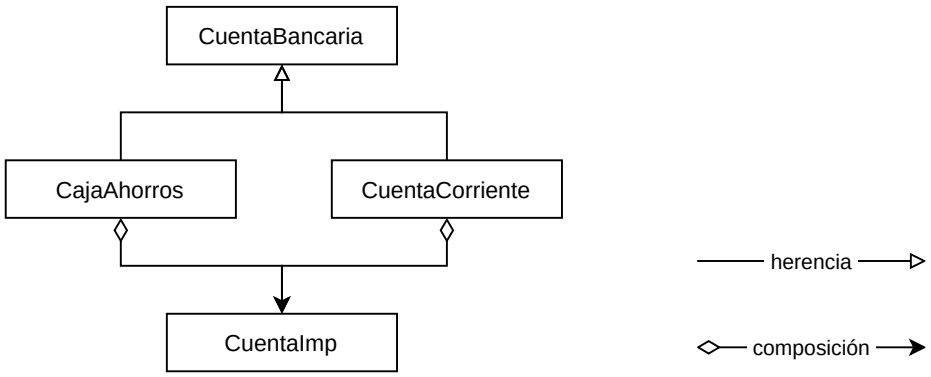

Figura 1: Representación gráfica de las relaciones entre los módulos

```
CuentaCorriente.extraer(Monto m) {miImp.extraerImp(m);}

CuentaCorriente.saldo() {return miImp.saldoImp();}
```

Las relaciones entre los módulos que hemos definido se pueden representar gráficamente como en la Figura 1. Es decir, `CajaAhorros` y `CuentaCorriente` son herederos de `CuentaBancaria`; y los primeros se componen con un objeto de tipo `CuentaImp`. Muchos desarrolladores consideran que hacer un diseño de software es básicamente dibujar algo como lo que vemos en la figura. Nosotros creemos que la información que transmite la figura no es suficiente como para definir un diseño. No es suficiente como para que un programador pueda realizar una implementación como la que se muestra en esta sección. Si el diseño fuese solo la figura, el programador podría hacer una implementación que no verificaría las propiedades de diseño para el cambio que buscamos. Y sin esas propiedades el diseño estaría mal. La figura no explica cuáles son las interfaces de los módulos y la forma en que `CajaAhorros` y `CuentaCorriente` delegan parte de la implementación en `CuentaImp`. En general, no tiene sentido intentar representar gráficamente esta información; es mucho más eficaz y eficiente recurrir a notaciones basadas en texto. Para las interfaces nosotros utilizamos 2MIL y para explicar la delegación de funcionalidad usamos la Guía de Módulos donde debemos explicar la funcionalidad de cada módulo. Se podría argumentar que escribir esta documentación alarga los tiempos de desarrollo. Nosotros creemos que no es así. Tarde o temprano toda la documentación de diseño que proponemos se transforma en código fuente: tarde o temprano el programador debe escribir las interfaces de sus módulos; tarde o temprano el programador debe decidir cuál es la funcionalidad de cada módulo. Si el programador solo cuenta con la figura, implementará un diseño sin las propiedades que buscamos o deberá preguntar al diseñador cómo quiere que tal o cual cosa sea implementada. En el primer caso la calidad del proyecto será inferior mientras que en el segundo el equipo perderá tiempo en reuniones y el proyecto perderá valor porque lo que allí se discuta o aclare no quedará escrito[2]. Nada de esto implica que la figura no tenga valor; solo decimos que su valor no es tan grande como se sugiere en otras metodologías de diseño. Una figura como esta sirve para comunicar los trazos más gruesos del diseño del sistema.

[2]En general, la comunicación oral de información técnica tiende a ser mal interpretada, olvidada y de inferior calidad a la escrita.

La Figura 1 NO es el diseño. La descripción del diseño requiere información que la figura no incluye. Documentar un diseño no alarga los tiempos de desarrollo, los reduce.

Tengan en cuenta que para usar correctamente este diseño se debe componer cada objeto de tipo `CajaAhorros` o `CuentaCorriente` con un objeto *diferente* de tipo `CuentaImp`.

```
CuentaBancaria cc, ca;
CuentaImp ci1, ci2;
ci1 = new CuentaImp;
ci2 = new CuentaImp;
cc = new CuentaCorriente(ci1);
ca = new CajaAhorros(ci2);
```

Obviamente que si vamos a definir este diseño deberíamos hacerlo desde el inicio. Aquí lo fuimos haciendo en etapas por cuestiones pedagógicas. Lo más importante del diseño lo hicimos en las Secciones 1 y 3. Eso es lo fundamental. Eso determina si el diseño es bueno o no lo es. Lo hecho en esta sección no es determinante para la calidad final del código. Probablemente para un programa tan simple como este sea suficiente con mantener copias del código en lugar de reunirlo en `CuentaImp`. Si es seguro que el código que es común a `CajaAhorros` y `CuentaCorriente` siempre lo será (es decir *no* es un ítem de cambio) entonces se puede recurrir a herencia de clases para evitar mantener dos copias. En este caso el código común se implementa en `CuentaBancaria` y no es necesario definir ni componer `CuentaImp`.

6. Sexto problema

Vamos a resolver el problema de que los módulos como `FormNombreDNI_SP` y `FormNombreDNI_EN` definidos en la Sección 4 ocultan *dos* ítems de cambio cada uno. Cada módulo oculta o implementa el título del formulario en un idioma y la implementación del formulario en sí.

Aplicando la metodología de Parnas, deberíamos tener dos módulos: uno que resuelve el problema del idioma del título y otro que implementa el formulario. El que implementa el formulario deberá imprimir el título *pero sin saber en qué idioma*. O sea, el módulo que implementa el formulario tiene que imprimir el título sí o sí (*no* es un ítem de cambio) pero no tiene que inmiscuirse en qué idioma hay que imprimirlo (*es* un ítem de cambio, porque hay varios idiomas). Como técnica general de diseño vamos a usar la combinación entre herencia y composición.

Entonces definimos un módulo para cada título (sí un módulo para cada título).

MODULE	`TituloFormNombreDNI`
EXPORTS	`titulo():String`

MODULE	`TituloFormNombreDNI_SP` INHERITS FROM `TituloFormNombreDNI`

MODULE	`TituloFormNombreDNI_EN` INHERITS FROM `TituloFormNombreDNI`

La implementación de `titulo()` es muy simple en cada caso.

```
String TituloFormNombreDNI_SP.titulo() {return 'Ingrese su nombre y DNI';}

String TituloFormNombreDNI_EN.titulo() {return 'Enter your name and ID number';}
```

Ahora tenemos que usar composición para decirle al formulario cuál es título que tiene que imprimir. Para eso podemos usar el constructor del módulo y una variable de estado que mantenga una referencia a un objeto de tipo `TituloFormNombreDNI`.

```
FormNombreDNI(TituloFormNombreDNI t) {miTitulo = t;}
```

La interfaz del módulo encargado de implementar el formulario es la misma que usamos en las secciones anteriores (recordar que los constructores no forman parte de la interfaz en lo que refiere a herencia).

MODULE	`FormNombreDNI`
IMPORTS	`TituloFormNombreDNI`
EXPORTS	`FormNombreDNI(i TituloFormNombreDNI)`
	`dibujar()`
	`leerEntrada()`

La diferencia es que ahora le asignamos una funcionalidad levemente diferente a `dibujar()`: tiene que imprimir un título que lo debe obtener a partir de `miTitulo`.

```
dibujar() {
  print(miTitulo.titulo());
  // dibujar el formulario con los campos para nombre y DNI
}
```

En `main()` configuramos adecuadamente el formulario en tiempo de ejecución ligando el título al idioma local.

```
main() {
  TituloFormNombreDNI t;
  FormNombreDNI form;
  case LANG is
    'sp': t = new TituloFormNombreDNI_SP;
    'en': t = new TituloFormNombreDNI_EN;
    // más idiomas
  endcase
  form = new FormNombreDNI(t);
  workflow(form,...);
  .....
}
```

Con este diseño es muy simple agregar soporte para nuevos idiomas: basta con implementar un heredero de `TituloFormNombreDNI` y agregar una línea a la sentencia `case` de `main()`.

Este diseño respeta el principio de diseño conocido como "Diseños abiertos y cerrados".

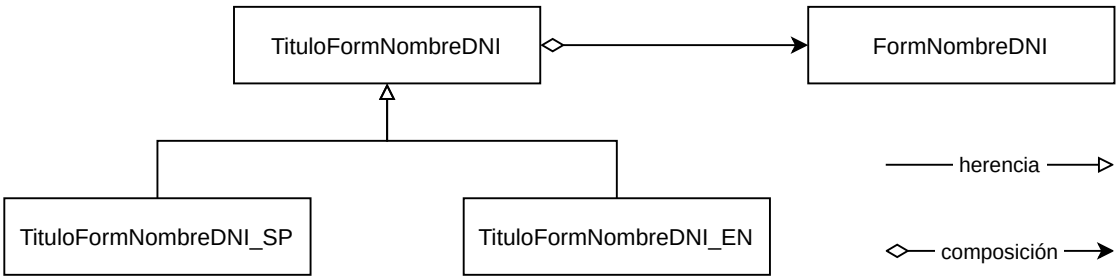

Figura 2: Representación gráfica de las relaciones entre los módulos

Recuerden que el título del formulario no es lo único que depende del idioma local. También están las etiquetas de los campos de texto para el nombre y el DNI y la etiqueta del botón `Aceptar`. Piensen cómo diseñar el formulario teniendo en cuenta estas etiquetas.

La representación gráfica de las relaciones entre los módulos que definimos en este problema se pueden ver en la Figura 2.