Diseño de Software

Maximiliano Cristiá

Ingeniería de Software 2

FCEIA — UNR

2022

Índice

1. Dos diseños distintos, ¿cuál es el mejor?

A continuación presentaremos dos diseños distintos para el mismo problema y analizaremos cuál de ellos es el mejor.

Consideremos el problema de las cajas de ahorro que hemos visto a lo largo del año, al cual le adicionamos el siguiente requisito:

- *Se debe registrar en una bitácora cada depósito mayor o igual a $1.000.000.*

Recordemos que diseñar un sistema de software consiste en:

1. Descomponer el sistema en elementos de software
2. Asignar una funcionalidad a cada elemento
3. Establecer la relaciones entre esos elementos

Con el fin de simplificar la presentación, describiremos los diseños en términos de conceptos de programación orientada a objetos y solo nos enfocaremos en una parte del sistema o los requerimientos. Aun así, debe quedar muy claro que un diseño *no* se debe dar en términos de una tecnología de implementación particular. Lo hacemos así porque aun no hemos introducido los conceptos y notación específicos del diseño de software.

1.1. Primer diseño

La descomposición del primer diseño consiste en una clase, `CajaAhorros`, que tiene la siguiente interfaz:

- `depositar(Dinero m)`: suma la cantidad `m` al saldo actual de la cuenta; si `m` es mayor o igual a $1.000.000 se invoca la rutina interna `registrarDepósito()`. Esta última registra los datos del depósito (e.g. número de caja de ahorros, fecha, hora, etc.) en un archivo de texto.
- `extraer(Dinero m)`: resta la cantidad `m` al saldo actual de la cuenta si este es mayor o igual a `m`.
- `Dinero saldo()`: retorna el saldo actual de la caja de ahorros.

Observar que al dar la interfaz de la clase hemos también asignado la funcionalidad. Como el diseño consiste de un único elemento no podemos establecer relaciones entre ellos.

1.2. Segundo diseño

La descomposición del segundo diseño consiste de dos clases: `CajaAhorros` y `RegistroDepGrandes`. La interfaz de `CajaAhorros` es la siguiente:

- `depositar(Dinero m)`: suma la cantidad `m` al saldo actual de la cuenta.
- `extraer(Dinero m)`: resta la cantidad `m` al saldo actual de la cuenta si este es mayor o igual a `m`.
- `Dinero saldo()`: retorna el saldo actual de la caja de ahorros.

La interfaz de `RegistroDepGrandes` es la siguiente:

- El constructor de la clase recibe un parámetro de tipo `CajaAhorros` que guarda en una variable de estado (o variable miembro) que llamaremos `ca`.
- `depositar(Dinero m)`: si `m` es mayor o igual a $1.000.000 registra los datos del depósito (e.g. número de caja de ahorros, fecha, hora, etc.) en un archivo de texto, y luego invoca `ca.depositar(m)`; caso contrario invoca directamente `ca.depositar(m)`.
- `extraer(Dinero m)`: invoca `ca.extraer(m)`.
- `Dinero saldo()`: invoca `ca.saldo()`.

Notar que ambas clases tienen la misma interfaz.

La relación entre ambas clases se establece por *composición de objetos* (ver Sección 6.3). Es decir, se declara un objeto de tipo `CajaAhorros` y uno de tipo `RegistroDepGrandes`, luego el primero se pasa como parámetro del constructor del segundo. En términos de código se da lo siguiente[1]:

```
CajaAhorros c;
RegistroDepGrandes r.RegistroDepGrandes(c);
```

De allí en más el sistema trabaja sobre `r`. Por ejemplo si queremos depositar 1.000 pesos en `c` hacemos `r.depositar(1000)`. En consecuencia cuando se efectúa un depósito, `r` controla si el monto es mayor o igual a 1.000.000 en cuyo caso deja el registro correspondiente pero también hace el depósito sobre `c`, como lo indica la función de `RegistroDepGrandes.depositar()`.

[1] Asumimos que el constructor de una clase tiene el mismo nombre que la clase.

1.3. Análisis

¿Cuál de los dos diseños es el mejor? ¿Cuál es el criterio para determinar el mejor diseño? El criterio es el principio de Diseño para el Cambio. El mejor diseño es aquel que permite la implementación de cambios (modificaciones) de manera tal que:

- Cada cambio se efectúe al menor costo posible
- La introducción de cada cambio no degrade la integridad conceptual del sistema

Entonces, para determinar cuál de los dos diseños es el mejor veremos qué costo tendría introducir algunos cambios probables.

1. No es necesario dejar un registro de los depósitos grandes.
2. La bitácora se guarda en una base de datos (en lugar de en un archivo de texto).
3. También hay que dejar un registro de las extracciones de más de $100.000.

Para incorporar el primer cambio, en el primer diseño deberíamos eliminar la sentencia condicional que comprueba si hay que dejar registro o no y la rutina interna `registrarDepósito()`. Esto implca tener que volver a verificar todas las subrutinas de `CajaAhorros` dado que estamos modificando su implementación. Por el contrario, en el segundo diseño lo único que tenemos que hacer es declarar `r` como una `CajaAhorros`. No tenemos que volver a verificar nada de `CajaAhorros` puesto que no hemos modificado su implementación.

La incorporación del segundo cambio en el primer diseño tiene más o menos los mismos problemas que el primer cambio. Es decir, debemos modificar la implementación de `CajaAhorros` lo que implica tener que verificar nuevamente la nueva implementación. Observar que esa implementación incluye funcionalidad básica de una caja de ahorros (por ejemplo, mantener actualizado el saldo) que de igual modo debemos volver a verificar. Por el contrario, introducir esta segunda modificación en el segundo diseño solo implica modificar y verificar `RegistroDepGrandes`. No tenemos que hacer nada sobre `CajaAhorros`; la implementación de la funcionalidad básica no cambia.

En cuanto al tercer cambio, siguiendo la idea del primer diseño deberíamos introducir una sentencia condicional en `extraer()` similar a la que tenemos en `depositar()`. Como modificamos la implementación de `CajaAhorros` deberíamos testear todo nuevamente. O sea tenemos problemas similares a los que se producen con los dos primeros cambios. Siguiendo la idea del segundo diseño, deberíamos definir una nueva clase, `RegistroExtGrandes`, con interfaz e implementación similar a la de `RegistroDepGrandes` e integrarla también de forma similar. Es decir, mediante composición, esta vez, componiendo una `CajaAhorros` con un `RegistroExtGrandes` con un `RegistroDepGrandes`. Claramente, no deberíamos volver a verificar nada de lo que ya está implementado.

Ahora supongamos que este programa se vende a distintos bancos. Supongamos que algunos tienen que registrar los depósitos grandes, otros las extracciones grandes, otros ambos y otros ninguno. Con el primer diseño tendríamos que entregar todo el código a todos los bancos y además deberíamos incluir nuevas sentencias condicionales que determinen si ese cliente tiene que registrar algo o no. . . ¡y volver a verificar todo! Tengan en cuenta que en este diseño cada cambio degrada la integridad conceptual del sistema. Claramente, comenzamos con una clase relativamente pequeña y fácil de comprender que con el tiempo se va haciendo cada vez más grande y compleja.

En cambio, con el segundo diseño tendríamos 4 configuraciones (o versiones) que surgen de combinar las 3 clases que tenemos más el programa principal. Cada una de estas configuraciones se compilaría por separado dando lugar a 4 binarios diferentes, cada uno con la funcionalidad mínima que requiere cada banco.

Es evidente la superioridad del segundo diseño frente al primero: cada cambio tiene menor costo en el segundo que en el primero, y en el segundo no se degrada la integridad conceptual del sistema mientras que en el primero sí. En lo que resta del apunte veremos cuáles son los principios, conceptos y técnicas que permiten construir de forma sistemática diseños que tienen estas propiedades.

2. Diseño basado en ocultación de la información (DBOI)

El DBOI se basa en uno de los principios de la Ingeniería de Software: Diseño para el Cambio. Es decir, no suponer jamás que los requerimientos de un sistema serán dados de una vez y para siempre sino que cambiarán muchas veces durante la vida del sistema. Por lo tanto, se debe concebir al sistema de forma tal que:

- Pueda ir incorporando los nuevos requerimientos evitando que su integridad conceptual se degrade.
- Los cambios puedan incorporarse con el menor costo posible.

Resulta evidente, entonces, que antes de comenzar con el diseño propiamente dicho se deben analizar las posibles líneas evolutivas del sistema y los cambios probables en los requerimientos. Por otro lado, cada vez que debamos introducir un cambio deberemos modificar una o más unidades de implementación de software (código fuente, archivos de configuración, etc.). Cuando una unidad de software implementa una funcionalidad específica se la llama *módulo*.

Definición 1. *Un módulo es una unidad de implementación de software que provee una unidad coherente de funcionalidad [5]. También se puede definir como una unidad de implementación de software que provee un conjunto de servicios [6]. Parnas define módulo como una asignación de trabajo para un programador o un grupo de programadores [9].*

Como mencionamos en la definición de diseño en el capítulo anterior, el diseño de un sistema se compone de muchos elementos de software. Una de las clases de elementos fundamentales en todo diseño es el módulo pues es en donde, en definitiva, se deberán materializar los cambios que seguramente el cliente exigirá, en tanto los módulos son las unidades de implementación (es decir, donde está el código fuente). Como el Diseño para el Cambio sugiere tener en cuenta los cambios probables para incorporarlos con el menor costo posible, y los cambios se deben realizar dentro de los módulos del sistema, entonces se busca descomponer el sistema en módulos (es decir, diseñar los módulos) de forma tal que:

- Cada módulo se pueda implementar independientemente de los restantes.
- Cada módulo pueda ser comprendido completamente sin necesidad de comprender los otros en su totalidad.
- Sea posible cambiar la implementación de un módulo sin conocer la implementación de los otros y sin afectarlos.

- Sea posible incorporar un cambio importante como un conjunto de cambios pequeños a distintas módulos.

Si bien la descomposición de un sistema en módulos es, tal vez, la parte más importante del diseño, este no termina aquí pues, como dijimos, hay otras clases de elementos que forman parte del diseño (tales como procesos, subrutinas, etc.). Además, las relaciones entre los elementos, que también forman parte del diseño, son tanto o más importantes que los elementos mismos. De hecho más adelante veremos que para lograr descomponer el sistema en módulos de forma que se cumplan los objetivos anteriores, es necesario que los módulos se relacionen entre sí de una forma muy particular.

2.1. La metodología de Parnas para descomponer un sistema en módulos

David L. Parnas en los primeros años de la década del 70 [11], observó que al descomponer un sistema en módulos se alcanzan los objetivos anteriores si se siguen los siguientes pasos:

1. Se identifican los ítem con probabilidad de cambio presentes en los requerimientos.
2. Se analizan la diversas formas en que cada ítem puede cambiar.
3. Se asigna una probabilidad de cambio a cada variación analizada.
4. Se aislan en módulos separados los ítem cuya probabilidad de cambio sea alta; implícitamente este punto indica que en cada módulo se debe aislar un **único** ítem con probabilidad de cambio.
5. Se diseñan las interfaces de los módulos de manera que resulten insensibles a los cambios anticipados.

Aplicar esta metodología no es siempre fácil. Parnas la propone como forma de minimizar el costo de desarrollo y mantenimiento del sistema y requiere que el diseñador estime la probabilidad de que se den los diversos cambios posibles. Estas estimaciones se basan en la experiencia del diseñador y de otros diseñadores y requieren conocimiento sobre el dominio de aplicación así como también entender la dinámica de la tecnología vinculada al software y hardware.

A esta forma de diseñar se la denomina *diseño basado en ocultación de información*. Obviamente, la información que se oculta es la implementación de cada ítem que *probablemente* cambiará en el futuro. En otras palabras: cada módulo de la descomposición se caracteriza por su conocimiento de una decisión de diseño que oculta a los demás módulos; su interfaz se elige de manera tal de revelar lo menos posible sobre su maquinaria interna [11].

Principio de Diseño: Principio de Ocultación de la Información

Los ítem con alta probabilidad de cambio son el fundamento para descomponer un sistema en módulos. Cada módulo de la descomposición debe ocultar un **único** *ítem con alta probabilidad de cambio, y debe ofrecer a los demás módulos una interfaz insensible a los cambios anticipados.*

Definición 2 (DBOI)**.** *Un sistema respeta el diseño basado en ocultación de la información (DBOI) si cada uno de sus módulos fue diseñado aplicando el POI y se siguieron adecuadamente cada uno de los pasos anteriores.*

En las secciones que siguen estudiaremos con detalle cada uno de los pasos de la metodología de Parnas.

2.2. Análisis de cambio

Los primeros pasos de la metodología del DBOI involucran analizar los cambios que pueden darse en los requerimientos. Comencemos por analizar el concepto de ítem de cambio. Entendemos por ítem de cambio cualquier cosa relativa a los requerimientos funcionales y no funcionales del sistema que suponemos puede modificarse en el futuro por diversas causas. Como indica la metodología, cada módulo del sistema debe aislar un único ítem con alta probabilidad de cambio. Pero un ítem de cambio puede hacer referencia a una porción enorme o pequeña del sistema dependiendo del nivel de abstracción con que se lo describa. Por ejemplo, un ítem de cambio de un cierto sistema puede ser el hardware con el que interactúa o puede dividirse en un ítem de cambio por cada dispositivo de hardware. Como la cantidad de código necesaria para implementar el ítem de cambio está en relación directa con la complejidad o el tamaño de las entidades abarcadas por ese ítem, es necesario tener un criterio para determinar cuándo hemos descripto los ítem de cambio con suficiente detalle como para considerar que no deben ser subdivididos. No obstante, dar ese criterio es equivalente a dar un criterio para determinar la envergadura máxima que puede tener cada módulo en que se subdivide el sistema. Parnas, en este sentido, sugiere que el sistema debe ser subdividido en módulos hasta que cada uno tenga un tamaño y complejidad tales que si el programador a cargo de su implementación se retira del proyecto, lo que había hecho hasta ese momento puede ser descartado de forma tal que el nuevo programador empieza desde cero; en otras palabras, tiene que ser más barato empezar desde cero que entender lo hecho hasta el momento y continuarlo. Entonces, al describir cada ítem de cambio debemos pensar en la envergadura que tendrá el módulo que lo aisle. No nos podemos permitir módulos grandes o complejos (o lo que es lo mismo ítem de cambio mal descriptos) porque es caro. En relación al ejemplo del hardware como ítem de cambio, subdividirlo en varios ítem específicos sería lo correcto.

Para saber cuándo un ítem puede cambiar y cómo puede hacerlo es conveniente tener en cuenta, al menos, cada uno de los siguientes puntos:

- Contracción y extensión de los requisitos (es decir pensar que algunos clientes pueden querer un sistema más grande, complejo o completo del que se está diseñanado mientras que otros pueden querer un sistema más pequeño, más simple, con menos funciones, etc.)
- Cambio de/en los algoritmos
- Cambio en la representación de las estructuras de datos y en la forma de organizarlos
- Cambio en la máquina abstracta subyacente (hardware, sistema operativo, compilador, runtime, etc.)
- Cambio en los dispositivos periféricos
- Cambio en el entorno socio-cultural (moneda, impuestos, fechas, idioma, etc.)
- Cambios propios del dominio de aplicación
- Cambios propios del negocio de la compañía desarrolladora
- Interconexión con otros sistemas

Al mismo tiempo, se debe recordar que “cada módulo de la descomposición se caracteriza por su conocimiento de una decisión de diseño que oculta”, por lo que cualquier decisión que se tome respecto a la instrumentación de cualquier requerimiento es un ítem de cambio.

Por ejemplo, una decisión de diseño es representar cierto ente del mundo real mediante tal o cual estructura de datos (con referencia al ejemplo de la sección **??**, representar la lista de personas como una secuencia de registros de cierta forma en un archivo de texto abarca varias decisiones de diseño); de igual modo, la implementación de un determinado proceso real con cierto algoritmo es otra decisión de diseño.

Si uno toma esto en términos absolutos, todo puede cambiar. Por esta razón es que en la metodología propuesta por Parnas, se indica que se le asigne a cada ítem de cambio una probabilidad de que ocurra y se pide que sólo se aislen aquellos con mayor probabilidad.

2.3. Interfaz e implementación

Para entender los pasos 4 y 5 es necesario entender primero qué forma o propiedades tienen los módulos. Un módulo puede ser visto, usado o accedido por otros módulos del sistema u otros sistemas. Un módulo de un sistema de software consta de dos secciones: *interfaz* e *implementación*. La interfaz de un módulo es todo aquello que los otros módulos pueden ver, usar o acceder en ese módulo. La implementación es la forma en que la interfaz se realiza; muchas veces se habla de *secreto* del módulo. Si un módulo puede ver, usar o acceder un elemento en otro módulo es porque este último *exportó* ese elemento. O sea que la interfaz de un módulo es todo lo que el módulo exporta.

Definición 3. *La interfaz de un módulo puede definirse como el conjunto de servicios que el módulo exporta [6]. También puede definirse como todas las interacciones que tiene el módulo con su entorno (es decir, los otros módulos) [5].*

Algunos de los servicios que un módulo puede exportar son: declaraciones de tipos, subrutinas, variables, constantes, etc.

Algunas de las interacciones con el entorno son: llamadas a subrutinas, tiempo de ejecución de esas subrutinas, especificación funcional de esas subrutinas, invariantes que preserva el módulo, etc.

Si un servicio pertenece a la interfaz de un módulo se dice que el servicio es público.

Definición 4. *La implementación de un módulo es la forma en que se logra que la interfaz funcione según lo perciben los otros módulos. Cualquier elemento de la implementación se dice que es privado.*

Claramente la interfaz de un módulo captura la visión externa del módulo. Cuando un módulo presenta una interfaz tal que los otros módulos no pueden suponer razonablemente nada sobre la implementación del módulo, se dice que la *implementación está encapsulada*. Interfaz e implementación no son conjuntos disyuntos. Más aun, si la interfaz no está definida correctamente, los otros módulos podrán hacer supuestos que involucran la implementación, lo que degenera en un sistema cuyo diseño no es un DBOI. Consideremos el siguiente ejemplo para clarificar estos conceptos. Un módulo exporta un arreglo de enteros de longitud fija con el fin de ordenar sus componentes; en otras palabras el arreglo es parte de la interfaz del módulo. Debido a cuestiones de eficiencia, el arreglo es copiado en una lista simplemente enlazada antes de ser ordenado. Se ordena la lista y luego el resultado es volcado en el arreglo. Ningún módulo externo sabe de la existencia de la lista, por lo tanto no es parte de la interfaz y claramente forma parte de la implementación. Ahora supongamos que la lista desaparece y se ordena el arreglo de forma directa. Entonces el arreglo es parte de la interfaz y de la implementación al mismo tiempo.

2.4. Abstracción y encapsulamiento

Volviendo a la metodología propuesta por Parnas (puntos 4 y 5), vemos que los ítem con alta probabilidad de cambio se pueden aislar en módulos independientes si cada uno de estos ítem se convierte en la implementación de un módulo y no un su interfaz, y la interfaz se diseña de forma tal que la implementación esté encapsulada. Pero entonces el problema pasa a ser cómo se construyen interfaces que oculten o encapsulen la implementación. Afortunadamente existen dos técnicas que ayudan a alcanzar este objetivo: *abstracción* y *encapsulamiento*.

Ambas técnicas se aplican sobre la interfaz pero de forma diferente. La abstracción consiste en lograr que la interfaz provea la menor cantidad de servicios posible y de la manera más abstracta posible. En tanto que el encapsulamiento es el proceso por el cual se ocultan todos los detalles de la implementación que permanecen visibles en los servicios exportados. *La forma básica de aplicar la abstracción al diseño de un módulo es definir la interfaz como un conjunto de subrutinas (que, por definición, son lo único que otros módulos pueden ver, usar o acceder) y luego remover todas aquellas subrutinas que pueden ser realizadas por otras*. Tener en la interfaz subrutinas que puede ser realizadas por otras da lugar a lo que se conoce como *interfaces gruesas*. Finalmente, mediante el encapsulamiento de la interfaz se logran remover los últimos detalles de la implementación que permanecen visibles desde el exterior del módulo. Normalmente es necesario revisar la especificación funcional de cada subrutina (recordar que esto también forma parte de la interfaz) y los parámetros de cada una de ellas. Por un lado se busca que la especificación funcional de cada subrutina esté lo más cerca posible de los requerimientos funcionales y no implique, razonablemente, ninguna implementación particular. Por el otro, se intenta eliminar restricciones propias de una implementación particular que afloran en la interfaz.

Nuevamente recurriremos a un ejemplo para clarificar estos conceptos. Un módulo que debe almacenar ciertos datos, debe proveer una forma para que otros módulos puedan recuperarlos, y debe permitir que un dato sea borrado o modificado. Se decide almacenar los datos en un arreglo suficientemente largo. Una primera interfaz podría ser exportar el arreglo mismo pero la técnica de abstracción nos indica que la interfaz debe estar compuesta por una subrutina que permite agregar un dato, y otras que borran, modifican y recuperan el i-esimo dato almacenado (donde `i` es de tipo `Int`); además se especifica que los últimos tres datos recuperados son almacenados en un *cache* como forma de mejorar la eficiencia de la subrutina de recuperación. Pero la subrutina para modificar un dato puede ser lograda aplicando sucesivamente las subrutinas para borrar y agregar ese dato, entonces, debido a la técnica de abstracción, quitamos esa subrutina de la interfaz. Ahora, aplicando la técnica de encapsulamiento, vemos que la especificación de la subrutina para recuperar datos da detalles más allá de los requerimientos (con referencia al *cache*); por lo tanto, deberíamos especificarla diciendo, simplemente, que retorna el dato solicitado. Pero aun queda un detalle por eliminar de la interfaz: el índice para recuperar los datos. Los requerimientos indican que los módulos externos deben ser capaces de recuperar los datos pero no mencionan que deba existir un acceso directo y mucho menos que el índice deba pertenecer a un subconjunto de los números enteros (como lo es `Int` en la mayoría de los lenguajes de programación). Por lo tanto, la recuperación por acceso directo indexada por un `Int` constituye una violación al principio de ocultación de la información (claramente es una decisión de diseño que debe permanecer oculta). Concretamente, los módulos externos pueden suponer razonablemente que el módulo no podrá almacenar más de `Int` datos. Para eliminar ese parámetro debe eliminarse la subrutina mencionada, reemplazándola por tres subrutinas: una que retorna el primer dato almacenado, otra que indica si quedan datos por recuperar o

no, y otra que retorna el siguiente dato al último recuperado. De esta forma ningún módulo externo puede suponer razonablemente que el contenedor indexa los datos con un `Int` o que puede almacenar una cantidad determinada.

Ahora es conveniente cuestionarse algunas cosas:

- ¿Qué se gana al prohibir que en la interfaz aparezcan las estructuras de datos?

  En el ejemplo que vimos, los programadores pueden preferir usar esa estructura a usar subrutinas de la interfaz, por lo que si más tarde se debe modificar la estructura, se deberán cambiar las subrutinas de la interfaz **más** todos los módulo externos que usan la estructura de forma directa. Claramente esto va en contra del principio de Diseño para el Cambio.

- ¿Qué se gana al remover de la interfaz subrutinas que pueden ser realizadas por otras; es decir, al evitar las interfaces gruesas?

  Todas las subrutinas de la interfaz acceden directamente al secreto del módulo. En consecuencia, cuantas menos haya, menor será el costo ante un cambio en el secreto del módulo. En el ejemplo anterior, la rutina para modificar un dato accedería la estructura donde estos están almacenados. Entonces si hay que cambiar la estructura de datos, hay que cambiar el código de la subrutina que los modifica. En cambio si esa subrutina se remueve de la interfaz y se implementa como:

  ```
  int modificar(.....)
  { borrar(.....);
    agregar(.....);
    return .....;
  }
  ```

  entonces cuando haya que modificar el secreto del módulo no habrá que cambiar el código de `modificar`.

- ¿Qué se gana al ajustar la especificación de las subrutinas a los requerimientos?

  Si la subrutina que recupera los datos implementa un *cache* y esto es informado en la especificación de la interfaz, entonces cualquier módulo externo tiene derecho a trabajar asumiendo esa especificación. En consecuencia si más tarde esa implementación se cambia, los módulos externos tendrán problemas para funcionar correctamente (por ejemplo si trabajan en un entorno de tiempo real). Además, la interfaz que incluye el *cache* es menos reusable (por ejemplo, un módulo externo que requiera un tiempo de recuperación igual para todos los elementos no puede usar nuestro almacén de datos).

  Por otro lado, el *cache*, de ser necesario, se podría implementar fuera del módulo. Básicamente se define un módulo *wrapper* que:

  - Tiene las mismas subrutinas en su interfaz que el módulo original
  - La especificación de la subrutina de recuperación de datos describe un *cache*
  - El módulo implementa el *cache*
  - El módulo redirige directamente todas las peticiones a las subrutinas del módulo original, excepto la de recuperación la cual primero usa el cache y luego, si es necesario, llama a la subrutina original

- ¿Qué se gana al eliminar el acceso directo y la indexación por `Int`?
  Todos los módulos externos que usen nuestro almacén deberán declarar una o varias variables de tipo `Int` para usar como índice de acceso. Por lo tanto, si se cambia el `Int` en la interfaz del almacén por un `LongInt` o un `SmallInt`, todos los módulos externos deberán ser modificados.

Ejercicios

Ejercicio 1. Imagine una empresa que desea un sistema para facturar sus ventas. Suponga que la empresa de desarrollo de software para la que usted trabaja ve una posibilidad de negocio en desarrollar este sistema para luego venderlo a diversas empresas y no solo a la que ahora efectúa el pedido. Explique qué posibilidades de contracción o extensión en los requerimientos debería tener en cuenta para abarcar un mercado importante.
Ejercicio 2. Ejemplifique ítem con probabilidad de cambio e indique cuáles de las categorías de cambio son las más importantes a tener en cuenta en los siguientes dominios de aplicación.

- Aplicaciones Web para comercio electrónico.
- Sistemas para teléfonos celulares.
- Sistemas para facturación en supermercados.
- Clientes de correo electrónico.

Ejercicio 3. En la sección 2.1 decíamos que representar la lista de personas del ejemplo de la sección **??** como una secuencia de registros de cierta forma incluye varias decisiones de diseño. Indique cuáles son esas decisiones de diseño y explique coloquialmente cómo sería un DBOI.
Ejercicio 4. Explique cuál es la interfaz y la implementación de cada uno de los módulos definidos en el ejemplo de la sección **??**. Indique si la implementación de esos módulos está oculta a los otros módulos.
Ejercicio 5. En la sección 2.4 cuestionamos las modificaciones a la interfaz del módulo que debe mantener en orden de llegada ciertos elementos. Respecto de esos cuestionamientos:

1. ¿Qué se pierde al no permitir estructuras de datos en la interfaz? ¿Qué es peor, permitirlas o no? Suponiendo que opta por prohibirlas, ¿cómo haría para evitar los problemas que detectó en la primera pregunta?
2. ¿Qué se pierde al no permitir interfaces gruesas? ¿Qué es peor, permitirlas o no? Suponiendo que opta por prohibirlas, ¿cómo haría para evitar los problemas que detectó en la primera pregunta?
3. ¿Qué se pierde al eliminar la indexación por `Int`? ¿Qué es peor, permitirlo o no? Suponiendo que opta por prohibirlo, ¿cómo haría para evitar los problemas que detectó en la primera pregunta?

3. Diseño del software de control de una estación de peaje

En esta sección presentamos un ejemplo de cierta envergadura y complejidad en el cual aplicamos la metodología de diseño sugerida por Parnas. En primer término introducimos los requerimientos del cliente, luego analizamos los ítem con probabilidad de cambio y finalmente describimos los módulos en que se divide el diseño. En la sección 11 completamos la documentación del ejemplo.

3.1. Los requerimientos

Requerimientos generales. Una estación de peaje desea automatizar el proceso de cobro, emisión de tiques y funcionamiento de las barreras. Con este fin instalará un sistema controlado por una computadora. Cada carril cuenta con una impresora de tiques, una barrera y una máquina que recibe monedas y billetes y entrega cambio. Hay un carril en cada sentido. La barrera debe bajarse 5 segundos después de que el conductor retiró el tique. Todos los demás requerimientos funcionales son los estándar para este tipo de problema, excepto porque a todos los vehículos se les cobra la misma tarifa.

La impresora. El sensor de la impresora deber ser consultado para saber si el conductor retiró el tique.

La máquina receptora de dinero. Los sensores de la máquina para recepción de dinero envían una señal cada vez que una moneda o billete es introducido, pero el sistema debe consultar a la máquina para conocer la denominación de la moneda o billete recibido. La máquina sólo devuelve cambio en monedas, de a una por vez. Las monedas de vuelto son depositadas manualmente por un operador en cinco cilindros, uno para cada denominación, de capacidad conocida. La máquina emite una señal cuando la bandeja que contiene los cilindros es retirada, es reinsertada y cuando un cilindro se vacía.

3.2. Análisis de cambio

Los ítem con probabilidad de cambio que consideraremos para este problema son los siguientes:

- Hardware de los dispositivos (sensores y *actuadores*[2]). Consideraremos que los dispositivos se pueden cambiar por otras marcas o modelos pero con las mismas funciones.
- Representación interna del dinero cobrado y devuelto.
- Requisito temporal sobre la barrera; en general el requisito para bajar la barrera.
- Frecuencia de consulta del sensor de la impresora.
- Cobro diferenciado por tipo de vehículo.
- Medios de pago (tarjetas de crédito/débito, obleas precargadas, etc.).

  Tener en cuenta que algunos medios de pago comunican ciertos datos al sistema para completar la facturación. Por ejemplo, el lector de tarjeta de crédito retorna el número de la tarjeta y otros datos que deben ser enviados al autorizador para que este autorice la transacción; a su vez el sistema que se comunica con el autorizador retorna ciertos datos como el número de la transacción. Estos datos son leídos y utilizados para confeccionar el tique que se entregará al cliente así como también algunos de ellos son almacenados en el sistema de contabilidad (que no forma parte de este ejemplo).

  No es posible conocer ahora todos los datos que serán comunicados por los distintos medios de pago, ni sus tipos; tampoco se sabe ahora si la empresa requerirá más datos de los que piden actualmente.

[2]Un sensor es un dispositivo de entrada mientras que un actuador es un dispositivo de salida.

En un proyecto real el análisis debería ser más detallado y las alternativas de cambio deberían estar mejor descriptas y justificadas. Mejor aun, sería conveniente justificar por qué no se tomaron en cuenta ciertos cambios. Veremos otros ítem de cambio en los ejercicios.

3.3. Módulos del sistema – Introducción a 2MIL

Como se explicó en las secciones anteriores cada módulo debe ocultar un secreto y en general esto se logra definiendo la interfaz de cada módulo a través de un conjunto de subrutinas que son las únicas que pueden acceder al secreto. De aquí que el análisis de cambio sea la referencia obligada para desarrollar el diseño y a la vez el criterio de corrección del mismo. Para describir (documentar) las interfaces de los módulos usaremos un lenguaje muy simple llamado 2MIL[3].

3.3.1. Las barreras

Comenzaremos por definir el módulo que ocultará el hardware de las barreras (parte del primer ítem de cambio). El texto 2MIL es el siguiente.

MODULE	`BarreraCarrilNorte`

EXPORTS	`subir()`
	`bajar()`
	`inicializar()`

COMMENTS	subir(), sube la barrera; bajar(), baja la barrera; inicializar(), pone la barrera en un estado inicial conocido.

Cada palabra reservada tiene el siguiente significado:

MODULE Es el nombre del módulo. Puede ser una cadena de caracteres cualquiera[4]. Este nombre se usará en futuras referencias.

EXPORTS Son las *signaturas* de las subrutinas de la interfaz del módulo. La signatura de una subrutina, también llamada *prototipo* o *encabezado*, está constituida por el nombre, los parámetros y los valores de retorno de la misma. Cada subrutina se lista en una línea separada.

COMMENTS Texto libre para comentar aspectos del módulo o su interfaz que se consideren complicados. No abusar de esta cláusula. En la sección 10 veremos los documentos apropiados para describir otros aspectos de cada módulo. Además tener en cuenta que la caja que encierra la descripción "formal" de cada módulo permite intercalar todo el texto informal que se desee entre la descripción de los módulos[5]; tampoco abusar de esta posibilidad.

Existen otras cláusulas del lenguaje que iremos explicando a medida que sea necesario. Es importante remarcar que si bien el significado de la cláusula EXPORTS es informal, se impone una restricción clara y simple que cualquier implementación está obligada a respetar si se desea

[3] 2MIL es una adaptación del lenguaje TDN presentado en [6].

[4] Las restricciones sobre los identificadores que se usan en 2MIL debería imponerlas un estándar del lenguaje, una herramienta que implemente algunas verificaciones sobre los "programas" que se escriben o el lenguaje de programación con el cual se vaya a implementar el sistema.

[5] Más o menos siguiendo la idea de Z.

seguir los lineamientos del diseño. En otras palabras, en la implementación final del sistema debe verificarse que ninguna porción del código que no pertenezca a la implementación de las subrutinas declaradas en la cláusula EXPORTS acceda al secreto del módulo; la única forma de hacerlo debe ser a través de ellas.

Ahora volvamos al ejemplo que nos ocupa. El módulo `BarreraCarrilNorte` oculta el hardware que controla la barrera instalada en el carril que va en sentido Norte, como lo prescribe la metodología de Parnas. Claramente, un cliente de este módulo podrá subir, bajar y llevar la barrera a un estado conocido pero no podrá conocer jamás cómo se hace exactamente para ordenarle al motor que lleve a cabo cada una de estas acciones. Evidentemente tendremos también:

MODULE	`BarreraCarrilSur`

EXPORTS	`subir()`
	`bajar()`
	`inicializar()`

COMMENTS	subir(), sube la barrera; bajar(), baja la barrera; inicializar(), pone la barrera en un estado inicial conocido.

Pero entonces cabe preguntarse, ¿por qué dos módulos para algo tan semejante? ¿Por qué no un único módulo que controle ambas barreras? Por ejemplo podríamos definir el módulo:

MODULE	`Barreras`

EXPORTS	`subir(i Int)`
	`bajar(i Int)`
	`inicializar(i Int)`

COMMENTS	El índice entero en las subrutinas indica la barrera sobre la cual se quiere operar (0:Norte, 1:Sur o cualquier otra convención). Las subrutinas tienen la misma función que en los módulos anteriores.

En este caso cada una de las subrutinas contendría el código necesario para controlar todas las barreras y, fundamentalmente, el módulo conocería el secreto (hardware) de todas las barreras. ¿Cuáles son las ventajas de la solución anterior frente a esta otra posibilidad? Son las siguientes:

- Tener un módulo para cada barrera implica que la implementación de cada subrutina será más simple que aquella capaz de manejar todas las barreras.
- Si las barreras son diferentes o cambia una pero no cambia la otra, entonces el módulo que las maneja a todas se hace todavía más complejo y, ante una modificación parcial, hay que "tocar" muy cuidadosamente las porciones del código que deban modificarse (sin que esto impacte negativamente en las porciones que no requieren modificaciones). Además, ante una modificación parcial hay que testear todo el módulo.
- En el peor de los casos si el hardware es idéntico entonces ambos módulos pueden compartir la implementación.

Pero entonces, ¿qué ocurre si son 20, 50 o 1000 barreras o dispositivos similares? La respuesta a esta pregunta la veremos en las secciones 4 y 6.

Continuando con el diseño del sistema para la estación de peaje vemos que, dada su simetría (en todos los carriles hay dispositivos equivalentes), tendremos dos módulos con la misma interfaz para cada carril. Para simplificar la exposición daremos únicamente la descripción 2MIL de los módulos correspondientes al carril Norte.

3.3.2. La impresora

El siguiente módulo oculta el hardware de la impresora de tiques, que es otro de los ítem con probabilidad de cambio considerados.

MODULE	ImpresoraCarrilNorte
IMPORTS	Ticket
EXPORTS	imprimir(i Ticket) inicializar() retiro():Bool
COMMENTS	retiro() se llama para saber si el conductor retiró el tique; inicialmente retorna false, cuando es invocada y retorna true, a la llamada siguiente retorna false.

En este módulo aparece la cláusula IMPORTS que tiene el efecto de hacer accesible a este módulo los servicios de todos los módulos que se listan a continuación (en este caso únicamente el módulo TICKET). La `i` antes del parámetro de imprimir() significa que dicho parámetro es de *entrada*, es decir que la salida de esa subrutina depende del valor que asuma en cada invocación ese parámetro. La definición de TICKET la veremos un poco más adelante.

3.3.3. Medios de pago

El siguiente paso es resolver el problema de los medios de pago. Sin embargo, antes de introducirnos en el diseño de cada módulo conviene prestar atención a la estructura conceptual general que utilizaremos para determinar cuándo el cliente ha pagado, como se muestra en la Figura 1. Según esta Estructura Conceptual los datos fluyen desde las unidades funcionales en contacto con el hardware de los distintos medios de pago, hacia las unidades funcionales que procesan o dan semántica de negocio a esos datos. Por el contrario, son las unidades funcionales superiores las que invocan los servicios provistos por las inferiores. Cada una de estas unidades funcionales normalmente se convertirá en un módulo[6].

El sistema determinará si el conductor pagó el peaje llamando a una subrutina provista por "Recepción pago" de la Figura 1. Esta subrutina, a su vez, llamará iterativamente a todos los medios de pago disponibles. Por otro lado, cada medio de pago interactuará con su hardware y la tabla de precios de peaje para determinar cuánto pagó el conductor y dar el vuelto si corresponde. De esta forma encapsulamos cada decisión de diseño: (a) los módulos inferiores solo interactúan con hardware, (b) los módulos intermedios determinan el monto del pago, dan vuelto o utilizan otros módulos para corroborar el pago (por ejemplo con tarjeta), (c) la tabla de precios solo guarda ese secreto y (d) el módulo superior toma la decisión final sobre el pago que puede involucrar a uno o más medios de pago y diferentes precios según el tipo

[6]La Estructura Conceptual no es exactamente parte de la documentación de diseño de un sistema aunque es útil para comprender a grandes rasgos el problema o las áreas complicadas del problema. Las unidades funcionales no son módulos, sino que suelen descomponerse en uno o más módulos.

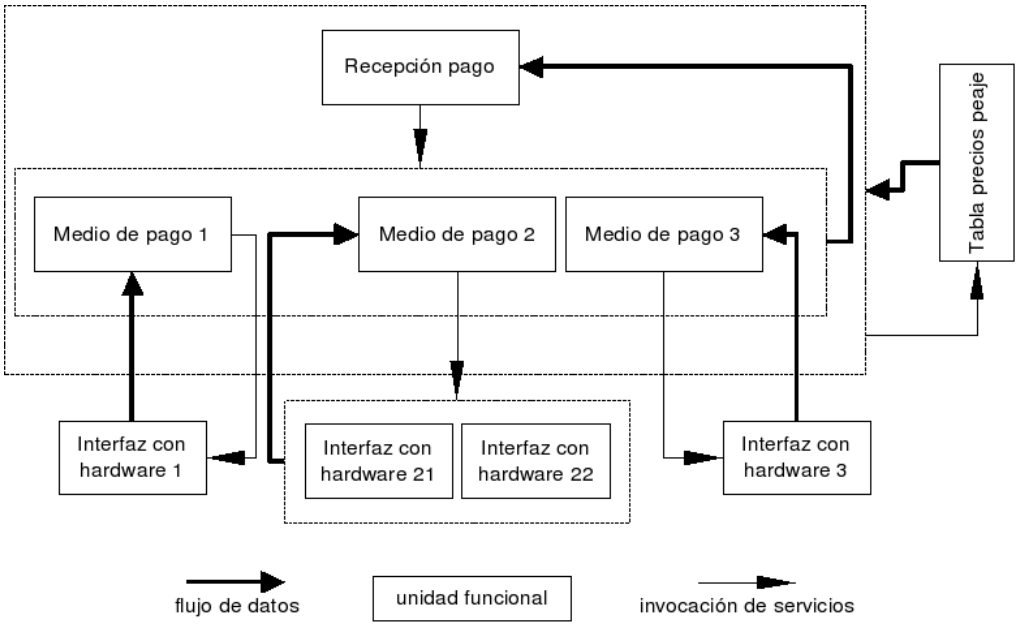


Figura 1: Estructura Conceptual general para los medios de pago. Las "cajas" que agrupan tareas o funciones sirven únicamente para ahorrar flechas.

de vehículo (ya veremos este punto) o las políticas de precios y cobro de la empresa. Poner la lógica de los módulos intermedios en el superior implicaría que este debería ser fuertemente modificado cada vez que se agrega o quita un medio de pago; otras estructuras más simples serían igualmente menos flexibles y/o más costosas de modificar. De esta forma, si aparecen nuevos medios de pago se deben agregar módulos para ocultar el hardware y un módulo que controle y avise que el conductor ha pagado con ese medio de pago, y no se debe modificar ningún módulo existente.

Esta forma de estructurar un sistema, es decir que nuevos requerimientos se implementan con nuevos módulos y no con modificaciones a módulos existentes, es uno de los principios básicos del diseño de software.

Principio de Diseño: Diseños Abiertos y Cerrados

El diseño debe ser tal que nuevas funcionalidades se puedan incorporar en nuevos módulos sin tener que modificar los módulos existentes. Los módulos de un diseño deben estar abiertos *a extensiones, pero* cerrados *a modificaciones.*

Al mismo tiempo podemos observar que esta arquitectura no surge caprichosa o mágicamente sino que se deduce, aplicando la metodología de Parnas, a partir de los ítem con probabilidad de cambio. Por último queremos resaltar que esta estructura responde al concepto de máquinas abstractas que Parnas propone como uno de los mecanismos básicos para lograr un Diseño para el Cambio [10].

En esta instancia del desarrollo del sistema solo debemos diseñar los módulos que conciernen al pago en efectivo utilizando la máquina receptora de dinero. Por lo tanto debemos diseñar

cuatro módulos. Pero ahora cambiamos la estrategia de diseño de *top-down* a *bottom-up*. Entonces, comenzamos por la máquina receptora de dinero. Presentaremos dos diseños: el primero más simple pero parcialmente incorrecto y el segundo algo más complejo pero mejor.

Primer diseño para la máquina receptora de dinero. Si bien este es un diseño simple, no es del todo correcto desde el punto de vista del principio del Diseño para el Cambio. El problema es que en la implementación del módulo `MaquinaMBCarrilNorte` hay llamadas *explícitas* a funciones de la interfaz de un módulo de nivel superior, lo que rompe la estructura de máquinas abstractas que recién mencionamos. Veamos la definición del módulo.

MODULE	`MaquinaMBCarrilNorte`
IMPORTS	`Valor, PagoEfectivoCarrilNorte`
EXPORTS	`esperarEventos()`
	`denominacion():Valor`
	`capacidadCilindro(i Valor):Int`
	`entregarMoneda(i Valor)`
	`inicializar()`

esperarEventos() es la subrutina que espera a que el hardware emita alguna señal (recepción de una moneda o billete, cilindro vacío, bandeja retirada, o insertada). Esta subrutina es semejante a un "manejador de interrupciones": al ser invocada asocia una o más subrutinas privadas del módulo con el hardware de manera que se puedan manejar las interrupciones (para más detalles ver, por ejemplo, [3, páginas 128 a 144]). Las subrutinas privadas son las que contienen las llamadas al módulo de nivel superior para indicarle, por ejemplo, que se ha recibido una moneda de forma tal que este módulo superior pueda llevar el total recibido. Luego, el módulo superior llamará a denominacion() para saber el valor de la moneda o billete recibido. Secuencias de invocaciones como esta implican un acoplamiento mutuo entre ambos niveles de la estructura de la Figura 1, lo que refleja un diseño deficiente.

capacidadCilindro() retorna la capacidad máxima de cada cilindro; notar que con el hardware actual todos los cilindros tienen la misma capacidad y que la máquina no es capaz de informar este dato, pero como asumimos que el hardware puede cambiar incluimos esta función cuya primera implementación será constante e independiente del hardware. Si hubiéramos codificado este dato en el cliente de `MaquinaMBCarrilNorte` un cambio en el hardware de la máquina hubiera impactado en el cliente; esta situación sería producto de no haber aplicado consistentemente la metodología de Parnas. La subrutina entregarMoneda() acciona sobre el hardware de forma tal que entregará una moneda del valor indicado sacándola del cilindro apropiado; si el cilindro está vacío la subrutina tendrá un comportamiento inesperado. Finalmente, inicializar() lleva la máquina a un estado conocido.

Segundo diseño para la máquina receptora de dinero. Uno de los problemas fundamentales del diseño anterior es que rompe la noción de máquina abstracta. Esto se produce dado que existen llamadas *explícitas* en el código de `MaquinaMBCarrilNorte` a subrutinas en la interfaz de un módulo de nivel superior. La forma clásica de evitar este problema es reemplazar las llamadas explícitas por llamadas *implícitas*. Una forma de implementar llamadas implícitas es mediante punteros a función cuyas referencias se definen dinámicamente en tiempo de

ejecución. Concretamente por cada llamada implícita se incluye, en la interfaz del módulo, una subrutina que espera un puntero a función. Normalmente, estas subrutinas se invocan una única vez. Al ser invocadas, lo único que hacen es asignar el valor del puntero recibido a uno de los punteros a función internos. Cada puntero interno es parte del código que maneja una interrupción. Por lo tanto, cuando se produce una interrupción, se invoca a la función apuntada por el puntero. A este mecanismo se lo denomina *callback*. El nombre refiere a que una subrutina le avisa a otra que debe llamarla. Este mecanismo se mejora introduciendo el patrón de diseño Command, como veremos más adelante.

Notar que si bien el efecto neto en tiempo de ejecución de ambos diseños es el mismo, no hay nada en el código fuente de `MaquinaMBCarrilNorte` que refiera a un módulo de nivel superior por lo que este diseño es más reusable que el anterior dado que el acoplamiento en ambos sentidos se ha reemplazado por acoplamiento en un único sentido.

La interfaz del módulo `MaquinaMBCarrilNorte` se muestra más abajo. Las subrutinas funcionan de la siguiente forma. nuevaMB() anuncia que la máquina ha recibido una moneda o billete; el anuncio lo hace llamado a la función apuntada por el puntero que recibe como parámetro. cilindroVacio() comunica que un cilindro de un valor específico se ha vaciado; ese valor se comunica pasando el parámetro en el puntero a función. bandejaRetirada() y bandejaInsertada() comunican sendas acciones sobre la bandeja que contiene los cilindros; se asume que cuando la bandeja es retirada no se puede entregar vuelto y que cuando es reinsertada todos los cilindros están cargados a su capacidad máxima.

MODULE	`MaquinaMBCarrilNorte`
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	`esperarEventos()`
COMMENTS	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.

Los parámetros reales correspondientes a los punteros a función esperados por las distintas subrutinas se asignan durante la inicialización del sistema (por ejemplo al comienzo de la función main() o como consecuencia de llamar a alguna función inicializar() de algún módulo superior, esto se verá más adelante), aunque pueden ser re-asignados posteriormente.

En la Figura 2 se muestra el pseudo-código de una posible implementación de algunas de las subrutinas del módulo `MaquinaMBCarrilNorte`.

Pago en efectivo. Siguiendo la estructura mostrada en la Figura 1 definiremos un módulo que controlará el pago en efectivo (acumulará el total y entregará el vuelto) y de alguna forma

```
void nuevaMB(void* f) {nmb := f;}

void cilindroVacio(void* f(Valor v)) {cv := f;}

void incializar() {
  // inicializar el hardware si hace falta

  params[] := 0;

  // lanzar en otro hilo la subrutina privada esperarEventos()
}

void esperarEventos() {
  while (true) {
    if (params[0]) {
      params[0] := 0;
      case params[1] is {
        1: nmb();          // código 1 --> moneda o billete insertado
        2: cv(params[2]);  // código 2 --> se vació un cilindro
        ...
      }
    }
  }
}

Valor denominacion() { return params[3]; }
```

Figura 2: Código de ejemplo de algunas subrutinas del módulo `MaquinaMBCarrilNorte`. Se pretende mostrar el uso de los *callbacks*; ver también la implementación del módulo `PagoEfectivoCarrilNorte` en la Figura 3. Se asume que `params` es un arreglo que el módulo define para comunicarse con el hardware de la máquina; esta lo escribe con valores relativos a su funcionamiento. Por ejemplo, en `params[0]` escribe un `1` cada vez que se produce un nuevo evento y en `params[1]` el código del evento. El `while (true)...` es a modo ilustrativo; en una implementación real habría una condición de terminación.

comunicará “hacia arriba” que el pago en efectivo a finalizado.

MODULE	PagoEfectivoCarrilNorte
IMPORTS	Valor, MaquinaMBCarrilNorte, Ticket, TablaPreciosCarrilNorte, Monto
EXPORTS	hayNuevaMB() noHayCambioDe(i Valor) noHayCambio() hayCambio() pagoEfectivo():Monto ticket():Ticket inicializar()
COMMENTS	pagoEfectivo() retorna un Monto inválido en tanto no ha finalizado el pago en efectivo y retorna el Monto pagado una vez que se considera que el conductor no entrega más dinero; una vez que retorna un Monto válido a la siguiente invocación retorna uno inválido. Faltan controles para hacer algo útil en caso de que no se pueda dar vuelto y para determinar cuál fue la última moneda o billete insertado.

La subrutina hayNuevaMB() es la que se pasa como parámetro a nuevaMB(); noHayCambioDe() se le pasa como parámetro a cilindroVacio(); noHayCambio() se le pasa a bandejaRetirada(); hayCambio() se le pasa a bandejaInsertada(). (Es importante aclarar que todo esto debería estar documentado más formalmente, ver secciones 10.7 y 11.3.) La subrutina pagoEfectivo() será usada por el módulo de nivel superior en la estructura de la Figura 1. De esta forma, cada vez que el conductor inserta una moneda o billete, nuevaMB() llama a hayNuevaMB() y esta llama a denominacion() para saber el valor de la moneda o billete; hayNuevaMB() actualiza lo que el conductor lleva pagado; cuando hayNuevaMB() determina que el conductor ha pagado lo que debe pagar (utilizando la interfaz de `TablaPrecios`), llama a entregarMoneda() las veces que sea necesario para darle el vuelto; en ese mismo momento cambia el estado del módulo de forma tal que pagoEfectivo() retorne el `Monto` correcto. Si pasado cierto tiempo el conductor no entrega una cantidad suficiente de dinero, se asume que el pago se completará con otro medio de pago y pagoEfecitvo() retornará el monto pagado[7]. Una vez que pagoEfectivo() retorna un `Monto` significativo, ticket() retorna un tique válido.

En la Figura 3 se muestra el pseudo-código de una posible implementación de algunas subrutinas del módulo `PagoEfectivoCarrilNorte`.

La tabla de precios. Los precios de los peajes es otra de las decisiones de diseño que decidimos ocultar. Por lo tanto definimos el siguiente módulo.

[7]Queda por resolver el caso en que el conductor se arrepiente de pagar en efectivo y desea cancelar el pago. Podemos asumir que existe un botón en la máquina que indica pago cancelado y retorna todas las monedas o billetes ingresados hasta el momento. Si esto es así habría que ampliar la interfaz de `MaquinaMBCarrilNorte` y la de `PagoEfectivoCarrilNorte`. Más allá del mecanismo específico es importante tener en cuenta en el diseño el requerimiento ”se cancela el pago en efectivo” de forma tal que luego sea posible implementarlo de varias formas distintas según las especificidades del caso particular de esta u otra estación de peaje.

```
void incializar() {
  nuevaMB(hayNuevaMB());           // configuramos MaquinaMB con los callbacks
  cilindroVacio(noHayCambioDe());
  bandejaRetirada(noHayCambio());
  bandejaInsertada(hayCambio());

  // asumimos un timer que usamos para saber cuándo el conductor
  // no ingresa más monedas; lo inicializamos a 10 u.t.
  settimer(10);

  total := 0;                      // inicialización de variables de estado
  ticketok := false;

  MaquinaMBCarrilNorte.inicializar();
}

void hayNuevaMB() {
    stoptimer();
    ticketok := false;
    total := total + denominacion();
    starttimer();
}

Monto pagoEfectivo() {
  if (timertimeout() && total != 0) {
    total2 := total;
    ticketok := true;
    total := 0;
    darVuelto();             // subrutina privada
    return total2;           // puede no ser lo que debe pagar
                             // pero hace tiempo que no pone una moneda
  }
  else  return error;
}

Ticket ticket() {
  if (ticketok)
    // devolver el ticket con los datos
  else  return error;
}
```

Figura 3: Código de ejemplo de algunas subrutinas del módulo `PagoEfectivoCarrilNorte`. Se pretende mostrar el uso de los *callbacks*.

MODULE	`TablaPreciosCarrilNorte`
IMPORTS	`Monto`
EXPORTS	`debePagar():Monto`
COMMENTS	El módulo debería tener una interfaz más completa para alterar la tabla de valores.

Notar que definimos un módulo por carril lo cual puede ser útil si la empresa decide cobrar más a los vehículos de un carril que a los del otro. Pero la razón es otra: debePagar() retorna lo que el último vehículo detectado en ese carril debe pagar (hasta tanto no se haya retirado el tique). La detección del vehículo será responsabilidad de un módulo que no definiremos porque para la versión del sistema que el cliente solicita no es necesario. De hecho, en esta versión, debePagar() es una función constante respecto del tipo de vehículo y respecto del tiempo.

El ticket. Uno de los ítem de cambio se relaciona con la imposibilidad de saber en este momento del desarrollo del sistema qué datos debe comunicar "hacia arriba" cada medio de pago. Por lo tanto debemos ocultar todas las posibilidades en un módulo específico que denominamos `Ticket`. Como las posibilidades son muchas y desconocidas debemos definir una interfaz muy general. Tenemos a nuestro favor que los datos que se pueden comunicar se pueden representar como cadenas de caracteres. Por lo tanto pensamos en una tabla de pares (*atributo*, *valor*) que cada módulo completará, consultará y modificará según el conocimiento que tenga sobre la transacción, hasta que finalmente llega a la impresora la cual imprime su contenido.

MODULE	`Ticket`

EXPORTS	`agregarAtributo(i String)`
	`agregarValor(i String)`
	`primero()`
	`siguiente()`
	`hayMas():Bool`
	`atributo():String`
	`valor():String`
	`eliminar()`

COMMENTS	eliminar() borrar el par apuntado en ese momento.

Entonces, por ejemplo, el módulo `PagoEfectivoCarrilNorte` puede agregar tres pares `(dinero recibido, 5)`, `(total,1.80)` y `(vuelto,3.20)`, lo que más tarde podría ser impreso por la impresora. Si el pago fuera con tarjeta crédito el módulo equivalente al de pago en efectivo podría agregar todos los pares propios de un pago con tarjeta.

Las subrutinas primero(), siguiente() y hayMas() constituyen un iterador; las subrutinas atributo() y valor() recuperan el atributo y el valor apuntado por el iterador en ese momento.

La decisión final sobre el pago. Ahora toca el turno de definir el módulo de mayor nivel en la estructura de la Figura 1. La responsabilidad de este módulo es determinar si el conductor ha pagado por cualquier medio de pago e informar esto al módulo que ordenará se emita el tique y se suba la barrera en el momento adecuado. El módulo es el siguiente.

MODULE	RecepcionPagoCarrilNorte
IMPORTS	Monto, PagoEfectivoCarrilNorte
EXPORTS	pago() ticket():Ticket inicializar()
PRIVATE	pagoConEfectivo():Bool pagoConTarjetaCredito():Bool pagoConTarjetaDebito():Bool
COMMENTS	Para hacer más interesante esta parte del problema suponemos que se han habilitado más de un medio de pago.

Antes de explicar el módulo veamos la nueva cláusula PRIVATE. Dentro de esta cláusula se listan los recursos o servicios que no están en la interfaz del módulo sino que son divisiones internas del módulo con el fin de darle alguna estructura más manejable. En general no es necesario mencionar esta cláusula (no lo hemos hecho hasta ahora) porque no atañe al diseño propiamente dicho ya que no impacta en la visión externa del módulo.

En la Figura 4 mostramos una versión muy simple del código que correspondería al caso en que la empresa permite pagar cada peaje con un único medio de pago a elección del conductor. Las variables `monto` y `ticket` son variables locales al módulo accesibles para cualquier subrutina del módulo. Como se puede ver, la subrutina `pago()` retorna solo cuando el conductor ha pagado la cantidad esperada usando un medio de pago. Claramente, en una implementación real habría que hacer algo si el conductor "nunca" termina de pagar—en ese caso probablemente `pago()` debería retornar un error. En tanto que en la Figura 5 se muestra la implementación de una política de cobro más flexible pues permite que el conductor pague parcialmente el peaje con diferentes medios de pago. Estos dos ejemplos bastan para mostrar que con este diseño la lógica de pago() se mantiene simple y lo único que hay que hacer para modificar la política de cobro es modificarla.

3.3.4. Control general del carril

Finalmente nos queda definir el módulo que oculte el requisito para levantar y bajar la barrera e imprimir el tique. Claramente este es el módulo que controlará el funcionamiento global del sistema. La subrutina iniciarControl() estará en ejecución permanentemente esperando que algún conductor pague el peaje para imprimir el tique, levantar la barrera, esperar que el conductor retire el tique, esperar cinco segundos y finalmente bajar la barrera. El módulo es el que sigue y en la Figura 6 mostramos como podría ser la implementación de `iniciarControl()`.

MODULE	ControlCarrilNorte
IMPORTS	Ticket, Monto, BarreraCarrilNorte, ImpresoraCarrilNorte, ...
EXPORTS	iniciarControl() inicializar()
COMMENTS	inicializar() llama a las subrutinas de inicialización de otros módulos.

El sistema se pone en funcionamiento desde `main()` el cual llama primero a inicializar() y luego a iniciarControl() en ambos carriles. (No se incluye una forma de detener el sistema; se puede detener matando el proceso desde el sistema operativo; se asume que esto se hace sólo

```
void pago() {
  while ~(pagoConEfectivo()
          || pagoConTarjetaCredito()
          || pagoConTarjetaDebito());
}

bool pagoConEfectivo() {
  if (pagoEfectivo() == debePagar()) {
    ticket := PagoEfectivoCarrilNorte::ticket();
    resultado := true;
  }
  else resultado := false;
  return resultado;
}
```

Figura 4: Código de ejemplo para la subrutina pago() y pagoConEfectivo(). En este caso la política de cobro de la empresa es que se puede pagar con un único medio de pago. Faltaría hacer algo si el conductor no paga o intenta pagar menos.

```
void pago() {
  monto := 0;
  while (monto < debePagar()) {
    pagoConEfectivo();
    if (monto < debePagar()) pagoConTarjetaCredito();
    if (monto < debePagar()) pagoConTarjetaDebito();
  }
}

pagoConEfectivo() {
  monto := monto + pagoEfectivo();
  if (monto == debePagar()) {
    ticket_tmp := PagoEfectivoCarrilNorte::ticket();
    ticket := completar_ticket(ticket_tmp);
  }
}
```

Figura 5: Esta porción de código implementa una política en la cual se le permite al conductor pagar cada peaje con varios medios de pago. La función `completar_ticket()` va completando el `Ticket` retornado por cada medio de pago de forma tal que luego se pueda, si se desea, imprimir un tique que indica cuánto se pagó por cada medio de pago y otros datos relevantes.

```
void iniciarControl() {
  while (true) {
    pago();
    subir();
    imprimir(RecepcionPagoCarrilNorte::ticket());
    esperarRetiroTicket();    //Falta definir módulo
    bajarBarrera();           //Falta definir módulo
  }
}
```

Figura 6: Código de ejemplo para la subrutina iniciarControl(). Las subrutinas esperarRetiroTicket() y bajarBarrera() están en módulos que no se muestran en este ejemplo.

en situaciones de emergencia.)

Ejercicios

Ejercicio 6. Al comienzo del ejemplo dijimos que no convenía definir el módulo `Barreras` para ocultar el hardware de todas ellas, sino que lo mejor es tener un módulo para cada una. También dijimos que si el hardware de todas las barreras es idéntico, los módulos pueden compartir la implementación. Explique cómo podría hacer eso.

Ejercicio 7. Diseñar y documentar el o los módulos donde deberían estar las subrutinas esperarRetiroTicket() y bajarBarrera().

Ejercicio 8. Mostrar una implementación sencilla de las funciones esperarRetiroTicket() y bajarBarrera().

Ejercicio 9. Explique por qué no conviene poner la lógica de las funciones mencionadas en los problemas anteriores en los módulos `ImpresoraCarrilNorte` y `BarreraCarrilNorte`, respectivamente.

Ejercicio 10. ¿Cuántos ítem de cambio oculta el módulo `ImpresoraCarrilNorte`? ¿La impresora imprime tiques o líneas o caracteres?

Ejercicio 11. ¿Cuántos ítem de cambio oculta el módulo `PagoEfectivoCarrilNorte`? El algoritmo para dar el vuelto, ¿no se debe ajustar si cambia la denominación de las monedas o si una nueva máquina tiene menos o más cilindros? ¿Es un error de diseño haber incluido este algoritmo en este módulo? ¿Es uno de los ítem de cambio considerados?

Ejercicio 12. Compruebe que llegados a este punto el diseño incorpora correctamente todos los ítem con probabilidad de cambio que se analizaron.

Ejercicio 13. Incluya en el diseño el requerimiento "el conductor puede cancelar el pago en efectivo presionando un botón de la máquina receptora de dinero".

4. Superioridad y limitaciones del DBOI

Cualquiera que haya leido algún libro sobre DOO encontrará una semejanza muy grande con el DBOI. Conceptos tales como interfaz, implementación, abstracción y encapsulamiento

son utilizados en los libros clásicos sobre DOO. Sin embargo, existe una diferencia metodológica muy importante entre el DBOI y el DOO como se lo define habitualmente. Según la mayor parte de la literatura dedicada al DOO los módulos se determinan por la existencia de entidades físicas en el sistema (como sensores) o son conjurados a partir de la experiencia y/o intuición del diseñador [1]. Por el contrario, según la metodología propuesta por Parnas y basada en el POI los módulos se determinan como consecuencia de ocultar los ítem con alta probabilidad de cambio. Esto no es un detalle menor: cómo obtener los módulos del sistema **es** diseñar, por lo tanto un método de diseño que no indique una forma rigurosa para obtener lo módulos está lejos de ser un buen método de diseño.

Sin embargo el DBOI tiene limitaciones. En realidad, no es del todo correcto hablar de limitaciones del DBOI, mejor sería decir que, como lo hemos planteado, esta forma de diseño tiene algunos inconvenientes que pueden ser fácilmente evitados. Nos referimos, concretamente, a la falta de una forma simple de generar instancias de módulos; lo que es semejante a pensar los módulos como *tipos*. Por ejemplo, si tenemos que diseñar un sistema que recibirá señales de 20 sensores iguales (cf. a la cantidad de carriles mencionada en el ejemplo de la sección 3), tendríamos dos posibilidades:

- Incluir en todas las subrutinas de la interfaz del módulo correspondiente (`Sensores`) un parámetro para indicar el sensor sobre el cuál se debe aplicar la subrutina
- Generar 20 módulos idénticos

Ninguna de las dos alternativas tiene la elegancia y simplicidad que podría tenerse. Además, si deseáramos que otros módulos reciban un "sensor" como parámetro, no tendríamos forma de expresarlo. Por lo tanto, sería conveniente contar con una sintaxis y una semántica que nos permitieran tratar a cada módulo como un *tipo* del cual se pueden generar instancias y a estas usarlas como parámetros en subrutinas. En otras palabras, es conveniente acercarnos a las ideas más recientes sobre Tipos Abstractos de Datos (TAD), lenguajes de programación orientados a objetos, etc.

5. Diseño basado en tipos abstractos de datos (DTAD)

La única diferencia entre el DBOI y el Diseño basado en Tipos Abstractos de Datos (DTAD) es que en este último se asume que cada módulo del diseño es, genera o define un *tipo*. Al ser cada módulo un tipo entonces es posible definir variables y parámetros que tengan ese tipo. Por lo demás, el DTAD es idéntico al DBOI: su metodología, notación, conceptos, etc.

Definición 5 (Tipo)**.** *En el contexto del DTAD, un tipo es un módulo.*

Concretamente, si A es un tipo de un DTAD, entonces $a : A$ significa que a es una instancia de A. Si a es una instancia de A y f es una subrutina en la interfaz de A, entonces $a.f(\dots)$ es lo mismo que $f(a, \dots)$. Además, si b es otra instancia de A (es decir tiene un nombre diferente al de a) entonces $a.f$ no tiene necesariamente el mismo valor o efecto que $b.f$. Esto se interpreta diciendo que cada instancia de A tiene su propio *estado* y que el efecto o resultado de cualquier subrutina en la interfaz de A puede depender del estado de la instancia sobre la cual se aplica.

5.1. Breve comentario sobre la estación de peaje en DTAD

Recordemos que en la sección 3 dijimos que:

1. Queríamos ocultar la cantidad de carriles de la estación.
2. Teníamos que definir dos módulos (`Norte` y `Sur`) iguales para cada módulo que generáramos.

Al pensar en cada módulo como un tipo, ahora es suficiente con definir, por ejemplo:

MODULE	`Barrera`
IMPORTS	`Carril`
EXPORTS	`subir()`
	`bajar()`
	`miCarril(i Carril)`
	`inicializar()`
COMMENTS	miCarril() enlaza cada instancia del módulo con el dispositivo de hardware que opera el carril que se le pase como parámetro.

La subrutina miCarril() asociará cada instancia del tipo con el carril sobre la cual opera. Esto es fundamental en tipos como `Barrera` donde la implementación debe interactuar con un dispositivo físico particular. En efecto, en tiempo de ejecución se creará una instancia de `Barrea` por cada carril (por ejemplo `norte` y `sur`). Pero el código de cada instancia es el mismo que el de las demás, por lo que `norte.subir()` y `sur.subir()` ejecutan las "mismas" sentencias, entonces, ¿cómo logramos que `norte.subir()` suba la barrera del carril norte y `sur.subir()` la del carril sur?[8] Se logra en dos pasos: primero, el programador de `Barrera` parametriza el código de todas las subrutinas con el puerto o dispositivo sobre el cual tienen que actuar; segundo, programa miCarril() de forma tal que se fije ese parámetro. El código sería semejante al que se muestra en la Figura 7. Las variables `carril`, `fdp` y `puerto` son *atributos* privados o *variables de estado* del módulo por lo que solo pueden ser accedidos por las subrutinas del tipo. El valor de la variable `carril` se usa para determinar sobre qué dispositivo de hardware actuará el código.

Notar que este diseño sirve solo si todas las barreras tienen las siguientes características: se comunican a través de puertos USB, el puerto se debe abrir en modo `WRITE`, para subirlas hay que enviar un uno y para bajarlas un cero.

5.2. Evitando una mala idea

Imagine que cada carril tiene una barrera diferente por lo que no se puede usar el código de la Figura 7 para manejar ambas. Muchos programadores (y diseñadores) estarían tentados en hacer algo como lo que se muestra en la Figura 8. Incluso el código podría ser más complejo si, por ejemplo, fuese necesario escribir un uno para subir la barrera conectada al puerto `com0` y otra cosa para hacerlo en la otra barrera. Sea como fuere es una muy mala idea. No lo es hoy, ni lo es para un módulo tan pequeño como `Barrera`, pero lo será cuando el programador que lo escribió no trabaje más con nosotros, lo será cuando tengamos que mantener ese código, lo será si queremos vender el sistema a una empresa que tiene 10 tipos de barreras diferentes, lo será cuando vendamos el sistema a 10 empresas diferentes que usan distintos modelos de barreras (es decir va a ser un problema cuando el sistema se transforme en *negocio*). Con el tiempo el

[8] Ponemos "mismas" entre comillas porque si bien en ambos casos se ejecuta el mismo algoritmo, puede haber dos copias idénticas del mismo programa, con lo cual en un sentido no son exactamente las mismas.

```
miCarril(Carril c) {
  carril := c;
  puerto := strconcat("/dev/usb",carril);
}

inicializar() {fdp := open(puerto, "WRITE");}

subir()       {write(fdp, "1");}

bajar()       {write(fdp, "0");}

main() {
  Barrera norte, sur;
  norte.miCarril("0");
  sur.miCarril("1");
  norte.subir();
  sur.bajar();
}
```

Figura 7: Pseudo-código del tipo `Barrera`

código de este módulo aumentará en tamaño y en complejidad, habrá estructuras `if` por todos lados y será cada vez más complejo entenderlo para poder seguir modificándolo.

¿Cuál es la solución entonces? Eliminar el tipo `Barrera` y definir dos nuevos tipos, por ejemplo `BarreraACME` y `BarreraEMCA`, con la misma interfaz pero diferente implementación[9]. Observar que necesitamos dos tipos porque cada barrera es de una marca o tipo diferente y no porque son dos barreras. También notar que los dos tipos no aparecen caprichosamente sino como consecuencia de seguir al pie de la letra la metodología de Parnas: ocultamos el hardware de *cada* barrera en un *único* módulo; *un* ítem de cambio, *un* módulo.

MODULE	`BarreraACME`
IMPORTS	`Carril`
EXPORTS	`subir()`
	`bajar()`
	`miCarril(i Carril)`
	`inicializar()`
COMMENTS	Esta podría ser la barrera para el carril norte.

[9]Si hubiese más marcas o tipos de barreras habría que definir más tipos, todos con la misma interfaz.

```
miCarril(Carril c) {
  carril := c;
  if (carril = "0")
    puerto := "/dev/usb0";
  else
    puerto := "/dev/com0";
}

inicializar() {fdp := open(puerto, "WRITE");}

subir()       {write(fdp, "1");}

bajar()       {write(fdp, "0");}
```

Figura 8: Un error muy común de programadores y diseñadores cuando hay que introducir cambios.

```
bajarBarrera(BarreraACME b) {..............}

bajarBarrera(BarreraEMCA b) {..............}
```

Figura 9: Código de un cliente de las barreras. No podemos usar la misma función bajarBarrera() porque `BarreraACME` y `BarreraEMCA` son dos tipos diferentes.

MODULE	`BarreraEMCA`
IMPORTS	`Carril`
EXPORTS	`subir()`
	`bajar()`
	`miCarril(i Carril)`
	`inicializar()`
COMMENTS	Esta podría ser la barrera para el carril sur.

Sin embargo, este diseño no es el óptimo porque complica el código de los clientes que deben usar las instancias de los tipos de las barreras. Por ejemplo, la Figura 9, muestra un esqueleto del código que corresponde al módulo que debe esperar 5 segundos después de que el conductor retiró el tique para bajar la barrera[10]. El código de los clientes se complica por la misma razón que el código de los dos módulos se simplifica: `BarreraACME` y `BarreraEMCA` son dos tipos por lo que una función que espere un parámetro de uno de los tipos no puede recibir un parámetro del otro tipo.

¿Cuál es la solución entonces? ¿Tener un único tipo `Barrera` y complicarlo tanto como lo vaya exigiendo la evolución del sistema? ¿Tener diferentes tipos para los diferentes tipos de barreras y complicar los clientes que deben usar las barreras? La idea es tener diferentes tipos pero no tener que complicar los clientes, sin embargo para lograrlo necesitamos pasar al Diseño Orientado a Objetos.

[10]Corresponde a un módulo que debe diseñar el lector en el ejercicio 7.

Ejercicios

Ejercicio 14. Agregue a los ítem de cambio de la sección 3.2 el siguiente:

- Un carril puede no tener la misma clase de componentes de hardware que los otros. Por ejemplo, un carril puede no tener una máquina receptora de monedas pero puede tener el hardware para leer obleas precargadas, en tanto que otro carril puede tener ambas cosas o no tener ninguna.

Rediseñe el sistema en términos de DTAD de manera tal de tener en cuenta este nuevo ítem con probabilidad de cambio.

Ejercicio 15. La función `main()` de la Figura 7 es cliente de `Barrera`, pero como tal está cometiendo un error. Encuéntrelo y repárelo.

Ejercicio 16. Detalle un poco más el código de las dos funciones de la Figura 9. ¿Hay alguna diferencia importante entre ellos? ¿Dónde están las diferencias? ¿Podría usar el mismo código para instancias de ambos tipos de barrera? Investigue cómo el sistema de archivos virtual (VFS) de Linux es capaz de manejar diferentes sistemas de archivos físicos como EXT3, FAT, NTFS, etc., ver por ejemplo [3]. ¿Podría utilizar la misma técnica para solucionar el problema de la duplicación del código de los clientes de las barreras?

6. Diseño orientado a objetos (DOO)

El problema de la duplicación del código de los clientes así como también otros problemas (menores) de diseño pueden resolverse aplicando el concepto de *herencia* desarrollado con el correr de los años por la comunidad de profesionales y científicos dedicados al diseño y programación orientados a objetos.

Definición 6 (Herencia (de interfaces))**.** *Sean A y B dos tipos de un DTAD. Decimos que B es un heredero de A si toda subrutina de la interfaz de A es una subrutina de la interfaz de B.*

Si B es un heredero de A, f es una subrutina en la interfaz de A y $b : B$ entonces $b.f$ está definido. Esto trae como corolario que si g es una subrutina cualquiera tal que uno de sus parámetros formales es de tipo A, entonces también aceptará un parámetro real de tipo B (porque se supone que externamente un tipo (módulo) queda definido por su interfaz, por lo que para un componente externo dos instancias con la misma interfaz deberían poder ser utilizadas de la misma forma aunque sean de tipos diferentes).

Definición 7 (Diseño Orientado a Objetos (DOO))**.** *Un DTAD en el cual se utiliza el concepto de herencia definido en la Definición 6, pasa a ser un Diseño Orientado a Objetos (DOO).*

Definición 8 (Objeto)**.** *En el contexto del DOO, un objeto es una instancia de un tipo (módulo).*

Definición 9 (Supertipo y subtipo)**.** *Si A y B son tipos de un DOO y B es un heredero de A decimos que B es un subtipo de A y que A es un supertipo de B. Si $A_1, \ldots, A_n$ son tipos de un DOO tales que A_i es un heredero de A_{i-1} para todo i, entonces decimos que A_i es un subtipo (supertipo) de A_j para todo $i > (<) j$.*

Definición 10 (Método)**.** *En el contexto del DOO, un método es una subrutina en la interfaz de un tipo.*

Es posible que en la literatura se den otras definiciones de herencia. A esta forma de herencia se la denomina *herencia de interfaces* en contraposición con la *herencia de clases*. La herencia de clases permite que la implementación de un subtipo acceda a la implementación de cualquiera de sus supertipos. Por el contrario, la herencia de interfaces describe cuándo se puede usar un objeto de un tipo en lugar de un objeto de otro tipo [4]. Claramente la noción de herencia de interfaces está mucho más relacionada con el concepto de diseño que venimos desarrollando porque trabaja sobre la visión externa de cada componente, en tanto que la otra noción lo hace sobre la implementación, que no es parte del diseño.

La Definición 6 es la única que admitiremos porque las diferentes formas de herencia de clases permiten violar el POI y por lo tanto entran en contradicción conceptual con todo lo antedicho. En otras palabras, según nuestra definición de herencia, si B es un heredero de A la modificación de la implementación de A no afecta a B. Al mismo tiempo la Definición 6 conserva dos características fundamentales asociadas al concepto de herencia:

- Los herederos pueden cambiar la definición de las subrutinas de su interfaz.
- Si B es un heredero de A y g es una subrutina cualquiera tal que uno de sus parámetros formales es de tipo A, entonces también aceptará un parámetro real de tipo B.

Estas propiedades permiten que los herederos redefinan[11] sus métodos y que los clientes que esperan objetos de un tipo puedan recibir objetos de sus subtipos sin notarlo. Esta definición de herencia enfatiza la visión externa de los módulos por encima de sus posibles implementaciones: dos módulos que comparten una parte de su interfaz son indistinguibles para los clientes que solo acceden a la parte en común, sea cual fuere la implementación de ambos. Por esta razón es que se soluciona el problema de la duplicación del código de los clientes (ver más detalles en la sección 6.1). Esta propiedad del DOO y de la herencia de interfaces responden al siguiente principio de diseño.

Principio de Diseño: Principio de sustitución de Liskov

Si S es un subtipo de T, entonces los objetos de tipo T pueden ser sustituidos por objetos de tipo S sin alterar ninguna de las propiedades importantes del programa [8].

Por lo general la mejor forma de aplicar la herencia es la siguiente [4, 5, 11]:

1. Se considera un cierto ítem con alta probabilidad de cambio.
2. Se define un tipo que lo oculta, pero que en general no será implementado; sólo provee una interfaz. Este tipo se implementa si no tiene herederos; ver el paso siguiente.
3. Para las variantes del ítem de cambio se definen herederos del tipo que por lo general tienen la misma interfaz, es decir ni si quiera agregan subrutinas.
4. Se implementan los herederos.
5. Los clientes de todos estos tipos se definen únicamente en términos del supertipo; es decir esperan parámetros o definen variables cuyo tipo es el del supertipo.

Esta forma de utilizar la herencia se resume en el siguiente principio de diseño [4].

[11] Redefinir es sinónimo de reimplementar.

Principio de Diseño: Programación Orientada a Objetos (POO)

Programe para una interfaz, no para una implementación. Es decir, no se deben declarar las variables con el tipo de los herederos sino con el tipo de los supertipos.

La definición de herencia que elegimos junto a la forma en que debería utilizarse tienen enormes ventajas tanto a nivel de diseño como de implementación pues [4]:

1. Los clientes no tienen que conocer los tipos específicos de los objetos que usan, basta con que estos adhieran a la interfaz que esperan los clientes.
2. Los clientes desconocen la implementación de dichos objetos; sólo conocen las interfaces de los supertipos.
3. Reduce de manera significativa las dependencias de implementación entre subsistemas.

Sin embargo, una desventaja de la herencia de interfaces es que se deben reimplementar todos los métodos en todos los herederos; aunque al combinarla adecuadamente con composición de objetos este problema se minimiza (ver más en la sección 6.3).

En 2MIL la herencia se expresa de la siguiente forma:

MODULE	B INHERITS FROM A
EXPORTS	*las subrutinas que no estén en A* *no es necesario declarar las restantes de A*

6.1. Un ejemplo de aplicación de la herencia

El problema de la duplicación (o multiplicación, en el peor de los casos) del código de los clientes, mencionado en la sección 5.2, se resuelve utilizando herencia. Para el caso de las barreras definimos los dos tipos `BarreraACME` y `BarreraEMCA` pero como subtipos del tipo `Barrera`. En 2MIL las interfaces se documentan de la siguiente forma.

MODULE	`Barrera`
IMPORTS	`Carril`

EXPORTS	`subir()`
	`bajar()`
	`miCarril(i Carril)`
	`inicializar()`

COMMENTS	Ninguno de los métodos de este tipo se implementa; el tipo se utiliza únicamente para definir la interfaz común a todas las marcas o modelos de barreras.

MODULE	`BarreraACME` INHERITS FROM `Barrera`

MODULE	`BarreraEMCA` INHERITS FROM `Barrera`

```
bajarBarrera(Barrera b) {
  Timer t;
  t.set(5);
  t.onTimeoutCall(b.bajar())
  t.start();
}
```

Figura 10: El código de un cliente de `Barrera` no debe hacer referencia a sus herederos.

Esta descripción dice que todos los métodos de los herederos deben reimplementarse. Notar que este diseño preserva todas las propiedades del DTAD pero permite que el diseño y el código de los clientes se simplifique. En particular el pseudo-código que debe bajar la barrera luego de 5 segundos de retirado el tique se muestra en la Figura 10. Notar que el parámetro formal de la subrutina es de tipo `Barrera` y no de ninguno de sus herederos.

El diseño anterior corresponde a un punto del desarrollo en el cual *ya* se sabe que hay dos marcas o modelos de barreras diferentes. Pero supongamos que al comienzo del desarrollo hay dos barreras idénticas y que luego de que el sistema fue desplegado aparece el segundo modelo de barrera, ¿cómo se pasa del primer diseño/implementación al segundo? En el diseño inicial: se define el tipo `Barrera` igual al que está más arriba solo que esta vez se implementan todos sus métodos y los clientes de este tipo se diseñan e implementan usando variables de tipo `Barrera` únicamente. Cuando se cambia la marca o el modelo de una de las barreras: se definen los dos subtipos `BarreraACME` y `BarreraEMCA` como herederos de `Barrera`, se mueve la implementación de `Barrera` a uno de los dos herederos según corresponda, se implementan los métodos del otro heredero y los clientes se dejan sin cambios (`Barrera` queda sin implementación). Resalta inmediatamente el bajo costo que tiene la modificación (dado que la mayor parte del código existente sigue sirviendo), cómo se preserva la integridad conceptual del diseño original y cómo queda abierta la puerta para realizar nuevos cambios sin perder la calidad.

6.2. Aplicabilidad y limitaciones de la herencia

Así como los libros "clásicos" sobre DOO en general no dan una técnica rigurosa, cuantificable y disciplinada para la obtención de los módulos del diseño, también hacen un uso abusivo de la herencia poniéndola, muchas veces, en el centro de la metodología de diseño cuando en realidad tiene un papel secundario y limitado. La herencia por sí sola, cualquiera sea su definición, no es el núcleo de un buen diseño. Aplicar la herencia como mecanismo primordial para obtener un diseño suele dar como resultado un diseño estático, difícil de comprender, mantener y modificar. Este efecto empeora considerablemente si se consideran variantes de la herencia de clases pues estas ponen el foco en la implementación y no en el diseño (considerado como visión externa de cada componente).

Reiteramos: el criterio central para descomponer un sistema en partes (diseñar) es el propuesto por Parnas, reseñado en la sección 2.1 y ejemplificado en la sección 3. Los conceptos que agregan el DTAD y el DOO son mejoras de segundo nivel, no constituyen el núcleo de la metodología.

El uso abusivo de la herencia tiende a generar malos diseños. En efecto, la herencia es un concepto estático porque las relaciones de herencia quedan congeladas en tiempo de compilación. No se pueden alterar las relaciones de herencia dinámicamente. Por lo tanto, un diseño

```
BarreraACME.miCarril(Carril c) {
  carril := c;
  puerto := strconcat("/dev/usb",carril);
}

BarreraEMCA.miCarril(Carril c) {
  carril := c;
  puerto := strconcat("/dev/com",carril);
}
```

Figura 11: Implementación de miCarrill() para cada modelo de barrera.

fuertemente basado en la herencia tenderá a generar un sistema no muy flexible en tiempo de ejecución. Por otro lado, un uso exagerado de la herencia da lugar a jerarquías hereditarias de varios niveles lo que resulta difícil de comprender y mantener y, por lo general, implica que los módulos no ocultarán un único ítem con alta probabilidad de cambio sino varios. Nuevamente, si se utilizan variantes de la herencia de clases todo este panorama empeora porque, por ejemplo, un cambio en la implementación de un supertipo incide en la implementación de todos sus subtipos.

La herencia debe utilizarse en su justa medida, no es *el* concepto del DOO ni es *la* bala de plata del diseño. La herencia se complementa muy bien con la *composición de objetos*.

6.3. Composición de objetos

Una de las razones por la que los programadores prefieren herencia de clases es que permite reutilizar código, mientras que la herencia de interfaces permite reutilizar tipos. Nosotros creemos que lo que se gana al usar herencia de clases se pierde en cuanto hay que hacer un cambio al sistema. Ya hemos analizado que los mayores costos se presentan durante el mantenimiento del sistema y que la fase de programación es o debería ser la más corta y por lo tanto la más económica. La herencia de clases permite ahorrar durante la programación pero puede ser un verdadero problema durante el mantenimiento.

Veamos entonces una forma alternativa de reutilizar código usando la herencia de interfaces. Supongamos que entre la implementación de `BarreraACME` y la de `BarreraEMCA` la única diferencia es en el método miCarril(), el cual se implementa, para cada módulo, como se muestra en la Figura 11. Allí se puede apreciar que la única diferencia es que se usa un puerto `usb` o `com`. Si la implementación de todos los otros métodos es la misma para ambos modelos de barrera sería molesto tener que mantener dos copias de exáctamente el mismo código.

El código que es común a ambos modelos de barrera puede verse como un único ítem de cambio. Por lo tanto podemos ponerlo en un módulo diferente que podemos llamar `BarreraImp`. La interfaz de este nuevo módulo puede ser similar a la de `Barrera` pero no tiene por qué ser igual, como se muestra a continuación. En la Figura 12 se puede ver la implementación de los métodos de este nuevo módulo.

```
inicializarImp(String p) {fdp := open(p, "WRITE");}

subirImp()               {write(fdp, "1");}

bajarImp()               {write(fdp, "0");}
```

Figura 12: Implementación de los métodos del módulo `BarreraImp`.

```
inicializar() {bImp.inicializarImp(puerto);}

subir()       {bImp.subirImp();}

bajar()       {bImp.bajarImp();}
```

Figura 13: Implementación de los otros métodos de `BarreraACME` y la de `BarreraEMCA`.

MODULE	`BarreraImp`

EXPORTS	`subirImp()`
	`bajarImp()`
	`inicializarImp(i String)`

COMMENTS	Notar que no se exporta el método miCarril().

Lo que nos queda por hacer es que las implementaciones de `BarreraEMCA` y `BarreraACME` sean capaces de usar la implementación provista por `BarreraImp`. Sin embargo, esto es muy fácil de hacer pues lo único que necesitamos es que `BarreraEMCA` y `BarreraACME` tengan acceso a un objeto de tipo `BarreraImp`, lo cual se logra declarando una variable de tal tipo como parte del estado de `BarreraEMCA` y `BarreraACME`. Entonces, cada vez que se cree un objeto `BarreraACME` o `BarreraEMCA` se creará un nuevo objeto `BarreraImp` que vivirá dentro (o formará parte del estado) del primero. Por este motivo se dice que un objeto `BarreraACME` o `BarreraEMCA` está *compuesto* por un un objeto `BarreraImp`. En este caso la implementación de los restantes métodos de `BarreraACME` y la de `BarreraEMCA` es la que se muestra en la Figura 13, donde `bImp` es el objeto de tipo `BarreraImp`. Como puede verse lo único que hacen estos métodos es *redirigir las peticiones de servicio* hacia la instancia de `BarreraImp`. Si en algún momento las implementaciones de uno o más de estos métodos difieren para ambos tipos de barrera, lo único que debe hacerse es modificarlos no redirigiendo las peticiones hacia `bImp`. En la Figura 14 se representan gráficamente las relaciones entre los cuatro módulos.

Esta forma de implementar la funcionalidad de las barreras se denomina *composición de objetos* [4]. En general, la composición de objetos se logra haciendo que el estado o el comportamiento de objetos de ciertos tipos dependan del estado o del comportamiento de objetos de otros tipos[12]. Esta dependencia se logra haciendo que un tipo declare variables de otro tipo o que mantenga referencias a objetos del otro tipo. Pasar un objeto como parámetro a un método de otro tipo no necesariamente es composición; lo es, sólo si el parámetro se convierte en parte del estado del objeto que lo recibe.

Al relacionar los tipos mediante composición estos no tienen una relación estática, sino que

[12]Booch en [2] denomina *agregación* a lo que nosotros denominamos composición.

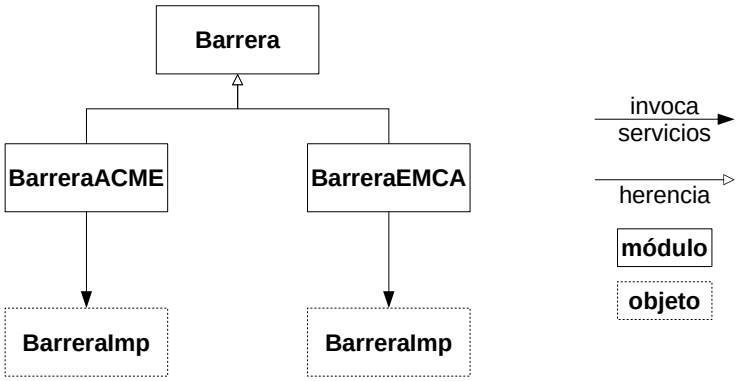


Figura 14: Relaciones de herencia y composición para las barreras.

sus objetos pueden mantener relaciones dinámicas (es decir en tiempo de ejecución). En otras palabras, la composición de objetos es una relación dinámica y no estática como la herencia (es decir que las relaciones de herencia son inmodificables luego de la compilación). O sea que la composición de objetos es una relación entre *objetos* y no entre módulos o tipos como lo es la relación de herencia. Esto implica que la composición de objetos permite que *algunos* objetos de un tipo puedan relacionarse en *algún momento* con objetos de otros tipos. En tanto que la relación de herencia entre los mismos tipos solo permite que *todos* los objetos de un tipo estén *siempre* relacionados con los del otro tipo.

La aplicación del concepto de composición de objetos nos lleva a otro principio de diseño.

Principio de Diseño: Composición de objetos

Favorecer la composición de objetos frente a la herencia (de clases) [4].

Es decir que se debe privilegiar el uso de composición de objetos como mecanismo para compartir código por sobre la herencia de clases.

Las ventajas de la composición frente a la herencia de clases son:

1. se logran diseños mucho más flexibles, sobre todo en tiempo de ejecución;
2. se preserva el encapsulamiento de cada módulo, no se rompe el principio de ocultación de información;
3. requiere interfaces cuidadosamente definidas lo que implica tomar el diseño en serio;
4. las jerarquías de herencia tienden a ser pequeñas y manejables

En tanto que la principal desventaja es que el sistema estará altamente parametrizado y su comportamiento dependerá de las relaciones dinámicas entre objetos de diversos tipos en lugar de estar definido estáticamente en módulos. Esto hace que los diseños sean más difíciles de comprender que si estuvieran regidos por relaciones estáticas. En particular, ciertos análisis estáticos de código no son posibles si el diseño está fuertemente basado en composición de objetos, en lugar de herencia.

El balance adecuado se logra combinando composición de objetos con herencia de interfaces, como es recurrente en los patrones de diseño [4].

Finalmente vale la pena hacer la siguiente observación. Imagine por un momento que la implementación de miCarril() es la mostrada en la Figura 8 (es decir, se implementa la diferencia entre los modelos de barrera en un único módulo mediante una sentencia condicional). Ahora supongamos que una empresa de peaje tiene varios carriles pero todos con el mismo modelo de barrera. Entonces al entregarle la implementación de la Figura 8 le estaríamos entregando más código del necesario. En cambio si la implementación se divide en los módulos `Barrera`, `BarreraACME`, `BarreraEMCA` y `BarreraImp`, es posible entregar el código necesario y nada más. Obviamente ante una diferencia tan pequeña todo este diseño parece una pérdida de tiempo para terminar entregando una línea de código menos. Sin embargo, se debería extrapolar esta situación a un sistema de porte industrial antes de descartar este tipo de diseños.

Ejercicios

Ejercicio 17. Investigue y clasifique las formas de herencia que soportan C++, Java y Eiffel.

Ejercicio 18. Explique cómo y cuándo se crea un objeto `BarreraACME` o `BarreraEMCA` según corresponda. Muestre le pseudo-código correspondiente.

Ejercicio 19. Considere el siguiente ítem de cambio y rediseñe el sistema para la estación de peaje.

- Es posible que el algoritmo general de procesamiento no sea el que se pide ahora. Es decir, no necesariamente primero se le cobrará al conductor, luego se imprimirá el tique, luego se levantará la barrera, se esperarán 5 segundos y finalmente se la bajará. Puede ocurrir que se decida no cobrar o que primero se levante la barrera y luego se imprima el tique, etc. Más aun, puede ocurrir que la empresa quiera modificar este comportamiento dinámicamente sin tener que parar y reiniciar el sistema.

Ejercicio 20. Muestre cómo y cuándo deberían crearse las instancias `bImp` que usan los módulos `BarreraACME` y `BarreraEMCA`.

Ejercicio 21. Suponga que otra empresa de peaje desea contar la cantidad de veces que se suben algunas barreras (de cualquier tipo). Muestre el diseño y el pseudo-código. ¿Y si esta nueva funcionalidad se debe poder activar y desactivar en tiempo de ejecución? ¿Y si otra empresa quiere contar la cantidad de veces que se bajan algunas barreras? ¿Y si una tercera empresa quiere contar las subidas y bajadas de algunas barreras, las subidas de otras, las bajadas de otras y todo modificable en tiempo de ejecución? Muestre el diseño y el pseudo-código.

Ejercicio 22. Liste los módulos que deberían compilarse para generar un programa que solo controle barreras de marca ACME según el diseño basado en composición de objetos.

7. Algunos tópicos complementarios

En esta sección veremos algunos temas que no son esenciales para definir un buen diseño pero que pueden ayudar a mejorarlo o entender algunas cuestiones más o menos generales.

7.1. Efectos laterales

Se dice que un método o un objeto tiene o produce un *efecto lateral* cuando tiene una *interacción observable* con el *mundo exterior*. En este caso el mundo exterior es cualquier software

no orientado a objetos. Es decir que un objeto, a través de uno o más de sus métodos, produce un efecto lateral cuando debe interactuar con un programa que no puede hacerlo por medio de objetos. Como este otro programa no reconoce objetos (y por consiguiente tampoco sus tipos), el programa orientado a objetos deberá convertir objetos en no-objetos (cosas no orientadas a objetos, entes no tipados, o con tipos fuera del sistema de tipos del programa y del lenguaje de programación) o viceversa.

Dado que los sistemas operativos, los servicios de red, los motores de bases de datos relacionales, los manejadores de dispositivos, etc. rara vez son orientados a objetos, la mayoría de los programas orientados a objetos tienen una gran cantidad de efectos laterales. En otras palabras, gran parte del diseño y del código de un programa orientado a objetos está relacionado de una u otra forma con efectos laterales. Es decir que gran parte del código no trabaja con objetos o debe hacer conversiones en ambos sentidos entre objetos y no-objetos. En consecuencia, los efectos laterales *contaminan* el diseño pues incluyen código que no puede respetar el sistema de tipos ya que el entorno con el cual interactúa tampoco lo hace.

Algunos efectos laterales que aparecen en casi todos los programas útiles son:

- Procesar la entrada del usuario.

  En general el usuario se comunica con el programa por medio de cadenas de caracteres y eventos del mouse. Ninguna de estas cosas son objetos.

- Emitir la salida para el usuario.

  El usuario no puede ver objetos en la pantalla ni puede escuchar objetos a través de un parlante, por lo tanto un programa orientado a objetos deberá convertir los objetos en sonido o cadenas de caracteres o de enteros.

- Almacenar o recuperar datos de almacenamiento secundario.

  Ni los sistemas de archivos ni los motores de bases de datos son orientados a objetos (al menos los que se usan comúnmente). Ni un archivo ni una tabla son objetos ni tienen tipo (del sistema de tipos del programa orientado a objetos que los quiere utilizar). En consecuencia si un programa orientado a objetos debe persistir un objeto en un archivo o en una tabla, deberá convertirlo en no-objetos; y deberá hacer lo inverso cuando quiera recuperarlos.

Claramente, los efectos laterales son tan inevitables como contaminantes. Tan es así que hay un tipo de lenguajes de programación (no orientados a objetos) que confina los efectos laterales a porciones muy específicas del código que tienen un tratamiento especial; estos son los *lenguajes funcionales*. Precisamente esta es la estrategia que debe seguirse en el DOO: aislar las porciones de código que producen efectos laterales lo más posible y lo *antes* posible, en el caso de las entradas y lo más *tarde* posible en el caso de las salidas.

De hecho la metodología de Parnas predice este tipo de diseños: los efectos laterales son producto de representar cierto ente del mundo real con una cierta porción de código y con ciertas estructuras de datos. Esa representación es arbitraria y por ende es un ítem de alta probabilidad de cambio. En consecuencia debe ser aislado en un módulo con una interfaz abstracta insensible a los cambios anticipados. Una vez que el dato proveniente del exterior se ha *tipado*, es decir se ha convertido en un objeto (es decir, se ha anulado o resuelto el efecto lateral), entonces puede ser convenientemente manejado por el resto del sistema... hasta que debe ser expulsado al exterior. La Figura 15 intenta graficar esta situación.

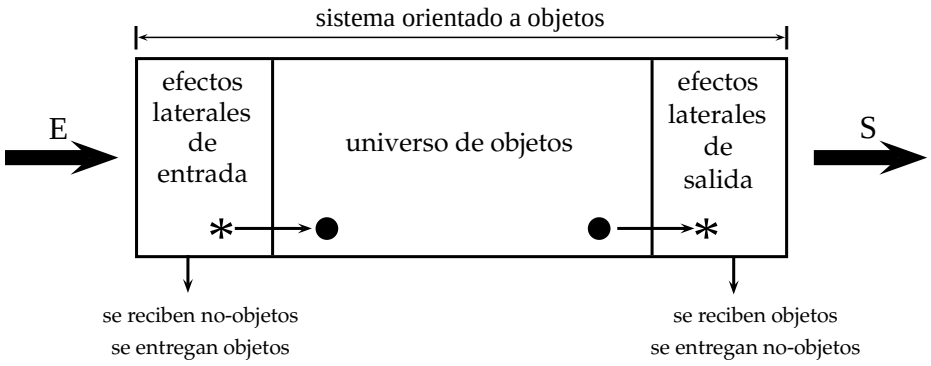


Figura 15: El diseño del sistema debe aislar las porciones de código que resuelven los efectos laterales. Por lo general, por cada tipo de efecto lateral habrá dos módulos: uno encargado de transformar no-objetos en objetos y otro de hacer lo inverso. El resto del sistema solo maneja objetos.

Cuando el sistema se diseña de forma tal que los efectos laterales se minimizan o confinan a porciones muy específicas del código, el testing es relativamente simple de realizar y mantener. Cuando los efectos laterales se distribuyen por todo el código, el testing se torna impracticable en el mejor de los casos, e imposible en el peor.

7.2. Generadores y observadores

En [7] los autores indican que los métodos de un TAD se dividen en dos categorías: generadores y observadores. Un *generador* es un método que modifica el estado de las instancias del TAD sin retornar nada. Un *observador* es un método que retorna un valor de otro TAD sin modificar el estado de la instancia sobre la cual se invoca. Si un método no cae dentro de ninguna de estas categorías entonces está mal definido (al menos desde el punto de vista teórico). Por otro lado, en la sección 2.4 se explicó que la interfaz de un módulo debe estar compuesta por la mínima cantidad de métodos que permitan proveer todos los servicios requeridos. Si los métodos se dividen en generadores y observadores, entonces la interfaz de un TAD debe estar compuesta por:

- La cantidad mínima de generadores tales que se pueda poner a una instancia dada en cualquiera de los estados admitidos por la especificación.
- La cantidad mínima de observadores tales que permitan observar el comportamiento externo especificado para el TAD.

Según estas definiciones un generador no puede retornar un error. Por ejemplo, el método autorizar() del módulo Edicto es un generador y por lo tanto solo puede modificar el estado de la instancia sobre la cual es invocado. En particular no puede retornar un error, por ejemplo, en el caso en que el usuario que recibe en el parámetro no exista. El módulo debería proveer alguna forma de que el invocante pueda determinar si la autorización fue exitosa o no.

7.3. Excepciones

Las excepciones no son un concepto perteneciente a la esfera del diseño por lo que solo mencionaremos un aspecto relacionado con el tema de la sección anterior. Muchos lenguajes de programación orientada a objetos introducen el mecanismo de las excepciones para comunicar errores durante la ejecución de métodos. Por otra parte, los valores de retorno de un método podrían usarse para comunicar errores. Surge entonces el dilema de cuál mecanismo usar para comunicar errores. Para dirimir este dilema es necesario tener en cuenta que cada método (tanto generadores como observadores) puede ser, matemáticamente hablando, una función total o una función parcial. Una función es parcial si no está definida para todo su dominio, caso contrario es total. Por ejemplo, el método (usualmente llamado head()) que toma una lista de elementos y retorna el primero de ellos, es una función parcial pues si la lista está vacía no puede retornar nada. Este método puede definirse de tres formas diferentes:

- head(**o** Elemento):Bool
  En este caso el valor de retorno indica si hay error o no; si no lo hay el valor retornado por el parámetro es válido, y es inválido en cualquier otro caso.
- head():Elemento – lanzando una excepción.
  Si el método es invocado sobre una lista vacía lanza una excepción indicando esta situación.
- head():Elemento – sin lanzar excepción.
  Los invocantes del método son responsables de verificar, previo a la llamada, que esta se puede realizar de forma segura. Si lo hacen en un estado incorrecto, el TAD al cual pertenece head() no garantiza nada.

En los dos primeros casos, el método se vuelve una función total, es decir brinda un comportamiento aceptable para todo su dominio. En el tercero sigue siendo una función parcial. Este último caso es razonable únicamente cuando los invocantes van a ser programados por el mismo equipo de ingenieros, quienes se responsabilizan por verificar la precondición antes de cada llamada.

Las dos primeras posibilidades son razonables en cualquier caso pero la segunda es la más conveniente. En efecto, las excepciones fueron creadas para dotar de un comportamiento razonable a los métodos parciales por lo que la segunda opción es la correcta.

Por este motivo, los generadores solo devolverán excepciones en tanto que los observadores utilizarán los valores de retorno para devolver valores útiles y las excepciones para comunicar errores.

8. Incidencia de los lenguajes de programación en un diseño

En general los libros "clásicos" de DOO dan todos sus ejemplos en uno o varios lenguajes orientados a objetos y muchos de ellos ponen a disposición del diseñador características específicas de cada lenguaje de programación. Más aun, en algunos de estos libros el diseño no es más que los primeros pasos de la programación. ¿Esto significa que si se hace un DOO la única forma de implementarlo es usando un lenguaje orientado a objetos? ¿Significa que el diseñador debe pensar diferente porque el lenguaje de implementación provea o no cierta característica? En esta sección queremos referirnos brevemente a esta cuestión porque consideramos que es otro de los mitos de la Ingeniería de Software.

Parnas propone su metodología a comienzos de los años 70 e inmediatamente después la aplica en un proyecto para la marina de los EE.UU conocido como A7E. El A7E es un avión de entrenamiento de la marina norteamericana que en aquellos años poseía una pequeña computadora de abordo IBM para controlar varias de las funciones de navegación y disparo. El proyecto encarado por el equipo de Parnas consistió en desarrollar el software de control para el A7E. Este tipo de software posee dos de las características más problemáticas de los sistemas de cómputo: debe cumplir requisitos temporales complejos sobre hardware muy limitado. Más aun, en aquella época ni si quiera se contaba con un compilador para el lenguaje de esa computadora.

Las ideas de Parnas son las precursoras del DOO [1]. ¿Cómo hizo el equipo de Parnas para implementar un DBOI de un sistema de tiempo real y con escaso poder de cómputo sin tener ni si quiera un lenguaje de alto nivel? Con disciplina de programación. Básicamente, los propios programadores realizaron los controles que haría un compilador para un lenguaje orientado a objetos. En nuestra opinión, este ejemplo extremo muestra a las claras que la imposibilidad de contar con un lenguaje orientado a objetos no es excusa para no diseñar e implementar los sistemas siguiendo la metodología de Parnas o sus extensiones más modernas. Obviamente, la implementación es más simple si se cuenta con un lenguaje que soporte las características del diseño, pero no tenerlo no determina el éxito o fracaso general del proyecto. Los proyectos de software fracasan por lo que no se hace o se hace mal antes o después de la programación y no, por no usar el último lenguaje de programación (en realidad muchas veces fracasan, precisamente, por querer usar el último lenguaje de programación).

Un compilador OO puede reemplazarse por revisiones de código, más ciertas técnicas de programación como punteros a función; en suma, por una buena disciplina de programación que adhiera al diseño existente. Más aun, nos ha tocado revisar diseños e implementaciones en lenguajes como Java que no eran OO cuando se pretendía que lo fueran. Incluso, la revisión de código no deja de ser necesaria porque puede ocurrir que el programador no entienda el diseño y, aun disponiendo de un lenguaje OO, haga una implementación errónea. Si este es el caso, cuando sea necesario incorporar un cambio, el costo no será el esperado.

Uno de los puntos a tener en cuenta cuando el lenguaje de implementación es OO es la semántica de la herencia soportada por ese lenguaje. Puede ocurrir que no soporte directamente herencia de interfaces o que estén disponibles variantes cuya semántica no sea la misma que la dada en la definición 6. En cualquier caso el diseñador debe instruir a los programadores sobre cómo o cuál definición de herencia disponible en el lenguaje deben utilizar y debe incorporar a las revisiones de código la verificación de esta adecuación.

9. Límites del DOO

Según lo hemos presentado, el DOO es la máxima expresión del DBOI. Al haber resuelto con el DOO elegantemente todos los problemas que se nos plantearon durante el diseño del software de control para la estación de peaje, podemos estar tentados de pensar que el DOO es la máxima expresión del diseño (y no ya únicamente del DBOI). Esto no es así. El DOO tiene varias deficiencias técnicas, metodológicas y expresivas que, mirando el desarrollo de un sistema globalmente, nos llevan a tomar el DOO como una forma de diseño de bajo nivel de abstracción.

El DOO permite conectar módulos únicamente por medio de llamada a procedimiento.

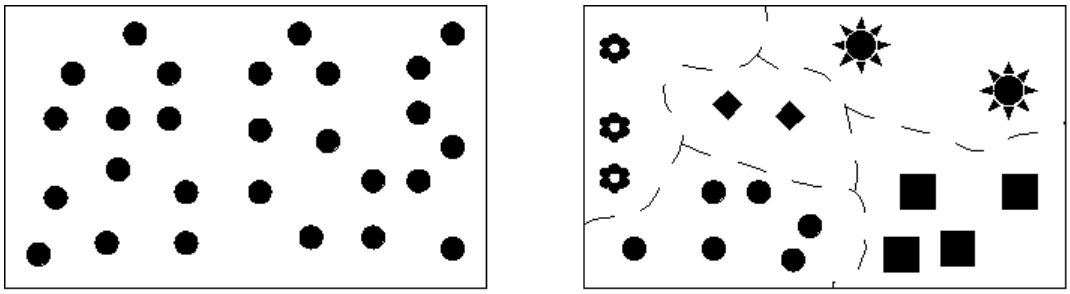


Figura 16: El espacio de soluciones del DOO (izquierda) es homogéneo e isotrópico. Para diseñar grandes sistemas se necesita un espacio de soluciones heterogéneo y no isotrópico (derecha).

Las interfaces de los módulos de un DOO quedan congeladas en tiempo de compilación. Los módulos de un DOO no tienen ningún tipo de restricción estructural (cualquier módulo puede relacionarse con cualesquiera otros) es decir el DOO presenta un espacio de soluciones homogéneo e isotrópico (Figura 16). Todas estas características, llegado el momento de diseñar grandes o complejos sistemas de software, son desventajas que pueden hacer al fracazar el proyecto.

La comunidad del DOO logró dar un paso en la dirección correcta al introducir los patrones de diseño [4]. Sin embargo, este salto en el nivel de abstracción y estructuración del espacio de soluciones, no es suficiente para diseñar grandes sistemas. Al mismo tiempo, no se puede lograr el nivel de abstracción y estructuración del espacio de soluciones adecuados sin salirse del DOO (Figura 16). En este sentido, se necesita echar mano de los conceptos de Arquitectura de Software y Estilos Arquitectónicos que veremos más adelante. Cuando terminemos de recorrer el camino que va desde el DBOI, pasando por el DOO y los patrones de diseño, hasta los estilos arquitectónicos, entenderemos que el DOO es, como dijimos, una forma de diseño de bajo nivel de abstracción pero que subyace sobre todas las formas de arquitectura y diseño. Es decir, los principios del DOO son aplicables a todos los sistemas pero en el nivel de abstracción correcto y en la etapa de desarrollo apropiada.

10. Documentación de diseño

Hasta el momento hemos tratado de aproximarnos al problema de diseñar un sistema desde un punto de vista conceptual, haciendo la menor referencia posible a notaciones y documentación. En esta sección veremos cómo debe documentarse un diseño.

El diseño, como fase del ciclo de desarrollo de un sistema, es una de las más importantes y posee los efectos de más largo alcance con respecto a la vida del sistema. Por lo tanto, un buen diseño es un factor clave en la definición de la duración del sistema y en el costo de incorporación de cambios; y simétricamente, un mal diseño puede transformar el mantenimiento del sistema en una pesadilla técnica y financiera. Por otro lado, quienes diseñan el sistema no necesariamente serán los que lo implementen y sus decisiones deberán estar disponibles para que diversos interesados en el sistema puedan revisarlas y comprenderlas. En consecuencia, documentar o describir de manera apropiada el diseño es tan importante como el diseño mismo. Tan es así que Parnas habla de "*design through documentation*" [9], a consecuencia de lo cual nosotros

enunciamos el siguiente principio de diseño.

Principio de Diseño: Diseñar es documentar

Un diseño sin documentación carece de utilidad práctica.

Un diseño no documentado carece de utilidad práctica porque la implementación debe basarse únicamente en la documentación de diseño. Sin documentación de diseño, los programadores no deberían iniciar la implementación

Como mencionamos en la definición de diseño dada en el capítulo anterior, el diseño incluye la especificación funcional de cada uno de sus elementos. La forma de documentar la función asignada a cada módulo es dando una especificación formal de su comportamiento. Como especificaciones formales ya lo hemos estudiado extensivamente, no mencionaremos más en este documento este aspecto de la documentación de diseño.

10.1. Documentos

La documentación de diseño se compone de documentos cada uno de los cuales describe el diseño del sistema desde un punto de vista particular, y de documentos que relacionan uno o más documentos. Cuantos más documentos se generen, más completa estará la documentación. Sin embargo, para ciertos sistemas no es necesario describir muchos de ellos. Por ejemplo, uno de los documentos típicos describe los procesos que ejecutarán para dar vida al sistema; si el sistema correrá lanzando un único proceso, entonces escribir ese documento es innecesario. Los documentos se clasifican en tres grandes categorías:

Documentos de módulos. Los elementos de estos documentos son módulos o unidades de implementación. Los módulos representan una forma basada en el código de considerar al sistema. Cada módulo tiene asignada y es responsable de llevar adelante una función.

Documentos de aspectos dinámicos. En estos documentos los elementos son componentes presentes en tiempo de ejecución y los conectores que permiten que los componentes interactúen entre sí[13].

Documentos con referencias externas. Estos documentos muestran la relación entre las partes en que fue descompuesto el sistema y elementos externos (tales como archivos, procesadores, personas, etc.)[14].

Conviene aclarar que en parte de la literatura se utilizan los términos vista y *estructura* como sinónimos.

Para los ejercicios de práctica y exámenes se pedirán uno o más de los siguientes documentos:

Documentos de módulos. Especificación de Interfaces, Estructura de Módulos, Guía de Módulos, Estructura de Herencia y Estructura de Uso.

Documentos de aspectos dinámicos. Estructura de Procesos, Estructura de Objetos, Diagrama de Interacciones.

Documentos con referencias externas. Estructura Física o de Despliegue.

[13] En [1] a estos documentos se los llama *vistas de componentes y conectores*.

[14] En [1] a estos documentos se los llama *allocation views*. En este caso optamos por una traducción más libre del término en inglés.

En cuanto a la documentación que relaciona dos o más de los documentos descriptos más arriba sólo diremos que si el nombre de una entidad en un documento coincide con el nombre de una entidad en otro documento, entonces se trata de la misma entidad vista desde dos ángulos diferentes. Por ejemplo, si el nombre de un módulo es *A* y el nombre de un proceso también es *A*, entonces se trata de lo mismo: una entidad que provee ciertos servicios y que se ejecuta como un proceso individual.

No siempre es necesario escribir todos los documentos. Se sugiere documentar al menos uno de cada categoría y siempre documentar la Especificación de Interfaces [5, 1].

10.2. Análisis de Cambio

Como sugiere la metodología propuesta por Parnas, antes de comenzar la descomposición del sistema en módulos es esencial estudiar los cambios futuros que probablemente se le exijan al sistema. Por tanto, es primordial documentar los ítem con probabilidad de cambio. Técnicamente hablando esta información no es parte del diseño pero es el fundamento para aquel, por lo que la incluimos como parte de la documentación de un diseño. Para documentar el análisis de cambio recurriremos a un listado donde se detallarán lo mejor posible cada una de las alternativas de cambio consideradas y se les asignará una probabilidad de cambio.

10.3. Estrategia de Cambio

Estrictamente hablando, este documento tampoco es parte de la documentación de diseño pero resulta esencial para mantener la correspondencia entre el código fuente y el diseño a medida que los ítem de cambio van apareciendo. En efecto, llega el momento en que uno de los cambios previstos se da y en consecuencia hay que modificar el sistema para incorporarlo. En la Estrategia de Cambio se documentan, para cada ítem de cambio, los pasos que deben seguir diseñadores y programadores para incorporarlo al sistema.

Por ejemplo, para incorporar un nuevo medio de pago al software de control de la estación de peaje visto en la sección 3, se debe:

1. Definir e implementar el módulo que abstrae el hardware necesario para ese nuevo medio de pago.
2. Definir e implementar el módulo que determina si el usuario pagó o no por ese medio de pago, cobra el dinero, consulta la lista de precios, etc.
3. Modificar el módulo **RecepcionPago** para que tenga en cuenta el nuevo medio de pago.

10.4. Especificación de Interfaces

Como ya mencionamos en la sección 3, las interfaces de los módulos del sistema se describen mediante un lenguaje semi-formal llamado 2MIL. En esta sección mostraremos únicamente la sintaxis y semántica de las construcciones del lenguaje que no aparecieron hasta el momento. Otras construcciones se verán en las secciones 10.5 y 10.10.

El módulo `Lista` descripto más abajo define la interfaz del tipo lista, parametrizado por el tipo de sus elementos. A los módulos de esta variedad se los llama *módulos genéricos*.

```
GENERIC MODULE   Lista(X)
IMPORTS          X
EXPORTS          add(i X)
                 head(): X
                 next(): X
                 more(): Bool
                 ...
```

A partir de un modulo genérico pueden crearse módulos (no son instancias, son módulos), como se muestra a continuación:

```
MODULE   ListaEmp IS Lista(Empleado)
```

De esta forma `ListEmp` es un módulo como cualquier otro del cual se pueden generar instancias.

En [4] se sugiere que normalmente la herencia es un concepto más poderoso y conveniente para lograr generalidad en un diseño que los módulos genéricos.

10.5. Estructura de Módulos

La estructura de módulos se construye sobre la base de la relación binaria entre módulos llamada *ES_SUBMODULO_DE* (o *ESD*). 2MIL provee la cláusula COMPRISES para definir esta relación, como se muestra en el siguiente ejemplo.

```
MODULE     System
COMPRISES  Client
           Server
```

En este caso $(Client, System) \in ESD$ y $(Server, System) \in ESD$. Un módulo comprendido en otro, puede a su vez comprender a otros módulos:

```
MODULE     Server
COMPRISES  ServerSideProtocol
           BusinessLogic
           LoadBalancing
```

La estructura de módulos es la representación de la relación *ESD*. Usualmente se recurre a una representación gráfica en forma de árbol en cuya raíz se ubica al módulo que comprende a todos los demás (típicamente llamado `System`, `Sistema` o que lleva el nombre del proyecto) y de este llega una flecha proveniente de cada uno de sus submódulos, y así sucesivamente, como se muestra en la Figura 17. Esta estructura muestra la organización del sistema en subsistemas o asignaciones de trabajo (es decir las partes del sistema que serán asignadas a diferentes equipos de trabajo). Siendo un resumen de la Guía de Módulos (sección 10.6), resulta muy útil pues habilita una aproximación jerárquica al diseño del sistema.

Los módulos que no son hojas de la estructura se llaman *módulos impropios* o *lógicos*, y las

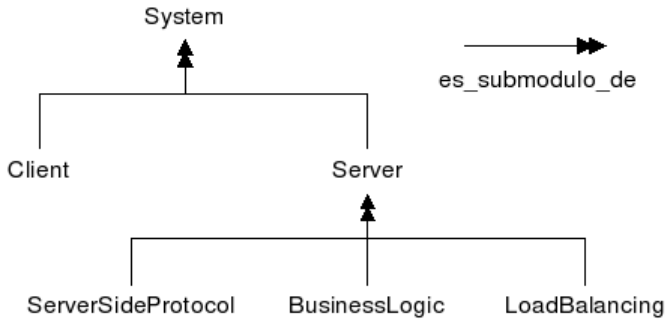


Figura 17: Representación gráfica de una estructura de módulos.

hojas se llaman *módulos propios* o *físicos*. Sólo los módulos físicos se implementan. Los módulos lógicos son un medio que permite organizar jerárquicamente el sistema. Los módulos lógicos no pueden participar en la herencia ni pueden instanciarse ni ser módulos genéricos. Ningún módulo puede ser submódulo de dos módulos diferentes.

El criterio para determinar cuándo un módulo debe ser submódulo de otro puede variar según las necesidades del diseñador [5]. Sea cual fuere el criterio utilizado debe quedar perfectamente documentado (ver final sección 10.6). No obstante, Parnas sugiere que el criterio sea una generalización del POI [9]: un módulo lógico debe contener a todos los módulos físicos o lógicos que deban ser modificados cuando uno de los cambios previstos sea exigido. Claramente, un módulo lógico puede ocultar más de un ítem de cambio pero esto no es un problema pues estos módulos no se implementan.

En algunos casos un módulo lógico representa un subsistema formado por todos los módulos que pertenecen a aquel. Usualmente un subsistema presenta una interfaz al resto del sistema que no necesariamente es la unión de las interfaces de todos los módulos que lo forman. Es decir, si f es una subrutina en la interfaz del módulo M el cual es parte del subsistema U, puede ser necesario restringir el uso de f sólo a otros módulos dentro de U de forma que f no sea invocada desde fuera de U. En 2MIL esta restricción se expresa de la siguiente forma.

MODULE	`Server`
COMPRISES	`ServerSideProtocol`
	`BusinessLogic`
	`LoadBalancing`
EXPORTS	`ServerSideProtocol`
	`turnLoadBalancingOff()`
COMMENTS	turnLoadBalancingOff() es parte de la interfaz de `LoadBalancing`.

Esto significa que *todas* las subrutinas en la interfaz del módulo `ServerSideProtocol` más la subrutina turnLoadBalancingOff() son las *únicas* subrutinas de los submódulos de `Server` que pueden ser accedidas por módulos que *no* pertenezcan a `Server`. Las subrutinas listadas en la cláusula EXPORTS de un módulo lógico tienen que estar sí o sí en la interfaz de alguno de sus submódulos físicos[15]. Si un módulo lógico no incluye la cláusula EXPORTS significa que

[15]Para estos casos es conveniente diseñar el módulo usando el patrón de diseño Facade [4], en cuyo caso la cláusula

todas las interfaces de sus submódulos pueden ser accedidas desde el exterior. La presencia de esta cláusula en un módulo lógico no significa de ninguna forma que ese módulo tenga una implementación real[16].

Usualmente los módulos lógicos de más alto nivel son asignados a diferentes grupos de programadores los que a su vez siguen la Estructura de Módulos para asignarse el trabajo internamente. Finalmente, uno o dos programadores se encargan de implementar los módulos físicos. Notar que el POI tiene un impacto crucial en este sentido pues un equipo a cargo de un módulo sólo necesita conocer la interfaz de algunos otros módulos para poder implementar el suyo, y a la vez, ese equipo puede decidir la mejor implementación sin comunicarla al resto de los equipos y sin que esto los afecte.

10.6. Guía de Módulos

La Guía de Módulos es un documento escrito en lenguaje natural que complementa la Especificación de Interfaces y la Estructura de Módulos. Fue propuesto por primera vez por Parnas [9] como el documento que sustenta a su metodología de diseño. El objetivo de este documento es permitir que los diseñadores, programadores y mantenedores del sistema puedan identificar las partes del software que ellos necesitan entender sin tener que leer detalles irrelevantes sobre las otras partes.

La guía se divide en capítulos, secciones, subsecciones, etc. donde cada una de estas divisiones corresponde a un nivel de la Estructura de Módulos (de aquí que resulte conveniente armar la Estructura de Módulos de forma tal que permita que la Guía de Módulos cumpla con su objetivo). Por ejemplo, si a los módulos introducidos en la sección 10.5 le agregamos los que siguen, y suponemos que los restantes módulos son físicos, la guía tendría la estructura mostrada en las Figuras 18 y 19.

MODULE	`Client`
COMPRISES	`UserInterface`
	`ClientSideProtocol`

MODULE	`BusinessLogic`
COMPRISES	`Customers`
	`Billing`
	`Payrole`
	`Providers`

La documentación a consignar en cada sección es la siguiente:

Módulos lógicos. Descripción del criterio por el cual el módulo contiene a sus "hijos". Muy breve descripción de los módulos existentes y de su función (uno o dos renglones por hijo).

EXPORTS listará únicamente el nombre de la fachada.

[16] Notar que usualmente este tipo de restricciones no son fáciles de imponer aun en lenguajes orientados a objetos, cf. secciones 8 y 9, también [4, punto 2 en página 174].

1. Module: Client

Criterio por el cual este módulo incluye a los otros.

1.1. Module: UserInterface

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

1.2. Module: ClientSideProtocol

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

2. Module: Server

Criterio por el cual este módulo incluye a los otros.

2.2. Module: BusinessLogic

Criterio por el cual este módulo incluye a los otros.

2.2.1. Module: Customers

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

2.2.2. Module: Billing

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

2.2.3. Module: Payrole

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

Figura 18: Estructura de una Guía de Módulos. Primera parte.

2.2.4. Module: Providers

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

2.1. Module: ServerSideProtocol

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

2.3. Module: LoadBalancing

Función. *Descripción funcional del módulo*

Secreto. *Descripción del secreto encapsulado en este módulo*

Figura 19: Estructura de una Guía de Módulos. Segunda parte

Módulos físicos. El texto se divide en dos secciones (como se muestra en las Figuras 18 y 19): Función y Secreto.

Función. En este apartado se da una especificación funcional del módulo lo más detallada posible. En principio se debe dar una especificación funcional de cada una de las subrutinas en la interfaz del módulo. Esta especificación funcional debe incluir, como mínimo, el significado de cada parámetro y una explicación de cada una de las salidas; en particular deben explicarse cada uno de los mensajes de error producidos (un mensaje de error no es únicamente algo que se imprime en la pantalla, sino los errores en general que emite la subrutina incluyendo valores de retorno, excepciones, etc.). En caso de que exista una especificación formal se debe incluir una referencia a aquella.

Secreto. En la segunda se explica la decisión de diseño que se decidió ocultar y la razón que llevó a ello; debe haber una referencia al documento Análisis de Cambio.

Si un módulo físico es heredero de otro, su descripción en la guía se hace al mismo nivel que la de su padre. Además de la información mencionada debe agregarse el motivo por el cual el módulo es un heredero y cuáles métodos se reimplementan.

10.7. Estructura de Uso

La Estructura de Uso fue propuesta originalmente por Parnas como medio para poder calcular los *subconjuntos útiles* y los *incrementos mínimos* de un sistema, y así habilitar la posibilidad de entregas incrementales.

Definición 11. *Un subconjunto útil de un sistema es un conjunto mínimo de subrutinas que proveen una funcionalidad útil para el destinatario del sistema. Es mínimo porque si se quita una de las subrutinas,*

la funcionalidad no puede realizarse. En la actualidad a los subconjuntos útiles se los denomina versiones del sistema.

Definición 12. *Un incremento mínimo con respecto a un subconjunto útil, es un conjunto mínimo de subrutinas tal que si es agregado al subconjunto útil genera un nuevo subconjunto útil.*

Una correcta ingeniería de subconjuntos útiles e incrementos mínimos permite realizar entregas incrementales al cliente, definir productos con menos o más funcionalidades orientados a diferentes clientes o hardware, etc.

La Estructura de Uso es la representación (usualmente gráfica o tabular) de la relación binaria entre unidades ejecutables *REQUIERE_EJECUCION_CORRECTA_DE* o *REC*. Las unidades ejecutables son subrutinas o programas (es decir cualquier unidad de código cuya ejecución pueda ser invocada desde algún punto de un programa).

Definición 13. *Si f y g son dos unidades ejecutables, $(f,g) \in REC$ sí y sólo sí f necesita de una implementación correcta de g para poder cumplir con su especificación.*

Una unidad ejecutable f necesita de una implementación correcta de otra unidad ejecutable, sólo si la especificación de f así lo indica.

Es importante resaltar que la relación *REC* está asociada con la especificación de las unidades ejecutables. Esto significa que si la especificación de una subrutina cambia (por ejemplo porque el cliente quiere una versión inicial diferente de la final) la Estructura de Uso probablemente también lo hará. Sin embargo, aun en estos casos, esta vista constituye una ayuda invaluable pues, precisamente, nos muestra cuáles especificaciones dependen de otras.

Según se desprende de la definición anterior "llamada a procedimiento" no siempre significa *REC*. En otras palabras si la subrutina f realiza una llamada a procedimiento de la subrutina g, esto no significa necesariamente que $(f,g) \in REC$. Más precisamente si la especificación de f sólo estipula que esta debe invocar a g entonces f no requiere la ejecución correcta de g. Por otro lado, puede ocurrir que $(f,g) \in REC$ pero que no exista una llamada a procedimiento de f a g. El caso más típico es cuando f y g interactúan por medio de memoria compartida.

Cabe preguntarse cuáles son las condiciones que deben darse para que $(f,g) \in REC$. Parnas sugiere que $(f,g) \in REC$ si se dan todas las condiciones que siguen:

- f es sustancialmente más simple al usar g.

  Si f usa a g significa que el código de g no estará dentro de f y por lo tanto f debería ser más simple de comprender.

- g no es sustancialmente más compleja al no permitírsele usar f.

  Como veremos más abajo, no es conveniente que se generen ciclos en la estructura de uso. Por lo tanto, si $(f,g) \in REC$ entonces no convendrá que $(g,f) \in REC$. Entonces puede ocurrir que g termine siendo demasiado compleja porque no puede usar a f.

- Existe un subconjunto útil que contiene a g pero no contiene a f.

- No existe un subconjunto útil razonable que contiene a f pero no contiene a g.

En un sistema de porte medio la cantidad de unidades ejecutables es tal que se torna poco práctico representar la Estructura de Uso gráficamente. Por lo tanto, se sugiere una representación matricial o tabular. Esto se logra con una tabla donde las filas y columnas listan, en el mismo orden, todas las unidades ejecutables del sistema y para aquellos pares que verifican la relación la celda de intersección se completa con algún símbolo adecuado. Tomaremos como

convención que la relación va de las filas en las columnas, es decir: si f es una unidad ejecutable listada en una fila y g es una unidad ejecutable listada en una columna, la celda de intersección debe completarse sí y sólo sí $(f, g) \in REC$. Claramente, la diagonal principal de la tabla debe quedar vacía. También es posible que todas las subrutinas en la interfaz de un módulo tengan el mismo patrón de uso en cuyo caso se pueden reemplazar por el nombre del módulo (con el fin de reducir el tamaño de la tabla).

Recalcamos que la relación *REC* está definida sólo entre unidades ejecutables, no entre módulos; los módulos aparecen en la estructura de uso únicamente con el fin de abreviar la representación.

Dada esta representación y la convención que hemos fijado, si f es una subrutina o programa, ubicada en la fila i, que debe estar en un subconjunto útil, S, entonces si la celda (i, j) (con $i \neq j$) está marcada y g es la unidad ejecutable listada en la columna j, entonces g debe estar en S; este procedimiento debe aplicarse también a g. En otras palabras todos los elementos en la clausura transitiva de f deben estar en S. La pertenencia de f a S la determina el ingeniero en base a la funcionalidad que se le debe entregar al cliente; pero la pertenencia de g se determina sintácticamente teniendo la Estructura de Uso. Algo semejante ocurre con los incrementos mínimos.

Otra propiedad fundamental de la relación *REC* es que no existan ciclos pues en ese caso, como dijera Parnas, "se tiene un sistema donde nada funciona hasta que todo funciona". Se da un ciclo cuando existe una secuencia de unidades ejecutables $f_1, \ldots, f_n$ tal que $(f_i, f_{i+1}) \in REC$ para todo $i \in [1, n-1]$ y $(f_n, f_1) \in REC$. En consecuencia es conveniente contar con alguna herramienta capaz de detectar esta clase de problemas tomando como entrada la Estructura de Uso. En caso de que se dé un ciclo, Parnas propone dividir en dos una de las unidades ejecutables de la secuencia (por ejemplo, f_i^1 y f_i^2) de forma tal que $(f_i^1, f_i^2) \notin REC$.

La Estructura de Uso es, tal vez, la vista más compleja de documentar y mantener pero al mismo tiempo es una de las que mayores beneficios otorga al equipo de ingenieros a cargo del proyecto. Además de constituir una base sólida para poder realizar entregas incrementales, es extremadamente valiosa durante la fase de testing pues permite determinar la secuencia en que deben ser probadas las unidades ejecutables de forma tal que las últimas en ser testeadas llamen a las que ya lo fueron.

Ver la sección 11 para un ejemplo.

10.8. Estructura de Procesos

La Estructura de Procesos permite tener una idea de cómo se comportará el sistema en tiempo de ejecución. Es una visión de alto nivel de abstracción puesto que no permite ver dentro de cada proceso. Por ejemplo, esta vista no permite saber cuántos o cuándo serán creados los objetos de cada tipo. Es una de las estructuras propuestas originalmente por Parnas. Aquí, consideraremos procesos con un único hilo de control por lo que procesos con múltiples hilos de control deben ser considerados como varios procesos.

Esta estructura o vista es la representación (usualmente gráfica) de la relación binaria entre procesos *SINCRONIZA_CON* o *SINC*.

Definición 14. *Un proceso P sincroniza con otro proceso Q sí y sólo sí P emite un evento que Q espera. En este caso decimos que* $(P, Q) \in SINC$. *Notar que la definición de sincronización es idéntica a la utilizada en CSP. Si dos procesos, P y Q, son tales que* $(P, Q) \notin SINC$ *entonces decimos que son independientes y por lo tanto pueden ejecutar en completo paralelismo.*

Es importante remarcar que procesos, módulos y programas son tres elementos diferentes que no necesariamente mantienen una relación simple entre ellos. Dado un módulo M puede ocurrir o no que haya un programa P que sea el resultado de compilar sólo M y, aun en ese caso, no necesariamente habrá un proceso Q que sea el resultado de poner en ejecución sólo a P. Justamente, para relacionar estas entidades es que se requieren documentos que relacionen diferentes vistas del sistema; este tópico lo abordaremos en Arquitecturas de Software[17].

En sistemas complejos o grandes, contar con esta estructura permite analizar las posibilidades de programación de procesos (*process scheduling*), existencia de abrazo mortal (*deadlock*), paralelización de procesos, etc. En suma, permite optimizar o predecir el comportamiento del sistema en tiempo de ejecución aun antes de haberlo programado.

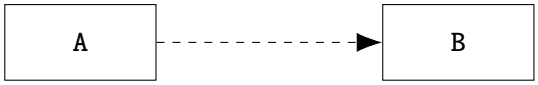
10.9. Estructura de Objetos

Esta vista aplica sólo a los DTAD y DOO pues relaciona las instancias (objetos) de los diversos tipos existentes en el sistema. Como ya observamos tanto en la Especificación de Interfaces como en la Estructura de Módulos, no hay posibilidad de conocer cuántas instancias de cada módulo se crearán (lo que muchas veces se hace recién en tiempo de ejecución) y qué relación hay entre ellas. La Estructura de Objetos viene a completar esta falta.

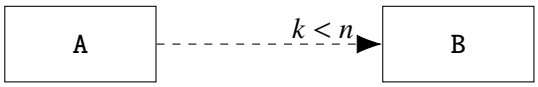
Esta vista se relaciona con la Estructura de Procesos pues los objetos se crean dentro de un proceso por lo cual es importante dejar registrado en relación a cuál de los procesos del sistema pertenecen los objetos que se mencionan.

Dividimos la Estructura de Objetos dos documentos.

Creación de Objetos. En este documento se describe cuáles módulos crean objetos de otros tipos. La forma más común de hacerlo es con un gráfico como el siguiente.



donde se indica que el módulo `A` crea uno o más objetos de tipo `B`. Si conocemos la cantidad de objetos que se crean de forma más precisa podemos anotar la flecha con el número o expresión correspondiente.



También es posible documentar esta información en forma de tabla como se muestra en la Tabla 1.

Como los objetos pueden ser creados y destruidos durante la ejecución del sistema, hay que dejar en claro qué se entiende por la cantidad de objetos que se crearán: ¿coexistirán simultáneamente? ¿Es el total que se creará durante la vida del proceso? No importa cuál sea la semántica que se le dé, lo importante es que se documente y que la información consignada sea consistente con esa semántica. Tener en cuenta que diferentes semánticas habilitan distintas posibilidades de análisis.

[17] Por el momento sólo utilizamos la convención mencionada al final de la sección 10.1.

Creación de Objetos			
Creador	Tipo	Cantidad	Condiciones
A	C	m	$m+k=10$
A	D	3	
B	D	k	$m \leq 10$

Cuadro 1: Ejemplo de una estructura de Creación de Objetos

Relaciones entre Objetos. En un segundo documento se describen las relaciones entre los objetos de los distintos tipos. No siempre es posible hacerlo y no siempre es necesario hacerlo para todos los objetos del sistema. Generalmente solo se documentan las relaciones más complejas o que son más difíciles de comprender. Las relaciones pueden ser *COMPUESTO_CON*, que indica alguna forma de composición según lo visto en la sección 6.3, y *ENLAZA_CON* (también llamada *INVOCA_SERVICIOS_DE* o *ISD*), que engloba a las relaciones por las cuales unos objetos utilizan los servicios de otros [2]. Por ejemplo, por cada carril en la estación de peaje habrá sí o sí un objeto `Barrera` y otro `ControlCarril` sin que haya ningún tipo de composición, sin embargo el objeto `ControlCarril` enlaza con el objeto `Barrera` pues utiliza los servicios de este.

La relación *ENLAZA_CON* no es *REC* ni viceversa. La diferencia fundamental está en que la primera se establece entre objetos, es decir elementos que existen en tiempo de ejecución, en tanto que la segunda se establece entre unidades ejecutables, es decir elementos que existen en tiempo de compilación. Además, si un objeto usa los servicios de otro (es decir, enlaza con otro) no necesariamente requiere la ejecución correcta de este pues puede ser que solo requiera la presencia de la interfaz. Finalmente, como ya dijimos, *REC* permite realizar una ingeniería de versiones en tanto que *ENLAZA_CON* permite estudiar ciertos aspectos de desempeño o comprender cómo funciona el sistema en tiempo de ejecución (en ocasiones, sobre todo cuando se utilizan patrones de diseño, es muy complejo entender cómo se obtendrá cierto resultado o efecto con solo leer la Especificación de Interfaces y la Guía de Módulos).

Notar que un objeto dado no tiene que estar compuesto o enlazado con un único objeto, sino que puede relacionarse con cero, uno, o más de uno. En consecuencia, las relaciones *COMPUESTO_CON* y *ENLAZA_CON* pueden ser de varios tipos: Rel (relación), Fun (función), PFun (función parcial), Iny (función inyectiva), PIny (función parcial inyectiva), Sur (función suryectiva), PSur (función parcial suryectiva), Biy (función biyectiva). Por ejemplo, cada objeto `Cliente` se compone de un único objeto `CuentaCorriente` (Fun) ; o sólo algunos objetos `Cliente` están asociados con un único objeto `CuentaCorriente` (PFun).

Para documentar estas relaciones se puede usar gráficos o tablas. La relación de composición se grafica como se indica a continuación.

lo que significa que un objeto de tipo `B` contiene (en términos de composición) a un objeto de tipo `A`. Si la composición es de más de un objeto se usa una flecha con un círculo en la punta.

Composición de Objetos					
Módulos	**Module 1**	**Module 2**	**Module 3**	... ...	**Module** *N*
Module 1		PFun			
Module 2					Rel
Module 3					
.....	.....	.....	.....	... ...	.....
.....	.....	.....	.....	... ...	.....
Module *N*			PIny		

Cuadro 2: Ejemplo de una tabla para representar la estructura de Composición de Objetos.

También podemos anotar la flecha con un parámetro o expresión que indique la cantidad de objetos que se componen.

También podemos usar una tabla con la estructura de la Tabla 2. En cada celda se consigna la relación entre los objetos del tipo de la fila con el tipo de la columna. La diagonal principal debe quedar en blanco. En sistemas complejos puede ser necesario describir restricciones entre los objetos involucrados en las diferentes relaciones, lo que puede hacerse aparte o en una columna de la tabla. Ver la sección 11 para un ejemplo completo.

10.10. Estructura de Herencia

Esta estructura muestra las relaciones de herencia que hay entre los diferentes tipos. Usualmente sólo hay unas pocas "líneas hereditarias" que involucran a unos pocos tipos por lo que suele ser adecuado una representación gráfica en forma de árbol para cada familia. Si se describieron las interfaces con 2MIL esta estructura se puede obtener por análisis sintáctico.

Los módulos que solo ofrecen una interfaz (es decir que no se implementan) no deben importar ningún recurso.

10.11. Estructura Física

También llamada Estructura de Despliegue [5], esta vista permite relacionar elementos de software con plataformas de ejecución tales como procesadores, discos, redes, etc. Usualmente los elementos de software son procesos de la Estructura de Procesos. La relación más común que se utiliza para documentar esta vista es *ALOJADO_EN*, la cual relaciona un elemento de software (generalmente un proceso) con un elemento de hardware (generalmente un procesador o computadora).

Para sistemas pequeños se sugiere una representación gráfica en forma de grafo donde los nodos son computadoras y los arcos líneas de comunicación (Ethernet, telefónicas, etc.). En cada nodo se dibujan, además, los procesos que serán alojados en esa computadora. Para sistemas más grandes o muy distribuidos una representación tabular suele ser más manejable.

Hay otras variantes de la Estructura Física que relacionan, por ejemplo, repositorios de datos con discos, objetos con computadoras, etc.; o que documentan relaciones tales como *MIGRA_HACIA*, *COPIA_MIGRA_HACIA*, etc.

10.12. Líneas y cajas

Muchas de las estructuras que hemos mencionado se pueden documentar gráficamente. Estos gráficos toman la forma de un grafo donde los nodos se dibujan como cajas que pueden ser procesos, unidades ejecutables, módulos, etc. y las flechas representan diferentes relaciones (*REC*, *ESD*, herencia, etc.). Si bien un gráfico suele ser muy bueno como vehículo para comunicar el diseño de un sistema, en el ámbito informático los gráficos de diseño ya casi no tienen *significado* [1]. Esto se debe a que no se expresa explícita y rigurosamente el significado (semántica) de las cajas y las flechas. El mismo tipo de flecha o caja se usa en distintos documentos para representar diferentes estructuras sin explicar en ningún caso qué significan las flechas y qué son las cajas. Es muy poco o muy pobre lo que estos gráficos pueden comunicar fehacientemente a los distintos involucrados en el desarrollo y mantenimiento del sistema

Por lo tanto, todo gráfico debe ir acompañado de una referencia que explique qué son las cajas y qué significan las flechas. Esta explicación debe apuntar a dar un significado claro, unívoco y preciso de esos elementos, como hemos hecho en esta sección. Si el gráfico lleva el nombre de una de las estructuras que hemos introducido en este apunte, tal vez esta explicación pueda obviarse pero nunca debe olvidarse que aun estas estructuras "estándar" tiene algunas variantes semánticas importantes.

El problema de la falta de significado de gráficos suele ser recurrente en el uso práctico (no tanto en la teoría) de UML.

10.13. Nota sobre la protocolización de la documentación de diseño

Todo lo que hemos mencionado en esta sección sobre documentación de diseño refiere únicamente al contenido técnico de cada documento. Sin embargo, cada documento debe estar precedido por datos protocolares tales como: nombre y apellido de quién es el responsable del documento, propósito, número de versión, fecha de la última modificación, etc., etc., etc.

Ejercicios

Ejercicio 23. Defina un módulo genérico para manejar diferentes marcas o modelos de barreras. ¿Qué tanto puede lograr? ¿Es mejor o peor que una jerarquía de herencia? ¿Por qué?

Ejercicio 24. Complete la estructura de módulos de la Figura 17 con los módulos introducidos en la sección 10.6.

Ejercicio 25. Explique la relación que hay entre la Estructura de Módulos y la Guía de Módulos. ¿Cómo debe armarse la Estructura de Módulos en relación a su Guía de Módulos? Tenga en mente el objetivo de la Guía de Módulos.

Ejercicio 26. La Figura 17 representa la relación *ESD* aunque usualmente las flechas van en sentido opuesto. Defina el significado de las flechas en sentido opuesto.

Ejercicio 27. ¿Dónde ubicaría los módulos genéricos en la Guía de Módulos? Justifique.

Ejercicio 28. Dé dos ejemplos de llamada a procedimiento que no implique *REC* y dos situaciones en que haya *REC* pero no sea producto de llamada a procedimiento.

Ejercicio 29. Si dibuja la Estructura de Uso como un grafo, ¿qué significa que a un método apunten muchas flechas? ¿Qué significa que de un método partan muchas flechas? ¿A cuáles de los dos tipos de métodos mencionados en las preguntas anteriores debería dedicarle más tiempo el diseñador? Justifique todas sus respuestas.

Ejercicio 30. ¿Qué análisis se pueden hacer teniendo la Estructura de Objetos?

Ejercicio 31. Suponga que *a ENLAZA_CON b*. ¿Significa esto que el flujo de datos es desde *a* hacia *b* únicamente? Justifique y ejemplifique.

11. Documentación de diseño del software para controlar la estación de peaje

En esta sección daremos los documentos que faltan para documentar el diseño del software de control para la estación de peaje. Tomaremos como referencia la versión definida en la sección 3 más los módulos que el lector debe definir en el ejercicio 7, pero considerando un DOO con una única marca o modelo para cada dispositivo de hardware excepto para la barrera. Como en el requerimiento original, suponemos que hay sólo dos carriles. El Análisis de Cambio y la Especificación de Interfaces ya fueron documentados en secciones anteriores por lo que no los incluiremos en esta sección.

11.1. Estructura de Módulos

El criterio aplicado para definir la estructura de módulos es el sugerido por Parnas en [9]. Documentamos la Estructura de Módulos usando 2MIL.

MODULE	`EstacionPeaje`
COMPRISES	`Hardware`
	`MediosPago`
	`ControlCarril`

MODULE	`Hardware`
COMPRISES	`Barrera`
	`BarreraACME`
	`BarreraEMCA`
	`Impresora`
EXPORTS	`Barrera, Impresora`

`Hardware` exporta `Barrera` para reforzar el hecho de que los clientes deben utilizar únicamente ese módulo y no sus herederos. Notar que esto no significa que no se puedan definir variables de los subtipos de `Barrera` en el código presente fuera del módulo `Hardware`.

De todo lo concerniente a los medios de pago, el resto del sistema sólo precisa acceder a `RecepcionPago` y `Ticket`.

MODULE	MediosPago
COMPRISES	HardwareMediosPago
	SoftwareMediosPago
	TablaPrecios
	Ticket
EXPORTS	RecepcionPago, Ticket

Los dos módulos lógicos que siguen se definen a la espera de otros medios de pago.

MODULE	HardwareMediosPago
COMPRISES	MaquinaMB

MODULE	SoftwareMediosPago
COMPRISES	PagoEfectivo
	RecepcionPago
EXPORTS	RecepcionPago

El módulo CondicionBajarBarrera debería definirse en el ejercicio 7.

MODULE	ControlCarril
COMPRISES	Control
	CondicionBajarBarrera

11.2. Guía de Módulos

Dado que la Guía de Módulos toma la forma de un documento con capítulos, secciones, subsecciones, etc. es muy difícil documentarla dentro de un documento que ya tiene esas diviciones. Por lo tanto optamos por documentarla en letra más pequeña y con títulos, subtítulos, etc. centrados y sin numeración.

EstacionPeaje

El sistema de software para control de la estación de peaje consiste de tres módulos que se describen a continuación.

Hardware

Este módulo contiene los módulos que deben ser modificados si se reemplaza algún dispositivo de hardware, excepto los relacionados con los medios de pago, por uno similar. Los submódulos de este módulo proveen al resto del sistema un hardware virtual. Los secretos ocultos en estos módulos son las diversas formas en que los dispositivos de hardware deben ser usados.

Barrera

Módulo abstracto que provee únicamente una interfaz para utilizar diferentes marcas o modelos de barreras.

BarreraACME

Oculta las interfaz hardware/software para interactuar con la barrera ACME UpDown 3000.

BarreraEMCA

Oculta las interfaz hardware/software para interactuar con la barrera EMCA DownUp 0003.

Impresora

Oculta la interfaz hardware/software para interactuar con la impresora ACME Jetprint 40. Además, oculta la forma en que se consulta el estado del tique para imprimirlo y el formato en que se lo imprime.

MediosPago

El módulo lógico `MediosPago` agrupa todos los módulos relacionados con los medios de pago habilitados por la empresa para pagar el peaje. Incluye desde los módulos que ocultan el hardware del sistema hasta los que implementan la política de cobro con uno o más medios de pago. También se incluye la lista de precios y el módulo que implementa el tique que finalmente se le entrega al cliente.

Si la empresa habilita nuevos medios de pago, los módulos que abstraigan esos requerimientos deben agregarse como submódulos de este módulo.

Los secretos que oculta este módulo van desde las interfaces hardware/software con los dispositivos de hardware, hasta la política de cobro pasando por los criterios que se aplican para determinar cuándo ha finalizado el pago por algún medio de pago.

HardwareMediosPago

Aquí se agrupan los módulos que abstraen el hardware que se utiliza para que el conductor pague el peaje, tal como máquinas receptoras de dinero, lectores de tarjetas de crédito/débito, etc. Por el momento, el único módulo físico es el que oculta el hardware de la máquina receptora de dinero.

MaquinaMB

Oculta las interfaz hardware/software para interactuar con la máquina receptora de dinero ACME CoinMachine.

SoftwareMediosPago

Este módulo lógico agrupa los módulos que implementan algún medio de pago por sobre los módulos que abstraen el hardware de los mismos. No incluye la lista de precios, ni el tique.

Los secretos que oculta son las condiciones por las cuales se considera que el pago por cada medio de pago ha finalizado, cualquier devolución que deba hacerse al cliente, lo que se incluirá en el tique correspondiente, y la política de cobro (es decir si se permite pagar con varios medios de pago, descuentos, etc.).

PagoEfectivo

Este módulo oculta las condiciones que determinan cuándo se considera que el conductor ha finalizado el pago en efectivo insertando monedas o billetes en la máquina correspondiente. También oculta cómo se entrega el vuelto.

RecepcionPago

Oculta la política de cobro autorizada por la empresa. Actualmente la política permite pagar con un único medio de pago.

TablaPrecios

Oculta las estructuras de datos y algoritmos que implementan la lista de precios así como su ubicación física.

Ticket

Oculta las estructuras de datos y algoritmos que implementan el tique que se enviará a la impresora. Se espera que cada medio de pago utilizará este módulo para agregar los datos que sean pertinentes a ese medio de pago.

ControlCarril

Este módulo incluye los módulos que se encargan de llevar adelante el algoritmo general de procesamiento desde que el conductor se aproxima a la casilla de peaje hasta que se retira.

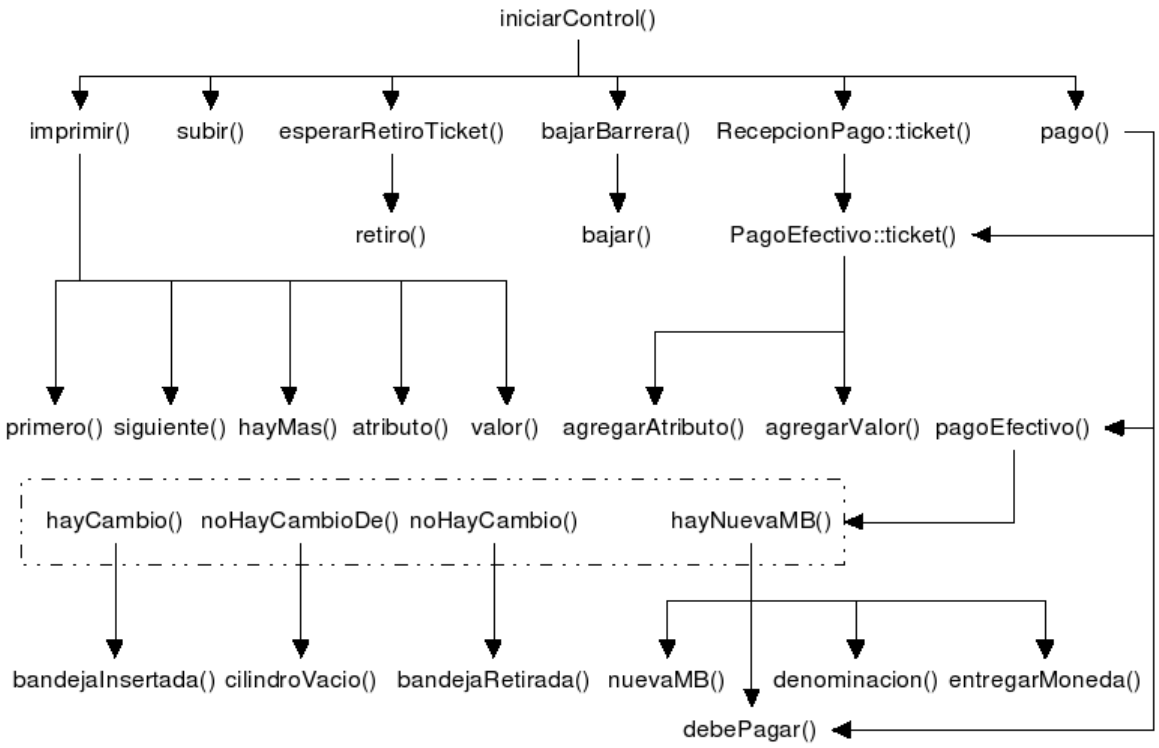


Figura 20: Estructura de Uso para el software de control de la estación de peaje. Las flechas indican *REC*. La caja de líneas discontinuas permite reemplazar las flechas que deberían llegar a cada uno de los métodos por una sola flecha.

Incluye los módulos que ocultan las distintas condiciones que deben darse para subir y bajar la barrera.

Control

Oculta el algoritmo general de procesamiento.

CondicionBajarBarrera

Este módulo oculta las condiciones (de negocio) para poder bajar la barrera.

11.3. Estructura de Uso

La Estructura de Uso se documenta parcialmente en la Figura 20, el resto queda como ejercicio para el alumno. Notar que pagoEfectivo() requiere la ejecución correcta de hayNuevaMB(), noHayCambioDe(), noHayCambio() y hayCambio() a pesar de que no hay una llamada a procedimiento. La razón es que para que pagoEfectivo() retorne el monto que realmente ha pagado el conductor, es necesario que los otros métodos funcionen correctamente pues son los que van acumulando el valor pagado o indican que el pago no puede efectuarse normalmente.

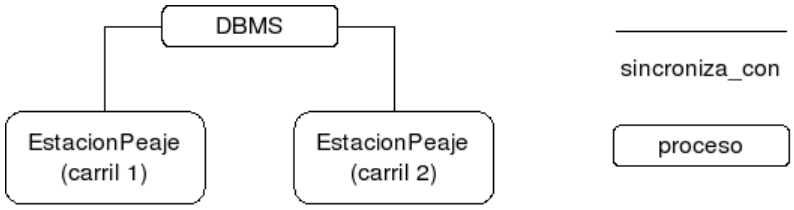


Figura 21: Estructura de Procesos para el software de control de la estación de peaje.

11.4. Estructura de Procesos

Suponemos que en cada carril se instalará una computadora en la cual correrá una versión del sistema. A cada una de estas computadoras se conectarán todos los dispositivos de hardware instalados en cada carril[18]. Con el fin de hacer la Estructura de Procesos un poco más interesante, consideramos que cada computadora de carril se conecta, vía una red Ethernet, a una computadora donde corre un DBMS que mantiene la contabilidad de la empresa por lo que luego de emitido el tique, este debe enviarse a aquella.

La Estructura de Procesos se documenta gráficamente en la Figura 21. Notar que hemos usado el mismo nombre, `EstacionPeaje`, para designar un módulo y un proceso. La convención establecida en el último párrafo de la sección 10.1, indica que ambos elementos refieren a lo mismo pero desde dos puntos de vista diferentes.

11.5. Estructura de Objetos

La Estructura de Objetos para el sistema la documentamos en la Tabla 3 que describe la creación de objetos, y en la Figura 22, que describe la relación *ENLAZA_CON*, en ambos casos para los objetos de cada proceso. No utilizamos composición de objetos.

Puede ocurrir que el objeto `RecepcionPago` enlace con el objeto `Ticket` dependiendo de la política de cobro que se implemente. Otros medios de pago podrían enlazar con el objeto `Ticket`. No consideramos que el objeto `MaquinaMB` enlaza con el objeto `PagoEfectivo`, a pesar de que hay invocación a métodos, pues se lo hace a través de *callbacks*.

11.6. Estructura de Herencia

La estructura de herencia de esta versión del sistema es muy simple pues solo la hemos utilizado para ocultar las dos marcas de barreras. La Figura 23 documenta parcialmente en forma gráfica el texto 2MIL introducido en la sección 6.1. Faltaría indicar cuáles métodos se reimplementan y cuáles no; de todas formas está información se consigna en la Guía de Módulos.

Ejercicios

Ejercicio 32. Grafique la Estructura de Módulos según la relación *ESD*.

Ejercicio 33. Complete la Estructura de Uso documentado los métodos inicializar(). ¿Qué pasa con los métodos Ticket::eliminar() y MaquinaMB::capacidadCilindro()?

[18] En rigor, todas estas suposiciones o hechos deben documentarse en la Estructura Física.

Creación de Objetos		
Semántica: cantidad de objetos que coexistirán como máximo durante la vida de un proceso **EstacionPeaje**.		
Tipo	**Cantidad**	**Condiciones**
BarreraACME	m	$m + k = 1$
BarreraEMCA	k	
CondicionBajarBarrera	1	
Control	1	
Impresora	1	
MaquinaMB	1	
PagoEfectivo	1	
RecepcionPago	1	
TablaPrecios	1	
Ticket	1	

Cuadro 3: Creación de Objetos para el software de control de la estación de peaje.

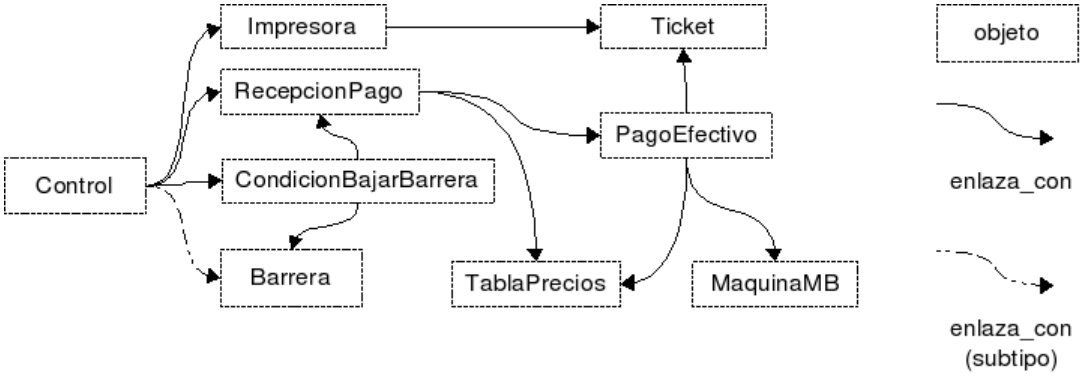


Figura 22: Enlace de Objetos para el software de control de la estación de peaje.

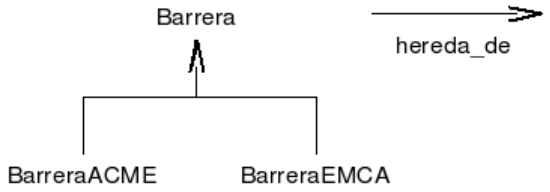


Figura 23: Estructura de Herencia para el software de control de la estación de peaje.

Ejercicio 34. Suponga que inicialmente el cliente desea conectar a la computadora del carril solo la barrera y operar manualmente el resto de los requerimientos (un operador de cabina pulsa Enter para indicar la entrega del tique). Determine el subconjunto mínimo que debería implementar. El conjunto que usted propone, ¿se corresponde con la Estructura de Uso documentada? ¿Por qué su subconjunto útil no incluye a imprimir(), por ejemplo? ¿Qué fue lo que cambió de iniciarControl() que le permite calcular ese subconjunto útil? El cambio en iniciarControl(), ¿implica que hay que modificar la Estructura de Uso?

Ejercicio 35. Respecto del problema anterior, determine el incremento mínimo para incluir la máquina traga monedas aunque el tique lo imprime manualmente un operador de cabina.

Ejercicio 36. Compruebe que la Estructura de Uso verifica el criterio señalado por Parnas para diseñarla.

Ejercicio 37. En la sección 3 no mencionamos claramente que habría una computadora por carril y que en cada una de ellas correría una instancia del sistema, como aclaramos en la sección 11.4. ¿Induce esta información algún cambio en las interfaces definidas? ¿En la implementación? Justifique. En caso afirmativo, modifique las interfaces que sea necesario y/o indique cómo debe modificarse la implementación.

Ejercicio 38. Suponga que hay una única computadora para los dos carriles (además de la computadora central) a la cual se conectan todos los dispositivos de ambos carriles. Determine cuáles documentos debe modificar y modifíquelos.

Ejercicio 39. ¿Qué significa, concretamente, que `EstacionPeaje` sea a la vez un módulo y un proceso? ¿Es esto un error? ¿Cómo se genera el proceso `EstacionPeaje`, qué código ejecuta?

Ejercicio 40. Considere el requerimiento de enviar los tiques al DBMS de la computadora central, mencionado en la sección 11.4. Rediseñe y documente el sistema. La aparición tardía de este requerimiento, ¿qué enseñanza le deja? ¿Qué ítem de cambio olvidamos tener en cuenta?

Ejercicio 41. Complete la documentación del software para la estación de peaje teniendo en cuenta los ejercicios 14 y 19.

Ejercicio 42. Documente la Estructura Física para el software de la estación de peaje.

Referencias

[1] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional, April 2003.

[2] Grady Booch. *Aná1lisis y diseño orientado a objetos con aplicaciones*. Addison Wesley, 1998.

[3] Daniel Bovet and Marco Cesati. *Understanding the Linux Kernel, Second Edition*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2002.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Patrones de diseño*. Addison Wesley, 2003.

[5] David Garlan, Felix Bachmann, James Ivers, Judith Stafford, Len Bass, Paul Clements, and Paulo Merson. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2nd edition, 2010.

[6] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of software engineering (2nd ed.)*. Prentice Hall, 2003.

[7] John V. Guttag and James J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[8] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994.

[9] D. L. Parnas, P. C. Clements, and D. M. Weiss. The modular structure of complex systems. In *Proceedings of the 7th international conference on Software engineering*, ICSE '84, pages 408–417, Piscataway, NJ, USA, 1984. IEEE Press.

[10] David L. Parnas. Designing software for ease of extension and contraction. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 264–277, Piscataway, NJ, USA, 1978. IEEE Press.

[11] David Lorge Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.