Programación modular I

Ricardo Pérez López

IES Doñana, curso 2020/2021

Índice general

1. Introducción

1.1. Modularidad

La **programación modular** es una técnica de programación que consiste en descomponer y programar nuestro programa en partes llamadas **módulos**.

- El concepto de *módulo* hay que entenderlo en sentido amplio: cualquier parte de un programa se puede considerar «módulo».

Equivale a la técnica clásica de resolución de problemas basada en 1) descomponer un problema en subproblemas; 2) resolver cada subproblema por separado; y 3) combinar las soluciones para así obtener la solución al problema original.

La **modularidad** es la propiedad que tienen los programas escritos siguiendo los principios de la programación modular.

El concepto de modularidad se puede estudiar a nivel *metodológico* y a nivel *práctico*.

A nivel metodológico, la modularidad nos proporciona una herramienta más para controlar la complejidad (como la *abstracción*, que de hecho puede servir como técnica de modularización).

Todos los mecanismos de control de la complejidad actúan de la misma forma: la mente humana es incapaz de mantener la atención en muchas cosas a la vez, así que lo que hacemos es centrarnos en una parte del problema y dejamos a un lado el resto momentáneamente.

- La **abstracción** nos permite controlar la complejidad permitiéndonos estudiar un problema o su solución por *niveles*, centrándonos en lo esencial e ignorando los detalles que a ese nivel resultan innecesarios.
- Con la **modularidad** buscamos descomponer conceptualmente el programa en partes que se puedan estudiar y programar por separado, de forma más o menos independiente, lo que se denomina **descomposición lógica**.

Ambos conceptos son combinables, como veremos luego.

A nivel práctico, la modularidad nos ofrece una herramienta que nos permite partir el programa en partes más manejables.

A medida que los programas se hacen más y más grandes, el esfuerzo de mantener todo el código dentro de un único *script* se hace mayor.

No sólo resulta incómodo mantener todo el código en un mismo archivo, sino que además resulta intelectualmente más difícil de entender.

Lo más práctico es repartir el código fuente de nuestro programa en una colección de archivos fuente que se puedan trabajar por separado, lo que se denomina **descomposición física**.

Un módulo es, pues, una parte de un programa que se puede estudiar, entender y programar por separado con relativa independencia del resto del programa.

Por tanto, podría decirse que **una función es un ejemplo de módulo**, ya que se ajusta a esa definición (salvo quizás que no habría *descomposición física*, aunque se podría colocar cada función en un archivo separado y entonces sí).

Sin embargo, descomponer un programa en partes usando únicamente como criterio la descomposición en funciones no resulta adecuado en general, ya que muchas veces nos encontramos con varias funciones que no actúan por separado, sino de forma conjunta entre ellas formando un todo interrelacionado.

Además, un módulo no tiene por qué ser simplemente una abstracción funcional, sino que también puede contener datos (almacenados en variables y constantes) manipulables desde dentro del módulo y posiblemente también desde fuera.

Por ejemplo, supongamos un conjunto de funciones que manipulan números racionales.

Tendríamos funciones para crear racionales, para sumarlos, para multiplicarlos, para simplificarlos, etc.

Todas esas funciones trabajarían conjuntamente, incluso usándose unas a otras, y actuarían sobre colecciones de datos que representarían números racionales de una forma apropiada.

Por consiguiente, aunque resulta muy apropiado considerar cada función anterior por separado, es más conveniente considerarlas como un todo: el *módulo de manipulación de números racionales*.

Otro ejemplo: supongamos un módulo encargado de realizar operaciones trigonométricas sobre ángulos.

Esos ángulos serían números reales que podrían representar grados o radianes.

Ese módulo contendría las funciones encargadas de calcular el seno, coseno, tangente, etc. de un ángulo pasado como parámetro a cada función.

Para ello, el módulo podría contener un dato (almacenado en una variable dentro del módulo) que indicará si las funciones anteriores deben interpretar los ángulos como grados o como radianes.

Ese dato constituiría el estado interno del módulo, al ser un valor que se almacena dentro del mismo módulo (pertenece a éste) y puede cambiar su comportamiento dependiendo del valor que tenga el dato.

Además, probablemente también necesite conocer el valor de π, que podría almacenar el módulo internamente como una constante.

De lo dicho hasta ahora se deducen varias **conclusiones importantes**:

- Un módulo es **una *parte* de un programa**.
- Los módulos nos permiten descomponer el programa en **partes más o menos independientes y manejables por separado**.
- Una **función** cumple con la definición de módulo pero, en general, **en la práctica no es una buena candidata** para ser considerada un módulo por sí sola.
- Los módulos, en general, agrupan colecciones de **funciones interrelacionadas**.
- Los módulos, **además de funciones, pueden contener datos** en forma de **constantes y variables** manipulables desde el interior del módulo así como desde el exterior del mismo usando las funciones que forman el módulo.
- Las variables del módulo constituyen el **estado interno** del módulo.
- A nivel práctico, los módulos se programan físicamente en **archivos separados** del resto del programa.

1.2. Beneficios de la programación modular

El tiempo de desarrollo se reduce porque grupos separados de programadores pueden trabajar cada uno en un módulo con poca necesidad de comunicación entre ellos.

Se mejora la productividad del producto resultante, porque los cambios (pequeños o grandes) realizados en un módulo no afectarían demasiado a los demás.

Comprensibilidad, porque se puede entender mejor el sistema completo cuando se puede estudiar módulo a módulo en lugar de tener que estudiarlo todo a la vez.

2. Diseño modular

2.1. Partes de un módulo

Desde la perspectiva de la programación modular, un programa está formado por una colección de módulos que interactúan entre sí.

Puede decirse que un módulo proporciona una serie de *servicios* que son *usados* o *consumidos* por otros módulos del programa.

Así que podemos estudiar el diseño de un módulo desde **dos puntos de vista complementarios**:

- El **creador** o **implementador** del módulo es la persona encargada de la programación del mismo y, por tanto, debe conocer todos los detalles internos al módulo, necesarios para que éste funcione (es decir, su **implementación**).
- Los **usuarios** del módulo son los programadores que desean usar ese módulo en sus programas. También se les llama así a los módulos de un programa que usan a ese módulo (lo necesitan para funcionar).

  A la parte que un usuario necesita conocer para poder usar el módulo se le denomina la **interfaz** del módulo.

Los **usuarios** están interesados en usar al módulo como una **entidad abstracta** sin necesidad de conocer los *detalles internos* del mismo, sino sólo lo necesario para poder consumir los servicios que proporciona (su **interfaz**).

Concretando, un módulo tendrá:

- Un **nombre** (que generalmente coincidirá con el nombre del archivo en el que reside).
- Una **interfaz**, formada por un conjunto de **especificaciones de funciones** que permiten al usuario consumir sus servicios, así como manipular y acceder al estado interno desde fuera del módulo.

  Es posible que la interfaz también incluya **constantes**.
- Una **implementación**, formada por:
  - Su posible estado interno en forma de **variables** locales al módulo.
  - La **implementación (el cuerpo) de las funciones** que aparecen en la interfaz.
  - Un conjunto de **funciones auxiliares** que no aparecen en la interfaz porque pensadas para ser usadas exclusivamente por el propio módulo de manera interna, pero no por otras partes del programa.

2.1.1. Interfaz

La **interfaz** es la parte del módulo que el **usuario** del mismo necesita conocer para poder utilizarlo.

Es la parte **expuesta, pública o visible** del mismo.

También se la denomina su **API** (*Application Program Interface*).

Debería estar perfectamente **documentada** para que cualquier potencial usuario tenga toda la información necesaria para poder usar el módulo sin tener que conocer o acceder a partes internas del mismo.

En general **debería estar formada únicamente por funciones** (y, tal vez, **constantes**) que el usuario del módulo pueda llamar para consumir los servicios que ofrece el módulo.

Esas funciones deben usarse como *abstracciones funcionales*, de forma que el usuario sólo necesite conocer las **especificaciones** de las funciones y no sus *implementaciones* concretas (el *cuerpo* o código de las funciones).

Acabamos de decir que la interfaz de un módulo debería estar formada únicamente por **funciones** (y, tal vez, *constantes*).

En teoría, la interfaz podría estar formada también por (algunas o todas las) **variables locales al módulo**, pero en la práctica eso no resulta apropiado, ya que cualquier cambio posterior en la representación interna de los datos almacenados en esas variables afectaría al resto de los módulos que acceden a dichas variables.

Más adelante estudiaremos este aspecto en profundidad cuando hablemos del **principio de ocultación de información**.

2.1.2. Implementación

La **implementación** es la parte del módulo que queda **oculta a los usuarios** del mismo.

Es decir: es la parte que los usuarios del módulo no necesitan (ni deben) conocer para poder usarlo adecuadamente.

Está formada por:

- Todas las **variables locales al módulo** que almacenan su estado interno.
- La **implementación (el *cuerpo*) de las funciones** que forman la interfaz.
- Las funciones que utiliza el propio módulo para gestionarse a sí mismo y que no forman parte de su interfaz (**funciones *auxiliares***).

La implementación debe poder cambiarse tantas veces como sea necesario sin que por ello se tenga que cambiar el resto del programa.

2.1.2.1. Resumen

$$\text{Interfaz del módulo} \begin{cases} \text{Especificación de funciones} \\ \text{Posibles constantes} \end{cases}$$

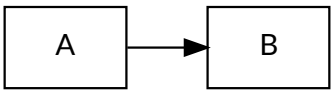
Implementación del módulo
- Implementación de funciones
- Variables locales
- Funciones auxiliares

2.2. Diagramas de estructura

Los diferentes módulos que forman un programa y la relación que hay entre ellos se puede representar gráficamente mediante un *diagrama de descomposición modular* o **diagrama de estructura**.

En el diagrama de estructura, cada módulo se representa mediante un rectángulo y las relaciones entre cada par de módulos se dibujan como una línea con punta de flecha entre los dos módulos.

Una flecha dirigida del módulo *A* al módulo *B* representa que el módulo *A* *utiliza* o *llama* o *depende* del módulo *B*.



A depende de *B*

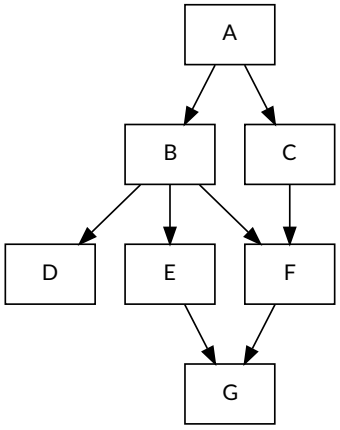


Diagrama de estructura

2.3. Programación modular en Python

En Python, un módulo es otra forma de llamar a un *script*. Es decir: «módulo» y «*script*» son sinónimos en Python.

Los módulos contienen instrucciones: definiciones y sentencias.

El nombre del archivo es el nombre del módulo con extensión `.py`.

Dentro de un módulo, el nombre del módulo (como cadena) se encuentra almacenado en la variable global `__name__`.

Cada módulo tiene su propio **ámbito local**, que es usado como el **ámbito global** de todas las instrucciones que componen el módulo.

Por tanto, el autor de un módulo puede crear variables globales o funciones en el módulo sin preocuparse de posibles colisiones accidentales con las variables globales o funciones de otros módulos.

2.3.1. Importación de módulos

Para que un módulo pueda usar a otros módulos tiene que **importarlos** usando la orden `import`. Por ejemplo, la siguiente sentencia importa el módulo `math` dentro del ámbito actual:

```
import math
```

Al importar un módulo de esta forma, lo que se hace es incorporar la definición del propio módulo (el módulo *importado*) en el ámbito actual del módulo o *script* que ejecuta el `import` (el módulo *importador*).

O dicho de otra forma: se incorpora al marco actual (es decir, el marco del ámbito donde se ejecuta el `import`) la ligadura entre el nombre del módulo importado y el propio módulo, por lo que el módulo importador puede acceder al módulo importado a través de su nombre, como una referencia al propio módulo.

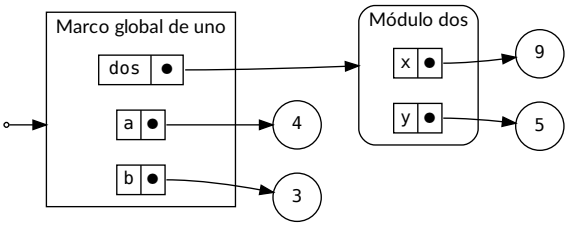
De esta forma, lo que se importa dentro del marco actual no es el contenido del módulo importado, sino una **referencia** al módulo.

Por ejemplo, si tenemos:

```
# uno.py

import dos

a = 4
b = 3
```

```
# dos.py

x = 9
y = 5
```

al final de la ejecución del script `uno.py` tendremos:



Importación del módulo `dos` en `uno`

Eso significa que **los módulos en Python son internamente un dato más**, al igual que las listas o las funciones: se pueden asignar a variables, se pueden borrar de la memoria con `del`, etc.

Significa, además, que los módulos tienen su propio ámbito.

Sin embargo, al importar un módulo como tal dentro del ámbito actual, no se crea un nuevo marco donde se almacenan sus definiciones.

En su lugar, se crea una estructura tipo diccionario que contiene las correspondencias entre los nombres de los elementos definidos y sus valores.

Esa estructura representa al módulo en memoria («*es*» el módulo), y se almacena en el montículo como cualquier otro dato.

El dato módulo permanecerá en memoria mientras haya una referencia que apunte a él, como siempre.

2.3.1.1. Espacios de nombres

Decimos que un módulo es un **espacio de nombres**.

Los espacios de nombres son estructuras que almacenan correspondencias entre un nombre y un valor, de forma que no haya nombres duplicados (o sea, nombres con más de un valor).

Hasta ahora, los únicos espacios de nombres que habíamos visto eran los **marcos** que se crean automáticamente cuando se entra en un cierto ámbito y se destruyen cuando se sale de él.

Los **módulos** también son espacios de nombres (como los marcos) porque tienen que almacenar sus definiciones locales, pero no se comportan como los marcos, sino como datos.

Los módulos se crean en el montículo cuando se importan y se destruyen como cualquier otro dato, cuando no hay más referencias que le apunten y el recolector de basura reclama su espacio.

En posteriores unidades veremos que hay otros espacios de nombres muy importantes: principalmente, las clases y los objetos.

Siempre que tengamos una referencia a un espacio de nombres, podemos acceder a los valores que contiene usando el operador punto (`.`), indicando el espacio de nombres y el nombre del elemento local al que queramos acceder:

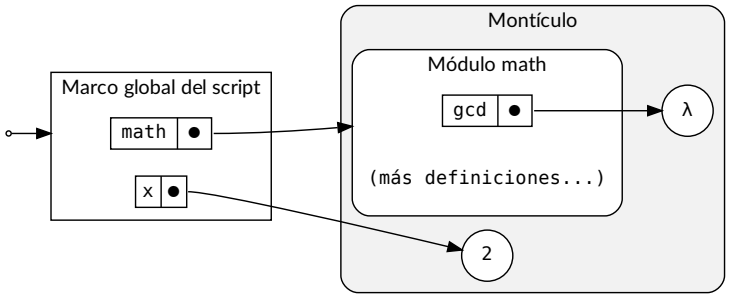
espacio_de_nombres `.` *elemento_local*

Por tanto, para acceder al contenido del módulo importado, indicaremos el nombre de ese módulo seguido de un punto (`.`) y el nombre del contenido al que queramos acceder:

módulo`.`*contenido*

Por ejemplo, para acceder a la función `gcd` definida en el módulo `math`, haremos:

```
import math

x = math.gcd(16, 6)
```



Entorno en la última línea del script anterior

Se recomienda (aunque no es obligatorio) colocar todas las sentencias `import` al principio del módulo importador.

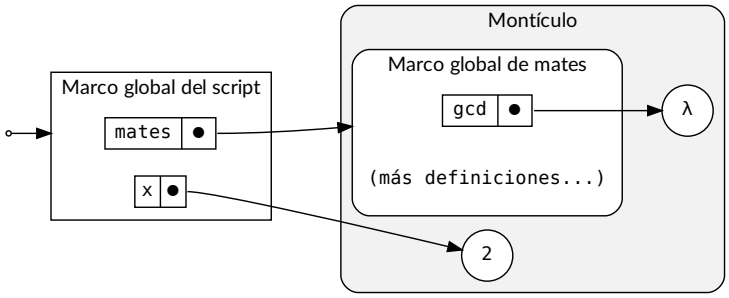
Se puede importar un módulo dándole al mismo tiempo otro nombre dentro del marco actual, usando la sentencia `import` con `as`.

Por ejemplo:

```
import math as mates
```

La sentencia anterior importa el módulo `math` dentro del módulo actual pero con el nombre `mates` en lugar del `math` original. Por tanto, para usar la función `gcd` como en el ejemplo anterior usaremos:

```
x = mates.gcd(16, 6)
```



Resultado de ejecutar las dos últimas líneas anteriores

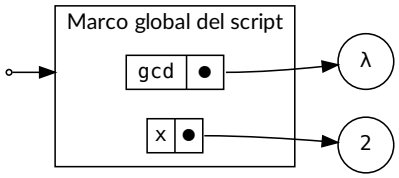
Existe una variante de la sentencia `import` que nos permite importar directamente las definiciones de un módulo en lugar del propio módulo. Para ello, se usa la orden `from`.

Por ejemplo, para importar sólo la función `gcd` del módulo `math`, y no el módulo en sí, haremos:

```
from math import gcd
```

Por lo que ahora podemos usar la función `gcd` directamente dentro del módulo importador, sin necesidad de indicar el nombre del módulo importado:

```
x = gcd(16, 6)
```



Resultado de ejecutar las dos últimas líneas anteriores

De hecho, ahora el módulo importado no está definido en el módulo importador (es decir, que en el marco global del módulo importador no hay ninguna ligadura con el nombre del módulo importado).

En nuestro ejemplo, eso significa que el módulo `math` no existe ahora como tal en el módulo importador, es decir, que ese nombre no está definido en el ámbito del módulo importador.

Por tanto, si hacemos:

```
x = math  # error
```

da error porque no hemos importado el módulo como tal, sino sólo una de sus funciones.

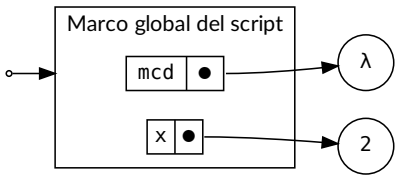
También podemos usar la palabra clave `as` con la orden `from`:

```
from math import gcd as mcd
```

De esta forma, se importa en el módulo actual la función `gcd` del módulo `math` pero llamándola `mcd`.

Por tanto, para usarla la invocaremos con su nuevo nombre:

```
x = mcd(16, 6)
```



Resultado de ejecutar las dos últimas líneas anteriores

Existe incluso una variante para importar todas las definiciones de un módulo:

```
from math import *
```

Con esta sintaxis importaremos todas las definiciones del módulo excepto aquellas cuyo nombre comience por un guión bajo (_).

Las definiciones con nombres que comienzan por _ son consideradas **privadas** o internas al módulo, lo que significa que no están concebidas para ser usadas por los usuarios del módulo y que, por tanto, no forman parte de su **interfaz**.

En general, los programadores no suelen usar esta funcionalidad ya que puede introducir todo un conjunto de definiciones desconocidas dentro del módulo importador, lo que incluso puede provocar que se «*machaquen*» definiciones ya existentes.

Para saber qué definiciones contiene un módulo, se puede usar la funcion `dir()`:

```
>>> import math
>>> dir(math)
['__doc__', '__loader__', '__name__', '__package__', '__spec__', 'acos',
'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'copysign',
'cos', 'cosh', 'degrees', 'e', 'erf', 'erfc', 'exp', 'expm1', 'fabs',
'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd', 'hypot',
'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'ldexp', 'lgamma', 'log',
'log10', 'log1p', 'log2', 'modf', 'nan', 'pi', 'pow', 'radians', 'sin',
'sinh', 'sqrt', 'tan', 'tanh', 'tau', 'trunc']
```

La función `dir()` puede usarse con cualquier espacio de nombres al que podamos acceder mediante una referencia (por tanto, también valdrá para clases y objetos cuando los veamos).

2.3.2. Módulos como *scripts*

Un módulo puede contener sentencias ejecutables además de definiciones.

Generalmente, esas sentencias existen para inicializar el módulo.

Las sentencias de un módulo se ejecutan sólo la primera vez que se encuentra el nombre de ese módulo en una sentencia `import`.

También se ejecutan si el archivo se ejecuta como un *script*.

Cuando se ejecuta un módulo Python desde la línea de órdenes como:

```
$ python3 fact.py <argumentos>
```

se ejecutará el código del módulo como si fuera un *script* más, igual que si se hubiera importado con un `import` dentro de otro módulo, pero con la diferencia de que la variable global `__name__` contendrá el valor `"__main__"`.

Eso significa que si se añade este código al final del módulo:

```
if __name__ == "__main__":
    <sentencias>
```

el módulo podrá funcionar como un *script* independiente.

Por ejemplo, supongamos el siguiente módulo `fact.py`:

```
def fac(n):
    if n == 0:
        return 1
    else:
        return n * fac(n - 1)

if __name__ == "__main__":
    import sys
    print(fac(int(sys.argv[1])))
```

Este módulo se podrá usar como un *script* separado o como un módulo que se pueda importar dentro de otro.

Si se usa como *script*, podremos llamarlo desde la línea de órdenes del sistema operativo:

```
$ python3 fac.py 4
24
```

Y si importamos el módulo dentro de otro, el código del último `if` no se ejecutará, por lo que sólo se incorporará la definición de la función `fac`.

3. Criterios de descomposición modular

3.1. Introducción

No existe una única forma de descomponer un programa en módulos (entiendo aquí por *módulo* cualquier parte del programa en sentido amplio, incluyendo una simple función).

Las diferentes formas de dividir el sistema en módulos traen consigo diferentes requisitos de comunicación y coordinación entre las personas (o equipos) que trabajan en esos módulos, y ayudan a obtener los beneficios descritos anteriormente en mayor o menor medida.

Nos interesa responder a las siguientes preguntas:

- ¿Qué criterios se deben seguir para dividir el programa en módulos?
- ¿Qué módulos debe tener nuestro programa?
- ¿Cuántos módulos debe tener nuestro programa?
- ¿De qué tamaño deben ser los módulos?

3.2. Tamaño y número

Se supone que, si seguimos al pie de la letra la estrategia de diseño basada en la división de problemas, sería posible concluir que si el programa se dividiera indefinidamente, cada vez se necesitaría menos esfuerzo hasta llegar a cero.

Evidentemente, esto no es así, ya que hay otras fuerzas que entran en juego.

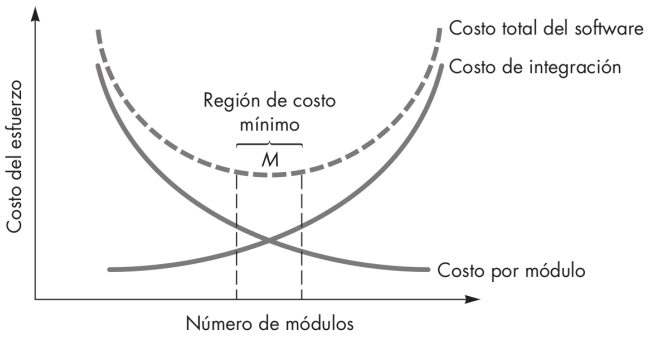
El coste de desarrollar un módulo individual disminuye conforme aumenta el número de módulos.

Dado el mismo conjunto de requisitos funcionales, cuantos más módulos hay, más pequeños son.

Sin embargo, cuantos más módulos hay, más cuesta integrarlos.

El tamaño de cada módulo debe ser el adecuado: si es demasiado grande, será difícil hacer cambios en él; si es demasiado pequeño, no merecerá la pena tratarlo como un módulo, sino más bien como parte de otros módulos.

El valor *M* es el número de módulos ideal, ya que reduce el coste total del desarrollo.



Esfuerzo frente al número de módulos

Las curvas de la figura anterior constituyen una guía útil al considerar la modularidad.

Debemos evitar hacer pocos o muchos módulos para así permanecer en la cercanía de M.

Pero, ¿cómo saber cuál es la cercanía de M? ¿Cómo de modular debe hacerse el programa?

Debe hacerse un diseño con módulos, de manera que el desarrollo pueda planearse con más facilidad, que los cambios se realicen con más facilidad, que las pruebas y la depuración se efectúen con mayor eficiencia y que el mantenimiento a largo plazo se lleve a cabo sin efectos indeseados de importancia.

Para ello nos basaremos en los siguientes **criterios**.

3.3. Abstracción

La **abstracción** es un proceso mental que se basa en estudiar un aspecto del problema a un determinado nivel centrándose en lo esencial e ignorando momentáneamente los detalles que no son importantes en este nivel.

Igualmente, nos permite comprender la esencia de un subsistema sin tener que conocer detalles innecesarios del mismo.

La utilización de la abstracción también permite trabajar con conceptos y términos que son familiares en el entorno del problema sin tener que transformarlos en una estructura no familiar.

La abstracción se usa principalmente como una técnica de **manejo y control de la complejidad**.

Cuando se considera una solución modular a cualquier problema, se pueden definir varios **niveles de abstracción**:

- En niveles **más altos** de abstracción, se enuncia una solución en términos más generales usando el lenguaje del entorno del problema.

  A estos niveles hay menos elementos de información, pero más grandes e importantes.

- En niveles **más bajos** de abstracción se da una descripción más detallada de la solución.

  A estos niveles se revelan más detalles, aparecen más elementos y se aumenta la cantidad de información con la que tenemos que trabajar.

La barrera de separación entre un nivel de abstracción y su inmediatamente inferior es la diferencia entre el *qué* y el *cómo*:

- Cuando estudiamos un concepto a un determinado nivel de abstracción, estudiamos *qué* hace.
- Cuando bajamos al nivel inmediatamente inferior, pasamos a estudiar *cómo* lo hace.

Esta división o separación puede continuar en niveles inferiores, de forma que siempre puede considerarse que cualquier nivel responde al *qué* y el nivel siguiente responde al *cómo*.

Recordemos que un módulo tiene siempre un doble punto de vista:

- El punto de vista del *creador* o implementador del módulo.
- El punto de vista del *usuario* del módulo.

La abstracción nos ayuda a **definir qué módulos constituyen nuestro programa** considerando la relación que se establece entre los *creadores* y los *usuarios* de los módulos.

Esto es así porque **los usuarios de un módulo quieren usar a éste como una abstracción**: sabiendo *qué* hace (su función) pero sin necesidad de saber *cómo* lo hace (sus detalles internos).

El responsable del *cómo* es únicamente el **creador** del módulo.

Los módulos definidos como abstracciones son más fáciles de usar, diseñar y mantener.

3.4. Ocultación de información

David Parnas introdujo el **principio de ocultación de información** en 1972.

Afirmó que el criterio principal para la modularización de un sistema debe ser la **ocultación de *decisiones de diseño* complejas o que puedan cambiar en el futuro**, es decir, que los módulos se deben caracterizar por ocultar *decisiones de diseño* a los demás módulos.

> Por tanto: todos los elementos que necesiten conocer las mismas *decisiones de diseño*, deben pertenecer al mismo módulo.

Al ocultar la información de esa manera se evita que los usuarios de un módulo necesiten de un conocimiento íntimo del diseño interno del mismo para poder usarlo, y los aísla de los posibles efectos de cambiar esas decisiones posteriormente.

Implica que la modularidad efectiva se logra **definiendo un conjunto de módulos independientes que intercambien sólo aquella información estrictamente necesaria para que el programa funcione**.

Dicho de otra forma:

- **Para que un módulo *A* pueda usar a otro *B*, *A* tiene que conocer de *B* lo menos posible**, lo imprescindible.

  > El uso del módulo debe realizarse únicamente por medio de **interfaces** bien definidas que no cambien (o cambien poco) con el tiempo y que no expongan detalles internos al exterior.

- Por tanto, *B* debe **ocultar** al exterior sus detalles internos de **implementación** y exponer sólo lo necesario para que otros lo puedan utilizar.

  Ésto aísla a los usuarios de los posibles cambios internos que pueda haber posteriormente en *B*.

> Es decir: cada módulo debe ser una **caja negra** recelosa de su privacidad que tiene «aversión» por exponer sus interioridades a los demás.

La **abstracción** y la **ocultación de información** se complementan:

- La **ocultación de información** es un **principio de diseño** que se basa en que los módulos deben ocultar a los demás módulos sus decisiones de diseño y exponer sólo la información estrictamente necesaria para que los demás puedan usarlos.
- La **abstracción** puede usarse como una **técnica de diseño** que nos ayuda a cumplir con el principio de ocultación de información, porque nos permite descomponer el programa en módulos

y **nos ayuda a identificar qué detalles hay que ocultar** y qué información hay que exponer a los demás.

Al **usuario** de un módulo…

- … le interesa la **abstracción** porque le permite usar el módulo sabiendo únicamente *qué* hace sin tener que saber *cómo* lo hace.
- … le interesa la **ocultación de información** porque cuanta menos información necesite conocer para usar el módulo, más fácil y cómodo le resultará usarlo.

Al **creador** de un módulo…

- … le interesa la **abstracción** como técnica porque le ayuda a determinar qué información debe ocultar su módulo al exterior.
- … le interesa la **ocultación de información** porque cuantos más detalles necesiten conocer los usuarios para poder usar su módulo, menos libertad dentrá de poder cambiar esos detalles en el futuro (cuando lo necesite o cuando lo desee) sin afectar a los usuarios de su módulo.

3.5. Independencia funcional

La independencia funcional se logra desarrollando módulos de manera que cada módulo resuelva una funcionalidad específica y tenga una interfaz sencilla cuando se vea desde otras partes de del programa (idealmente, mediante paso de parámetros).

- De hecho, la interfaz del módulo debe estar destinada únicamente a cumplir con esa funcionalidad.

Al limitar su objetivo, el módulo necesita menos ayuda de otros módulos.

Y por eso el módulo debe ser tan independiente como sea posible del resto de los módulos del programa, es decir, que dependa lo menos posible de lo que hagan otros módulos, y también que dependa lo menos posible de los datos que puedan facilitarle otros módulos.

Dicho de otra forma: los módulos deben centrarse en resolver un problema concreto (ser «monotemáticos»), deben ser «antipáticos» y tener «aversión» a relacionarse con otros módulos.

Los módulos independientes son más fáciles de desarrollar porque la función del programa se subdivide y las interfaces se simplifican, por lo que se pueden desarrollar por separado.

Los módulos independientes son más fáciles de mantener y probar porque los efectos secundarios causados por el diseño o por la modificación del código son más limitados, se reduce la propagación de errores y es posible obtener módulos reutilizables.

De esta forma, la mayor parte de los cambios y mejoras que haya que hacer al programa implicarán modificar sólo un módulo o un número muy pequeño de ellos.

Es un objetivo a alcanzar para obtener una modularidad efectiva.

La independencia funcional se mide usando dos métricas: la **cohesión** y el **acoplamiento**.

El objetivo de la independencia funcional es **maximizar la cohesión y minimizar el acoplamiento**.

3.5.1. Cohesión

La **cohesión** mide la fuerza con la que se relacionan los componentes de un módulo.

Cuanto más cohesivo sea un módulo, mejor será nuestro diseño modular.

Un módulo cohesivo realiza una sola función, por lo que requiere interactuar poco con otros componentes en otras partes del programa.

En un módulo cohesivo, sus componentes están fuertemente relacionados entre sí y pertenencen al módulo por una razón lógica (no están ahí por casualidad), es decir, todos cooperan para alcanzar un objetivo común que es la función del módulo.

Un módulo cohesivo mantiene unidos (*atrae*) los componentes que están relacionados entre ellos y mantiene fuera (*repele*) el resto.

En pocas palabras, un módulo cohesivo debe tener un único objetivo, y todos los elementos que lo componen deben contrubuir a alcanzar dicho objetivo.

Aunque siempre debe tratarse de lograr mucha cohesión (por ejemplo, una sola tarea), con frecuencia es necesario y aconsejable hacer que un módulo realice varias tareas, siempre que contribuyan a una misma finalidad lógica.

Sin embargo, para lograr un buen diseño hay que evitar módulos que llevan a cabo funciones no relacionadas.

La siguiente es una escala de grados de cohesión, ordenada de mayor a menor:

- Cohesión funcional
- Cohesión secuencial
- Cohesión de comunicación

  [Hasta aquí, los módulos se consideran **cajas negras**.]
- Cohesión procedimental
- Cohesión temporal
- Cohesión lógica
- Cohesión coincidental

No hace falta determinar con precisión qué cohesión tenemos. Lo importante es intentar conseguir una cohesión alta y reconocer cuándo hay poca cohesión para modificar el diseño y conseguir una mayor independencia funcional.

Cohesión funcional: se da cuando los componentes del módulo pertenecen al mismo porque todos contribuyen a una tarea única y bien definida del módulo.

Cohesión secuencial: se da cuando los componentes del módulo pertenecen al mismo porque la salida de uno es la entrada del otro, como en una cadena de montaje (por ejemplo, una función que lee datos de un archivo y los procesa).

Cohesión de comunicación: se da cuando los componentes del módulo pertenecen al mismo porque trabajan sobre los mismos datos (por ejemplo, un módulo que procesa números racionales).

Cohesión procedimental: se da cuando los componentes del módulo pertenecen al mismo porque siguen una cierta secuencia de ejecución (por ejemplo, una función que comprueba los permisos de un archivo y después lo abre).

Cohesión temporal: se da cuando los componentes del módulo pertenecen al mismo porque se ejecutan en el mismo momento (por ejemplo, una función que se dispara cuando se produce un error, abriría un archivo, crearía un registro de error y notificaría al usuario).

Cohesión lógica: se da cuando los componentes del módulo pertenecen al mismo porque pertenencen a la misma categoría lógica aunque son esencialmente distintos (por ejemplo, un módulo que agrupe las funciones de manejo del teclado o el ratón).

Cohesión coincidental: se da cuando los componentes del módulo pertenecen al mismo por casualidad o por razones arbitrarias, es decir, que la única razón por la que se encuentran en el mismo módulo es porque se han agrupado juntos (por ejemplo, un módulo de «utilidades»).

3.5.2. Acoplamiento

El **acoplamiento** es una medida del grado de interdependencia entre los módulos de un programa.

Dicho de otra forma, es la fuerza con la que se relacionan los módulos de un programa.

El acoplamiento depende de:

- La complejidad de la interfaz entre los módulos
- El punto en el que se entra o se hace referencia a un módulo
- Los datos que se pasan a través de la interfaz

Lo deseable es tener módulos con poco acoplamiento, es decir, módulos que dependan poco unos de otros.

De esta forma obtenemos programas más fáciles de entender y menos propensos al *efecto ola*, que ocurre cuando se dan errores en un sitio y se propagan por todo el programa.

Los programas con alto acoplamiento tienden a presentar los siguientes problemas:

- Un cambio en un módulo normalmente obliga a cambiar otros módulos (consecuencia del *efecto ola*).
- Requiere más esfuerzo integrar los módulos del programa ya que dependen mucho unos de otros.
- Un módulo particular resulta más difícil de reutilizar o probar debido a que hay que incluir en el lote a los módulos de los que depende éste (no se puede reutilizar o probar por separado).

La siguiente es una escala de grados de acoplamiento, ordenada de mayor a menor:

- Acoplamiento por contenido
- Acoplamiento común
- Acoplamiento externo
- Acoplamiento de control
- Acoplamiento por estampado

- Acoplamiento de datos
- Sin acoplamiento

No hace falta determinar con precisión qué cohesión tenemos. Lo importante es intentar conseguir una cohesión alta y reconocer cuándo hay poca cohesión para modificar el diseño y conseguir una mayor independencia funcional.

Acoplamiento por contenido: ocurre cuando un módulo modifica o se apoya en el funcionamiento interno de otro módulo (por ejemplo, accediendo a datos locales del otro módulo). Cambiar la forma en que el segundo módulo produce los datos obligará a cambiar el módulo dependiente.

Acoplamiento común: ocurre cuando dos módulos comparten los mismos datos globales. Cambiar el recurso compartido obligará a cambiar todos los módulos que lo usen.

Acoplamiento externo: ocurre cuando dos módulos comparten un formato de datos impuesto externamente, un protocolo de comunicación o una interfaz de dispositivo de entrada/salida.

Acoplamiento de control: ocurre cuando un módulo controla el flujo de ejecución del otro (por ejemplo, pasándole un *conmutador* booleano).

Acoplamiento por estampado: ocurre cuando los módulos comparten una estructura de datos compuesta y usan sólo una parte de ella, posiblemente una parte diferente. Esto podría llevar a cambiar la forma en la que un módulo lee un dato compuesto debido a que un elemento que el módulo no necesita ha sido modificado.

Acoplamiento de datos: ocurre cuando los módulos comparten datos entre ellos (por ejemplo, parámetros). Cada dato es una pieza elemental y dicho parámetro es la única información compartida (por ejemplo, pasando un entero a una función que calcula una raíz cuadrada).

Sin acoplamiento: ocurre cuando los módulos no se comunican para nada uno con otro.

3.6. Reusabilidad

La **reusabilidad** es un factor de calidad del software que se puede aplicar tambien a sus componentes o módulos.

Un módulo es **reusable** cuando puede aprovecharse para ser utilizado (tal cual o con muy poca modificación) en varios programas.

A la hora de diseñar módulos (o de descomponer un programa en módulos) nos interesa que los módulos resultantes sean cuanto más reusables mejor.

Para ello, el módulo en cuestión debe ser lo suficientemente general y resolver un problema patrón que sea suficientemente común y se pueda encontrar en varios contextos y programas diferentes.

Además, para aumentar la reusabilidad, es conveniente que el módulo tenga un bajo acoplamiento y que, por tanto, no dependa de otros módulos del programa.

Esos módulos incluso podrían luego formar parte de una *biblioteca* o *repositorio* de módulos y ponerlos a disposición de los programadores para que puedan usarlos en sus programas.

A día de hoy, el desarrollo de programas se basa en gran medida en seleccionar y utilizar módulos (o bibliotecas, *librerías* o *paquetes*) desarrollados por terceros o reutilizados de otros programas elaborados por nosotros mismos anteriormente.

Es decir: la programación se ha convertido en una actividad consistente principalmente en ir combinando componentes intercambiables.

Eso nos permite acortar el tiempo de desarrollo porque podemos construir un programa a base de ir ensamblando módulos reusables como si fueran las piezas del engranaje de una máquina.

Bibliografía

Pressman, Roger S, Darrel Ince, Rafael Ojeda Martín, and Luis Joyanes Aguilar. 2004. *Ingeniería Del Software: Un Enfoque Práctico*. Madrid: McGraw-Hill.

Python Software Foundation. n.d. “Sitio Web de Documentación de Python.” https://docs.python.org/3.