Programación modular I

Ricardo Pérez López

IES Doñana, curso 2019/2020

1. Introducción

1.1. Modularidad

1.1. Modularidad

- La **programación modular** es una técnica de programación que consiste en descomponer y programar nuestro programa en partes llamadas **módulos**.
- Equivale a la técnica clásica de resolución de problemas basada en descomponer un problema en subproblemas y resolver cada subproblema por separado 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 de forma similar a como lo hace la abstracción.
- 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**.

- **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 descomponer físicamente 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, podríamos considerar que una función es un ejemplo de módulo, ya que se ajusta a esa definición (salvo 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 funcional no resulta adecuado en general, ya que muchas veces nos encontramos con funciones que no actúan por separado, sino de forma conjunta formando un todo interrelacionado.
- Además, un módulo no tiene por qué ser simplemente una abstracción funcional, sino que también puede tener su propio estado interno, manipulable desde dentro del módulo pero 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... Y todas esas funciones trabajarían conjuntamente, actuando sobre la misma colección de datos (la representación interna que usa el módulo para implementar los números racionales).
- Esos datos (es decir, esa representación interna) también formarían parte del módulo y constituirían su estado interno.
- Por consiguiente, aunque resulta muy apropiado considerar cada función anterior por separado, también resulta evidente que debemos considerarlas como formando un todo conjunto con los datos que manipulan: el *módulo de manipulación de números racionales*.

- De lo dicho hasta ahora se deducen varias conclusiones importantes:
  - Un módulo es una parte del programa.
  - Los módulos nos permiten descomponer el programa en **partes independientes y manejables por separado**.
  - Una función no es una candidata con suficiente entidad como para ser considerada un módulo, en general.
  - Los módulos, generalmente, agrupan colecciones de **funciones interrelacionadas**.
  - Los módulos, generalmente, también poseen un **estado interno** en forma de estructuras de datos, manipulable desde el interior del módulo así como desde el exterior del mismo usando las funciones que forman el módulo.
  - A nivel práctico, los módulos se programan físicamente en **archivos separados** del resto del programa.

1.2. Beneficios de la programación modular

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

2.1.1 Interfaz

2.1.2 Implementación

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**:
  - La **implementación** del módulo se ocupa de la programación de los detalles internos al módulo, necesarios para que éste funcione
  - Los **usuarios** del módulo (normalmente, otros módulos), como entidades externas al mismo, utilizan el 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.

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

- 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 funciones que permiten manipular y acceder al estado interno desde fuera del módulo.
  - Una **implementación**, formada por:
    - Su estado interno en forma de variables internas y locales al módulo.

      Como contiene variables, también puede contener constantes.
    - Un conjunto de funciones que manipulan el estado interno dentro del módulo.

      Es decir: funciones pensadas para ser usadas internamente por el propio módulo pero no por otras partes del programa.

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.
- A menudo 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 la **especificación** de la función y no su *implementación* concreta.

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 formado por todas las variables locales al módulo que almacenan su estado interno, junto con las funciones que utiliza el propio módulo para gestionarse a sí mismo y que no forman parte de su interfaz.
- La implementación debe poder cambiarse tantas veces como sea necesario sin que por ello se tenga que cambiar el resto del programa.

2.2. Programación modular en Python

2.2. 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 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 funciones definidas en el módulo.
- Por tanto, el autor de un módulo puede usar variables globales en el módulo sin preocuparse de posibles colisiones accidentales con las variables globales del usuario del módulo.

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 módulo actual:

```
import math
```

- Al importar un módulo de esta forma, lo que se hace es incorporar la definición del propio módulo (el módulo *importado*) dentro del ámbito global del módulo o *script* actual (el módulo *importador*).
- O dicho de otra forma: se incorpora al marco global del módulo importador 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.
- De esta forma, lo que se importa dentro del marco global actual no es el conjunto de las definiciones que forman el módulo importado, sino el módulo en sí.

- 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.
- Podemos acceder al contenido de un módulo importado de esta forma, indicando el nombre del módulo seguido de un punto (`.`) delante del nombre del contenido al que queramos acceder.
- Por ejemplo, para acceder a la función `gcd` definido en el módulo `math` haremos:

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

- 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 la palabra clave `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)
```

- 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)
```

- 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.
- 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)
```

- 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.

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`:

```python
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

3.1. Introducción

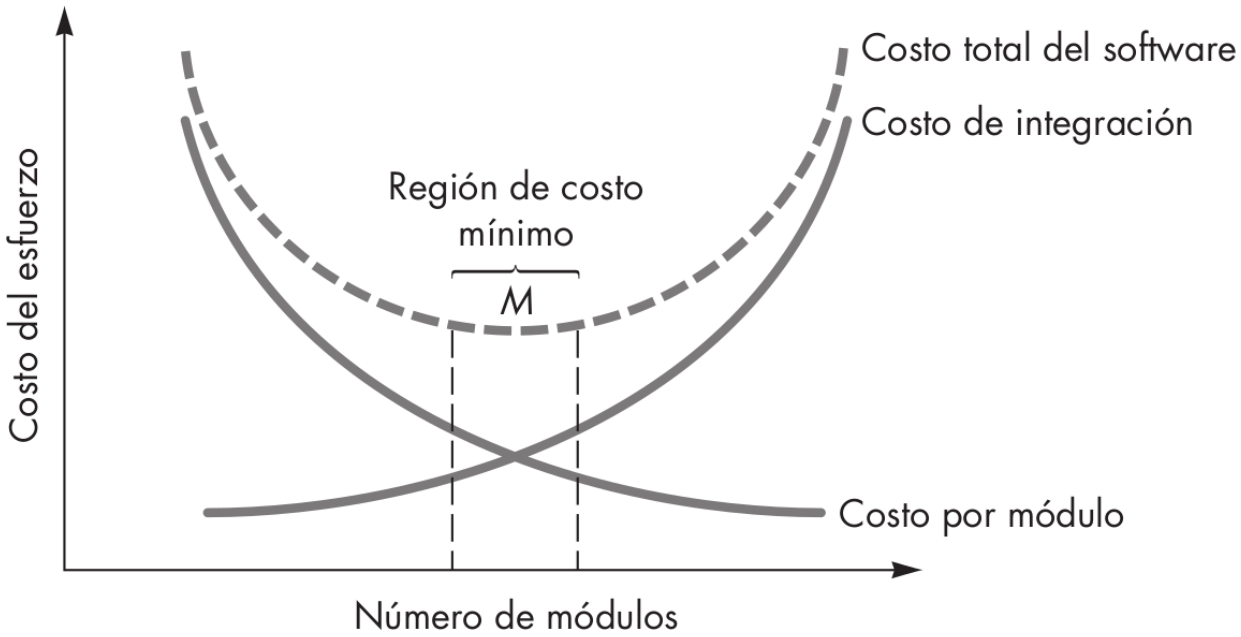
- No existe una única forma de descomponer un programa en módulos.
- Las diferentes formas de dividir el sistema en módulos traen consigo diferentes requisitos de comunicación y coordinación entre las personas (o equipos) que trabajan en esos módulos, y ayudan a obtener los beneficios descritos anteriormente en mayor o menor medida.
- ¿Qué criterios se deben seguir para dividir el programa en módulos?
- ¿Cuántos módulos debe tener nuestro programa?
- ¿De qué tamaño deben ser los módulos?

3.2. Tamaño y número

3.2. Tamaño y número

- Según el punto de vista de 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 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.
- Deben hacerse módulos, pero con cuidado para permanecer en la cercanía de M.
- Debe evitarse hacer pocos o muchos módulos.
- Pero, ¿cómo saber cuál es la cercanía de M? ¿Cuán modular debe hacerse el software?
- 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

3.3. Abstracción

- Cuando se considera una solución modular a cualquier problema, se pueden definir varios niveles de abstracción:
  - En el nivel más alto de abstracción, se enuncia una solución en términos más generales usando el lenguaje del entorno del problema.
  - En niveles más bajos de abstracción se da una descripción más detallada de la solución.
- La noción psicológica de **abstracción** permite concentrarse en un problema a algún nivel de generalización sin tener en consideración los datos irrelevantes de bajo nivel.
- Es decir: 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.

- 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.
- 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 entidades 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 éste como una abstracción: sabiendo qué hace (su función) pero sin necesidad de saber cómo lo hace (sus detalles internos).
- Los módulos definidos como abstracciones son más fáciles de usar, diseñar y mantener.

3.4. Ocultación de información

3.4. Ocultación de información

- David Parnas introdujo el concepto 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**, es decir, que los módulos se deben caracterizar por ocultar decisiones de diseño a los demás módulos.
- Ocultar la información de esa manera aísla a los usuarios de un módulo de necesitar un conocimiento íntimo del diseño interno del mismo para poder usarlo.
- También 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.

- La abstracción ayuda a definir qué entidades constituyen el software.
- La ocultación define y hace cumplir las restricciones de acceso a los componentes de un módulo; es decir: define qué cosas deben ser públicas y qué no.
- En definitiva: que **para que un módulo *A* pueda usar a otro *B*, *A* tiene que conocer de *B* lo menos posible**, lo imprescindible. *B* debe ocultar al exterior los detalles internos de implementación y exponer sólo lo necesario para que otros lo puedan utilizar. Ésto aísla a los clientes de los posibles cambios internos que pueda haber posteriormente en *B*.
- Cada módulo es una **caja negra** recelosa de su privacidad que tiene «aversión» por exponer sus interioridades a los demás.

- La diferencia entre la **abstracción** y la **ocultación de información** se puede resumir también de la siguiente forma:
  - Al **usuario** de un módulo le interesa la **abstracción** porque le permite usar el módulo conociendo de él sólo lo imprescindible e ignorando los detalles superfluos de implementación.
  - Al **creador** del módulo le interesa el **principio de ocultación de información** porque si los usuarios de su módulo conocen más detalles de los necesarios para poder usarlo, el creador no podrá cambiarlos luego (cuando lo necesite o cuando lo desee) sin afectar a los usuarios de su módulo.

3.5. Independencia funcional

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.
- 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**.

3.6. Reusabilidad

4. Diagramas de estructura