Buenas prácticas de programación

Gisela Ramos, Lic. Manuel Camejo, Charly Talavera, Dr. Mariano Mendez
Facultad de Ingeniería, Universidad de Buenos Aires
26 de agosto de 2018

1. El Software

La palabra software tiene origen en el préstamo del inglés software, compuesto por *soft* = blando y *ware* = utensilios, objetos. Término creado por los ingenieros informáticos americanos por analogía a hardware (V.). Según la RAE (Real Academia Española) el **software** es un conjunto de programas, instrucciones y reglas informáticas para ejecutar ciertas tareas en una computadora.

Si bien su origen y definición son correctos, el software va mucho más allá de ese pequeño párrafo. Una muy buena caracterización de lo que es el software se encuentra en el artículo de F. Brooks, cuyo título es **No Hay balas de Plata**. En ese hermoso artículo Brooks hace un análisis desde el punto de vista aristotélico de lo que es el software. Aristóteles sostiene en Metafísica, Libro Quinto, XXX :

> *Accidente se dice de lo que se encuentra en un ser y puede afirmarse con verdad, pero que no es, sin embargo, ni necesario ni ordinario. Supongamos que cavando un hoyo para poner un árbol, se encuentra un tesoro. Es accidental que el que cava un hoyo encuentre un tesoro; porque ni es lo uno consecuencia ni resultado necesario del otro, ni es ordinario tampoco que plantando un árbol se encuentre un tesoro. Supongamos también, que un músico sea blanco; como no es ni necesario, ni general, a esto llamamos accidente. Por tanto, si sucede una cosa, cualquiera que ella sea, a un ser, aun en ciertas circunstancias de lugar y de tiempo, pero sin que haya causa que determine su esencia, sea actualmente, sea en tal lugar, esta cosa será un accidente. El accidente no tiene pues ninguna causa determinada tiene sólo una causa fortuita; y lo fortuito es lo indeterminado. Por accidente se arriba a Egina, cuando no se hizo ánimo de ir allí, sino que le ha llevado a uno la tempestad o los piratas. El accidente se produce, existe, pero no tiene la causa en sí mismo, y sólo existe en virtud de otra cosa. La tempestad ha sido causa de que hayáis arribado a donde no queríais, y este punto es Egina. La palabra accidente se entiende también de otra manera; se dice de lo que existe de suyo en un objeto, sin ser uno de los caracteres distintivos de su esencia: tal es la propiedad del triángulo, de que sus tres ángulos valgan dos ángulos rectos. Estos accidentes pueden ser eternos; los accidentes propiamente dichos no lo son; ya hemos dado la razón de esto en otra parte.*

En pocas palabras, para decir que algo es hay que definir su esencia y su accidente. Con este enfoque, F. Brooks define a la esencia del software con 4 propiedades:

1. **Complejidad** Las entidades de software son más complejas debido a su tamaña que, posiblemente, cualquier otra construcción humana porque no hay dos partes iguales (al menos por encima del nivel de instrucción). Si la hay, podemos convertir a las dos partes similares en una subrutina, abierta o cerrada. A este respecto, los sistemas de software difieren profundamente de los ordenadores, edificios o automóviles, donde abundan los elementos repetidos.

2. **Conformidad** La gente del Software no es la única que encara la complejidad. Los físicos tratan con objetos terriblemente complejos, incluso a los niveles "fundamentales"de la física de partículas. Las labores del físico descansan, sin embargo, en una profunda fe de que hay principios unificadores que deben encontrarse, o en los quarks o en las teorías de campo unificado. Einstein argumentaba que deben haber explicaciones simplificadas de la naturaleza, porque Dios no es caprichoso ni arbitrario.

3. **Variabilidad** El software está constantemente sometido a presiones para cambiarlo. Por supuesto, esto también afecta a los edificio, coches u ordenadores. Pero las cosas manufacturadas no cambian casi nunca una vez fabricadas; son sobrepasadas por modelos posteriores, o cambios esenciales se

incorporan en copias de número de serie superior del mismo diseño básico. Es raro que un automóvil deba ser revisado por un defecto de diseño; cambios en los ordenadores una vez que están funcionando, son algo más frecuentes. Pero son muchísimo menos frecuentes que cambios en el software ya en funcionamiento.

4. **Invisibilidad** El software es invisible e invisualizable. La realidad del software no es un algo espacial. Aquí, no hay representaciones geométricas como las que que tiene la tierra en forma de mapas, los chips de silicio en forma de diagramas o los computadores en esquemas de conexiones. En el momento que intentamos crear diagramas de las estructuras de software, descubrimos que no está constituido por uno, sino por varios gráficos en distintas direcciones, superpuestos unos sobre otros. Los diversos gráficos pueden representar el flujo de control, el flujo de datos, patrones de dependencia, secuencias temporales, relaciones entre los nombres. Estos gráficos no suelen ser planares y mucho menos jerárquicos. De hecho, una de establecer control conceptual sobre estas estrucutras es forzar que todo encaje sobre uno (o más de uno) gráficos jerárquicos.

Teniendo en cuenta estos aspectos **esenciales del software**, los programadores deben siempre enfocarse en reducir lo más posible parte alguno de ellos. Hay algunos como la Invisibilidad que no pueden ser tratados, el software es invisible e intangible y no se puede hacer nada contra ello, así como también no se puede trabajar sobre el aspecto de conformidad. Por otro lado la complejidad y la variabilidad son dos aspectos que pueden ser atacados y se pueden reducir o por lo menos atenuar los resultados que estos causan al software.

En las próximas secciones daremos algunos principios básicos que ayudan a reducir la complejidad y la variabilidad del software en alguna medida. Estos principios surgen de la experiencia de muchas personas y han sido recopilados para que puedan aprenderlos en una etapa temprana de la formación como programadores.

2. Nombres Descriptivos

Existen **dos** *cosas difíciles en la Informática:*
1- Poner buenos nombres
2- La Concurrencia
3-Errarle por uno

Se recomienda usar nombres descriptivos, tanto para nombrar funciones como variables, ya que le aporta legibilidad y claridad al código. Esto ayudará también si se trabaja en grupo, el compañero (y uno mismo) sabrá para qué fue declarada la variable/función.

No es recomendable usar nombres muy largos, si se necesita agregar una explicación para la variable, se agrega un comentario breve en la misma línea.

Mala práctica

```
#define MAX 30
#define ELEMENTO 'H'
```

Buena práctica

```
#define MAX_NOMBRE 30
#define HIDROGENO 'H'
```

Los nombres son demasiado genéricos. ELEMENTO puede ser cualquier cosa y MAX también. sus nombres al ser tan genéricos dificultan sin un contexto determinar que representan.

Mala práctica

```
bool funcion1(int numero){
    return (numero%2 == 0);
}

```

Buena práctica

```
bool es_impar(int numero){
    return (numero%2 == 0);
}
```

Mala práctica

```
bool funcion_que_calcula_si_un_numero_es_par(int numero){
    return (numero%2 == 0);
}
```

Es visiblemente evidente que la longitud del nombre de la función es bien auto-descriptivo, pero seguramente se volverá una tortura tener que escribir ese nombre mas de una vez. Entonces, ¿Cuán largo debe ser el nombre de una función? El nombre de una función debe seguir los mismo lineamientos que los nombres de las variables pero además el mismo debe indicar una acción atómica. Con atómica se refiere que lo que la funcion diga hacer debe ser solamente un concepto.

Mala práctica

```
int x;
int xx;
char v[MAX];

```

Buena práctica

```
int i;
int contador_personas;
char nombre_alumno[MAX_NOMBRE];

```

En este ejemplo es evidente que los nombres de las variables del lado izquierdo del código no aportan información sobre lo que la variable almacena.

2.1. Reglas para Poner Nombres

Los siguientes son lineamientos de como deben ser los nombres de las variables, Gracias Uncle Bob !! (Robert C. Martin) por tu libro Clean Code:

1. **Nombres Reveladores**: Esto significa que los nombres deben aportar significado a las variables. Por ende las variables del estilo `x`, `xx`, `xxx`, `xxxx`, no son recomendables, y un buen programador debería evitarlas.

2. **Evitar la Desinformación**: El nombre de la variable no debe ayudar a la incomprensión del código fuente. por ejemplo una variable boolean que se llame `verdadero` introduce desinformación dentro del contexto del código fuente que de utilice.

```
#define FALSO 0
int verdadero;

...
if (verdadero==FALSO) {

}
```

3. **Nombres Distinguibles**: Un hecho clásico, cuando un programador se encuentra con dos variables que contienen valores relacionados y debido a la restricción de la unicidad del nombre de las variables, es cambiar el nombre de una por alguna regla arbitraria. Por ejemplo, si se quiere copiar una cadena de caracteres a otra cadena se podría estar tentado en armar una función que reciba dos cadenas y copie el contenido de una a otra:

```
void copiar_string( char a1[], char a2[]){
}
```

4. **Nombres Fáciles de Buscar**: Normalmente los nombres de variables que corresponden a un carácter por ejemplo `int i`, cuando se intente buscar esa variable en un programa medianamente grande, el motor de búsqueda del editor se va a parar en cada i que se haya escrito.

5. **No Te Hagas el Canchero**: Esta regla se refiere a que es preferible un nombre claro que un nombre que parezca canchero, irónico o gracioso pero solo entiende quien lo programo.

6. **Una Sola Palabra por Concepto**:En español esta regla aplica bastante ya que es un idioma en el cual un concepto puede ser representado por varios vocablos. ejemplo, si se utiliza la palabra obtener como nombre de una variable no deberían utilizarse sus sinónimos para el mismo concepto (lograr, conseguir, alcanzar, conquistar, ganar. ganar, ingresar, cobrar, percibir, obtener. adquirir, lograr, mercar, coger, comprar, apropiarse, cazar, conseguir, obtener, agenciarse, apoderarse, pescar, procurarse, atrapar, alcanzar, adueñarse, lucrar) ya que posiblemente se preste a confusión .

7. **Nombres Pronunciables**: Los nombres deben poder pronunciarse `DtaRcd32` si esa variable representa el valor del codigo de cliente debería simplemente llamarse `codigo_cliente`.

8. **Evitar los Nombres Genéricos**: Los nombres deben hacer que una variable auto-describa que concepto almacena en el contexto del programa, los nombres genéricos no reducen la complejidad del código fuente sino que lo hacen menos entendibles.

3. Declaración de variables

Se pueden declarar distintas variables de un mismo tipo en una línea. Es recomendable cuando las variables tienen el mismo uso (por ejemplo, índice de iteración) y no entorpezca la lectura del código.

Mala práctica

```
int i, contador_personas, suma, resta
    , numero1, numero2, j, k;
char opcion, nombres_alumnos[
    MAX_NOMBRE], tablero[MAX_FILAS][
    MAX_COLUMNAS], sexo_femenino,
    sexo_masculino;
```

Buena práctica

```
#define SEXO_FEMENINO 'F'
#define SEXO_MASCULINO 'M'

int i, j, k;
int suma, resta;
int numero1, numero2;

char opcion;
char nombres_alumnos[MAX_NOMBRE];
char tablero[MAX_FILAS][MAX_COLUMNAS
    ];
```

En el ejemplo de la mala práctica se puede observar que por más que las variables corresponden al mismo tipo de dato, no implica que su uso será el mismo. Incluso se están trabajando con variables que involucran más de una dimensión.

En la buena práctica se observa que las variables están agrupadas por finalidad de las mismas (por ejemplo en la primera línea son variables que se usarán como índices) y a la vez por dimensiones (matrices y vectores).

4. Uso de constantes

Si se usarán constantes o literales, declararlos antes del main y las funciones. Esto servirá para que si en el futuro dicha constante tenga que ser modificada, sólo se hará en la declaración y se ahorrará trabajo al estar buscándola en todo el código. Como convención, deben estar escritos en mayúscula.

Mala práctica

```c
int main(){

        float radio;

        printf("Ingrese el valor del radio expresado en metros (m)\n");
        scanf("%f" , &radio);
        printf("El perimetro de una circunferencia de radio %f es: %f\n" , radio , 3,14 * radio * 2);
        printf("La superficie de una circunferencia de radio %f es: %f\n", radio , 3,14 * radio * radio);
        printf("El volumen de una esfera de radio %f es: %f\n" , radio , 4/3 * 3,14 * radio * radio * radio);

        return 0;
}
```

```c
int main(){

        int precio;

        printf("Ingrese el precio para calcular su IVA\n");
        scanf("%d" , &precio);
        printf("El IVA aplicado a $%d es: %d\n" , precio , 21 * precio / 100);

        return 0;
}
```

Buena práctica

```c
#include <stdio.h>

#define PI 3,14

int main(){

        float radio;

        printf("Ingrese el valor del radio expresado en metros (m)\n");
        scanf("%f" , &radio);
        printf("El perimetro de una circunferencia de radio %f es: %f\n" , radio , PI * radio * 2);
        printf("La superficie de una circunferencia de radio %f es: %f\n", radio , PI * radio * radio);
        printf("El volumen de una esfera de radio %f es: %f\n" , radio , 4/3 * PI * radio * radio * radio);

  return 0;
}
```

```c
#define IVA 21

int main(){

        int precio;

        printf("Ingrese el precio para calcular su IVA\n");
        scanf("%d" , &precio);
        printf("El IVA aplicado a $%d es: %d\n" , precio , IVA * precio / 100);

        return 0;
}
```

En el ejemplo de la mala praćtica se puede ver que si en el caso que se quisiera agregar más dígitos en el valor de pi, se deberá buscar en todo el programa dicho valor y agregarlo. Caso contrario en la buena práctica que sólo bastará con cambiar el valor de la constante en `PI`.

5. Inicialización de variables

Principalmente si se usarán como contadores o para posiciones en vectores, esto evitará que se esté trabajando con ?basura? durante la ejecución del programa y/o intentar acceder a una porción no asignada de memoria, siempre y cuando el usuario no tenga que inicializarla durante la ejecución del programa con los valores que necesite.

Mala práctica

```c
#include <stdio.h>
#define TOPE_SUMA 10

int main(){

        int suma;
        for (int i = 0 ; i < TOPE_SUMA ; i++){
                suma = suma + i;
        }
        printf("El resultado es: %d\n" , suma);

        return 0;
}
```

Buena práctica

```c
#include <stdio.h>
#define TOPE_SUMA 10

int main(){

        int suma = 0;     /* o inicializar con el valor que se desee, incluso con
                * el valor de otra variable del mismo tipo, segun se
                * necesite */
        for (int i = 0 ; i < TOPE_SUMA ; i++){
        suma = suma + i;
        }
        printf("El resultado es: %d\n" , suma);

        return 0;
}
```

Al no asignarle un valor a `suma` y luego sumar sobre dicha variable, no garantiza que se obtenga el resultado deseado, ya que el compilador de no sabe de antemano que valor desea el usuario. En este ejemplo, se desea sumar desde cero, y es por eso que le asignamos ese valor al declararla.

6. Emplear una sola convención

La convención elegida debe ser usada en todo el código y por todos los integrantes del grupo, si fuera el caso. Para la escritura de la declaración de variables y funciones, algunas de las convenciones más utilizadas son *snakecase* y *camelcase*. En el caso de apertura y cierre de llaves también se debe emplear una sola convención en todo el programa.

Mala práctica

Para la declaración de variables y funciones:

```c
int cantidad_alumnos, sumadorNotas;
int mes_1 , segundo_mes;
```

Para la apertura y cierre de llaves:

```c
if (numero1 < numero2)
{
        resultado = numero2 - numero1;
}else{
        resultado = numero1 - numero2;
}
```

Buena práctica

Para la declaración de variables y funciones:

```c
int cantidad_alumnos;  // (snakecase)
int mes_1 , mes_2;
```

ó

```c
int cantidadAlumnos;  // (camelcase)
```

Para la apertura y cierre de llaves:

```c
int sumar(int numero1 , int numero2){
        return numero1 + numero2;
}
```

ó

```c
int sumar(int numero1 , int numero2)
{
        return numero1 + numero2;
}
```

La diferencia que se nota en los ejemplos es que no se ve una uniformidad en las convenciones usadas. No sólo indica que no se respetaron las convenciones sino que también le quita claridad al código.

7. Variables globales

El uso de variables globales no es recomendable, ya que al ser parte del entorno global del programa dicha variable se puede modificar en cualquier parte del mismo, a su vez que todo el programa dependería

de ella y también dificultaría la lectura del código. **En este curso está PROHIBIDO usar variables globales**.

Mala práctica

```c
#include <stdio.h>

int resultado;
int sumar(int num1 , int num2){
    resultado = num1 + num2;
    return resultado;
}

int main(){

    resultado = sumar(numero1 , numero2 );
    printf("%d\n" , resultado);
    return 0;
}
```

Buena práctica

```c
#include <stdio.h>

int sumar(int num1 , int num2){
    return num1 + num2;
}

int main(){

    int resultado;

    resultado = sumar(numero1 , numero2);
    printf("%d\n" , resultado);
    return 0;
}
```

8. Indentación

Para darle más claridad al código, cuando comienza una estructura de control, dentro de la misma se deja un espacio tabulado. Esto aplica aún cuando la sentencia para la estructura es una sola.

Mala práctica

```c
int suma = 0;
while(suma < MAX_PERSONAS){
for(int  i = 0 ; i < MAX_PERSONAS ; i ++){
suma = suma + i;
printf("%d\n" , i);
}
}
```

```c
int divisor = 0;
int dividendo = 30;
int resultado;

if (dividendo == 0) {printf("Error\n");}
else {resultado = divisor/dividendo;}
```

Buena práctica

```c
int suma = 0;
while(suma < MAX_PERSONAS){
        for(int  i = 0 ; i < MAX_PERSONAS ; i++){
                suma = suma + i;
                printf("%d\n" , i);
        }
}
```

```c
int divisor = 0;
int dividendo = 30;
int resultado;

if (dividendo == 0){
        printf("Error\n");
}else{
        resultado = dividendo / divisor;
}
```

En el ejemplo de mala práctica se puede ver que es díficil saber dónde comienzan las instrucciones para la estructura de control, así como también la totalidad del código. En cambio en el ejemplo de la buena práctica se puede apreciar que es más claro saber qué instrucciones pertenecen a las estructuras de control y cuales no, sin importar si son varias instrucciones o una sola.

9. Una instrucción por línea

Si se escribe la totalidad del programa en una sola línea, el compilador no tendrá problemas en compilarlo y ejecutarlo (si no hubiesen errores de compilación y ejecución), se escribe de esta manera para hacer el seguimiento del código de una manera más sencilla.

Mala práctica

```c
for(int i = 0 ; i < MAX_PERSONAS ; i ++){ suma = suma + i; j = i + 1;}
```

Buena práctica

```
for(int i = 0 ; i < MAX_PERSONAS ; i ++){
        suma = suma + i;
        j = i + 1;
}
```

Al igual que la identación, el indicar una instrucción por línea hace que el seguimiento del código sea mucho más llevadero. En la mala práctica, por más que el programa funcione correctamente, es dificultoso hacer un seguimiento del mismo. En cambio en la buena práctica, es más visible ver al menos cuántas instrucciones hay.

10. Usar paréntesis para cada operación lógica

No sólo será más legible para el programador, sino que también para el compilador.

Mala práctica

```
if(numero < MAX_PERSONAS && opcion ==
    OPCION_SI){

}
```

Buena práctica

```
if( (numero < MAX_PERSONAS)  &&  (
   opcion == OPCION_SI) ){

}
```

11. Evitar comentarios redundantes en el código

```
int i = 1;                    /* asigno 1 a la variable 'i' */
```

12. Modularización ("Divide y vencerás")

Modularizar un programa con funciones sencillas no sólo da legibilidad al código, sino también ayuda a evitar repetición del mismo.

Ejemplo: se quiere calcular aproximadamente la cantidad de basura que se tira por día en Hogwarts. Se sabe que en cada habitación de cada casa conviven 3 alumnos, que hay un total de cuatro casas (Gryffindor, Hufflepuff, Ravenclaw y Slytherin) y que cada alumno genera aproximadamente 500gr de basura por día. La cantidad de habitaciones serán ingresadas por el usuario.

Mala práctica

```
#include <stdio.h>

int main(){

        int slytherin , gryffindor , ravenclaw , hufflepuff;
        int total_basura;

        printf("Ingrese cantidad de habitaciones para Gryffindor\n");
        scanf("%d" , &gryffindor);
        printf("Ingrese cantidad de habitaciones para Hufflepuff\n");
        scanf("%d" , &hufflepuff);
        printf("Ingrese cantidad de habitaciones para Ravenclaw\n");
        scanf("%d" , &ravenclaw);
        printf("Ingrese cantidad de habitaciones para Slytherin\n");
        scanf("%d" , &slytherin);

        total_basura = (slytherin + gryffindor + ravenclaw + hufflepuff) * 3 * 500;
        printf("El total de basura generada por dia (gr/dia) en Hogwarts es: %d\n" ,
    total_basura);

        return 0;
}
```

Buena práctica

```c
#include <stdio.h>

#define GRYF "Gryffindor"
#define HUFF "Hufflepuff"
#define RAVE "Ravenclaw"
#define SLY "Slytherin"
#define CANT_ALUMNOS 3
#define CANT_BASURA 500

void pedir_cantidad_habitaciones(int *habitaciones , char *nombre_casa){

        printf("Ingrese cantidad de habitaciones para %s\n" , nombre_casa);
        scanf("%d" , habitaciones);
}

int sumar_habitaciones(int slytherin , int gryffindor , int ravenclaw , int
    hufflepuff){

        int total;
        total = slytherin + gryffindor + ravenclaw + hufflepuff;
        return total;
}

int calcular_total_basura(int total_habitaciones){

        return (CANT_ALUMNOS * CANT_BASURA * total_habitaciones);
}

int main(){

        int total_habitaciones;
        int slytherin , gryffindor , ravenclaw , hufflepuff;
        int total_basura;

        pedir_cantidad_habitaciones(&gryffindor , GRYFF);
        pedir_cantidad_habitaciones(&hufflepuff , HUFF);
        pedir_cantidad_habitaciones(&ravenclaw , RAVE);
        pedir_cantidad_habitaciones(&slytherin , SLY);
        total_habitaciones = sumar_habitaciones(slytherin , gryffindor , ravenclaw ,
    hufflepuff);
        total_basura = calcular_total_basura(total_habitaciones);

        printf("El total de basura generada por dia (gr/dia) en Hogwarts es: %d\n" ,
    total_basura);

        return 0;
}
```

En el primer ejemplo se observa que hay mucha repetición de código y no hay uso de constantes. En el segundo ejemplo, el código se lo dividió en funciones sencillas e incluso más "generales"(ejemplo, `pedir_cantidad_habitaciones`. Se puede usar la misma función para luego escribir distintos textos), se declararon constantes tal como se explicó anteriormente y da lugar a que dichas funciones puedan usarse reiteradas veces dentro del programa.

13. Usar pre y post condiciones

Principalmente al trabajar en grupo, ayuda mucho saber qué hace una función sin necesidad de hacer un seguimiento de la misma. Esta práctica se aplica incluso cuando la función es muy sencilla.

```c
/*pre: funcion que recibe solo numeros positivos*/
/*post: el programa muestra por consola las estrellas enumeradas*/
```

```
void listar_estrellas(int cantidad_estrellas){
       for(int i = 1; i < cantidad_estrellas; i++){
               printf("estrella Nro.: %d",i);
       }
}
```

14. Una función que tenga interacción con el usuario, no debería devolver un tipo de dato que no sea void.

15. No declarar variables de más

No solo que puede llegar a confundir en la lectura del código, sino que también estaría ocupando memoria innecesaria.

16. Cortar estructuras iterativas correctamente

Las estructuras iterativas tienen una condición de corte, la cual debe ser respetada y debe ser la única puerta de salida de dicha estructura.

Usar instrucciones como `break` o `return` dentro de estructuras iterativas rompe con el normal flujo del algoritmo, obstruyendo la lectura y comprensión del código.

```
while(true){
    ...
    ...
    if (condicion){
          break;
      }
}
```

ó

```
for(i = 0 ; i < CANT_PERSONAS ; i++){
    ...
    ...
    if (condicion){
          return i;
      }
}
```

En realidad esto es una estructura iterativa do-while:

```
do {
    ...
    ...
} while (!condicion);
```