Actas de las Jenui, vol. 7. 2022.
Páginas: 295-302

Patrones y antipatrones para el primer acercamiento a la programación

Rosalía Peña, Álvaro Somolinos Yagüe
Departamento de Ciencias de la Computación
Universidad de Alcalá
Alcalá de Henares
rpr@uah.es, alvaro.somolinos@uah.es

Resumen

Muchos profesores hemos experimentado, que los estudiantes universitarios se sienten desorientados en su primer acercamiento a la programación (CS1).

En la búsqueda de soluciones a este problema, alguna bibliografía reciente propone presentar, explícitamente al alumnado, patrones y antipatrones de código. Otros autores apelan a proporcionar convenios de legibilidad y mantenibilidad, para conseguir códigos más estilosos.

Consideramos que la manera de evitar la desorientación y simultáneamente, mejorar el código que generan consiste en facilitar una visión global del conjunto de recursos que proporciona el lenguaje, junto a un conjunto reducido pero completo de los criterios de calidad del software y secuenciar consecuentemente el temario, seleccionando cuidadosamente los ejercicios propuestos.

Abstract

Many professors have experienced that university students feel disoriented in the first approach to programming (CS1).

Searching for solutions, some recent literature works suggests explicitly presenting code patterns and anti-patterns to students. Other authors appeal to provide readability or maintainability conventions to achieve more stylish codes.

We believe that the way to avoid disorientation and simultaneously improve the code they generate is to provide them with a reduced but complete set of software quality criteria, properly sequence the course topics and carefully select the proposed exercises.

Palabras clave

CS1, patrones, antipatrones, asesoramiento en calidad, programación.

1. Introducción

Pese a la dificultad que supone al novel acercarse a la programación [1], o precisamente debido a ello, hemos de planificar la materia teniendo presente que es *igualmente importante que el código del alumno funcione, como que lo escriba con estilo desde CS1* [9]. Hoare [8] reconoce dos estilos de programación: *hacerlo tan simple que obviamente no incluya deficiencias, o tan complejo que las deficiencias no sean obvias, pero el primero es más difícil.* El Zen de Python[1] (nuestro soporte) indica: *There should be one-- and preferably only one --obvious way to do it.* Conducir al novel por ese "único" camino supone un gran reto [13,14]. Por supuesto que un problema general y poco definido puede tener diferentes soluciones: iterativo/ recursivo, por ejemplo, pero en el proceso docente y especialmente cuando la materia es percibida como difícil, cada problema que proponemos pretende entrenar un aspecto adecuado al momento de evolución del curso. Si hemos concretado adecuadamente el enunciado, incluso los identificadores coincidirán [2,7,12] y las desviaciones de la propuesta esperada, probablemente, sean consecuencia de un conocimiento poco profundo o confuso de un recurso de programación [3]. Apreciamos además que las propuestas extrañas se repiten, edición tras edición y con independencia de la universidad y el país. La bibliografía se refiere a estas propuestas repetitivas como antipatrón. Si proceden de errores conceptuales, resulta claro que hay que atenderlas.

En ese sentido, las herramientas de evaluación de código basadas exclusivamente en la satisfacción de una batería de pruebas son inadecuadas para CS1. En los últimos años se avanza en herramientas y/o técnicas diseñadas para fomentar el estilo del código.

De Ruvo considera fundamental ofrecerles mucha retroalimentación sobre el estilo de su código, para evitar el establecimiento de malos hábitos [3].

[1] Accesible en http://recursospython.com/guias-y-manuales/zen-de-python-tim-peters/ ejecutando "import this" en el editor de Python.

En [13] y anteriores trabajos (realizados sobre diferentes colectivos), Wiese busca determinar si las directrices de estilo parecen arbitrarias al estudiante y si conviene una instrucción formal sobre el estilo que usan los expertos, mostrando antipatrones. Para cada antipatrón con que trabaja en sus experimentos enfrenta al estudiante a dos códigos con idéntica funcionalidad, uno que contiene un error y otro, diseñado por un experto, y les pregunta cuál les parece más legible y cuál sería el código desarrollado por el experto.

Por nuestra cuenta habíamos caracterizado los errores que usa (y otros muchos) Wiese en sus pruebas, lo que una vez más, confirmaba la universalidad de los antipatrones, pero llamó nuestra atención que les sorprenda que los estudiantes predigan cuál es la versión del experto, pero encuentren más legible la contraria. El experto no codifica sólo en términos de legibilidad, balancea un conjunto de criterios de calidad. Pensamos que, si Wiese hubiera preguntado qué versión satisface mejor los criterios de calidad del software, en vez de preguntar por la legibilidad exclusivamente, la respuesta de los estudiantes a las dos preguntas habría coincidido.

También nos chocó alguna de sus conclusiones y, para contrastar con ellos, reprodujimos su experimento. Opinamos que los antipatrones no requieren instrucción explícita. Creemos que si la requieren los criterios de calidad del software que conducen al experto a codificar como lo hace. Creemos que conocidos estos, las normas de estilo no le parecerán arbitrarias al estudiante, pues emanan de los criterios de calidad. Los antipatrones pueden usarse como ejemplos de la aplicación de los criterios de calidad en la toma de decisión del diseño de código. Consideramos que mostrarles explícitamente algún patrón de programación mejora la eficiencia del proceso docente, y a partir de ese punto, incentivar que induzcan ellos los siguientes.

2. Criterios de calidad en CS1

Sin criterios claros y concisos sobre la calidad del código, es lógico que el estudiante novel se sienta desorientado. Partiendo del modelo de calidad ISO/IEC25010[2] buscamos el menor subconjunto relevante de criterios para CS1.

En compromiso con la formación deontológica de nuestros estudiantes, es necesario derribar el mito de que la informática es un "mundo virtual" sin consecuencias [5]. Nada de virtual tiene sacar dinero de un cajero o viajar en avión. El programador es responsable de que su código proporcione, el 100% de las veces, el resultado correcto para cualquier conjunto de entrada viable en el mundo real, y sólo para valores posibles. Bajo el término eficacia[3] agrupamos, además de la adecuación funcional, una somera aproximación a fiabilidad y a la protección frente a errores del usuario. La eficacia es un criterio irrenunciable, pero no es el único relevante, desde la primera aproximación a la programación. Que el resultado sea correcto, no implica que el programa lo sea [9].

Tener la reusabilidad como meta proporciona excelentes oportunidades para desarrollar pensamiento computacional, practicando la generalización/ especialización hasta concretar el ámbito del subprograma. Es decir, tras el enunciado: "pide al usuario el mes", propondremos "pide al usuario el día" y/o equivalentes, incentivando que el estudiante subprograme de una vez por todas `entero_pedido (mini, maxi, mensaje)`. Este entrenamiento, proporciona código reusable para sus bibliotecas y, lo que es más importante, le conducirá a localizar soluciones reusables (es decir, a inducir los patrones por sí mismo) y a consolidar buenas prácticas de depuración [10], lo que a su vez mejora la eficacia.

La facilidad de mantenimiento (mantenibilidad) es nuestro tercer criterio de calidad: Más del 90% del tiempo de un programador está dedicado a leer/mantener código; Martin [11] lo comprobó monitorizando su propia dedicación. El porcentaje de la inversión en software destinado a mantenimiento frente al total sigue ascendiendo [4]. Este coste se debe mayormente a su actualización y ampliación y no a la reparación de errores, por tanto, no es previsible que decrezca [6]. La legibilidad contribuye a la mantenibilidad y mejora al adoptar sencillos convenios [2] para documentar, seleccionar identificadores, establecer el orden de las secciones entre otros. Independientemente de cuales sean los elegidos, entendemos que resulta fundamental entrenar la adhesión del ingeniero en ciernes a estándares y convenios.

Además de la legibilidad, contribuyen a la mantenibilidad la elección de algoritmos y estructuras de control todo lo simple que sea posible [3][4] y evitar la redundancia. Los patrones, con frecuencia, están orientados a lo primero. Atajamos la redundancia mediante subprogramación y el uso de constantes nominales.

La cuarta característica de calidad del software que dirige nuestro temario es la eficiencia. Este curso, no es el momento de estudiar formalmente algoritmia (ni hay tiempo, ni el comportamiento asintótico es ahora punto de mira), pero simples nociones del reloj de la unidad central de proceso, permiten entender que contar las operaciones ejecutadas proporciona otro

[2] https://iso25000.com/index.php/normas-iso-25000/iso-25010#:~:text=ISO%2FIEC%2025010,de%20un%20producto%20software%20determinado.

[3] Según la Real Academia Española, eficacia es la *capacidad de lograr el efecto que se desea o se espera*.

[4] También invitación en el Zen: *Simple is better than complex. Flat is better than nested*.

criterio para elegir entre dos algoritmos (con menos frecuencia es relevante el consumo de memoria).

3. Consecuencias de la vertebración por calidad

Tanto el orden de presentación de los temas, como la selección de los ejercicios propuestos ha de ser coherente con los criterios de calidad que deseamos que interioricen. Potenciar la reusabilidad condiciona el orden de los contenidos. La subprogramación ha de presentarse lo antes posible en el temario. Así mismo, exige que el estudiante organice código en bibliotecas. Incentivar la eficacia condiciona adelantar el diseño de los casos de prueba aun antes de pensar en el algoritmo, lo cual facilita que el algoritmo no solo sea preciso, sino también conciso. La eficacia también condiciona el elegir enunciados que se puedan resolver, de forma completa, con los recursos con que ya cuenta el estudiante. Fomentar el compromiso ético invita a proponer enunciados relevantes en el mundo real. Los enunciados deben contener elementos reusables, que requieran mantenimiento.

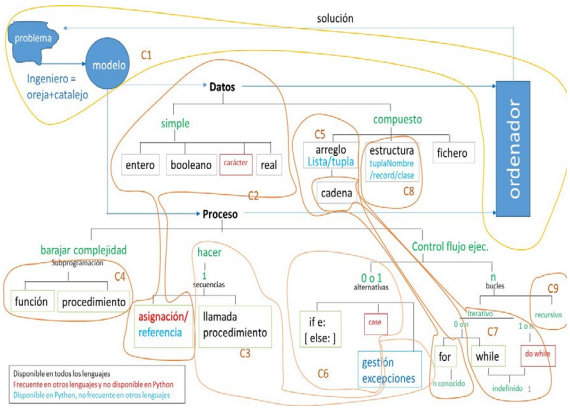

Figura 1: Planificación temporal de recursos PPME.

Evitar redundancias también refuerza el criterio de adelantar la subprogramación en el temario. Algunos patrones de programación, como “Bucle con condición múltiple y discriminación a la salida”, “Agrupar para simplificar”, “Asignación booleana”, o “Agrupar y seleccionar” reducen la anidación de instrucciones de control. Visualizar expresamente las precondiciones favorece realizar una buena descomposición funcional, evitando redundancia en código.

A lo largo del curso, nuestros estudiantes desarrollan las bibliotecas interfaz (de usuario), tiempo, dinero, estadística y punto que, en su conjunto, ofrecen una rica variedad de experiencias de programación y permiten montar, con un esfuerzo razonable del alumno, una aplicación completa a final de curso (por ejemplo, gestión de un centro docente), con un aspecto muy profesional, que les trasmita lo mucho que han aprendido.

4. Descripción de la experiencia

4.1. Contexto y participantes

Impartimos la docencia del primer acercamiento a la programación en una asignatura de 6 créditos, en un grupo de Ingeniería de Computadores (60 estudiantes) y otro grupo de Ingeniería Informática (32 estudiantes). El énfasis está puesto en la resolución de problemas, apoyándonos en el Paradigma de Programación Modular Estructurada (PPME), utilizando Python como sintaxis, pero pensando en facilitar a los estudiantes la migración a lenguaje C el siguiente semestre.

En las primeras sesiones del curso nos aproximamos a los aspectos imprescindibles del hardware para comprender el desarrollo de código. Establecimos y justificamos los criterios de calidad del software, recorrimos el mapa conceptual de los recursos disponibles (los ofrecidos por PPME), que reflejamos en la figura 1. Durante el curso, según resultaron relevantes, facilitamos los convenios de estilo y patrones de codificación y proporcionamos retroalimentación personalizada a códigos presentados por alumnos tanto en el foro de aula virtual como en entregas.

La reproducción de la experiencia [13] se llevó a cabo en las 2 últimas horas de la última sesión de clase y contó con la presencia de 40 estudiantes de los que solamente 20 enviaron el documento. No se incentivó la entrega de éste.

4.2. Secuenciación del experimento

La última sesión, empezó revisando los convenios de estilo del curso, se solicitó a los estudiantes que explicaran porqué se había establecido cada uno y repasamos algunos patrones (2 horas). Iniciamos la experiencia (otras 2 horas), recorriendo los siguientes pasos, para varios enunciados, tratados de uno en uno:

1. Pedimos que escribieran un conjunto adecuado de pruebas. Puesta en común y discusión.
2. Mostramos tres (o más) códigos que supuestamente lo resolvían, solicitando que, para cada código, revisaran si era eficaz. Había un código ineficaz, y uno (o varios) pares de códigos en que uno contenía un error a derribar y en el otro se había corregido.
3. Para el código ineficaz solicitamos clasificar el error (traducción/ ejecución/ lógico/ de estilo[5]) y

[5] Un error de estilo es un código que incumple uno de los convenios que hemos establecido.

a qué característica(s) de calidad afectaba. Puesta en común.

4. Frente a las dos versiones eficaces, solicitamos que reflejaran en su documento qué versión creían que elegiría un experto y cuál les parecía más mantenible.
5. Solicitamos que especificaran qué características de calidad variaban entre ambas versiones.

Al finalizar, se les propuso redactar, con sus palabras, una recomendación respecto a la diferencia que presentaban las versiones de cada pareja y que nos enviasen el documento que habían desarrollado durante la sesión, una vez concretadas sus recomendaciones.

```
#V1
def es_bisiesto(anno):
  """int-->bool
  OBJ: True si anno es bisiesto
  PRE: anno>=1582"""
  if anno%4==0:
    if anno%100==0:
      if anno%400==0:
        lo es=True
      else:
        lo_es=False
    else:
      lo_es=True
  else:
    lo es=False
  return lo_es
```

```
#V2
def es_bisiesto(anno):
  """int-->bool
  OBJ: True si anno es bisiesto
  PRE: anno>=1582"""
  if anno%4==0 and anno%100!=0 or anno%400==0:
    lo_es=True
  else:
    lo es=False
  return lo_es
```

Cuadro 1: (A) Condicionales compuestas frente a condicionales anidadas

```
#V2
def es bisiesto(anno):
  """int-->bool
  OBJ: True si anno es bisiesto
  PRE: anno>=1582"""
  if anno%4==0 and anno%100!=0 or anno%400==0:
    lo es=True
  else:
    lo_es=False
  return lo_es
```

```
#V3
def es bisiesto(anno):
  """int-->bool
  OBJ: True si anno es bisiesto
  PRE: anno>=1582"""
  return anno%4==0 and anno%100!=0 or \
         anno%400==0
```

Cuadro 2: (B) Asignación booleana

```
#V1
def muestra si bisiesto(anno):
  """int-->None
  OBJ: imprime si es bisiesto anno
  PRE: anno>=1582. es bisiesto definido"""
  if es bisiesto(anno) == True:
    print(anno, 'es bisiesto')
  else:
    print(anno, 'no es bisiesto')
```

```
#V2
def muestra si bisiesto(anno):
  """int-->None
  OBJ: imprime si es bisiesto anno
  PRE: anno>=1582. es bisiesto definido"""
  if es bisiesto(anno):
    print(anno, 'es bisiesto')
  else:
    print(anno, 'no es bisiesto')
```

Cuadro 3: (C) Bool es bool

```
#V2
def muestra_si_bisiesto(anno):
  """int-->None
  OBJ: imprime si es bisiesto anno
  PRE: anno>=1582. es bisiesto definido"""
  if es bisiesto(anno):
    print(anno, 'es bisiesto')
  else:
    print(anno, 'no es bisiesto')
```

```
#V3
def muestra_si_bisiesto(anno):
  """int-->None
  OBJ: imprime si es bisiesto anno
  PRE: anno>=1582"""
  if anno%4==0 and anno%100!=0 or anno%400==0:
    print(anno, 'es bisiesto')
  else:
    print(anno, 'no es bisiesto')
```

Cuadro 4: (D) Reúso en definición

```
#V1
"""PROGRAMA OBJ: tipo de triangulo {equila-
tero/isósceles/escaleno} formado a,bc"""
# captura de a,b,c
if a==b:
  if a==c:
    tipoTri = 'equilátero'
  else:
    tipoTri = 'isósceles'
elif c==b or c==a:
  tipoTri = 'isósceles'
else:
  tipoTri = 'escaleno'
print('el triángu-
lo',a,';',b,';',c,'es',tipoTri)
```

```
#V2
"""PROGRAMA OBJ: tipo de triangulo {equilate-
ro/isósceles/escaleno} formado """
# captura de a,b,c

if a==b and a==c: tipoTri = 'equilatero'
if a!=b and a!=c and b!=c: tipoTri = 'escaleno'
if a==b and a!=c and b!=c or a==c and \
   a!=b and  b!=c or c==b and a!=c and b!=a:
  tipoTri = 'isósceles'
print('el triángulo',a,';',b,';',c,'es',tipoTri)
```

Cuadro 5: (E) Condicionales anidados frente a planos

```
#V2
def imprime animal_chino(anno):
  """int-->None
  OBJ: imprime el animal del horóscopo chino
       de año anno
  PRE: anno>0"""
  if   anno%12== 4: animal = 'Rata'
  elif anno%12== 5: animal = 'Buey'
  elif anno%12== 6: animal = 'Tigre'
  elif anno%12== 7: animal = 'Conejo'
  elif anno%12== 8: animal = 'Dragón'
  elif anno%12== 9: animal = 'Serpiente'
  elif anno%12==10: animal = 'Caballo'
  elif anno%12==11: animal = 'Cabra'
  elif anno%12== 0: animal = 'Mono'
  elif anno%12== 1: animal = 'Gallo'
  elif anno%12== 2: animal = 'Perro'
  else:             animal = 'Cerdo'
  print(f'El animal de{anno} es {animal}')
```

```
#V3
def imprime animal_chino(anno):
  """int-->None
  OBJ: imprime el animal del horóscopo chino de
       año anno
  PRE: anno>0"""
  if   anno%12== 0: animal = 'Mono'
  elif anno%12== 1: animal = 'Gallo'
  elif anno%12== 2: animal = 'Perro'
  elif anno%12== 3: animal = 'Cerdo'
  elif anno%12== 4: animal = 'Rata'
  elif anno%12== 5: animal = 'Buey'
  elif anno%12== 6: animal = 'Tigre'
  elif anno%12== 7: animal = 'Conejo'
  elif anno%12== 8: animal = 'Dragón'
  elif anno%12== 9: animal = 'Serpiente'
  elif anno%12==10: animal = 'Caballo'
  else:             animal = 'Cabra'
  print(f'El animal de {anno} es {animal}')
```

Cuadro 6: (F) El orden afecta a la legibilidad

```
#V3
def imprime animal_chino(anno):
  """int-->None
  OBJ: imprime el animal del horóscopo
       chino de año anno
  PRE: anno>0"""
  if   anno%12== 0: animal = 'Mono'
  elif anno%12== 1: animal = 'Gallo'
  elif anno%12== 2: animal = 'Perro'
  elif anno%12== 3: animal = 'Cerdo'
  elif anno%12== 4: animal = 'Rata'
  elif anno%12== 5: animal = 'Buey'
  elif anno%12== 6: animal = 'Tigre'
  elif anno%12== 7: animal = 'Conejo'
  elif anno%12== 8: animal = 'Dragón'
  elif anno%12== 9: animal = 'Serpiente'
  elif anno%12==10: animal = 'Caballo'
  else:             animal = 'Cabra'
  print(f'El animal de {anno} es {animal}')
```

```
#V4
ANIMALES ZODIACO=('Mono','Gallo','Perro','Cerdo',
          'Rata','Buey','Tigre','Conejo',
          'Dragón','Serpiente','Caballo','Cabra')

def imprime_animal_zodiaco(anno):
  """int-->None
  OBJ: imprime el animal del horóscopo chino de
       ese año
  PRE: año > 0, ANIMALES ZODIACO definida """
  print(f'El animal de {año} es
        {ANIMALES_ZODIACO[anno%12]}')
```

Cuadro 7: (G) Patrón agrupar y seleccionar

4.3. Patrones y antipatrones evaluados

Los cuadros 1 a 9 reflejan las parejas de códigos eficaces, elegidas para la experiencia. Los códigos de cada par, correspondientes al antipatrón son, directamente, propuestas de alumnos en el foro de la asignatura. Algunos de los códigos son consecuencia de que el estudiante tiene un conocimiento superficial del recurso mal usado (sabe la sintaxis e incluso definirlo, pero no lo entiende en profundidad), o son falta de hábito de abstracción (generalización/ especialización). Las propuestas A, B, C, y E son ejemplos del primer motivo. Las propuestas D, F, G, H, I del segundo motivo. Por ejemplo: la A muestra dificultad en el manejo de los operadores bool, la B y la C están relacionadas el con concepto del propio tipo bool. Una variable bool puede ser asignada a una expresión bool; y constituye una expresión, como ocurre con cualquier otro tipo. En la E y clasificaciones por rangos, es la cláusula `else` lo que conocen, pero no terminan de entender.

En nuestro convenio de documentación, la primera línea describe los tipos de los parámetros de entrada y salida (necesaria al no disponer Python de control de tipo). El "OBJ" (objetivo) especifica el problema que resuelve el código. La "PRE" describe las precondiciones que deben satisfacer los argumentos de entrada (ya sea debido a restricciones del mundo real o provocadas por el diseño: aviso de consulta global a recursos compartidos con otras piezas). Para centrar la atención del lector en las diferencias entre los códigos de cada pareja, la parte común se muestra en menor intensidad.

4.4. Resultados

El cuadro 10 recoge la versión elegida por cada alumno (filas) como más mantenible (m) y como experta (e), para cada pareja de códigos enfrentados (columnas de la "A" la "I").

Aunque la muestra no es estadísticamente significativa, el conjunto de las respuestas parece indicar que la mayoría de estudiantes identifican la versión más mantenible como la que seleccionaría el experto, mientras que los hallazgos de Wiese indican gran discrepancia entre la versión que asignan al experto y la que valoran como más legible.

Requieren un comentario específico los siguientes pares de propuestas:

```
#V4
ANIMALES ZODIACO=('Mono','Gallo','Perro',
  'Cerdo','Rata','Buey','Tigre','Conejo',
  'Dragón','Serpiente','Caballo','Cabra')

def imprime animal_zodiaco(anno):
  """int-->None
  OBJ: imprime el animal del horóscopo chino
       de ese año
  PRE: año > 0, ANIMALES ZODIACO definida"""
  print(f'El animal de {año} es
        {ANIMALES_ZODIACO[anno%12]}')
```

```
#V5
def imprime animal_zodiaco(anno):
  """int-->None
  OBJ: imprime el animal del horóscopo chino
       de ese año
  PRE: año > 0"""
  ANIMALES ZODIACO=('Mono','Gallo','Perro',
    'Cerdo','Rata','Buey','Tigre','Conejo',
    'Dragón','Serpiente','Caballo','Cabra')

  print(f'El animal de {año} es
        {ANIMALES_ZODIACO[anno%12]}')
```

Cuadro 8: (H) Patrón encapsulación

```
#V1
def estacion(d,m):
  """int,int-->str
  OBJ: devuelve estación que corresponde a d/m
       (dia/mes)
       20/03 al 19/06 Primavera
       20/06 al 21/09 Verano
       22/09 al 20/12 Otoño
       21/12 al 19/03 Invierno
  PRE: d, m son parte de una fecha válida"""
  if (m==12 and d>=21)or m==1 or m==2 or \
     (m==3 and d<=19):
    estac = 'invierno'
  elif (m==3 and d>=20) or m==4 or m==5 or \
       (m==6 and d<=19):
    estac = 'primavera'
  elif (m==6 and d>=20) or m==7 or m==8 or \
       (m==9 and d<=21):
    estac = 'verano'
  else:
    estac = 'otoño'
  return estac
```

```
#V2
def estacion(d,m):
  """int,int-->str
  OBJ: devuelve estación que corresponde a
       d/m (dia/mes)
       20/03 al 19/06 Primavera
       20/06 al 21/09 Verano
       22/09 al 20/12 Otoño
       21/12 al 19/03 Invierno
  PRE: d, m son parte de una fecha válida"""
  plana = m * 100 + d #agrupo
  if   plana<=319: estacion= 'invierno'
  elif plana<=619: estacion= 'primavera'
  elif plana<=921: estacion= 'verano'
  elif plana<=1221:estacion= 'otoño'
  else:            estacion= 'invierno'
  return estacion
```

Cuadro 9: (I) Patrón agrupar para simplificar

\Par_cod	A		B		C		D		E		F		G		H	I		Val
Alum\	m	e	m	e	m	e	m	e	m	e	m	e	m	e		m	e	
1	V2	V2	V3	V3	V2	V2	V3	V3	V2	V1	V3	V3	V4	V4	T	V2	V2	10
2	V2	V2	V3	V2	V2	V2	V2	V2	V1	V1	V3	V3	V4	V4	T	V2	V2	10
3	V2	V2	V3	V3	V2	V2	V3	V2	V2	V1	V3	V3	V4	V4	?	V2	V2	9
4	V2	V2	V3	V3	V2	V2	V3	V2	V1	V1	V3	V3	V4	V4	?	V2	V2	9
5	V2	V2	V3	V3	V3	V3	V2	V2	V2	V1	V3	V3	V4	V4	T	V2	V2	10
6	V2	V2	V3	V3	V2	V2	V3	V2	V2	V1	V3	V3	V4	V4	T			10
7	V2	V2	V3	V3			V3	V2	V2	V1	V3	V3	V4	V4	?	V2	V2	9
8	V2	V2	V3	V3	V2	V2	V2	V2	V1	V2	V3	V3	V4	V4	T	V2	V2	10
9	V2	V2	V3	V3	V2	V2	V2	V2	V2	V2	V3	V3	V4	V4	T	V2	V2	10
10	V2	V2	V3	V3	V2	V2	V2	V2	V1	V1	V3	V3	V4	V4	T	V2	V2	9
11	V2	V2	V3	V3	V2	V2	V2	V2	V2	V1	V3	V3	V4	V4	?	V2	V2	10
12	V2	V2	V3	V2	V2	V2	V3	V2	V2	V1	V3	V3				V2	V2	8
13	V2	V2	V3	V3	V2	V2	V3	V3	V1	V1	V3	V3	V4	V4	?	V2	V2	10
14	V2	V2	V3	V3	V2	V2	V3	V3	V1	V1	V3	V3	V4	V4	?	V2	V2	9
15	V2	V2	V3	V3	V2	V2	V3	V3	V1	V1	V3	V3	V4	V4	T	V2	V2	10
16	V2	V2	V3	V3	V2	V2	V2	V2	V2	V2	V3	V3	V4	V4	T	V2	V2	10
17					V2	V2	V3	V2	V1	V1	V3	V3	V4	V4	?	V1	V1	10
18	V2	V2	V3	V3	V2	V2	V3	V2	V1	V1	V3	V3	V4	V4	T	V2	V2	7
19	V2	V2	V3	V3	V2	V2	V2	V2	V1	V1	V3	V3	V3	V3	?	V2	V2	10
20	V2	V2	V2	V2	V2	V2	V2	V2	V1	V1	V3	V3	V3	V4	T	V2	V2	10

Cuadro 10: Recopilación de las respuestas de los alumnos

D: Reuso en definición. Probablemente, no transmitimos que la función reusada ya residía en la biblioteca y era usada por otras aplicaciones.

E: Condicionales anidados frente a planos**.** La versión plana realiza operaciones innecesarias (por tanto, es menos eficiente). Los alumnos detectan que V1 es más eficiente y la asignan adecuadamente al experto. La reticencia a considerar mantenible la que consideran experta nos indica que a algunos les resulta difícil la anidación de condicionales y debemos prestarle atención

H: Patrón encapsulación. En esta propuesta, 9 estudiantes eligieron la versión esperada, otros 8 generalizaron respondiendo “depende”. Es decir, sintetizaron un patrón. Para confirmarlo, hemos evaluado la recomendación que redactaron, encontrando 11 estudiantes (reflejados con una T en el cuadro) que explican claramente cuando encapsular y cuando no. Hemos marcado con “?” la respuesta de aquellos que, aun habiendo elegido la versión adecuada para el caso no queda tan clara su recomendación. Es probable que estos alumnos también lo hayan entendido.

4.5. Evaluación de la experiencia

Por nuestra parte, valoramos que los estudiantes se mostraron muy participativos y la sesión resultó muy dinámica, dando tiempo a discutir muchos aspectos relevantes al estilo del código. Es posible que cooperaran otros factores como que era final de curso y algunas timideces ya estaban vencidas; la cercanía de la prueba de evaluación pudo resultar una motivación. De otra parte, la baja asistencia (quizá causada por el examen de Física convocado la hora siguiente, quizá por haber abandonado esta asignatura) puede haber desviado la muestra, de la población destino (alumnos de CS1) hacia un subgrupo más interesado y mejor preparado.

Apreciamos que para los estudiantes era significativamente más fácil localizar el error (paso 3), que determinar a qué criterios de calidad afectaba (paso 5) una modificación. Ya que esta habilidad mejoró al avanzar la sesión creemos que es rentable esta propuesta como recurso pedagógico.

Entendemos que conviene que el estudiante generalice, con sus palabras, la recomendación que suscita la comparación de las versiones eficaces de cada propuesta; porque verbalizarlo le ayuda a consolidar sus conceptos y por eso les pedimos que lo hicieran como trabajo personal. Como hemos comentado, se adelantaron a nuestra solicitud, en la propuesta H, casi al final de la experiencia, lo cual nos induce a valorar el recurso docente.

Entendemos que el momento empleado para la experiencia es el adecuado para que el estudiante tome ventaja de este recurso pedagógico, incluso aunque el momento haya contribuido a la baja asistencia, que minora la significancia de nuestra evaluación.

Por la otra parte, se pidió a los estudiantes que, en el documento que nos enviarían, evaluaran la rentabilidad de la sesión, en una escala de 0 a 10 y el dato aparece reflejado en la columna VAL del cuadro 10. Más de la mitad de las evaluaciones venían acompañadas de comentarios (no solicitados), desde dos lacónicos: “muchísimas gracias”, a dechado de alabanzas, pasando por el del alumno que evalúa más

bajo la experiencia, que indica: *"Esto es extremadamente útil…le doy un 7 porque después de la clase tenemos examen de Física que nos desconcentra, aun así, a mí me ha servido para aclararme más con la asignatura."*

Acogemos con reservas la valoración del alumnado, puesto que fue nominativa y puede estar condicionada.

5. Conclusiones

Apreciamos que comparar respecto a los criterios de calidad, parejas de código que resuelven el mismo problema, es un recurso pedagógico interesante para consolidar buen estilo, ya que les ayuda a percibir cómo ligeros (o menos ligeros) cambios en la codificación afectan a la calidad de su código. Su empleo es más propicio para finales de curso. Lateralmente, la experiencia ha mostrado que conviene prestar mayor atención a los condicionales anidados, pues les resultan especialmente difíciles de leer.

Confirmamos, como apunta [3], que los antipatrones corresponden a errores conceptuales. Pueden estar propiciados por la supresión o reducción a lo anecdótico del estudio de Lógica en Filosofía y del Algebra de Bool en Matemáticas en la formación preuniversitaria. Estos pilares son básicos para potenciar la capacidad del razonamiento, y deberíamos intentar influenciar, desde nuestra posición en las políticas de la educación preuniversitaria.

Apreciamos que visualizar los criterios de calidad del software, como eje vertebrador del desarrollo del curso y como criterios de la propia evaluación del estudiante, reducirán la sensación de desorientación y bloqueo del novel al conducirle en su proceso de diseño de soluciones. Pensamos que, como consecuencia, producirán código más elegante.

Este compromiso con los criterios de calidad del software implica una adecuada secuenciación en la presentación de los temas y una mimada selección de los ejercicios propuestos, así como de las propuestas de solución.

Referencias

[1] Jens Bennedsen y Michael E. Caspersen. Failure rates in introductory programming: 12 years later. En *ACM In roads* 10.2, pp. 30–36, 2019.

[2] Charis Charistsis, Chris Piech y John Mitchell. Assessing Function Names and Quantifying the Relationship Between Identifiers and Their Funtionality to Improve Them. En *Proceedings of the Eighth ACM Conference on Learning @ Scale,* pp. 291–294, junio 2021.

[3] Giuseppe De Ruvo, Ewan Tempero, Andrew Luxton-Reilly, Gerard Rowe y Nasser Giacaman. Understanding semantic style by analysing student code. En *Proceedings of the 20th Australasian Computing Education Conference*, pp. 73–82, enero 2018.

[4] Sayed Mehdi Hejazi y Nafiseh Hairahimi. Which factors affect software projects maintenance cost more? En *Acta Inform Med. 21(1),* pp. 63-6, marzo 2013.

[5] Casey Fiesler, Mikhaila Friske, Natalie Garrett, Felix Muzny, Jessie J. Smith y Jason Zietz. Integrating Ethics into Introductory Programming Classes. En *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education.* pp. 1027-1033, marzo 2021.

[6] Robert L. Glass. Facts and Fallacies of Software Engineering. En *Addison-Wesley Professional*, octubre 2002.

[7] Elena L. Glassman, Lyla Fischer, Jeremy Scott y Robert C. Miller. Foobaz: Variable name feedback for student code at scale. En *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, pp. 609-617, noviembre 2015.

[8] Charles A. R. *Hoare*. The Emperor's Old Clothes. En *The 1980 ACM Turing Award Lecture Communications of the ACM*, 24(2), pp. 75–83, febrero 1981.

[9] Saj-Nicole A. Joni y Elliot Soloway. But my program runs! Discourse rules for novice programmers. En *Journal of Educational Computing Research*, 2(1), pp. 95-125, 1986.

[10] Cagin Kazimoglu, Mary Kiernan, Liz Bacon y Lachlan Mackinnon. A serious game for developing computational thinking and learning introductory computer programming. En *Procedia - Social and Behavioral Sciences*, 47, pp. 1991-1999, 2012.

[11] Robert C. Martin. Clean Code: A Handbook of Agile Software Craftsmanship. *Prentice Hall*, 2008.

[12] Marcos Nascimento, Eliane Araújo, Dalton Serey y Jorge César A. de Figueiredo. Giving automated feedback about student code identifiers: a method based on the description of programming problem. En *Brazilian Symposium on Computers in Education*, vol. 30, no. 1, p. 537, noviembre 2019.

[13] Eliane S. Wiese, Anna N. Rafferty y Armando Fox. Linking Code Readability, Structure, and Comprehension among Novices: It's Complicated. En *Proceedings of the 41st ACM/IEEE International Conference on Software Engineering*, pp. 84-89, 2019.

[14] Michael Woodley y Sam Kamin. Programming studio: A course for improving programming skills in undergraduates. En *Proceedings of the 38th SIGSE Technical Symposium on Computer Education*, pp. 531–535, 2007.