

6. Codificación

Introducción

Principios y pautas

Errores comunes de codificación

1. Programación Estructurada ★
2. Ocultamiento de la información
3. Prácticas de programación
4. Estándares de codificación

Proceso de Codificación ★

1. Proceso de Codificación Incremental ★
2. Test Driven Development (TDD) ★
3. Programación de a pares ★
4. Control de código fuente y construcción ("built") ⚡

Refactorización ★

Ventajas:

Indicios de necesidad de refactorización

Tipos de Refactorizaciones ★

Verificación ⚡

Introducción

La codificación comprende la implementación del diseño elaborado. Se busca que sea de la mejor forma posible para:

- Mejorar el desempeño de la **etapa del testing**
- Facilitar el **mantenimiento**
- **Disminuir costos** (relacionado con testing y mantenimiento)
- Hacer que el **código sea fácil de leer y comprender**.



El objetivo de codificación NO es reducir los costos de implementación, si no del **TESTING** y **MANTENIMIENTO**.

Principios y pautas

Nuestro objetivo, como dijimos, es escribir programas simples y fáciles de leer con la menor cantidad de errores (bugs) posibles. Se mencionan aquí errores comunes dentro de la etapa de codificación, y los principios y pautas que buscan solucionar estos errores comunes.

Errores comunes de codificación

1. Memory leaks
2. Liberar memoria ya liberada
3. Desreferencias de punteros a NULL

4. Falta de unicidad en direcciones
5. Errores de sincronización (deadlocks, condiciones de carrera, sincronización inconsistente)
6. Segmentation faults (índice fuera de límites de arreglo)
7. Excepciones aritméticas (dividir por cero)
8. Pobre contemplación de casos bordes (ej: \leq por $<$, etc).
9. Uso ilegal de `&` en lugar de `&&`
10. Errores de manipulación de strings
11. Buffer overflow

A partir de estos errores, aparecieron los siguientes principios y pautas:


1. Programación Estructurada

Alienta a seguir cierta estructura de código para su simplificación en la escritura y lectura. Es consecuencia del mal uso de los go-tos.


El objetivo de este tipo de programación es simplificar la estructura de los programas de manera que sea más fácil razonar sobre ellos.

Dentro de este tipo de programación, podemos entender la estructura de programas en dos tipos que conviven entre sí:

1. **Estática:** es el orden de las sentencias en el código literal.

 La que se usa para justificar el comportamiento de un programa.

2. **Dinámica:** el orden real de ejecución de las sentencias en el código.

 La que se usa para hablar de la corrección de un programa (o sea, si se obtiene el valor esperado).

Entendiendo estas dos vistas sobre la estructura de un programa, lo que se busca con programación estructurada es **escribir programas cuya estructura dinámica es la misma que la estática**. En otras palabras, las sentencias se ejecutan en el mismo orden que las presenta el código. Esto permitiría la simplificación de:

- La comprensión de los programas
- El razonamiento sobre los programas

Para lograr esto, se utilizan **constructores de programación estructurada** no arbitrarios (los ifs, whiles pueden pensarse como constructores dentro de la programación estructurada).

2. Ocultamiento de la información

Se busca que la información guardada en estructuras de datos manejadas por los programas, sean ocultadas ante la globalidad de operaciones de un programa. **Sólo deben ser expuestas a pocas operaciones**.

Si se sigue este principio, **disminuimos acoplamiento**.

3. Prácticas de programación

Se siguen las siguientes prácticas

1. Constructores de control
2. No usar Go-tos
3. Ocultamiento de la información
4. Tipos definidos por usuario: facilita lectura del programa
5. Tamaño de módulos: disminuir los tamaños para aumentar la cohesión
6. Interfaz del módulo: debe ser simple
7. Robustez: tener en cuenta los casos bordes
8. Efectos secundarios: evitarlos y escribir documentación.

Entre otras:

Bloque “catch” vacío: Realizar siempre alguna acción por defecto en lugar de nada.

“if” o “while” vacío: Pésima práctica.

Switch case: Usar default.

Valores de retorno en lecturas: leer para lograr robustez.

“return” en “finally”: No usar.

Fuentes de datos confiables: Desconfiar (usar psw, hash, etc.).

Dar importancia a las excepciones: los casos excepcionales son los que tienden a hacer que el programa funcione mal.

4. Estándares de codificación

Este principio es usado para proveer pautas para los programadores a la hora de leer y escribir código. Algunas convenciones (orientadas a Java) son:

- Convenciones de Nombre:
 - Nombres de paquetes en minúscula.
 - Nombres de tipos que sean sustantivos comenzando con mayúscula.
 - Nombres de variables que sean sustantivos con minúscula.
 - Constantes en mayúscula.
 - Nombres de métodos comienzan con verbos con minúscula.
 - Variables y métodos booleanos prefijados con *is*.
- Archivos:
 - Las fuentes tiene extensión “.java”.
 - Cada archivo contiene sólo una clase externa con igual nombre.

- Longitud de la línea debe ser menor a 80 caracteres.
- Sentencias:
 - Inicializar variables cuando se declaran.
 - Declarar variables en el scope más pequeño posible.
 - Declarar conjuntamente variables que están relacionadas.
 - Declarar separadamente variables no relacionadas.
 - Las variables de clases nunca deben ser públicas.
 - Inicializar variables de los loops justo antes de éstos.
 - Evitar uso de *break* y *continue* en loops.
 - Evitar sentencias ejecutables en condicionales.
 - Evitar uso de *'do ... while'*.
- Comentarios y "layout":
 - Comentarios de una sola línea para un bloque deben alinearse con el bloque de código.
 - Debe haber comentarios para todas las variables más importantes describiendo qué representan.
 - Un bloque de comentario debe comenzar con una línea conteniendo sólo */** y finalizar con otra conteniendo sólo **/*.
 - Los comentarios en la misma línea de una sentencia deben ser cortos y alejados a la derecha.

Proceso de Codificación ★

Proceso que refiere a la etapa del desarrollo de software en la que los programadores escriben el código real que implementa la funcionalidad de los módulos diseñados previamente.

Repasando un poquito:

- La codificación comienza tan pronto se disponga de la especificación del diseño de los módulos.
- Se hace una asignación de módulos: los módulos se asignan a programadores individuales o equipos específicos.
- Se emplea un tipo de desarrollo
 1. Top-down: comenzar por módulos de niveles superiores
 2. Bottom-up: comenzar por módulos de niveles inferiores

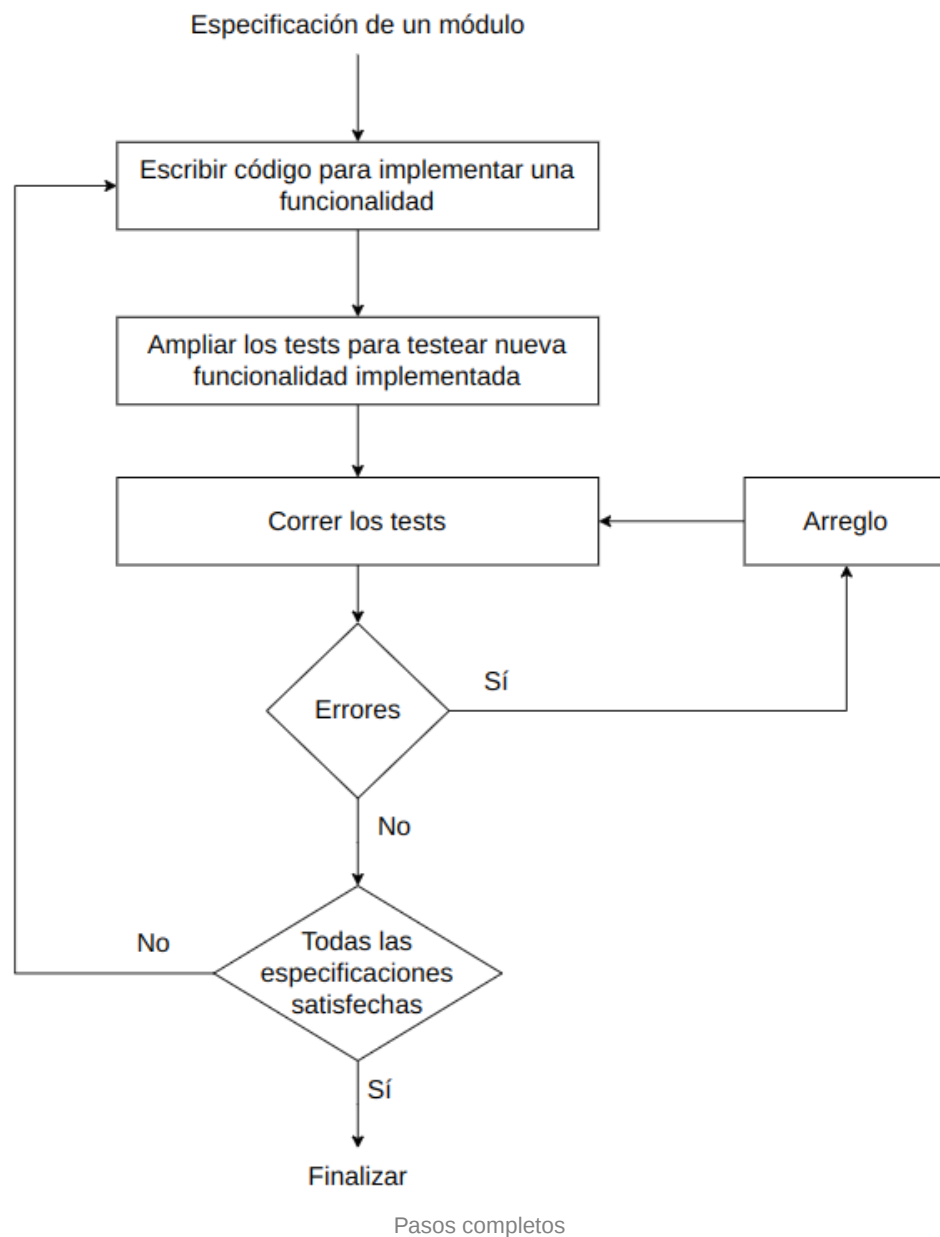
Ahora, pasando a la codificación en sí, la misma se puede llevar a cabo siguiendo alguno de los siguientes procesos. Entre ellos:

1. Proceso de Codificación Incremental ★

Consiste en implementar los módulos en etapas o incrementos sucesivos. Cada incremento agrega funcionalidad adicional al sistema, lo que permite una entrega temprana de partes funcionales del

software. Los pasos resumidos serían:

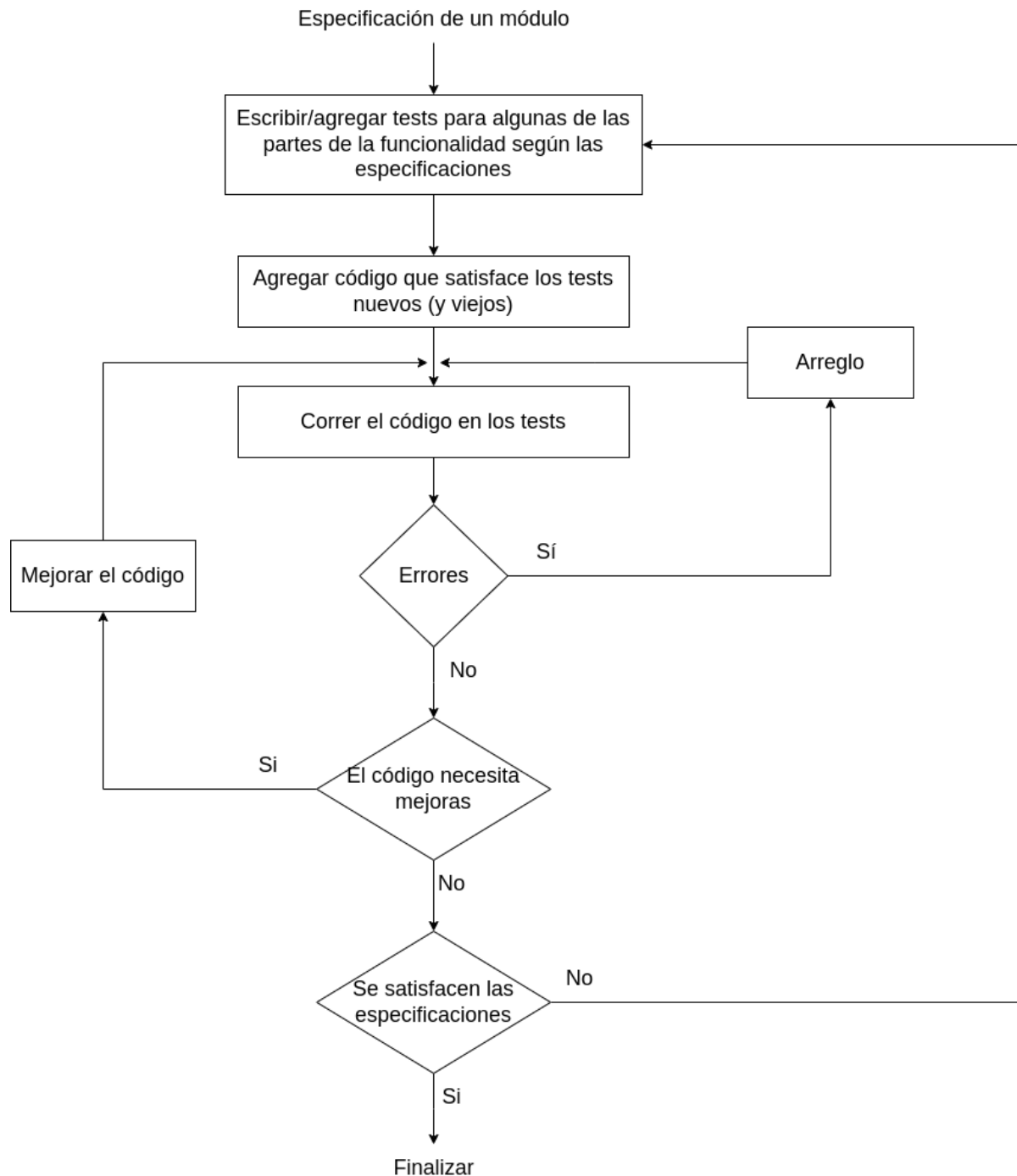
1. Escribir código del módulo
2. Realizar un test de la unidad
3. Si hay error: arreglar bugs y repetir tests.
4. Si no hay error: chequear que las especificaciones estén satisfechas
 - a. Si lo están: finalizar.
 - b. Si no lo están: volver a 1.



Como vemos, el uso de tests a la hora de la implementación es un ingrediente muy importante en la codificación. Veremos ahora un tipo de desarrollo dirigido por la elaboración de éstos tests:

2. Test Driven Development (TDD) ★

Metodología de desarrollo que se centra en escribir tests antes del código de la funcionalidad en sí. Los programadores primero escriben los tests, y luego los códigos que hacen que los tests funcionen. Pasos:



Tener en cuenta que la responsabilidad de asegurar cobertura de toda la funcionalidad radica en el **diseño de los casos de test** y no en la codificación.

Este approach es muy útil ya que se enfoca en cómo será usado el código a desarrollar dado que los tests se escriben primero. **Ayuda a validar la interfaz del usuario especificada en diseño.**

Entonces:

- La completitud del código depende de cuan exhaustivos sean los casos de test.
- El código necesitará refactorización para mejorar el código posiblemente confuso.

3. Programación de a pares

El código es escrito por dos programadores en lugar de uno solo.

- Los dos programadores diseñan en conjunto algoritmos, estructuras de datos, estrategias, etc.
- Una persona tipea el código, otra lo revisa.
- Se señalan errores y conjuntamente formulan soluciones
- Los roles se alternan periódicamente

4. Control de código fuente y construcción (“built”)

Se realiza un control de código fuente del programa y se construye sobre el mismo usando herramientas que consisten en un repositorio manipulable a través de operaciones básicas como:

- Check out
- Commit
- Update

Estos repositorios mantienen la historia completa de los cambios y todas las versiones más viejas pueden recuperarse. Promueve la coordinación en grandes proyectos.

Refactorización

La refactorización se aplica cuando un programa aumenta su funcionalidad, por lo que nuevo código es agregado, haciendo que los cambios deterioren el diseño viejo (pero funcional) que se tenía. A través de la refactorización, el código puede mejorarse.

Es mejorar el diseño de un código existente pero sin modificar el comportamiento observacional del mismo. O sea, internamente el programa cambia y mejora, pero externamente funciona igual.

Algo que se debe cumplir: si el código a refactorizar es testeado usando unos tests específicos, luego de la refactorización, esos tests deben seguir funcionando sin cambiarlos.



Tener en cuenta que la refactorización **no es corregir bugs**. Se refactoriza código que ya funciona.

Además, tener presente que refactorización se realiza separadamente de la codificación normal!!

Ventajas:

- Reduce acoplamiento
- Incrementa cohesión

- Mejora respuesta al principio abierto-cerrado
- Permite que el diseño mejore continuamente en lugar de degradarse con el tiempo.

A tener en cuenta:

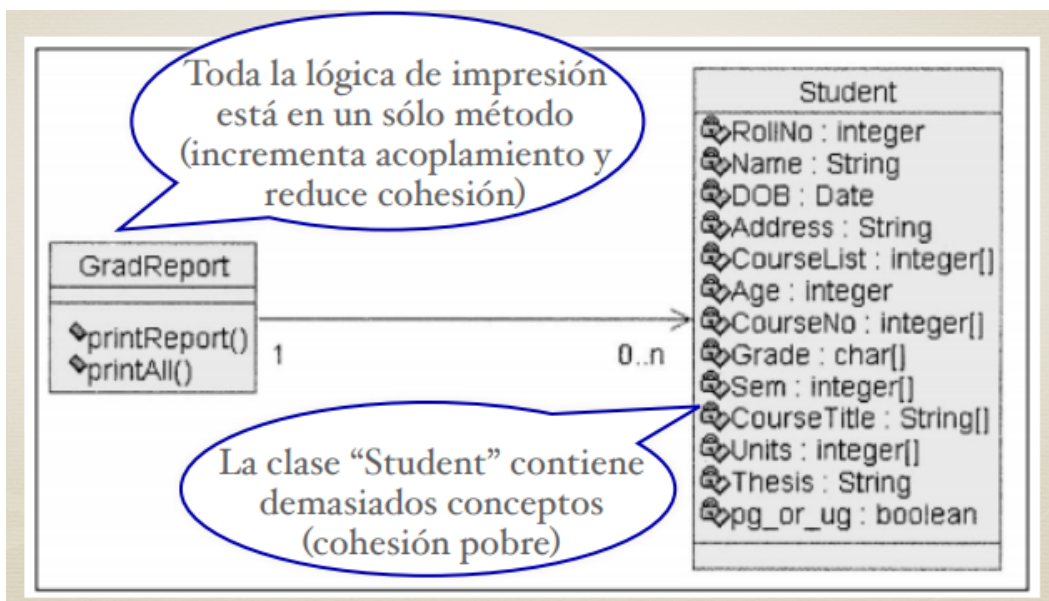
Como refactorizar significa repensar la organización que tiene un código, esto puede “romper” la funcionalidad ya existente. Para que esto no suceda:

- Refactorizar en pasos pequeños
- Disponer de tests automatizados para testear la funcionalidad ya existente.

Ejemplo de refactorización:

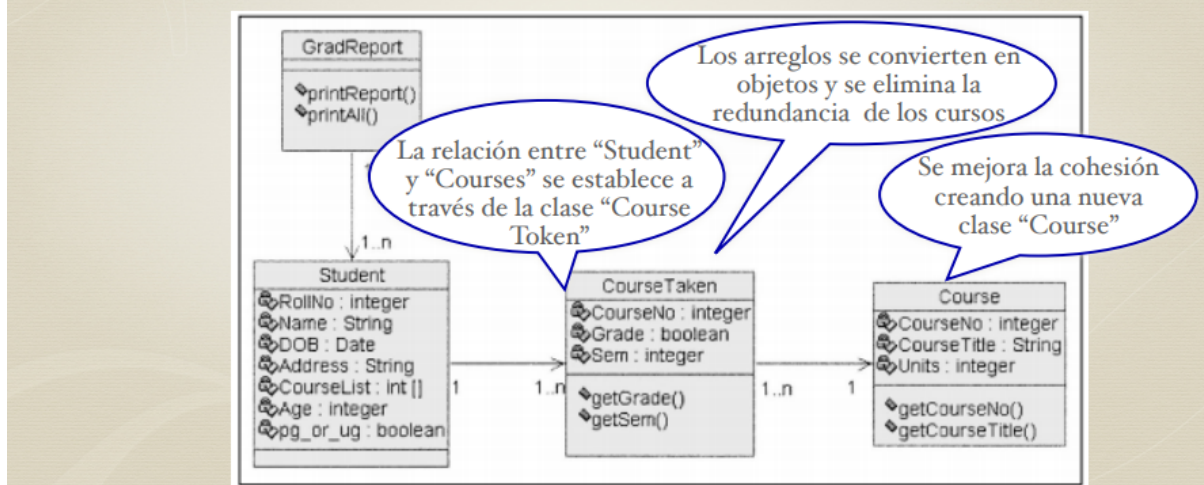
Tenemos un sistema que verifica si un alumno completó los requerimientos de graduación e imprime el resultado junto con las calificaciones. Los alumnos pueden ser de grado o de posgrado.

↓ Mal diseño a refactorizar:

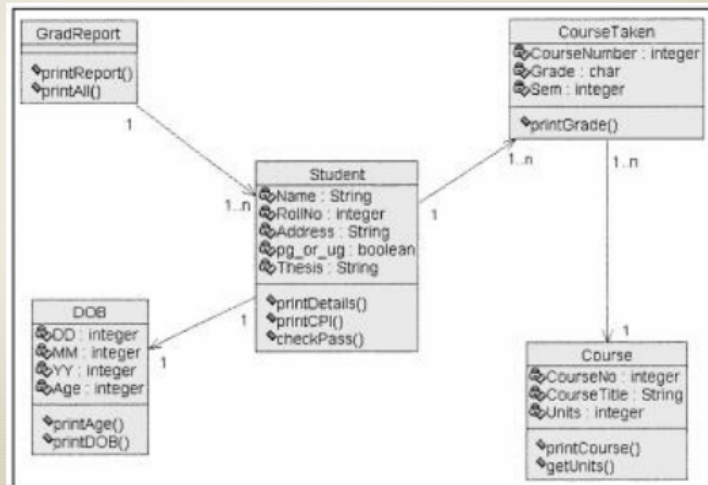


↓ Diseño refactorizado:

Primera refactorización

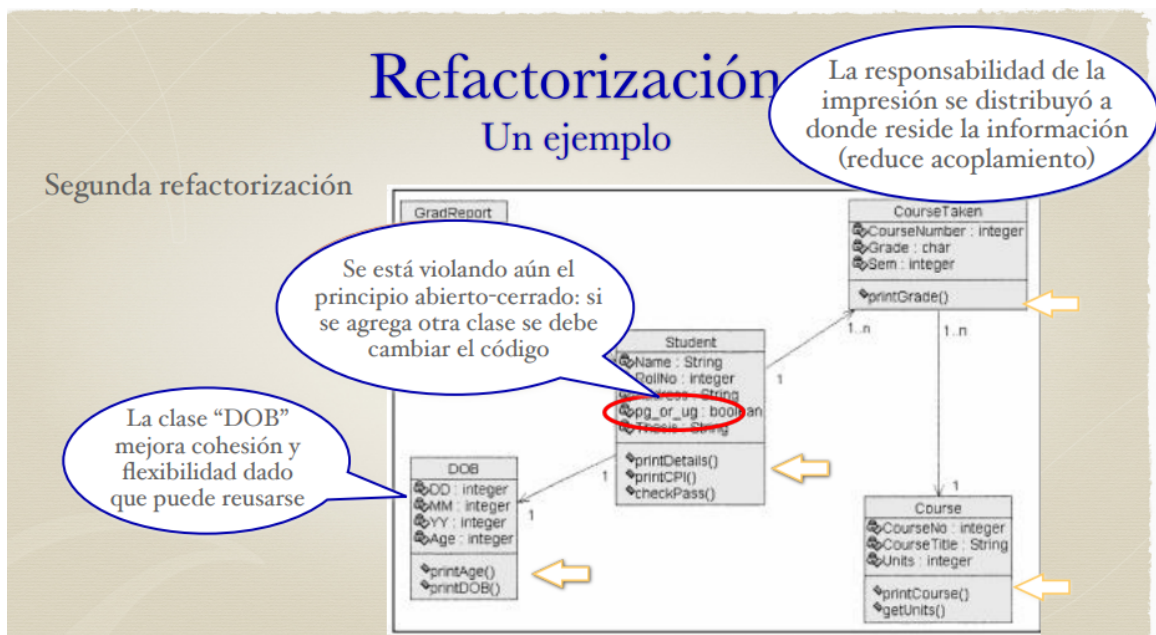


Segunda refactorización

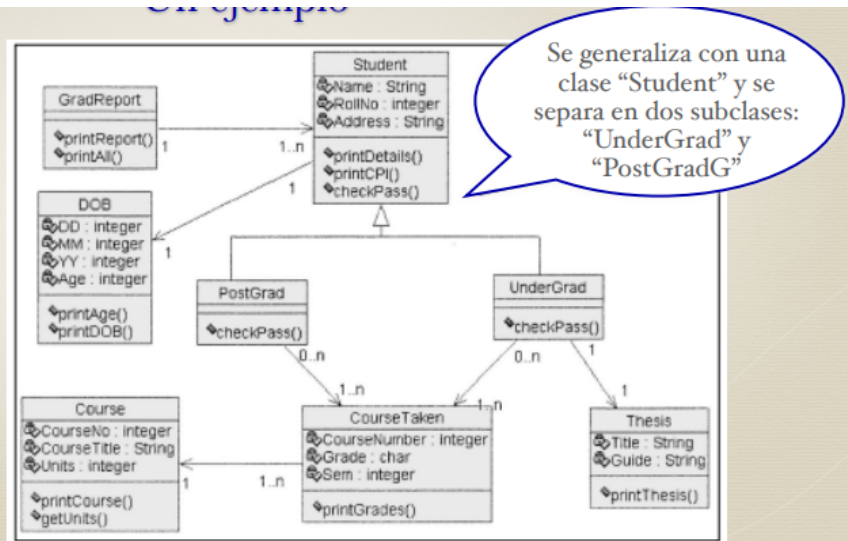


- Clase DOB mejora cohesión y flexibilidad para poder reusarse.
- La responsabilidad de la impresión se distribuyó a donde reside la información, por ejemplo, en CourseTaken. (reduce acoplamiento)
- El booleano pg_or_ug de student es conflictivo ya que si se agrega una nueva clase, se debe cambiar el código.

▼ Diapo de 2da refactorización pero con los comentarios de la profe



Refactorización final



Indicios de necesidad de refactorización

Son signos fáciles de localizar en el código que indican que se puede llegar a refactorizar. Consiste en un análisis caso por caso. Los posibles “malos olores” son:

- **Código duplicado:** si tenemos cosas repetidas, cuando haya que cambiar algo relacionado a eso, se deberán cambiar todas sus repeticiones.
- **Método largo:** no nos gusta que haya cosas que hacen MUCHAS cosas al mismo tiempo.
- **Clase grande:** hay que evitar encapsular muchos conceptos en las clases.
- **Lista larga de parámetros:** no es útil tener listas de parámetros grandes ya que interfaces complejas nos generan problemas.
- **Sentencias “switch”:** si se usa herencia, podríamos repetir las sentencias switch en varias partes del código
- **Generalidad especulativa:** se tiene una subclase innecesaria que es un calco de la superclase.
- **Demasiada comunicación entre objetos:** clases y métodos pueden no ser cohesivos.
- **Encadenamiento de mensajes:** un método llama a otro que llama a otro que llama ... (posible acoplamiento al dope)

Tipos de Refactorizaciones ★

Para mejorar el diseño, la refactorización suele ser en:

▼ 1 Métodos

- **Extracción de métodos:** se hace cuando un método es muy largo. Se quiere separar en métodos cortos cuya signatura indique lo que el método hace.

Parte del código del método largo se refactoriza a estos métodos mas pequeños, al igual que las variables referenciadas en el largo se transforman en parámetros de los pequeños.

También se realiza la extracción de métodos si un método retorna un valor y también cambia el estado del objeto (conviene dividir en dos métodos aquí).

▼ Ejemplo de esto último

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self.saldo = saldo_inicial

    def realizarTransaccion(self, monto):
        # Realiza la transacción y actualiza el saldo
        self.saldo -= monto
        # Retorna el saldo después de la transacción
        return self.saldo

# En este ejemplo, el método realizarTransaccion
# realiza dos tareas: modifica el estado del objeto
# (self.saldo -= monto) y también devuelve un valor
# (return self.saldo). Conviene hacer dos métodos, uno
# que cambie el estado del saldo, y otro que imprima el
# saldo.

# Cortesía de Chat GPT.
```

- Agregar/Eliminar parámetros: se usa para simplificar las interfaces donde sea posible. Se agregan parámetros solamente si los parámetros que ya están no proveen toda la información que se necesita. Eliminar parámetros si se agregaron “por las dudas” pero no se usan.

▼ Ejemplo

```
def calcular_area_rectangulo(dimensiones):
    ancho, alto = dimensiones
    return ancho * alto
```

▼ 2 Clases

- Desplazamiento de métodos: se mueve un método de una clase a otra cuando el mismo actúa demasiado con los objetos de la otra clase. Un ejemplo de esto es tener un método print en un objeto que no tiene mucho sentido que tenga ese método.
- Desplazamiento de atributos: si un atributo se usa más en otra clase, moverlo a esa clase. Esto mejora cohesión y acoplamiento.
- Extracción de clases: si una clase agrupa muchos conceptos, separar cada concepto en una clase distinta. Mejora cohesión.
- Reemplazar valores de datos por objetos: a veces una colección de datos se transforma en una entidad lógica. Conviene separarlos como una clase y definir objetos para accederlos.

▼ Ejemplo de esto

```
# Supongamos que estás desarrollando un sistema de
# gestión de biblioteca y tienes una clase llamada
# Libro que tiene una serie de atributos, como el
# título, el autor, el año de publicación y el ISBN.
# En lugar de mantener estos atributos como variables
# individuales en la clase Libro, puedes refactorizar
# el código para crear una nueva clase llamada
# InformacionLibro que contenga estos atributos como
# propiedades.
class InformacionLibro:
```

```

def __init__(self, titulo, autor, anio, isbn):
    self.titulo = titulo
    self.autor = autor
    self.anio = anio
    self.isbn = isbn

class Libro:
    def __init__(self, informacion, ubicacion):
        self.informacion = informacion
        self.ubicacion = ubicacion

# Crear una instancia de InformacionLibro y luego una instancia de Libro
info_libro = InformacionLibro("El Gran Gatsby", "F. Scott Fitzgerald", 1925, "978-0743273565")
libro1 = Libro(info_libro, "Estantería 1, Estante 2")

# Acceder a los atributos a través de la instancia de InformacionLibro
print(libro1.informacion.titulo)
print(libro1.informacion.autor)

```

▼ 3 Jerarquía de clases

- Reemplazar condiciones con polimorfismo: si se tiene un condicional dentro de una clase, en donde el comportamiento depende de algún indicador de tipo, no se está explotando el poder de la OO. Reemplazar tal análisis a través de una jerarquía de clases apropiada.

▼ Ejemplo de esto

```

# Supongamos que estamos construyendo un sistema
# de envío de productos y tenemos diferentes tipos
# de productos, como libros y electrónicos. Enfoquemos
# el problema primero utilizando condiciones y luego
# con polimorfismo:

# Enfoque con Condiciones:
class Producto:
    def __init__(self, nombre, tipo):
        self.nombre = nombre
        self.tipo = tipo

    def calcular_costo_envio(self):
        if self.tipo == "libro":
            return 5
        elif self.tipo == "electrónico":
            return 2
        else:
            return 10

# Enfoque con Polimorfismo:
class Producto:
    def __init__(self, nombre):
        self.nombre = nombre

    def calcular_costo_envio(self):
        pass

class Libro(Producto):
    def calcular_costo_envio(self):
        return 5

class Electronico(Producto):
    def calcular_costo_envio(self):
        return 2

```

- Subir métodos/atributos: si una funcionalidad o atributo esta duplicado en las subclases, puede subirse a la superclase.

▼ Ejemplo de esto

```
# Supongamos que estamos construyendo un sistema
# de gestión de vehiculos y tenemos diferentes tipos
# de vehículos, como automóviles, motocicletas y
# camiones. Cada tipo de vehículo tiene un atributo
# común, que es el número de ruedas. Primero,
# veamos una implementación sin refactorización:
class Vehiculo:
    def __init__(self):
        pass

class Automovil(Vehiculo):
    def __init__(self):
        self.ruedas = 4

class Motocicleta(Vehiculo):
    def __init__(self):
        self.ruedas = 2

class Camion(Vehiculo):
    def __init__(self):
        self.ruedas = 6

# Ahora, apliquemos la refactorización
# "Subir métodos/atributos" para evitar la duplicación
# de código:
class Vehiculo:
    def __init__(self, ruedas):
        self.ruedas = ruedas

class Automovil(Vehiculo):
    def __init__(self):
        super().__init__(4)

class Motocicleta(Vehiculo):
    def __init__(self):
        super().__init__(2)

class Camion(Vehiculo):
    def __init__(self):
        super().__init__(6)
```



Siempre con el objetivo de mejorar el acoplamiento, la cohesión y el principio abierto-cerrado. (diseño orientado a objetos)

Verificación

Antes de que el código se use por otros, se debe verificar. Tenemos distintas técnicas para hacer esto.

- Inspección de código

Es un proceso de revisión como cualquier otro. Hay un equipo que se enfoca en encontrar defectos y bugs. Es un proceso caro y se usan listas de control para enfocar la atención

▼ Items que se usan en la lista de control

¿Todos los punteros apuntan a algún lado?
¿Se inicializaron todas las variables y punteros?
¿Los índices de los arreglos están dentro de sus cotas?
¿Terminan todos los loops?
¿Hay defectos de seguridad?
¿Se verificaron los datos de entrada?
¿Se satisfacen los estándares de codificación?

- Test de unidad

Es testing pero enfocado en el módulo escrito por un programador. Normalmente este test de unidad es hecho por el mismo programador del módulo. Los tests se dividen en casos y requiere de la escritura de “drivers” que ejecuten el módulo con los casos de test.

- Análisis estático

Herramientas para analizar programas fuentes y verificar existencia de problemas. Suelen ser analizadores que dan “falsos positivos”. Son útiles para memory leaks, código muerto, punteros colgando, etc.

- Métodos formales

Apuntan a demostrar la corrección de programas, es decir, demuestran que los programas implementan la especificación formal dada. Es un área de investigación muy activa y su problema es la escalabilidad del programa (si es muy grande o no). Suele usarse en programas importantes bajo situaciones críticas (software de seguridad crítica, misión crítica, etc).

fin