+Instituto Superior de Formación Técnica Nº 151

Carrera: Analista de Sistemas
2 Año. Algoritmos y Estructuras de Datos II

Trabajo Práctico Nº 6	**Unidad 6**
Modalidad: Semi-Presencial	**Estratégica Didáctica:** Trabajo Grupal.
Metodología de Desarrollo: acordar	**Metodología de Corrección:** acordar docente
Carácter de Trabajo: Obligatorio – con Nota	**Fecha Entrega:** A confirmar por el Docente.

Unidad 6 - Trabajo Práctico - Refactoring

Desarrollo Teórico

1. Que entiende por deuda Técnica
2. Dar 3 Ejemplos de Origen de Deuda Técnica Basar en Ejemplos la Respuesta
3. ¿Como se Puede evitar la Deuda Técnica?
4. Resumir la Definición de Refactoring que da Martin Fowler.
5. Describa Como, Cuando Donde y Por Que debería Refactorizar el Código.
6. Explicar “Conceptualmente” el Flujo trabajo del Refactoring.
7. Que problemas puede traer la Refactorización.
8. Que entiende por “Code Smell”
9. Describa 3 Síntomas de este “Fenómeno” y las Soluciones.
10. Describir 3 Tips a Tener en cuenta a la hora de Refactorizar.
11. Describir Red-Green Refactoring.

Desarrollo Practico

Refactorizar y Resolver en Codigo.

1. Campo de encapsulado

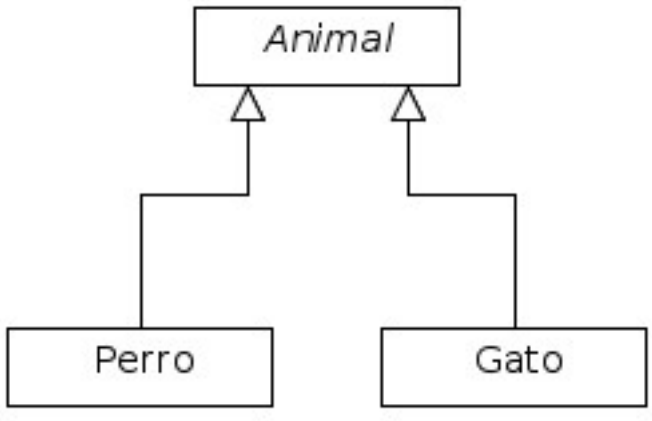
   Encapsulación de campo implica proporcionar métodos que se pueden utilizar para leer/escribir en/desde el campo en lugar de acceder directamente al campo. Estos métodos se denominan métodos accesorios. Encapsule el siguiente código:

```
class A
{
public:
  int var;
};
```

2. Generalización de tipos

Generalización de tipos es una técnica para hacer tipos más generalizados.
Ejemplos:

Mover un método de un hijo a una clase principal
Crear métodos en la Clase Hijos y refactorizar en la clase padre.



3. Métodos de composición

Extract (método)
Convierta el fragmento en un método cuyo nombre explique el propósito del método.

Ejemplo:

```
void printOwing(double amount)
{
    printBanner();

    //print details
    cout << "name:" <<  name << endl;
    cout << "amount" << amount << endl;;
}
```

4. Inline (método)

El cuerpo de un método es tan claro como su nombre. [5]. Cuando el cuerpo de un método es tan claro como el nombre, entonces debe deshacerse del método.
Ejemplo:

```
int getRating()
{
  return (moreThanFiveLateDeliveries()) ? 2 : 1;
}
boolean moreThanFiveLateDeliveries()
{
  return numberOfLateDeliveries > 5;
```

```
}
```

5. ## Temperatura en línea

Variable temporal que se asigna a una vez con una expresión simple.

Ejemplo:

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000);
```

6. ## Explicación de variables

Coloque el resultado de la expresión en variables temporales con un nombre que explique el propósito.

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
   (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 ) {
         // do something
}
```

La Cátedra.

Desarrollo Teórico

1. Que entiende por deuda Técnica

La deuda técnica es una metáfora creada por el informático y programador estadounidense Ward Cunningham, uno de los autores del movimiento ágil. Esta metáfora se usa para describir el problema que genera priorizar soluciones rápidas a corto plazo en el desarrollo de software; en lugar de desarrollar software pensando en el largo plazo, aunque requiera más esfuerzo y trabajo en el momento del desarrollo. En resumen, es el costo del trabajo futuro por haber elegido una solución rápida, en lugar de elegir una mejor solución que demanda más tiempo, pero implica más trabajo y más tiempo.

Al igual que la deuda financiera, la deuda técnica puede utilizarse tanto de forma positiva como negativa, a veces la deuda técnica es una necesidad que me permite solucionar un problema urgente que en un momento futuro pueda arreglar cuando este más cómodo. De manera negativa, siguiendo la metáfora, acumular deuda y pagar muchos intereses puede dejarme en quiebra y ser un desastre en mi vida.

2. Dar 3 Ejemplos de Origen de Deuda Técnica Basar en Ejemplos la Respuesta

Un origen de la deuda técnica es la externalización del código. Un ejemplo de este tipo de deuda técnica ocurrió cuando BioWare, un estudio de desarrollo de videojuegos, tercerizó las animaciones de los rostros del videojuego Mass Effect: Andromeda a un equipo de desarrollo en Rumanía, y el resultado fue un desastre. El problema fue tan grande que muchos de los más famosos influencers en YouTube especializados en videojuegos hacían compilaciones de los rostros, burlándose del problema.

Otro origen es la presión económica. Siguiendo con la industria del videojuego, el estudio polaco CD Projekt Red (creadores de la saga The Witcher) hizo un lanzamiento desastroso del juego Cyberpunk 2077 debido a la presión de los ejecutivos para que los desarrolladores lo sacaran al mercado, “como sea”. Además se acusó a los ejecutivos de realizar un “crunch” despiadado a los desarrolladores. El resultado fue que cuando salió al mercado, el juego tenía una cantidad descomunal de bugs y errores de todo tipo.

El último ejemplo es la falta de gestión de calidad en el desarrollo del software, esto fue especialmente desastroso en el caso de Boeing. Los accidentes de los aviones Boeing 737 Max (fueron en octubre de 2018 y en marzo de 2019) se debieron a un problema con un sistema de control de vuelo automatizado del avión, conocido como Sistema de Aumento de Características de Maniobra (MCAS). Este sistema fue diseñado para evitar que el avión se incline demasiado y entre en pérdida. Sin embargo, en los accidentes de los 737 Max, se encontró que el MCAS se activó incorrectamente debido a datos erróneos de un sensor que mide el ángulo de ataque del avión. Como resultado, el MCAS empujó repetidamente la nariz del avión hacia abajo, incluso cuando los pilotos intentaban elevar el avión. Esto llevó a que los aviones entraran en un picado catastrófico, con el resultado de dos aviones se estrellaron y todos los ocupantes de los mismos resultaron muertos. Creo que es un ejemplo extremo de los consecuencias que pueden aparecer por no gestionar la calidad de desarrollo de un software.

3. ¿Como se Puede evitar la Deuda Técnica?

La deuda técnica es inevitable, pero usando buenas prácticas de programación se puede mitigar considerablemente. Hay medidas preventivas que pueden reducirla como usar procesos estandarizados para la refactorización, y la gestión de calidad, usando herramientas actualizadas para medir y analizar errores. La elección de las tecnologías apropiadas para el proyecto en desarrollo es crucial. Otra medida es la capacitación continua de los desarrolladores es otra excelente idea para disminuirla, también hacer un diseño claro del código, distribuir correctamente responsabilidades en los equipos de trabajo e igualmente mantener una arquitectura informática por medio de vigilancia, medición y gestión de calidad constante.

4. Resumir la Definición de Refactoring que da Martin Fowler.

En el libro de Martin Fowler aparece esta definición de Refactoring:

“...
In my refactoring book, I gave a couple of definitions of refactoring.

Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

Refactoring (verb): to restructure software by applying a series of refactorings without changing its observable behavior. ,,,”

Refactoring es modificar la estructura de un software para hacerlo más entendible y hacer más eficiente su modificación, sin cambiar su comportamiento observable.

5. Describa Como, Cuando Donde y Por Que debería Refactorizar el Código.

La refactorización de código se puede definir como el proceso de editar, eliminar y simplificar implementaciones que ya existen, sin cambiar la funcionalidad que este ya tiene. Refactorizar puede mejorar la legibilidad del código, y también puede hacer que el proceso de pruebas y depuración de este sea mucho más eficiente. Este proceso no contempla corrección de errores, son dos etapas separadas y es muy importante mantenerlas separadas, ya que después de una refactorización, hay que verificar que esta no modifico el comportamiento del software, para asegurarnos que sigue funcionando correctamente.

Respondiendo al como, la refactorización es revisar el código existente haciéndole mejoras, sin cambiar su comportamiento. Para lograr esto hay varios métodos aplicables, como hacer testing y que el código pase las pruebas a la que es sometido,regla importante, ya que el código no que pasa las pruebas, no se debe refactorizar. Otro método es buscar el código que huele mal y luego simplificar el código. Después de la simplificación hay que volver a hacer pruebas, ya que no es lógico refactorizar y que el código empiece a fallar. Este proceso se debe repetir todas las veces que sea necesario hasta que el "olor" desaparezca.

El momento para refactorizar es "todo el tiempo" en una situación ideal. En general siempre que se pueda refactorizar, es decir, debe ser una práctica continua durante el ciclo de vida del desarrollo de software (aunque no siempre se pueda). Una metodología para saber cuando refactorizar es aplicar "la regla de tres" , también conocida como "Tres strikes y tú refactorizas", es una regla práctica para decidir cuándo se deben refactorizar piezas de código similares para evitar la duplicación. Esta regla fue popularizada por Martin Fowler en su libro "Refactoring" y atribuida a Don Roberts. Esta regla establece que cuando dos módulos de mi sistema de software son muy similares, no hay que refactorizar, pero si aparece un tercer modulo igual a los dos que ya son similares entre, hay que refactorizar. Otros momentos son cuando agrego una nueva funcionalidad, cuando aprendo algo sobre el código, cuando arreglo un error, y cuando el código "huele".

Hay que refactorizar en cualquier lugar que se necesite hacerlo, donde se haya identificado como problemático o difícil de entender. Un factor importante en este punto es el cliente, si el acepta que refactorizar es valioso y esta dispuesto a pagar la refactorización y tolerar demoras. Otro criterio es el equipo de trabajo, es decir si un compañero paso a otra área, proyecto o abandono la empresa, el código que escribió, debe ser refactorizado.

Refactorizar evita el deterioro del diseño, disminuye la entropía del mismo, simplifica el código, haciéndolo más legible, comprensible y menos propenso a errores. También disminuye el tiempo de depuración, por que en la refactorización aumenta la comprensión de lo que es y hace la aplicación desarrollada y fomenta la creatividad.

6. Explicar "Conceptualmente" el Flujo trabajo del Refactoring.

Primero se buscan los síntomas que indican problemas profundos del software ("bad smell"), luego se eligen las técnicas que se van a aplicar al refactoring, después se crean y ejecutan pruebas unitarias. Hago un paréntesis para aclara conceptos, una prueba unitaria es una prueba automatizada que tiene como objetivo verificar el funcionamiento de una unidad de código, que se define como la unidad más pequeña testeable de una aplicación. Dependiendo el paradigma, esta puede ser una rutina, una función, un método o una clase. Sigo con el flujo de trabajo, una vez que se realizan las pruebas unitarias, se aplica la

técnica ya elegida de refactoring, terminado el refactoring se vuelven a aplicar las pruebas para evitar detectar problemas en el código ya refactorizado. Por último el código refactorizado se implementa.

7. Que problemas puede traer la Refactorización.

Algunos de los problemas que pueden traer son refactorizar en exceso, en donde la relación costo-beneficio deja de ser rentable y los beneficios que traiga la refactorización son menores al tiempo y horas hombre invertido en él. Hacer refactorización cuando el código no pasa las pruebas nunca es una buena idea, pudiendo devolver al cliente un producto peor que antes de la refactorización.

Hacer refactorización con bases de datos es dificil y hacer que un cliente no pueda usarlas por algún error de la aplicación de refactorización puede tener consecuencias nefastas y finalmente refactorizar las interfaces públicas puede traer problemas para los módulos de código que las implementa.

8. Que entiende por “Code Smell”

El “code smell” es una característica del código que permite a los programadores experimentados intuir que el código no está limpio. Es una indicación superficial que suele hacer referencia a un problema más profundo en el sistema. En palabras más simples, cuando el código huele, significa que es código tiene problemas, dependiendo de lo que se tarde en “limpiarlo”, puede ser que haya que hacerle pequeñas modificaciones o en el peor de los casos desecharlo y tener que reescribirlo.

9. Describa 3 Síntomas de este “Fenómeno” y las Soluciones.

Una clase grande según Martín Fowler puede ser considerada “code smell”, por que una clase que fue creciendo y a la que se le han agregado demasiadas responsabilidades, puede ser difícil de entender y mantener. Además de romper con el primer principio SOLID(Single responsability), refactorizar una clase en grande en clases más pequeñas con una sola responsabilidad puede limpiar el “mal olor”.

Otro síntoma es la “cirugía de escopetas”, que emite un “mal olor” por que cada vez que hay un cambio pequeño en el código, se necesita hacer muchos cambios en varias clases o funciones. Las razones de este “code smell” pueden surgir por

- Mala separación de responsabilidades
- Violación del principio de single responsability
- Fallas en entender el comportamiento de mi clase o función
- Errores de diseño como el uso de un patrón no adecuado al problema.
- Hacer demasiado copy-paste de código de mi aplicación sin adaptarlo al modulo que estoy desarrollando.

Para solucionar “la cirugía de escopetas”, se pueden aplicar estos métodos, el encapsulamiento, la descomposición de clase, aplicar mejor el principio de abstracción para encapsular comportamiento en las clases abstractas o interfaces y elegir mejor el patrón de diseño a aplicar.

El último concepto es el de la prueba ansiosa, que se refiere a una prueba unitaria que verifica más de un comportamiento específico. Esto puede hacer que la prueba sea

difícil de entender y mantener, y puede llevar a resultados de pruebas poco claros. En el contexto del refactoring, la “Prueba ansiosa” se considera un “code smell” porque indica un posible problema en la estructura del código que podría beneficiarse de la refactorización. Específicamente, la presencia de pruebas ansiosas puede dificultar la refactorización del código que está siendo probado, ya que cambiar cualquier comportamiento podría requerir la actualización de varias pruebas ansiosas. Las pruebas ansiosas pueden hacer que sea difícil identificar exactamente qué comportamiento está fallando cuando una prueba falla, ya que cada prueba cubre múltiples comportamientos.

10. Describir 3 Tips a Tener en cuenta a la hora de Refactorizar.

1.- Buscar código duplicado y eliminarlo, puedo guardar ese código en un módulo e invocarlo cuando lo necesite.
2.- Fijarme en los comentarios, si hay partes del código que están profusamente conectadas, podría ser un indicativo de que esas partes necesitan refactorización.
3- Revisar las pruebas unitarias, por que son fundamentales para que el proceso de refactorización cumpla su cometido.

11. Describir Red-Green Refactoring.

El Red-Green Refactoring es un enfoque de desarrollo de software que se utiliza para mejorar el código paso a paso, manteniendo el código funcional en todo momento. El nombre de la técnica proviene de los colores rojo y verde de los semáforos. Se usa dentro de un enfoque mayor que es el Desarrollo Guiado por Pruebas (TDD). La sigla TDD(Test-Driven-Development), es una metodología que enfatiza la escritura de pruebas unitarias antes de escribir el código.

El red-green-refactoring, se divide en tres fases:

1.- No se le permite escribir ningún código de producción a menos que sea para hacer que una prueba unitaria fallida pase. Esto significa que se debe construir la prueba unitaria, que la primera vez debe fallar. Y recién después escribir código que pase esta prueba. Solo cuando la prueba se supera, se pasa a la siguiente fase.
2.- No se le permite escribir más de una prueba unitaria de lo que es suficiente para fallar; y los errores de compilación son errores. Esto significa que solo puedo enfocarme en una prueba a la vez. Algo así como el “paso a paso futbolero”. De manera más formal, no hago una prueba unitaria nueva, si la anterior no fue resuelta satisfactoriamente. Que el código no compila también es una falla y no se debe avanzar hasta que lo haga.
3.- No se le permite escribir más código de producción del que es suficiente para pasar la prueba unitaria que falla. Esto significa que no se debe escribir código innecesario, o sea no se debe agregar código adicional que no este relacionado con hacer pasar la prueba unitaria

Estas reglas ayudan a crear minimizar errores, haciendo que todo el código haya sido probado, y que las pruebas sigan el proceso iterativo incremental del desarrollo de software.